**Лабораторная работа 2.**

**Разработка программных модулей**

**Элементы сетевого программирования**

Цель: Ознакомиться с теорией и практикой создания элементов сетевого программирования в среде .Net.

**Теория**

**Классы WebRequest и WebResponse**

**Асинхронные запросы страниц**

Дополнительным средством класса WebRequest является способность запрашивать страницы асинхронно. Это средство существенно, потому что между отправкой запроса на хост и получением ответа может существовать ощутимая задержка. Такие методы, как WebClient.DownloadData() и WebRequest.GetResponse(), не вернут управления, пока не будет готов ответ сервера. Вряд ли захочется "замораживать" приложение на длительный период, и потому в таких сценариях лучше применять методы BeginGetResponse() и EndGetResponse().

Метод BeginGetResponse() работает асинхронно и возвращает управление практически мгновенно. "За кулисами" исполняющая среда асинхронно управляет фоновым потоком, чтобы получить ответ от сервера. Вместо возврата объекта WebResponse, метод BeginGetResponse() возвращает объект, реализующий интерфейс IAsyncResult. С этим интерфейсом можете продолжить работу или подождать, пока не станет доступным ответ, и затем вызвать EndGetResponse() для сбора результатов.

Можно также передать делегат обратного вызова в метод BeginGetResponse(). Целью делегата обратного вызова должен быть метод, возвращающий void и принимающий ссылку IAsyncResult в качестве параметра. Когда рабочий поток завершает получение результата, исполняющая среда вызывает делегат обратного вызова, чтобы проинформировать о завершении работы. Как показано в следующем коде, вызов EndGetResponse() в методе обратного вызова позволяет извлечь объект WebResponse (модифицируем первый пример):

private void request\_Click(object sender, RoutedEventArgs e)

{

// Создать объект запроса

WebRequest request = WebRequest.Create(txb\_url.Text);

request.BeginGetResponse(new AsyncCallback(OnResponse), request);

}

protected void OnResponse(IAsyncResult ar)

{

WebRequest request = (WebRequest)ar.AsyncState;

WebResponse response = request.EndGetResponse(ar);

// Читаем ответ

}

Обратите внимание, что для извлечения исходного объекта WebRequest методу BeginGetResponse() можно передать этот объект во втором параметре. Второй параметр является ссылкой на объект, и он известен как параметр состояния. Во время выполнения метода обратного вызова тот же объект состояния можно извлечь с использованием свойства AsyncState интерфейса IAsyncResult.

**Теория**

**Классы .NET для IP-адресов**

В Интернете серверы и клиенты идентифицируются по IP-адресу либо имени хоста (также известному, как DNS-имя). В общем случае, имя хоста — это дружественное для человека имя, которое вводится в поле адреса веб-браузера, такое как www.professorweb.ru или www.microsoft.com. IP-адрес — это идентификатор, используемый компьютерами для опознания друг друга. IP-адреса представляют собой идентификаторы, применяемые для обеспечения достижения запросами и ответами соответствующих машин. Один компьютер может даже иметь несколько IP-адресов.

В наше время IP-адреса обычно представлены 32-битным значением. Вот пример 32-битного IP-адреса: 192.168.1.100. Этот формат IP-адреса определен в протоколе Internet Protocol version 4.

Поскольку количество компьютеров и прочих устройств, претендующих на место в Интернете, неуклонно растет, была разработана более новая система адресации — Internet Protocol version 6. Протокол IPv6 предоставляет 64-битные IP-адреса. IPv6 потенциально может представить до 3e+28 уникальных адресов. Среда .NET Framework позволяет приложениям работать как с IPv4, так и с IPv6.

Чтобы имена хостов работали, сначала должен быть отправлен сетевой запрос на трансляцию имени хоста в IP-адрес; эту задачу решает один или более DNS-серверов.

DNS-сервер хранит таблицу, отображающую имена хостов на IP-адреса для всех известных ему компьютеров, а также IP-адреса других DNS-серверов, где можно искать имена хостов, которые ему неизвестны. Локальный компьютер должен всегда знать, по крайней мере, один DNS-сервер. Сетевые администраторы конфигурируют эту информацию при настройке компьютера.

Прежде чем отправлять запрос, компьютер сначала спрашивает у DNS-сервера IP-адрес, соответствующий введенному имени хоста. Получив корректный IP-адрес, компьютер может отправить ему запрос через сеть. Все это нормально работает "за кулисами", пока пользователь путешествует по Интернету.

В .NET Framework предлагается множество классов, которые помогают в процессе поиска IP-адресов и нахождении информации о компьютерах-хостах.

**Класс IPAddress**

Класс IPAddress представляет IP-адрес. Сам адрес доступен в виде свойства GetAddressBytes и может быть преобразован в десятичный формат с разделителями-точками с помощью метода ToString(). В этом классе также реализован статический метод Parse(), который эффективно выполняет преобразование, обратное ToString() — из десятичного формата с разделителями-точками в IPAddress:

IPAddress ip = IPAddress.Parse("234.58.78.9");

byte[] adress = ip.GetAddressBytes();

string ipString = ip.ToString();

В классе IPAddress есть несколько открытых, доступных только для чтения полей, которые возвращают предопределенные IP-адреса:

**IPAddress.None**

Возвращает адрес, который означает, что ни один сетевой интерфейс не должен использоваться. Это поле используется классом Socket, чтобы указать серверу не ожидать активности клиента.

**IPAddress.Loopback**

Возвращает предопределенный адрес обратной связи 127.0.0.1. Этот адрес используется не для соединения с сетью, а для локальных операций на одной машине.

**IPAddress.Broadcast**

Возвращает широковещательный IP-адрес. В широковещательных сообщениях можно посылать данные всем компьютерам в локальной сети.

**IPAddress.Any**

У компьютера может быть несколько сетевых плат с несколькими IP-адресами. Сокет использует IPAddress.Any, чтобы ожидать действия на любом из этих сетевых интерфейсов.

**Класс IPHostEntry**

Класс IPHostEntry инкапсулирует информацию об определенном компьютере-хосте. Этот класс делает имя хоста доступным через свойство HostName (которое возвращает строку), а свойство AddressList возвращает массив объектов IPAddress.

**Класс Dns**

Класс Dns способен взаимодействовать с DNS-сервером по умолчанию для извлечения IP-адреса. Он имеет два важных статических метода —GetHostEntry(), который использует DNS-сервер для получения деталей хоста по заданному его имени, и GetHostByAddress(), который также возвращает детали хоста, но на этот раз используя IP-адрес. Оба метода возвращают объект IPHostEntry.

Класс Dns отличается от классов IPAddress и IPHostEntry тем, что обладает способностью действительно взаимодействовать с серверами для получения информации. В отличие от него IPAddress и IPHostEntry — это скорее просто структуры данных с удобными свойствами, обеспечивающими доступ к лежащим в основе данным.

**Задание 1.**

Реализовать пример WPF-приложения, использующего возможности класса Dns:

public MainWindow()

{

InitializeComponent();

string hostname = "www.google.com", message = "IP адреса для домена " + hostname + "\n";

IPHostEntry entry = Dns.GetHostEntry(hostname);

foreach (IPAddress a in entry.AddressList)

message += " --> " + a.ToString() + "\n";

message += "\nАльтернативное имя домена: ";

foreach (string aliasName in entry.Aliases)

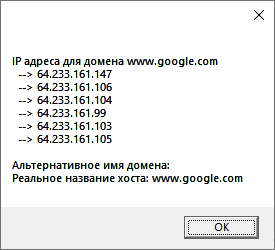
message += aliasName + "\n";

message += "\nРеальное название хоста: " + entry.HostName;

MessageBox.Show(message);

}

Результат работы:



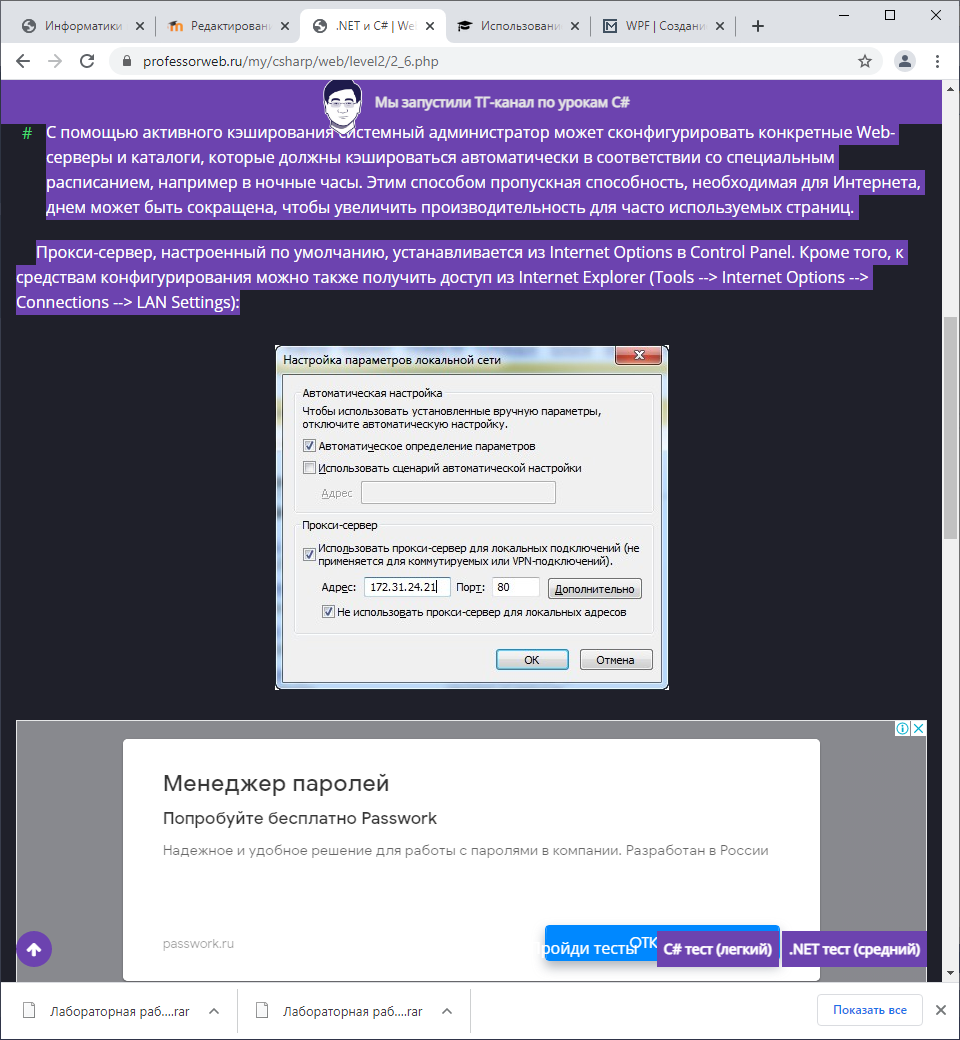
**Web- прокси**

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-pecypcы 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 172.31.24.21 и слушает порт 80. 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.

**Класс WebProxy**

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

Таблица. 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:

WebProxy wp = new WebProxy("192.168.10.2", true);

wp.Credentials = new NetworkCredential("user1","user1Password");

WebRequest wrq = WebRequest.Create("https://www.birsk.ru/");

wrq.Proxy = wp;

WebResponse wrs = wrq.GetResponse();

**Аутентификация и разрешения**

Если Web-cepвep 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("student","qwerty","birsk.ru");

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

WebRequest request = WebRequest.Create("http://www.birsk.ru/");

request.Credentials = credential;

Если нужна разная информация удостоверений личности для разных URI, можно использовать класс CredentialCache, как показано в следующем коде. С таким кэшем также определяется тип аутентификации для конкретного соединения. Здесь используется базовая аутентификация для Web-сайта www.unsecure.com и аутентификация на основе дайджестов для Web-сайта www.moresecure.com, для которого через сеть посылается не пароль, а хеш-код:

CredentialCache ccache = new CredentialCache();

ccache.Add(new Uri("http://www.birsk.ru/"), "Basic", new NetworkCredential("student", "qwerty"));

ccache.Add(new Uri("http://www.birskdo.ru/"), "Digest", new NetworkCredential("student", "qwerty","domain"));

**Разрешения**

Всякий раз когда используются сетевые классы, требуются разрешения. Для вопросов сетевого обмена рассмотрим три типа разрешения:

* Разрешение **DnsPermission** требуется для поиска имени DNS с помощью класса Dns.
* **WebPermission** используется классами из пространства имен System.Net, которые отправляют данные в Интернет и получают данные с помощью URI.
* **SocketРеrmission** используется для приема данных на локальном сокете или соединения с хостом через транспортный протокол.

