**Лабораторная работа 3.**

**Разработка программных модулей**

**Элементы сетевого программирования**

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

**Теория**

**Протокол UDP**

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

Internet Protocol (IP) является основным протоколом Интернета. Transmission Control Protocol (TCP) и UDP — это протоколы транспортного уровня, построенные поверх лежащего в основе протокола.

TCP/IP — это набор протоколов, называемый также "пакетом протоколов Интернета" (Internet Protocol Suite), состоящий из четырех уровней. Запомните, что TCP/IP не просто один протокол, а семейство или набор протоколов, который состоит из других низкоуровневых протоколов, таких, как IP, TCP и UDP. UDP располагается на транспортном уровне поверх IP (протокола сетевого уровня). Транспортный уровень обеспечивает взаимодействие между сетями через шлюзы. В нем используются IP-адреса для отправки пакетов данных через Интернет или другую сеть с помощью разнообразных драйверов устройств.

Прежде чем приступать к изучению работы UDP, обратимся к основной терминологии, которую нужно хорошо знать. Ниже вкратце определим основные термины, связанные с UDP:

**Пакеты**

В передаче данных пакетом называется последовательность двоичных цифр, представляющих данные и управляющие сигналы, которые передаются и коммутируются через хост. Внутри пакета эта информация расположена в соответствии со специальным форматом.

**Дейтаграммы**

Дейтаграмма — это отдельный, независимый пакет данных, несущий информацию, достаточную для передачи от источника до пункта назначения, поэтому никакого дополнительного обмена между источником, адресатом и транспортной сетью не требуется.

**MTU (Maximum Transmission Unit)**

MTU характеризует канальный уровень и соответствует максимальному числу байтов, которое можно передать в одном пакете. Другими словами MTU — это самый большой пакет, который может переносить данная сетевая среда. Например, Ethernet имеет фиксированный MTU, равный 1500 байтам. В UDP, если размер дейтаграммы больше MTU, протокол IP выполняет фрагментацию, разбивая дейтаграмму на более мелкие части (фрагменты) так, чтобы каждый фрагмент был меньше MTU.

**Порты**

Чтобы поставить в соответствие входящим данным конкретный процесс, выполняемый в компьютере, UDP использует порты. UDP направляет пакет в соответствующее место, используя номер порта, указанный в UDP-заголовке дейтаграммы. Порты представлены 16-битными номерами и, следовательно, принимает значения в диапазоне от 0 до 65 535. Порты, которые также называют конечными точками логических соединений, разделены на три категории:

* Хорошо известные порты - от 0 до 1023
* Регистрируемые порты — от 1024 до 49151
* Динамические / частные порты — от 49152 до 65535

Заметим, что порты UDP могут получать более одного сообщения в каждый промежуток времени. В некоторых случаях сервисы TCP и UDP могут использовать одни и те же номера портов, например 7 (Echo) или 23 (Telnet).

UDP использует следующие известные порты:

Таблица. UDP порты

|  |  |
| --- | --- |
| Номер порта | Описание |
| 15 | NETSTAT — Состояние сети |
| 53 | DNS — Сервер доменных имен |
| 69 | TFTP - Простейший протокол передачи файлов |
| 137 | Служба имен NetBIOS |
| 138 | Дейтаграммная служба NetBIOS |

Перечень портов UDP и TCP поддерживается агентством IANA (Internet Assigned Numbers Authority).

**IP-адреса**

Дейтаграмма IP состоит из 32-битных IP-адресов источника и назначения. IP-адрес назначения задает конечную точку для дейтаграммы UDP, а IP-адрес источника используется для получения информации о том, кто отправил сообщение. В пункте назначения пакеты фильтруются, и те из них, адреса источников которых не входят в допустимый набор адресов, отбрасываются без уведомления отправителя.