Приложения, установленные локально в системе, пользуются полным доверием, поэтому все разрешения доступны по умолчанию. Приложения платформы. .NET также могут запускаться с общего сетевого ресурса, а сборки могут загружаться из Интернета — в этих ситуациях многие разрешения недоступны по умолчанию. Следовательно, для этих приложений надо конфигурировать параметры настройки безопасности. Сначала обсудим программные аспекты безопасности, после чего рассмотрим, как можно конфигурировать разрешения.

**DnsPermission**

Когда используется класс Dns для поиска IP-адреса, требуется разрешение DnsPermission. Для него различаем только значения "признать" и "отвергнуть". Запросы DNS или абсолютно не ограничиваются, или не разрешаются совсем.

**WebPermission**

WebPermission требуется для таких классов, как WebRequest и WebResponse, чтобы отправлять данные в Интернет и получать их из Интернета.

В этом случае различаются разрешения "согласиться" (Accept) и "соединиться" (Connect). Разрешение Accept нужно для URI, используемых внутри классов и методов. Клиентским приложениям, использующим URI для соединения с сервером, требуется полномочие Connect. У класса WebPermission также есть список URI, с которыми можно соединиться, и список URI, с которыми можно согласиться.

**SocketPermission**

Разрешения SocketРеrmission нужны для классов сокетов из пространства имен System.Net.Sockets. Это разрешение — самое гибкое из трех классов сетевых разрешений.

Для серверных приложений, ожидающих запросы на соединения от клиентов, в конструктор передается значение перечисления NetworkAccess.Accept; клиентские приложения, соединяющиеся с серверами, используют значение NetworkAccess.Connect. Можем ограничить соединение конкретными хостами и номерами портов и определить используемый транспортный протокол.

**Использование атрибутов разрешения**

Если требуемое разрешение недоступно, программа завершается с исключением SecurityException, как только вызван привилегированный метод. До порождения исключения пользователь мог какое-то время работать с приложением, и может потерять данные, если не обработать исключение аккуратно. Хороший способ, позволяющий этого избежать, состоит в том, чтобы отметить сборку необходимыми разрешениями.

Если для получения данных из Интернета используется класс WebRequest, требуется разрешение WebPermission. Можно отметить классы и методы, требующие разрешения, атрибутом WebPermission (реализованным в классе WebPermissionAttribute) следующим образом:

[WebPermission(System.Security.Permissions.SecurityAction.Demand,

ConnectPattern = "www.professorweb.ru")]

class PermissionDemo

{

}

В данном случае исключение SecurityException возникает, как только создается экземпляр класса PermissionDemo. Если хотим, чтобы эта проверка выполнялась при старте программы, можно применить атрибут WebPermission ко всей сборке:

[assembly: WebPermission(System.Security.Permissions.SecurityAction.Demand,

ConnectPattern = "www.birsk.ru")]

Если этот атрибут применен к сборке, при старте программы среда выполнения проверяет, имеет ли программа необходимое разрешение. Если она не обладает требуемым разрешением, выполнение прекращается немедленно, прежде чем пользователь ввел (или потерял) информацию.

Со всеми атрибутами разрешения конструктору передается значение перечисления SecurityAction. Здесь рассмотрим только самые важные значения перечисления:

Таблица. Перечисление SecurityAction

|  |  |
| --- | --- |
| Значения перечисления | Описание |
| Demand и Deny | Значения перечисления SecurityAction.Demand и SecurityAction.Deny можно использовать с классами и методами. Значением Demand указывается, что классу или методу требуется разрешение, значение Deny определяет, что это разрешение не нужно. |
| RequestMinimum, RequestOptional и RequestRefuse | Значением перечисления Requestxxx можно пользоваться только в контексте сборки, его нельзя указывать с классами и методами. RequestMinimum определяет, что это разрешение обязательно для использования программы.  Значением RequestOptional сообщается, что программа может выполнить некоторую полезную работу без этого разрешения. В этом случае надо постепенно обработать исключение SecurityException.  Значение RequestRefuse определяет, что это разрешение не нужно. Это значение используется в тех случаях, когда возможно неправильное применение разрешения, например вызов сборок, для которых у нас нет исходных текстов и которым мы не доверяем полностью. |

С атрибутом WebPermission можно в дополнение к SecurttyAction установить следующие свойства:

**Accept и AcceptPattern**

В свойстве Accept определяется URI ресурса для использования с классом, методом или сборкой, к которым применен этот атрибут. В свойстве AcceptPattern указывается регулярное выражение, чтобы разрешить или запретить доступ к URI.

**Connect и ConnectPattern**

Два свойства ConnectXX аналогичны свойствам AcceptXX, но отличаются тем, что применяются для строки соединения с URI.

В классе SocketPermissionAttribute определены следующие дополнительные свойства:

**Access**

В этом свойстве определяется допустимый метод доступа к сети. Допустимы только два: Accept и Connect. Значение Accept используется для серверного приложения, слушающего и принимающего соединения клиентов, а значение Connect предназначено для клиента, соединяющегося с сервером.

**Host**

В свойстве Host с использованием синтаксиса DNS или IP-адреса устанавливается имя хоста, к которому относится разрешение.

**Port**

Это строковое свойство, задающее номер порта, для которого требуется разрешение. Свойство может использоваться для ограничения приложений-клиентов некоторыми конкретными серверами. Свойство имеет тип string, поскольку разные протоколы необязательно определяют номер порта как целое число.

**Transport**

С помощью свойства Transport ограничиваются сетевые соединения конкретным транспортным протоколом. Возможные значения: All, ConnectionLess, CormectionOriented, Tcp и Udp. Значение ConnectionLess позволяет использовать все протоколы, не требующие устанавливать соединения, например UDP; значение ConnectionOriented позволяет использовать протоколы, ориентированные на установление соединения, например TCP.

**Сокеты**

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 сокеты и дейтаграммные.

**Потоковые сокеты (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 сокеты 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 сокеты предпочтительнее дейтаграммных. Следовательно, если надежность связи между двумя приложениями имеет первостепенное значение, выбирайте потоковые сокеты.

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**

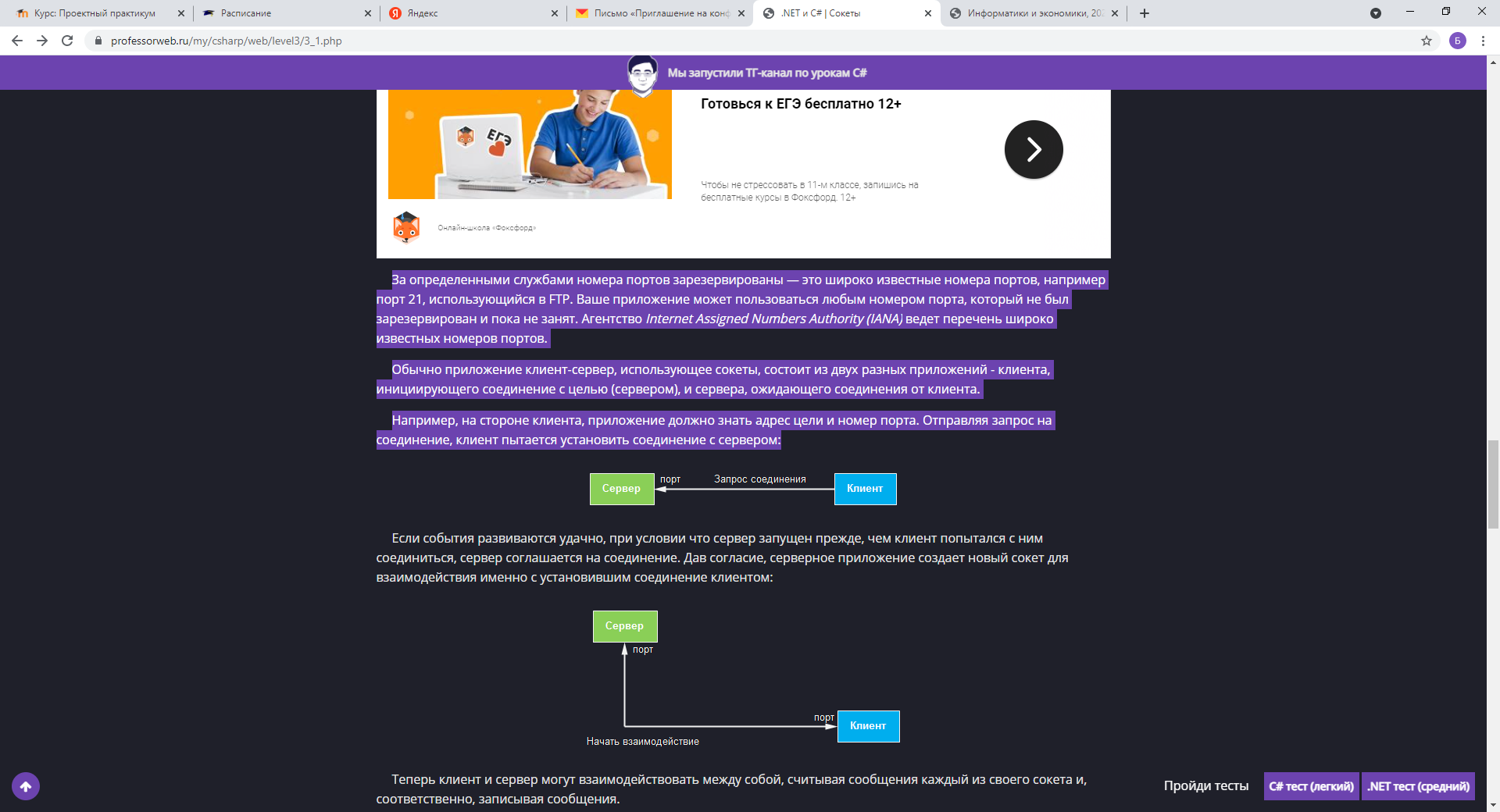
Порт определен, чтобы разрешить задачу одновременного взаимодействия с несколькими приложениями. По существу с его помощью расширяется понятие IP-адреса. Компьютер, на котором в одно время выполняется несколько приложений, получая пакет из сети, может идентифицировать целевой процесс, пользуясь уникальным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

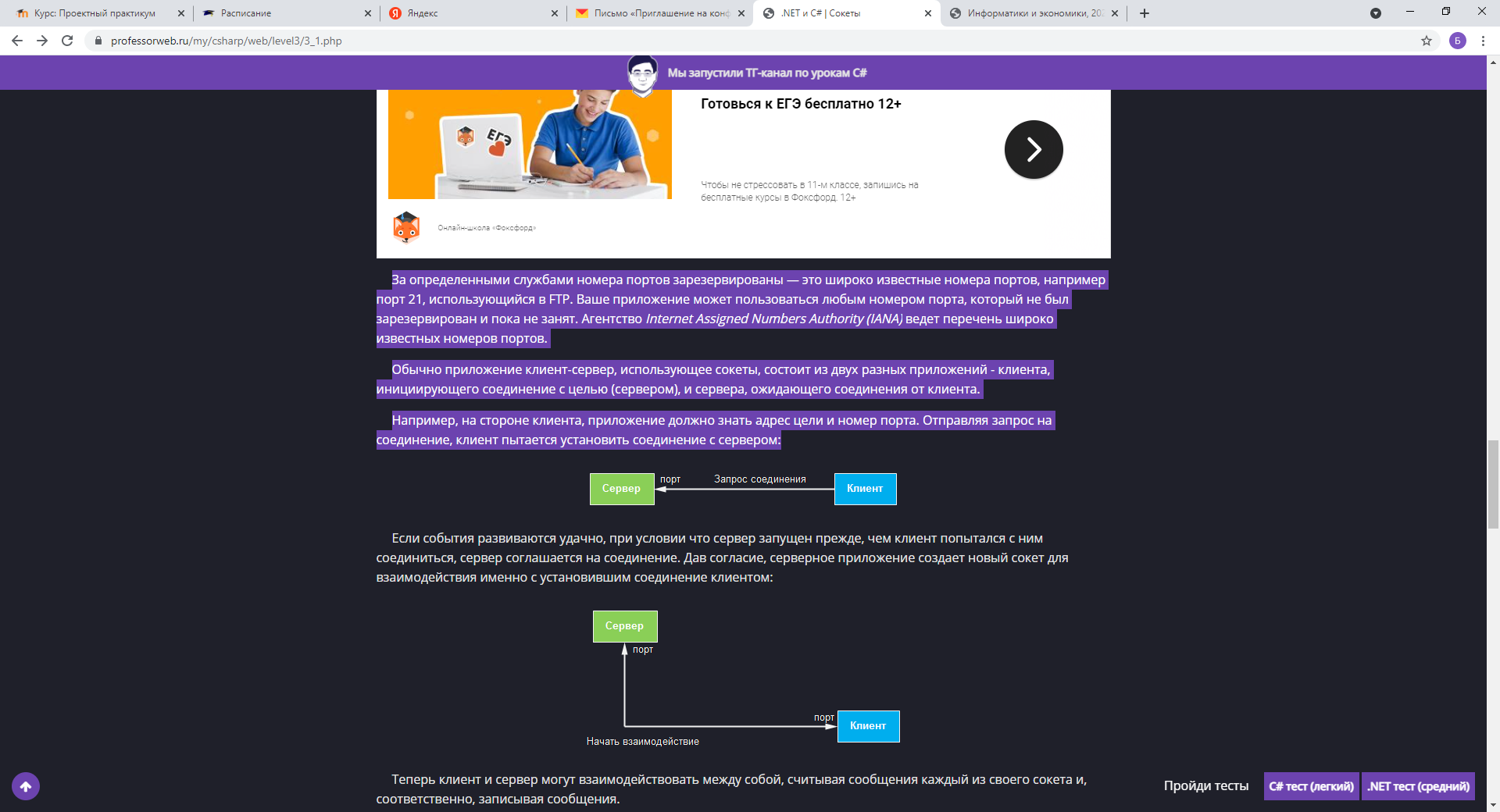
За определенными службами номера портов зарезервированы — это широко известные номера портов, например порт 21, использующийся в FTP. Ваше приложение может пользоваться любым номером порта, который не был зарезервирован и пока не занят. Агентство Internet Assigned Numbers Authority (IANA) ведет перечень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

Обычно приложение клиент-сервер, использующее сокеты, состоит из двух разных приложений - клиента, инициирующего 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.

Таблица. Классы для работы с сокетами

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

Таблица. Свойства и методы класса Socket.

|  |  |
| --- | --- |
| Свойство или метод | 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. |

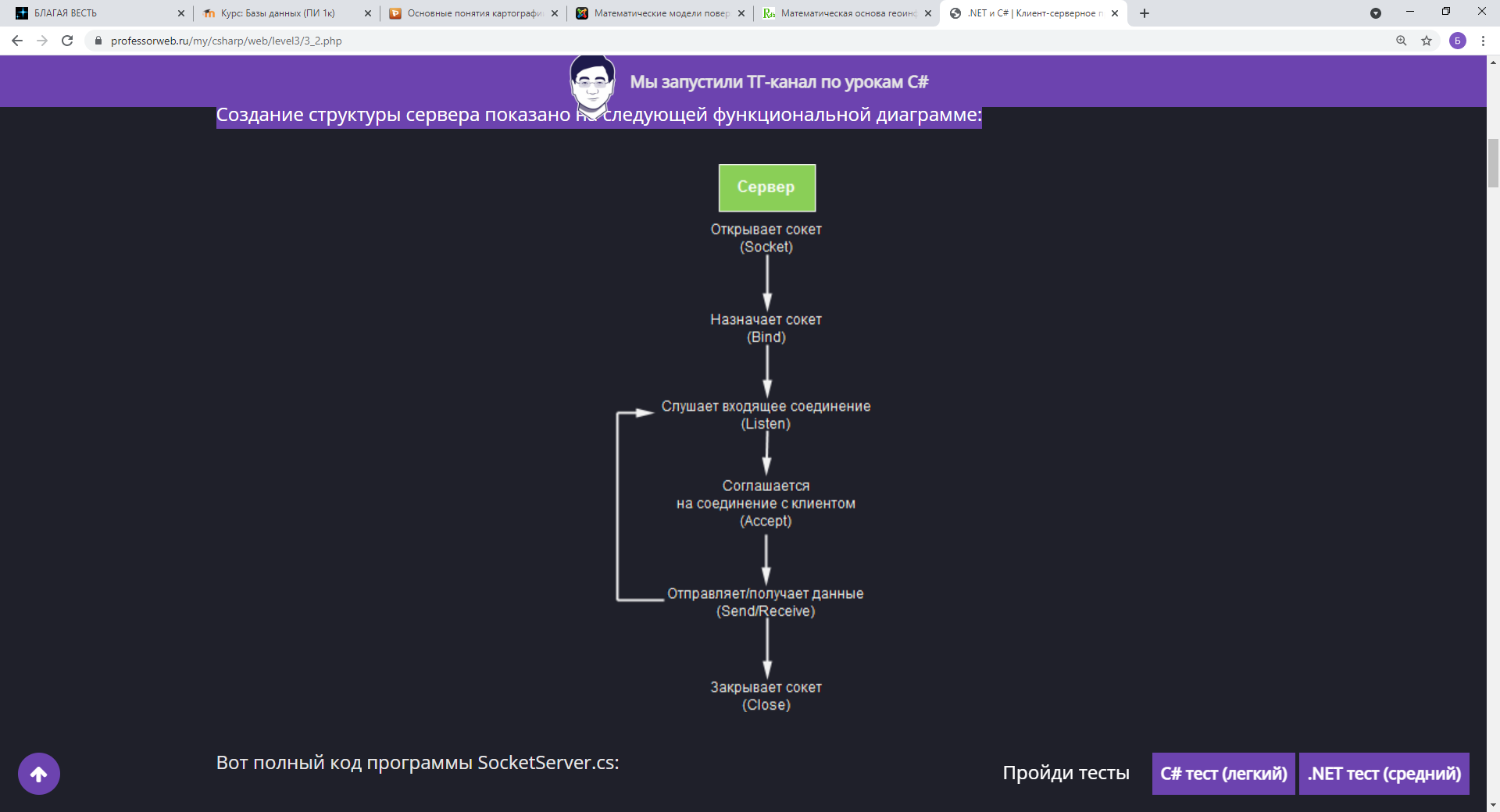
**Клиент-серверное приложение на потоковом сокете TCP**

В следующем примере используем TCP, чтобы обеспечить упорядоченные, надежные двусторонние потоки байтов. Построим завершенное приложение, включающее клиент и сервер. Сначала демонстрируем, как сконструировать на потоковых сокетах TCP сервер, а затем клиентское приложение для тестирования нашего сервера.

Следующая программа создает сервер, получающий запросы на соединение от клиентов. Сервер построен синхронно, следовательно, выполнение потока блокируется, пока сервер не даст согласия на соединение с клиентом. Это приложение демонстрирует простой сервер, отвечающий клиенту. Клиент завершает соединение, отправляя серверу сообщение <TheEnd>.

**Сервер TCP**

Создание структуры сервера показано на следующей функциональной диаграмме:



Код программы SocketServer.cs:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

namespace SocketServer

{

class Program

{

static void Main(string[] args)

{

// Устанавливаем для сокета локальную конечную точку

IPHostEntry ipHost = Dns.GetHostEntry("localhost");

IPAddress ipAddr = ipHost.AddressList[0];

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);

// Создаем сокет Tcp/Ip

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

// Назначаем сокет локальной конечной точке и слушаем входящие сокеты

try

{

sListener.Bind(ipEndPoint);

sListener.Listen(10);

// Начинаем слушать соединения

while (true)

{

Console.WriteLine("Ожидаем соединение через порт {0}", ipEndPoint);

// Программа приостанавливается, ожидая входящее соединение

Socket handler = sListener.Accept();

string data = null;

// Мы дождались клиента, пытающегося с нами соединиться

byte[] bytes = new byte[1024];

int bytesRec = handler.Receive(bytes);

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

// Показываем данные на консоли

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

// Отправляем ответ клиенту\

string reply = "Спасибо за запрос в " + data.Length.ToString()

+ " символов";

byte[] msg = Encoding.UTF8.GetBytes(reply);

handler.Send(msg);

if (data.IndexOf("<TheEnd>") > -1)

{

Console.WriteLine("Сервер завершил соединение с клиентом.");

break;

}

handler.Shutdown(SocketShutdown.Both);

handler.Close();

}

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

finally

{

Console.ReadLine();

}

}

}

}

Давайте 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 создает конечную точку для обслуживания.

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 IPEndPoint для сервера, комбинируя первый IP-адрес хост-компьютера, полученный от метода Dns.Resolve(), с номером порта:

// Устанавливаем для сокета локальную конечную точку

IPHostEntry ipHost = Dns.GetHostEntry("localhost");

IPAddress ipAddr = ipHost.AddressList[0];

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);

Здесь класс IPEndPoint представляет localhost на порте 11000. Далее новым экземпляром класса Socket создаем потоковый сокет. Установив локальную конечную точку для ожидания соединений, можно создать сокет:

// Создаем сокет Tcp/Ip

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:

**Dgram**

Поддерживает дейтаграммы. Значение Dgram требует указать Udp для типа протокола и InterNetwork в параметре семейства адресов.

**Raw**

Поддерживает доступ к базовому транспортному протоколу.

**Stream**

Поддерживает потоковые сокеты. Значение Stream требует указать Tcp для типа протокола.

Третий и последний параметр определяет тип протокола, требуемый для сокета. В параметре РrotocolType 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.

while (true)

{

Console.WriteLine("Ожидаем соединение через порт {0}", ipEndPoint);

// Программа приостанавливается, ожидая входящее соединение

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);

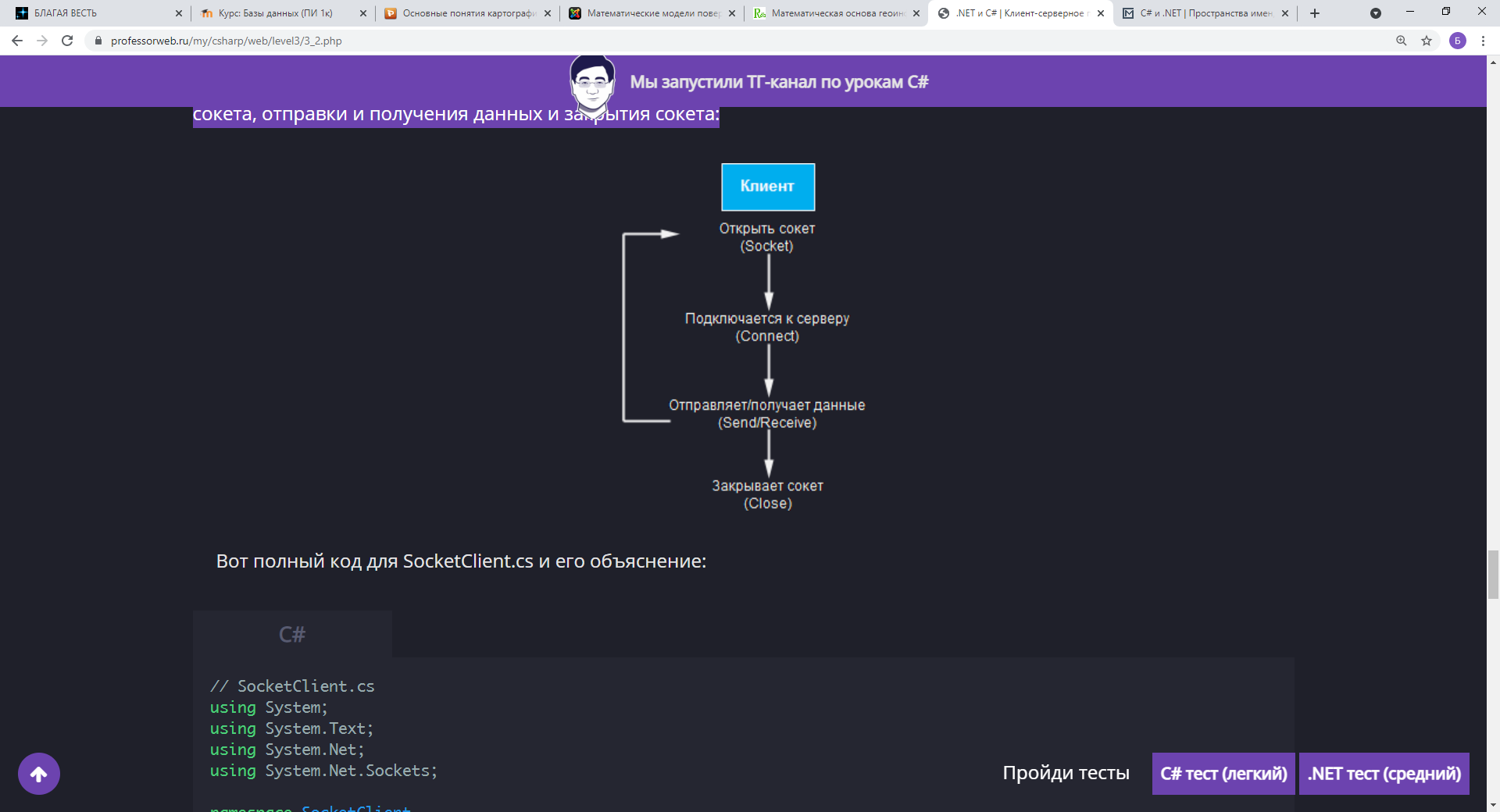
handler.Close();

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 на TCP**

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:



Код для SocketClient.cs и его объяснение:

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

namespace SocketClient

{

class Program

{

static void Main(string[] args)

{

try

{

SendMessageFromSocket(11000);

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

finally

{

Console.ReadLine();

}

}

static void SendMessageFromSocket(int port)

{

// Буфер для входящих данных

byte[] bytes = new byte[1024];

// Соединяемся с удаленным устройством

// Устанавливаем удаленную точку для сокета

IPHostEntry ipHost = Dns.GetHostEntry("localhost");

IPAddress ipAddr = ipHost.AddressList[0];

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, port);

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

// Соединяем сокет с удаленной точкой

sender.Connect(ipEndPoint);

Console.Write("Введите сообщение: ");

string message = Console.ReadLine();

Console.WriteLine("Сокет соединяется с {0} ", sender.RemoteEndPoint.ToString());

byte[] msg = Encoding.UTF8.GetBytes(message);

// Отправляем данные через сокет

int bytesSent = sender.Send(msg);

// Получаем ответ от сервера

int bytesRec = sender.Receive(bytes);

Console.WriteLine("\nОтвет от сервера: {0}\n\n", Encoding.UTF8.GetString(bytes, 0, bytesRec));

// Используем рекурсию для неоднократного вызова SendMessageFromSocket()

if (message.IndexOf("<TheEnd>") == -1)

SendMessageFromSocket(port);

// Освобождаем сокет

sender.Shutdown(SocketShutdown.Both);

sender.Close();

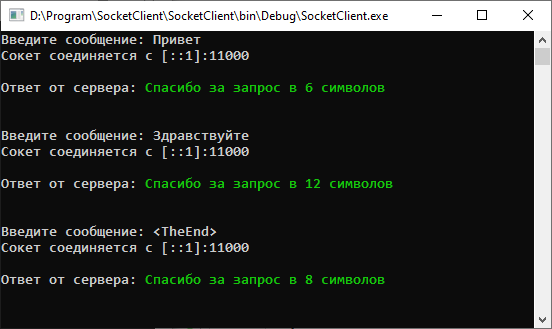
}

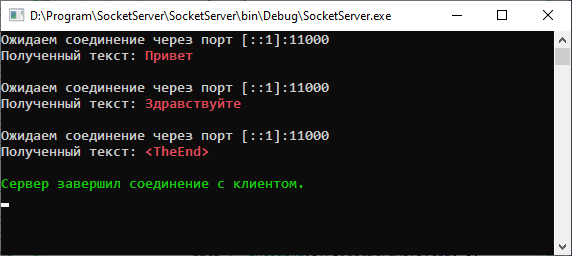
}

}

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

**Задание 2.** Реализовать клиент – серверное приложение на потоковом сокете TCP, представленного выше проекта. Реализовать в виде консольного проекта. Результат работы которой выводил результат в ниже показанном варианте:





**Задание 3:**

Реализовать программу сканирования портов, представленного ниже проекта.

**Программа сканирования портов**

В следующем примере создадим собственную программу сканирования портов, которая пытается соединиться с localhost по каждому порту, указанному в цикле. Сообщаем об успешных соединениях, а если установить соединение не удается, перехватываем порождаемое в этом случае исключение SocketException. В качестве графической среды используется WPF.

Сканер портов может использоваться для получения списка открытых портов на вашем компьютере. В открытых портах проявляется потенциальная слабость системы, которой могут воспользоваться приложения-нарушители. Вот полный код программы, включая исходную XAML-разметку:

<Window x:Class="SocketPortScaner.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:SocketPortScaner"

mc:Ignorable="d"

Title="Сканер портов" WindowState="Maximized">

<ListView Name="listview\_scaner" Margin="5">

<ListView.Resources>

<Style TargetType="{x:Type ListView}">

<Setter Property="ItemContainerStyle">

<Setter.Value>

<Style TargetType="ListViewItem">

<Setter Property="HorizontalContentAlignment" Value="Center"/>

</Style>

</Setter.Value>

</Setter>

</Style>

</ListView.Resources>

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn Header="Port ID" DisplayMemberBinding="{Binding Path=PortNumber}"

Width="150"/>

<GridViewColumn Header="Local Adress" DisplayMemberBinding="{Binding Path=Local}"

Width="250"/>

<GridViewColumn Header="Remote Adress" DisplayMemberBinding="{Binding Path=Remote}" Width="250"/>

<GridViewColumn Header="State" DisplayMemberBinding="{Binding Path=State}" Width="250"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

</Window>

C#

namespace SocketPortScaner

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

List<PortInfo> pi = MainWindow.GetOpenPort();

listview\_scaner.ItemsSource = pi;

}

private static List<PortInfo> GetOpenPort()

{

IPGlobalProperties properties = IPGlobalProperties.GetIPGlobalProperties();

IPEndPoint[] tcpEndPoints = properties.GetActiveTcpListeners();

TcpConnectionInformation[] tcpConnections = properties.GetActiveTcpConnections();

return tcpConnections.Select(p =>

{

return new PortInfo(

i: p.LocalEndPoint.Port,

local: String.Format("{0}:{1}", p.LocalEndPoint.Address, p.LocalEndPoint.Port),

remote: String.Format("{0}:{1}", p.RemoteEndPoint.Address, p.RemoteEndPoint.Port),

state: p.State.ToString());

}).ToList();

}

}

class PortInfo

{

public int PortNumber { get; set; }

public string Local { get; set; }

public string Remote { get; set; }

public string State { get; set; }

public PortInfo(int i, string local, string remote, string state)

{

PortNumber = i;

Local = local;

Remote = remote;

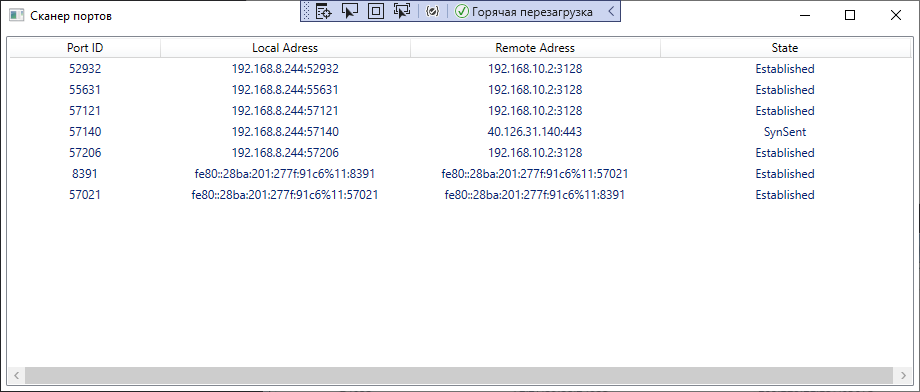
State = state;

}

}

}

Результат работы:



**Разрешение сокетов**

Среда .NET Framework, предоставляет многочисленные классы, помогающие разрабатывать в приложениях безопасный код. Многие из этих классов предлагают основанную на ролях безопасность и криптографию. Среда .NET Framework также дает объекты разрешений доступа к коду, являющиеся компоновочными блоками для защиты от несанкционированного доступа к коду. Они являются фундаментом, гарантирующим управляемому коду безопасность — может выполняться только тот код, который имеет разрешение выполняться в текущем контексте.

Каждое разрешение доступа к коду демонстрирует одно из следующих прав:

* Право доступа к защищенным ресурсам, например к файлам.
* Право выполнения защищенной операции, например обращения к управляемому коду.

Для мира Интернета и особенно для сетевых приложений классы System.Net предоставляют встроенную поддержку аутентификации и разрешений доступа к коду. Среда .NET Framework дает класс SocketPermission, обеспечивающий соблюдение разрешений доступа к коду.

Класс SocketPermission используется для управления правами на установление и принятие соединений, контролирующими доступ к сети через сокеты. Этот класс состоит из спецификации хоста и набора "действий", определяющих способы установления соединений с этим хостом. Он обеспечивает безопасность кода, контролируя значения имени хоста, IP-адреса и транспортного протокола.

Для сокетов C# имеется два способа обеспечить разрешение, поддерживающее безопасность:

* Императивно, используя класс SocketPermission
* Декларативно, используя класс SocketPermissionAttribute

Синтаксис императивной безопасности реализует разрешения, создавая новый экземпляр класса SocketPermission, чтобы при выполнении кода запросить конкретное разрешение, например право установить TCP-соединение. Этот способ обычно применяется, когда настройки безопасности изменяются при выполнении приложения. В декларативном синтаксисе используются атрибуты, позволяющие поместить информацию о безопасности в метаданные нашего кода, чтобы клиент, вызывающий код, мог воспользоваться рефлексией и узнать, какие разрешения требуются для кода.

**Императивная безопасность**

Этот синтаксис для обеспечения безопасности создает новый экземпляр класса SocketPermission. Синтаксис императивной безопасности можно использовать для выполнения требований и переопределений, но не запросов. Прежде чем вызвать соответствующий критерий безопасности, необходимо через конструктор инициализировать состояние класса SocketPermission, чтобы он представлял конкретную форму разрешения, которую вы ищете.

В следующем приложении демонстрируется основное использование класса SocketPermission. Поскольку этот код ведет себя как клиент, до выполнения этой программы нужно запустить приложение SocketServer.cs, созданное ранее:

class Program

{

static void Main(string[] args)

{

SendMessageFromSocket(11000);

}

static void SendMessageFromSocket(int port)

{

// Буфер для входящих данных

byte[] bytes = new byte[1024];

// Соединяемся с удаленным устройством

// Устанавливаем удаленную точку для сокета

IPHostEntry ipHost = Dns.GetHostEntry("localhost");

IPAddress ipAddr = ipHost.AddressList[0];

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, port);

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

// Настройка разрешений

SocketPermission permisSocket = new SocketPermission(NetworkAccess.Connect,TransportType.Tcp,"localhost",SocketPermission.AllPorts);

permisSocket.Assert();

try

{

// Соединяем сокет с удаленной точкой

sender.Connect(ipEndPoint);

Console.Write("Введите сообщение: ");

string message = Console.ReadLine();

Console.WriteLine("Сокет соединяется с {0} ", sender.RemoteEndPoint.ToString());

byte[] msg = Encoding.UTF8.GetBytes(message);

// Отправляем данные через сокет

int bytesSent = sender.Send(msg);

// Получаем ответ от сервера

int bytesRec = sender.Receive(bytes);

Console.Write("\nОтвет от сервера: ");

Console.WriteLine("{0}", Encoding.UTF8.GetString(bytes, 0, bytesRec) + "\n\n");

// Используем рекурсию для неоднократного вызова SendMessageFromSocket()

if (message.IndexOf("<TheEnd>") == -1)

SendMessageFromSocket(port);

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

finally

{

if (sender.Connected)

{

// Освобождаем сокет

sender.Shutdown(SocketShutdown.Both);

sender.Close();

}

}

Console.ReadLine();

}

}

Приведенный код показывает, как, используя императивный синтаксис, реализовать безопасность доступа к коду. Важность этого кода в данном случае заключается в вызове одного метода класса **SocketPermission - Assert()**, который указывает, что приложению разрешается давать согласие на запросы соединения от Интернета и местного ресурса.

**Декларативная безопасность**

Декларативная безопасность использует атрибуты .NET, чтобы поместить информацию о безопасности внутрь метаданных кода. Атрибуты можно поместить на уровне сборки, класса или члена и указать необходимый тип запроса, требования или переопределения. Для использования этого синтаксиса безопасности сначала через декларативный синтаксис нужно инициализировать данные состояния объекта SocketPermissionAttribute, чтобы он представлял форму разрешения, соблюдение которого обеспечивается в коде.

В следующем примере демонстрируется, как обеспечить выполнение разрешения с использованием SocketPermissionAttribute:

class Program

{

static void Main(string[] args)

{

LegalMethod();

Console.ReadLine();

}

[SocketPermission(SecurityAction.Assert, Access = "Connect",

Host = "localhost", Port = "All", Transport = "Tcp")]

public static void LegalMethod()

{

Console.WriteLine("Legal Method");

// Устанавливаем удаленную точку для сокета

IPHostEntry ipHost = Dns.GetHostEntry("localhost");

IPAddress ipAddr = ipHost.AddressList[0];

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);

Socket sender = new Socket(ipAddr.AddressFamily,

SocketType.Stream, ProtocolType.Tcp);

try

{

// Соединяем сокет с удаленной endPoint

sender.Connect(ipEndPoint);

Console.WriteLine("Сокет подключен к {0}",

sender.RemoteEndPoint.ToString());

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

finally

{

if (sender.Connected)

{

// Освобождаем сокет

sender.Shutdown(SocketShutdown.Both);

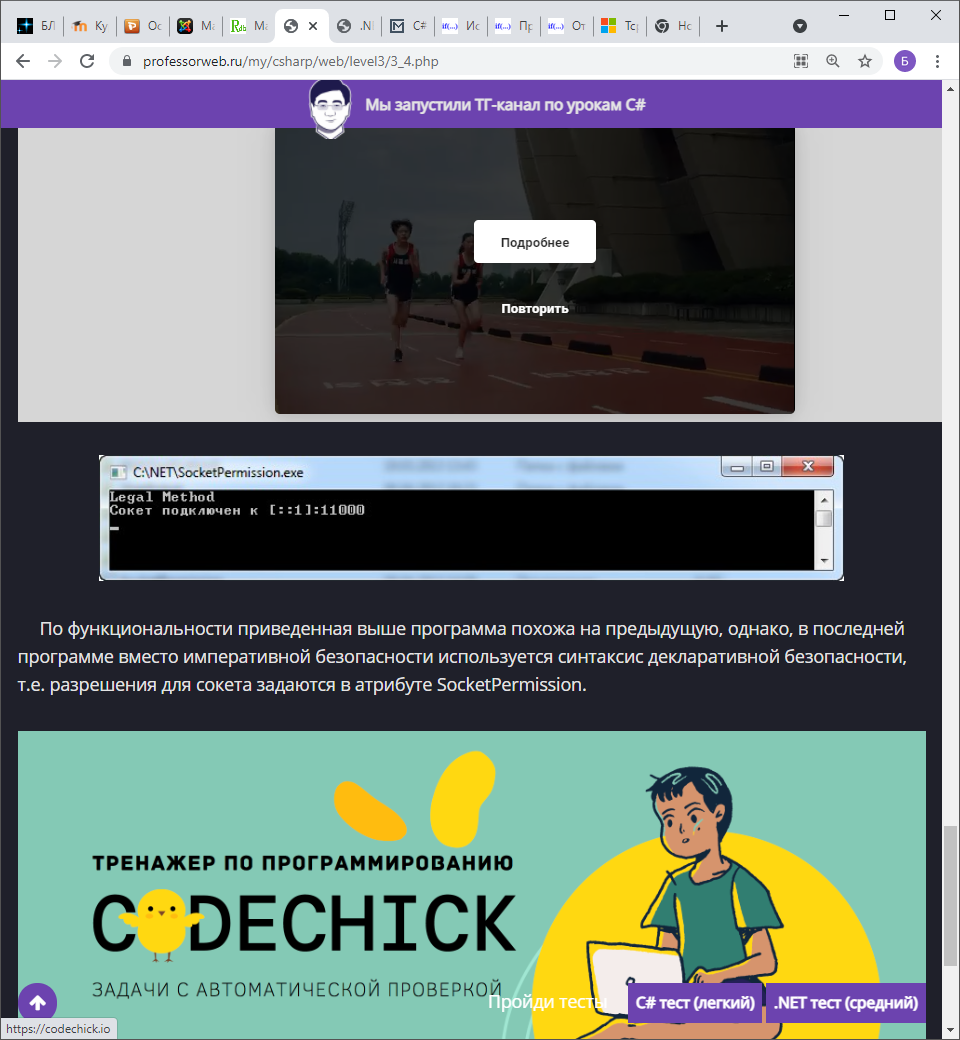
sender.Close();

}

}

}

}



По функциональности приведенная выше программа похожа на предыдущую, однако, в последней программе вместо императивной безопасности используется синтаксис декларативной безопасности, т.е. разрешения для сокета задаются в атрибуте SocketPermission.

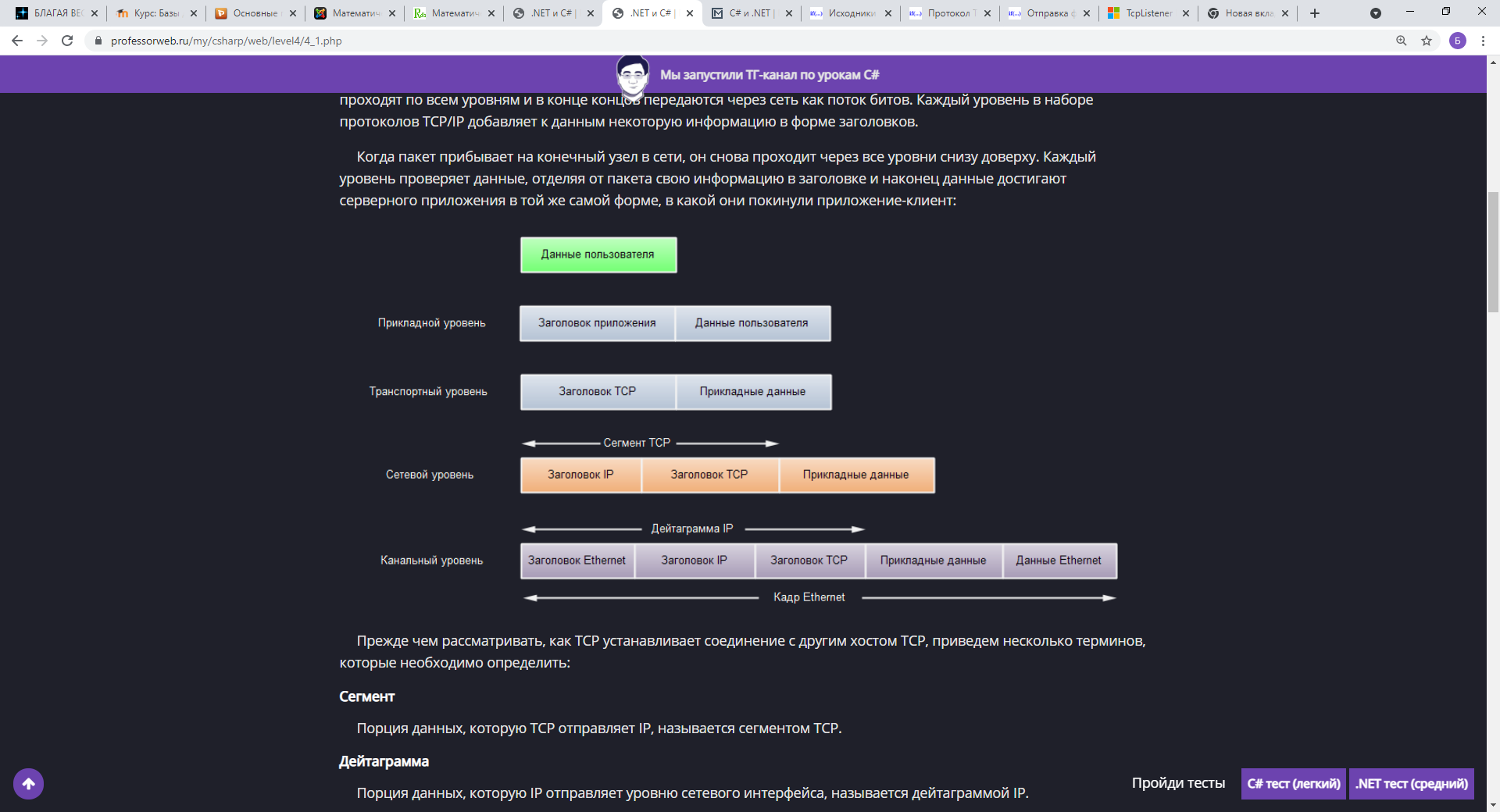
**Протокол TCP**

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:



Прежде чем рассматривать, как TCP устанавливает соединение с другим хостом TCP, приведем несколько терминов, которые необходимо определить:

**Сегмент**

Порция данных, которую TCP отправляет IP, называется сегментом TCP.

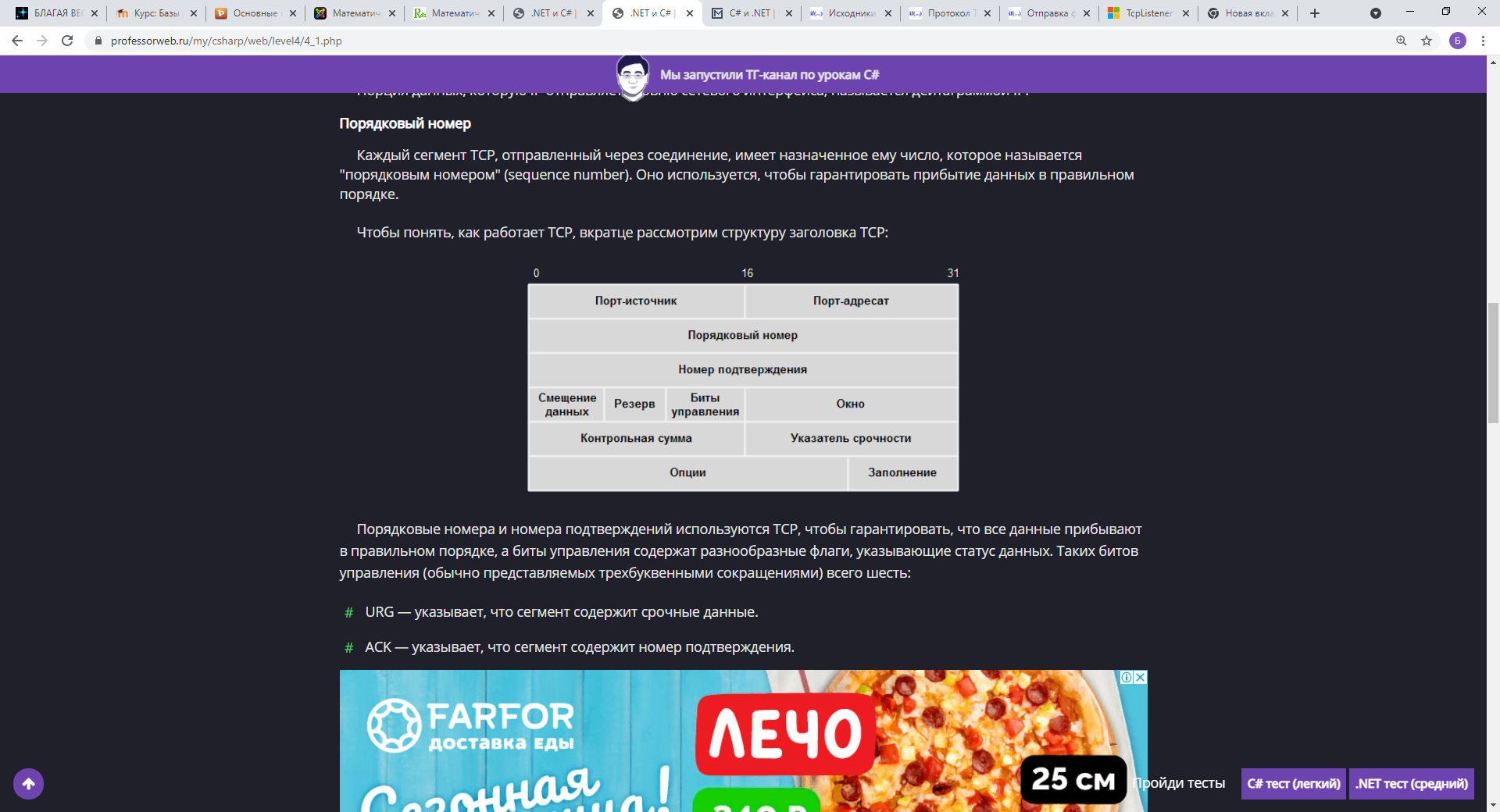
**Дейтаграмма**

Порция данных, которую IP отправляет уровню сетевого интерфейса, называется дейтаграммой IP.

**Порядковый номер**

Каждый сегмент TCP, отправленный через соединение, имеет назначенное ему число, которое называется "порядковым номером" (sequence number). Оно используется, чтобы гарантировать прибытие данных в правильном порядке.

Чтобы понять, как работает TCP, вкратце рассмотрим структуру заголовка TCP:



Порядковые номера и номера подтверждений используются TCP, чтобы гарантировать, что все данные прибывают в правильном порядке, а биты управления содержат разнообразные флаги, указывающие статус данных. Таких битов управления (обычно представляемых трехбуквенными сокращениями) всего шесть:

* URG — указывает, что сегмент содержит срочные данные.
* ACK — указывает, что сегмент содержит номер подтверждения.
* PSH — указывает, что данные нужно протолкнуть к получающему пользователю.
* RST — сбрасывает соединение.
* SYN — используется для синхронизации порядковых номеров.
* FIN — указывает конец данных.

Для установления соединения TCP использует процесс, называемый *"трехфазным квитированием" (Three-Phase Handshake)*. Как следует из названия, этот процесс включает три шага:

1. Клиент инициирует взаимодействие с сервером, посылая сегмент с установленным битом SYN. Этот сегмент содержит начальный порядковый номер клиента.
2. Сервер отвечает отправкой сегмента с установленными битами SYN и ACK. Этот сегмент содержит начальный порядковый номер сервера (не связанный с порядковым номером клиента) и номер подтверждения, на единицу больший порядкового номера клиента (т.е. равный следующему порядковому номеру, ожидаемому от клиента).
3. Клиент должен подтвердить этот сегмент отправкой обратно сегмента с установленным битом ACK. Номер подтверждения будет на единицу больше порядкового номера сервера, а порядковый номер будет равен номеру подтверждения сервера (т. е. на единицу больше начального порядкового номера клиента).

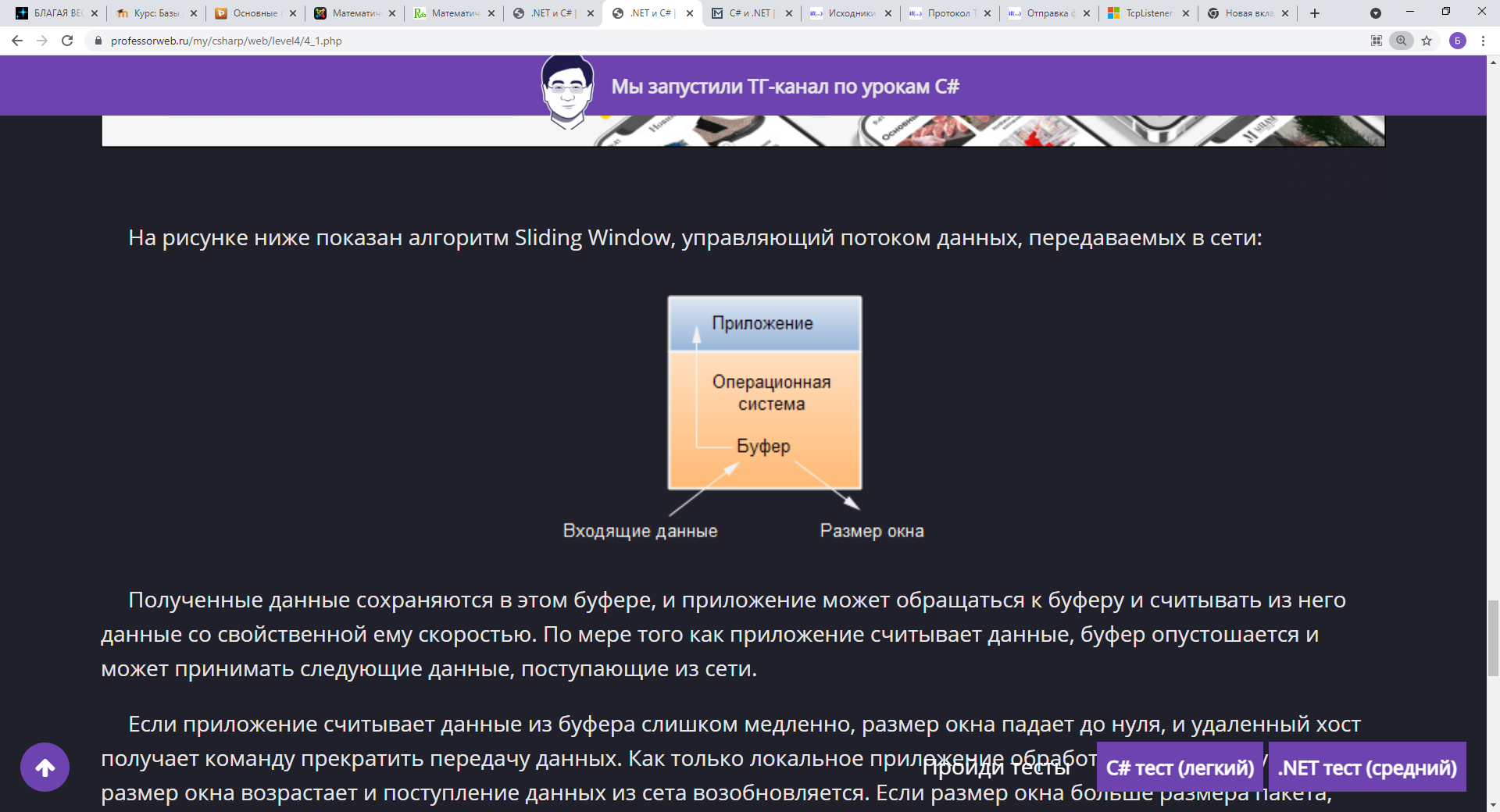
Теперь, узнав в общих чертах, как TCP устанавливает соединения, рассмотрим немного подробнее несколько операций TCP, чтобы понять, как TCP передает данные.

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").

На рисунке ниже показан алгоритм Sliding Window, управляющий потоком данных, передаваемых в сети:



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 сета 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-cepвep, слушающий порт 80 и отвечающий на запросы от нескольких компьютеров.

**Класс TcpClient**

Поддержка сокетов TCP на платформе .NET значительно усовершенствована по сравнению с предыдущей моделью программирования. Раньше большинство разработчиков, использовавших Visual С++, 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:

Таблица. Свойства и методы класса TcpClient.

|  |  |  |
| --- | --- | --- |
| Свойство или метод | Тип | Описание |
| LingerState | LingerOption | Устанавливает или возвращает объект LingerOption, содержащий информацию о том, будет ли соединение оставаться открытым после закрытия сокета и как долго. |
| NoDelay | bool | Указывает, будет ли сокет задерживать отправку и получение данных, если буфер, назначенный для отправки или получения данных, не заполнен. Если свойство имеет значение false, TCP задержит отправку пакета, пока не будет накоплен достаточный объем данных. Это средство помогает избежать неэффективной отправки через сеть слишком маленьких пакетов. |
| ReceiveBufferSize | int | Задает размер буфера для входящих данных (в байтах). Это свойство используется при считывании данных из сокета. |
| ReceiveTimeout | int | Задает время в миллисекундах, которое TCpClient будет ждать получения данных после инициирования этой операции. Если это время истечет, а данные не будут получены, возникнет исключение SocketException. |
| SendBufferSize | int | Задает размер буфера для исходящих данных. |
| SendTimeout | int | Задает время в миллисекундах, которое TcpClient будет ждать подтверждения числа байтов, отправленных удаленному хосту от базового сокета. При истечении времени SendTimeout порождается исключение SocketException. |
| Close() |  | Закрывает TCP-соединение. |
| Connect() |  | Соединяется с удаленным хостом TCP. |
| GetStream() |  | Возвращает объект NetworkStream, используемый для передачи данных между клиентом и удаленным хостом. |
| Active | bool | Указывает, есть ли активное соединение с удаленным хостом. |
| Client | Socket | Задает базовый объект Socket, используемый объектом TcpClient. Поскольку это защищенное свойство, к базовому сокету можно обращаться, если вы производите ваш класс от TcpClient. |

public TcpClient();

public TcpClient(IPEndPoint ipEnd);

public TcpClient(string hostname, int port);

Конструктор, используемый по умолчанию, инициализирует экземпляр TcpClient. Если экземпляр TcpClient создается так, то 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 ipAddr = IPAddress.Parse("140.16.82.66");

IPEndPoint endPoint = new IPEndPoint(ipAddr, 11000);

TcpClient newClient = new TcpClient(endPoint);

// Для создания соединения с сервером надо вызвать connect()

newClient.Connect(ipAddr, 11000);

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 newClient = 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():

public void Connect(IPEndPoint endPoint);

public void Connect(IPAddress ipAddr, int port);

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 newClient = new TcpClient();

// Устанавливаем соединение с IPEndPoint

IPAddress ipAddr = IPAddress.Parse("127.0.0.1");

IPEndPoint endPoint = new IPEndPoint(ipAddr, 11000);

// Соединяемся с хостом

newClient.Connect(endPoint);

1. Передача объекта IPAddress и номера порта:

// Создаем новый экземпляр TcpClient

TcpClient newClient = new TcpClient();

// Устанавливаем соединение с IPEndPoint

IPAddress ipAddr = IPAddress.Parse("127.0.0.1");

// Соединяемся с хостом

newClient.Connect(ipAddr, 11000);

1. Передача имени хоста и номера порта:

// Создаем новый экземпляр TcpClient

TcpClient newClient = new TcpClient();

// Соединяемся с хостом

newClient.Connect("127.0.0.1", 11000);

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

try

{

TcpClient newClient = new TcpClient();

// Соединяемся с сервером

newClient.Connect("140.16.82.66", 11000); // 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():

NetworkStream tcpStream = newClient.GetStream();

Предположим, что newClient — это экземпляр TcpClient, а соединение с хостом уже установлено. Иначе будет порождено исключение InvalidOperation. 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("Простой 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[newClient.ReceiveBufferSize];

int bytesRead = tcpStream.Read(bytes, 0, newClient.ReceiveBufferSize);

// Строка, содержащая ответ от сервера

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.

**Закрытие сокета TCP**

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

// Закрываем клиентский сокет

newClient.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.

**FTP клиент на C#**

**Обзор FTP**

File Transfer Protocol (FTP) — это протокол прикладного уровня, построенный поверх протокола транспортного уровня, обычно поверх TCP. Он используется для копирования файлов с удаленного сервера и на удаленный сервер.

Принцип работы FTP заключается в следующем: сначала открывается соединение TCP с сервером, отправляются текстовые команды для выполнения таких действий, как извлечение файла с сервера, и сервер возвращает трехразрядный код (вместе с сообщением, удобным для восприятия человеком), чтобы показать состояние запрошенного действия.

В FTP используются два разных соединения — управляющее соединение, на котором посылаются команды и получаются ответы сервера, и соединение для данных, используемое для самой передачи файлов с сервера или на сервер. По умолчанию сервер слушает команды от клиента на порту 21, а когда нужно отправлять данные, открывает второе соединение с портом 20 клиента.

**Активный и пассивный режимы**

Соединение для данных открывается, если только отправлена команда копирования файла. В активном режиме (который установлен по умолчанию) клиент должен слушать соединения. Когда потребуется отправить данные, FTP-сервер откроет соединение с этим сокетом и передаст данные к клиенту.

При таком подходе проблема заключается в том, что большинство конфигураций брандмауэров не позволит извне установить соединения с машинами за брандмауэром, а разрешит только те соединения, которые были инициализированы из-за брандмауэра. В FTP эта проблема решается вводом пассивного режима. В этом случае клиент отправляет команду, указывающую, что используется пассивный режим, и на нее сервер отвечает номером порта, на котором он слушает. Когда нужно отправить данные, вместо того чтобы слушать запрос от сервера, клиент может открыть соединение с указанным портом. В нашей реализации будет использоваться пассивный режим.

**FTP-команды**

Спецификация FTP определяет несколько команд для аутентификации, копирования файлов на сервер и с сервера и изменения каталога на сервере. В нашем коде не будем пользоваться всеми командами, поэтому рассмотрим только те, которые нам понадобятся:

Таблица. Команды FTP

|  |  |
| --- | --- |
| Команда | Описание |
| USER <имя пользователя> | Имя пользователя, которое нужно удостоверить на сервере |
| PASS <пароль> | Пароль, связанный с именем пользователя |
| RETR <имя файла> | Скопировать с сервера указанный файл |
| ST0R <имя файла> | Скопировать файл на сервер и сохранить его в указанном месте |
| TYPE <индикатор типа> | Формат данных. Может принимать одно значение из следующего перечня: A — ASCII, Е — EBCDIC, I — изображение (двоичные данные), L <размер байта> — локальный размер байта |
| PASV | Использовать пассивный режим |
| STAT | Принуждает сервер отправить сообщение состояния клиенту. Команда может использоваться в ходе передачи данных, чтобы указать состояние операции |
| QUIT | Закрывает соединение с сервером |

**Коды состояния FTP**

Трехразрядные коды FTP упорядочены в соответствии со степенью детализации, предоставленной разрядом кода: в первом разряде дается общее указание о состоянии команды; второй разряд указывает общий тип возникшей ошибки; в третьем разряде содержится более конкретная информация. Первый разряд может принимать следующие значения:

* 1 - Положительный предварительный ответ — запрошенное действие инициировано, и, прежде чем клиент сможет послать новую команду, будет отправлен еще один ответ.
* 2 - Положительный окончательный ответ — запрошенное действие завершено.
* 3 - Положительный промежуточный ответ — команда принята, но сервер, прежде чем приступить к выполнению, должен получить дополнительную информацию.
* 4 - Временный отрицательный ответ — команда отвергнута, но ошибка имеет временный характер, и команду можно повторить.
* 5 - Постоянный отрицательный ответ — команда отвергнута.

Вот некоторые конкретные ответы, которые будут обрабатываться:

|  |  |
| --- | --- |
| Код | Описание |
| 125 | Открыто соединение для данных — начинается передача. |
| 150 | Готовность к открытию соединения для данных. |
| 200 | Команда принята |
| 220 | Обслуживание готово для нового пользователя. |
| 227 | Вход в пассивный режим. |
| 230 | Пользователь вошел в систему. |
| 331 | Имя пользователя принято — отправляйте пароль. |

**Кодирование FTP-клиента**

Ниже представлен код FTP-клиента, реализованного на WPF с использованием собственной реализации класса FtpWebRequest в виде класса Client:

using System;

using System.Collections.Generic;

using System.IO;

using System.Net;

namespace FtpClient

{

public class Client {

private string password;

private string userName;

private string uri;

private int bufferSize = 1024;

public bool Passive = true;

public bool Binary = true;

public bool EnableSsl = false;

public bool Hash = false;

public Client(string uri, string userName, string password) {

this.uri = uri;

this.userName = userName;

this.password = password;

}

public string ChangeWorkingDirectory(string path) {

uri = combine(uri, path);

return PrintWorkingDirectory();

}

public string DeleteFile(string fileName) {

var request = createRequest(combine(uri, fileName), WebRequestMethods.Ftp.DeleteFile);

return getStatusDescription(request);

}

public string DownloadFile(string source, string dest) {

var request = createRequest(combine(uri, source), WebRequestMethods.Ftp.DownloadFile);

byte[] buffer = new byte[bufferSize];

using (var response = (FtpWebResponse)request.GetResponse()) {

using (var stream = response.GetResponseStream()) {

using (var fs = new FileStream(dest, FileMode.OpenOrCreate)) {

int readCount = stream.Read(buffer, 0, bufferSize);

while (readCount > 0) {

if (Hash)

Console.Write("#");

fs.Write(buffer, 0, readCount);

readCount = stream.Read(buffer, 0, bufferSize);

}

}

}

return response.StatusDescription;

}

}

public DateTime GetDateTimestamp(string fileName) {

var request = createRequest(combine(uri, fileName), WebRequestMethods.Ftp.GetDateTimestamp);

using (var response = (FtpWebResponse)request.GetResponse()) {

return response.LastModified;

}

}

public long GetFileSize(string fileName) {

var request = createRequest(combine(uri, fileName), WebRequestMethods.Ftp.GetFileSize);

using (var response = (FtpWebResponse)request.GetResponse()) {

return response.ContentLength;

}

}

public string[] ListDirectory() {

var list = new List<string>();

var request = createRequest(WebRequestMethods.Ftp.ListDirectory);

using (var response = (FtpWebResponse)request.GetResponse()) {

using (var stream = response.GetResponseStream()) {

using (var reader = new StreamReader(stream, true)) {

while (!reader.EndOfStream) {

list.Add(reader.ReadLine());

}

}

}

}

return list.ToArray();

}

public string[] ListDirectoryDetails() {

var list = new List<string>();

var request = createRequest(WebRequestMethods.Ftp.ListDirectoryDetails);

using (var response = (FtpWebResponse)request.GetResponse()) {

using (var stream = response.GetResponseStream()) {

using (var reader = new StreamReader(stream, true)) {

while (!reader.EndOfStream) {

list.Add(reader.ReadLine());

}

}

}

}

return list.ToArray();

}

public string MakeDirectory(string directoryName) {

var request = createRequest(combine(uri, directoryName), WebRequestMethods.Ftp.MakeDirectory);

return getStatusDescription(request);

}

public string PrintWorkingDirectory() {

var request = createRequest(WebRequestMethods.Ftp.PrintWorkingDirectory);

return getStatusDescription(request);

}

public string RemoveDirectory(string directoryName) {

var request = createRequest(combine(uri, directoryName), WebRequestMethods.Ftp.RemoveDirectory);

return getStatusDescription(request);

}

public string Rename(string currentName, string newName) {

var request = createRequest(combine(uri, currentName), WebRequestMethods.Ftp.Rename);

request.RenameTo = newName;

return getStatusDescription(request);

}

public string UploadFile(string source, string destination) {

var request = createRequest(combine(uri, destination), WebRequestMethods.Ftp.UploadFile);

using (var stream = request.GetRequestStream()) {

using (var fileStream = System.IO.File.Open(source, FileMode.Open)) {

int num;

byte[] buffer = new byte[bufferSize];

while ((num = fileStream.Read(buffer, 0, buffer.Length)) > 0) {

if (Hash)

Console.Write("#");

stream.Write(buffer, 0, num);

}

}

}

return getStatusDescription(request);

}

public string UploadFileWithUniqueName(string source) {

var request = createRequest(WebRequestMethods.Ftp.UploadFileWithUniqueName);

using (var stream = request.GetRequestStream()) {

using (var fileStream = System.IO.File.Open(source, FileMode.Open)) {

int num;

byte[] buffer = new byte[bufferSize];

while ((num = fileStream.Read(buffer, 0, buffer.Length)) > 0) {

if (Hash)

Console.Write("#");

stream.Write(buffer, 0, num);

}

}

}

using (var response = (FtpWebResponse)request.GetResponse()) {

return Path.GetFileName(response.ResponseUri.ToString());

}

}

private FtpWebRequest createRequest(string method) {

return createRequest(uri, method);

}

private FtpWebRequest createRequest(string uri, string method) {

var r = (FtpWebRequest)WebRequest.Create(uri);

r.Credentials = new NetworkCredential(userName, password);

r.Method = method;

r.UseBinary = Binary;

r.EnableSsl = EnableSsl;

r.UsePassive = Passive;

return r;

}

private string getStatusDescription(FtpWebRequest request) {

using (var response = (FtpWebResponse)request.GetResponse()) {

return response.StatusDescription;

}

}

private string combine(string path1, string path2) {

return Path.Combine(path1, path2).Replace("\\", "/");

}

}

}

<Window x:Class="FtpClient.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="FTP клиент" Style="{DynamicResource MainWindowStyle}">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<Border BorderThickness="0,0,1,0" BorderBrush="White" Margin="5">

<Grid Width="300">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="0.4\*"/>

<ColumnDefinition Width="0.6\*"/>

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<TextBlock Text="Адрес сервера" />

<TextBox x:Name="txt\_adres" Grid.Column="1" Text="ftp://77.222.61.135/"/>

<TextBlock Text="Логин" Grid.Row="1"/>

<TextBox x:Name="txt\_login" Grid.Row="1" Grid.Column="1" Text=""/>

<TextBlock Text="Пароль" Grid.Row="2"/>

<PasswordBox x:Name="txt\_password" Grid.Row="2" Grid.Column="1" Password="12345"/>

<Button x:Name="btn\_connect" Content="Соединиться по FTP" Padding="10" Margin="0,10"

Grid.Row="3" Grid.ColumnSpan="2" Width="180" HorizontalAlignment="Left" Click="btn\_connect\_Click\_1" />

</Grid>

</Border>

<ListView Grid.Column="1" Margin="5" x:Name="lbx\_files" ItemsSource="{Binding}">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn Header="Файл/папка" Width="500">

<GridViewColumn.CellTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal" Height="36" MouseLeftButtonDown="folder\_Click">

<Image Width="32" Height="32" Source="{Binding Type}" />

<TextBlock Foreground="#DDD" Text="{Binding Name}" Margin="12,0" />

</StackPanel>

</DataTemplate>

</GridViewColumn.CellTemplate>

</GridViewColumn>

<GridViewColumn Header="Размер" Width="200" DisplayMemberBinding="{Binding FileSize}"/>

<GridViewColumn Header="Дата создания" Width="200" DisplayMemberBinding="{Binding Date}"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

using System.Net;

using System.IO;

using System.Text.RegularExpressions;

namespace FtpClient

{

public partial class MainWindow : Window

{

string prevAdress = "ftp://";

public MainWindow()

{

InitializeComponent();

}

private void btn\_connect\_Click\_1(object sender, RoutedEventArgs e)

{

try

{

// Создаем объект подключения по FTP

Client client = new Client(txt\_adres.Text, txt\_login.Text, txt\_password.Password);

// Регулярное выражение, которое ищет информацию о папках и файлах

// в строке ответа от сервера

Regex regex = new Regex(@"^([d-])([rwxt-]{3}){3}\s+\d{1,}\s+.\*?(\d{1,})\s+(\w+\s+\d{1,2}\s+(?:\d{4})?)(\d{1,2}:\d{2})?\s+(.+?)\s?$",

RegexOptions.Compiled | RegexOptions.Multiline | RegexOptions.IgnoreCase | RegexOptions.IgnorePatternWhitespace);

// Получаем список корневых файлов и папок

// Используется LINQ to Objects и регулярные выражения

List<FileDirectoryInfo> list = client.ListDirectoryDetails()

.Select(s =>

{

Match match = regex.Match(s);

if (match.Length > 5)

{

// Устанавливаем тип, чтобы отличить файл от папки (используется также для установки рисунка)

string type = match.Groups[1].Value == "d" ? "DIR.png" : "FILE.png";

// Размер задаем только для файлов, т.к. для папок возвращается

// размер ярлыка 4кб, а не самой папки

string size = "";

if (type == "FILE.png")

size = (Int32.Parse(match.Groups[3].Value.Trim()) / 1024).ToString() + " кБ";

return new FileDirectoryInfo(size, type, match.Groups[6].Value, match.Groups[4].Value, txt\_adres.Text);

}

else return new FileDirectoryInfo();

}).ToList();

// Добавить поле, которое будет возвращать пользователя на директорию выше

list.Add(new FileDirectoryInfo("","DEFAULT.png","...","",txt\_adres.Text));

list.Reverse();

// Отобразить список в ListView

lbx\_files.DataContext = list;

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString() + ": \n" + ex.Message);

}

}

private void folder\_Click(object sender, MouseButtonEventArgs e)

{

if (e.ClickCount >= 2)

{

FileDirectoryInfo fdi = (FileDirectoryInfo)(sender as StackPanel).DataContext;

if (fdi.Type == "DIR.png")

{

prevAdress = fdi.adress;

txt\_adres.Text = fdi.adress + fdi.Name + "/";

btn\_connect\_Click\_1(null, null);

}

}

}

}

}

Вспомогательный класс

public class FileDirectoryInfo

{

string fileSize;

string type;

string name;

string date;

public string adress;

public string FileSize

{

get { return fileSize; }

set { fileSize = value; }

}

public string Type

{

get { return type; }

set { type = value; }

}

public string Name

{

get { return name; }

set { name = value; }

}

public string Date

{

get { return date; }

set { date = value; }

}

public FileDirectoryInfo() { }

public FileDirectoryInfo(string fileSize, string type, string name, string date, string adress)

{

FileSize = fileSize;

Type = type;

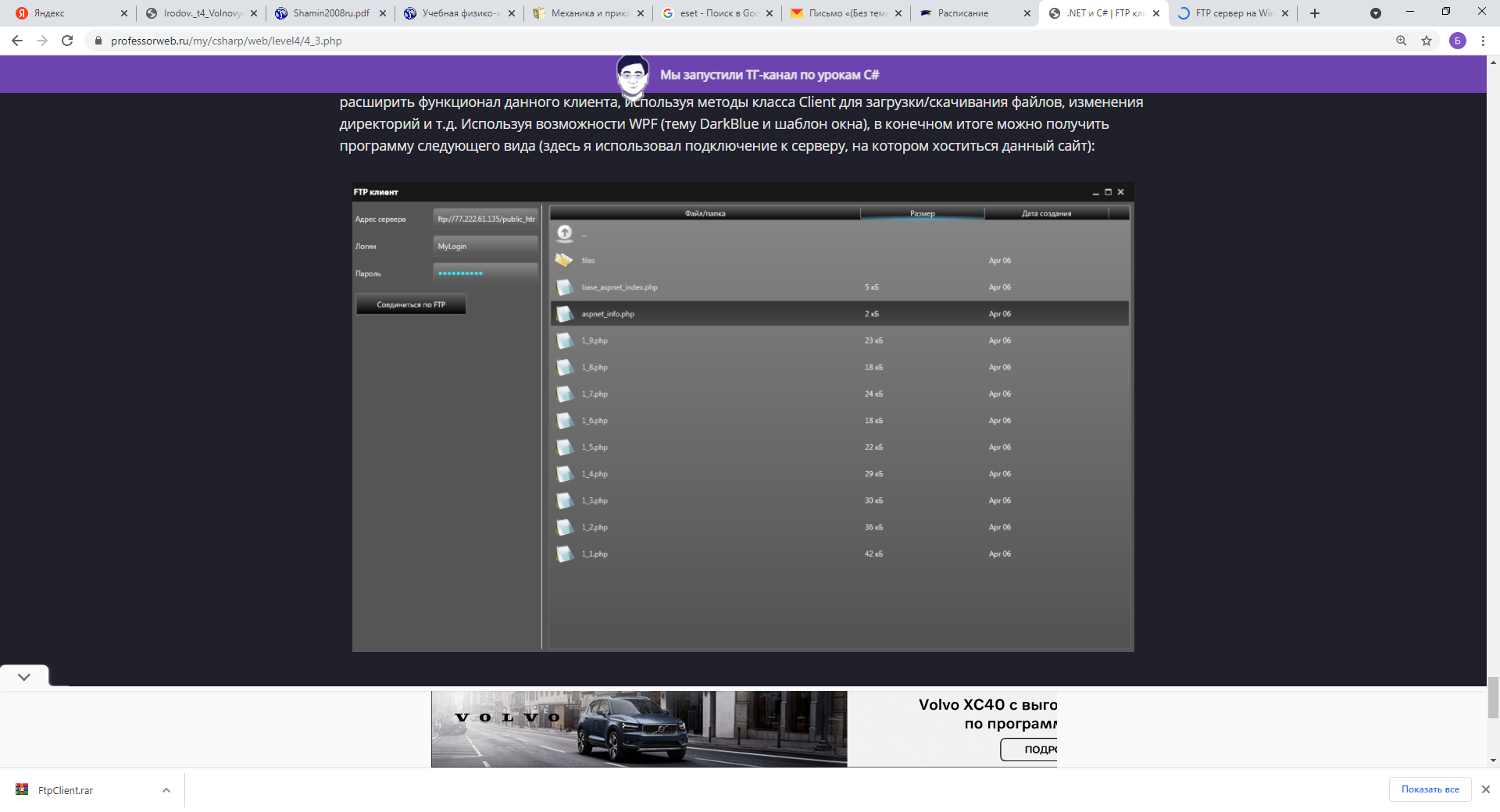
Name = name;

Date = date;

this.adress = adress;

}

}



**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:

Таблица. Свойства и методы класса TcpListener

|  |  |  |
| --- | --- | --- |
| Свойство или метод | Тип | Описание |
| LocalEndpoint | IPEndpoint | Это свойство возвращает объект IPEndpoint, который содержит информацию о локальном сетевом интерфейсе и номере порта, используемую для ожидания входящих запросов от клиентов. |
| AcceptSocket() |  | Дает согласие на ожидающий запрос соединения и возвращает объект Socket, используемый для взаимодействия с клиентом. |
| AcceptTcpClient() |  | Дает согласие на ожидающий запрос соединения и возвращает объект TcpClient, используемый для взаимодействия с клиентом. |
| Pending() |  | Указывает, есть ли ожидающие запросы соединения. |
| Start() |  | Принуждает TcpListener начать слушать запросы соединения. |
| Stop() |  | Закрывает слушающий объект. |
| Active | bool | Указывает, слушает ли в настоящий момент TcpListener запросы соединения. |
| Server | Socket | Возвращает базовый объект Socket, используемый объектом TcpListener, чтобы слушать запросы соединения. |

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. Второй конструктор принимает объект IPEndPoint, определяющий IP-адрес. Последний перегруженный конструктор принимает объект IPAddress и номер порта.

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 listener = 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 (listener.Pending())

{

Console.WriteLine("Соединение добавлено в очередь");

}

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.

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.

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

listener.Stop();

**Задание 4**

Проанализировать приложения реализованных в архивах:

* tcpclientservertextmessage\_vs11

Протокол TCP, отправка текстовых сообщений по сети.

Здесь реализованы TCP - клиент и TCP – сервер. Серверное и клиентское приложения отправки и извлечении сообщений по локальной сети. Сетевая работа построена на протоколе TCP. Классы TcpListener, TcpClient.

* tcpsendfiles2

Отправка файлов по сети имеет особенности по сравнению с отправкой текстовых сообщений. Размер файла в большинстве случаев значительно больше текста и вследствие этого передачу необходимо осуществлять частями. Передача происходит по протоколу TCP.

Программы позаимствованы из второго литературного источника.

**Задание 5**

Задачи:

Рассмотрим следующие задачи:

* Протокол TCP отправка сообщений по сети
* Протокол TCP отправка файлов по сети

**Протокол TCP отправка сообщений по сети**

Решение задачи: создание приложения по отправке текстовых сообщений по сети с помощью протокола TCP.

По методологии RUP разделим этап создания этого приложения. На несколько этапов в конце каждого достигаем нужного результата.

* **Первый этап**. Создание TCP – сервера и TСP – клиента в консоли, для обмена текстовых сообщений.
* **Второй этап** добавление клиенту и серверу асинхронной работы с сообщениями.
* **Третий этап** реализация чата обмена текстовыми сообщениями по сети с помощью протокола TCP. Реализовать TCP – сервера и TСP – клиента, за основу взять технологию WPF.
* **Четвертый этап** добавитьк чату возможность передачи файлов по сети с помощью протокола TCP.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. **Фленов М. Е.**Библия C#. — 4-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2019. — 512 с.: ил.
2. URL: https://www.interestprograms.ru/sources/csharp/os-windows/desktop/setevye-prilozheniya (дата обращения 10.04.2021)
3. URL: <https://professorweb.ru/my/csharp/web/level2/2_4.php> (дата обращения 10.04.2021)
4. URL: <https://professorweb.ru/my/csharp/web/level2/2_5.php> (дата обращения 10.04.2021)
5. URL: <https://professorweb.ru/my/csharp/web/level2/2_6.php> (дата обращения 10.04.2021)
6. URL: <https://professorweb.ru/my/csharp/web/level2/2_7.php> (дата обращения 10.04.2021)
7. URL: <https://professorweb.ru/my/csharp/web/level3/3_1.php> (дата обращения 10.04.2021)
8. URL: <https://professorweb.ru/my/csharp/web/level3/3_2.php> (дата обращения 10.04.2021)
9. URL: <https://professorweb.ru/my/csharp/web/level3/3_3.php> (дата обращения 10.04.2021)
10. URL: <https://professorweb.ru/my/csharp/web/level3/3_4.php> (дата обращения 10.04.2021)
11. URL: <https://professorweb.ru/my/csharp/web/level4/4_1.php> (дата обращения 10.04.2021)
12. URL: <https://professorweb.ru/my/csharp/web/level4/4_2.php> (дата обращения 10.04.2021)
13. <https://professorweb.ru/my/csharp/web/level4/4_3.php>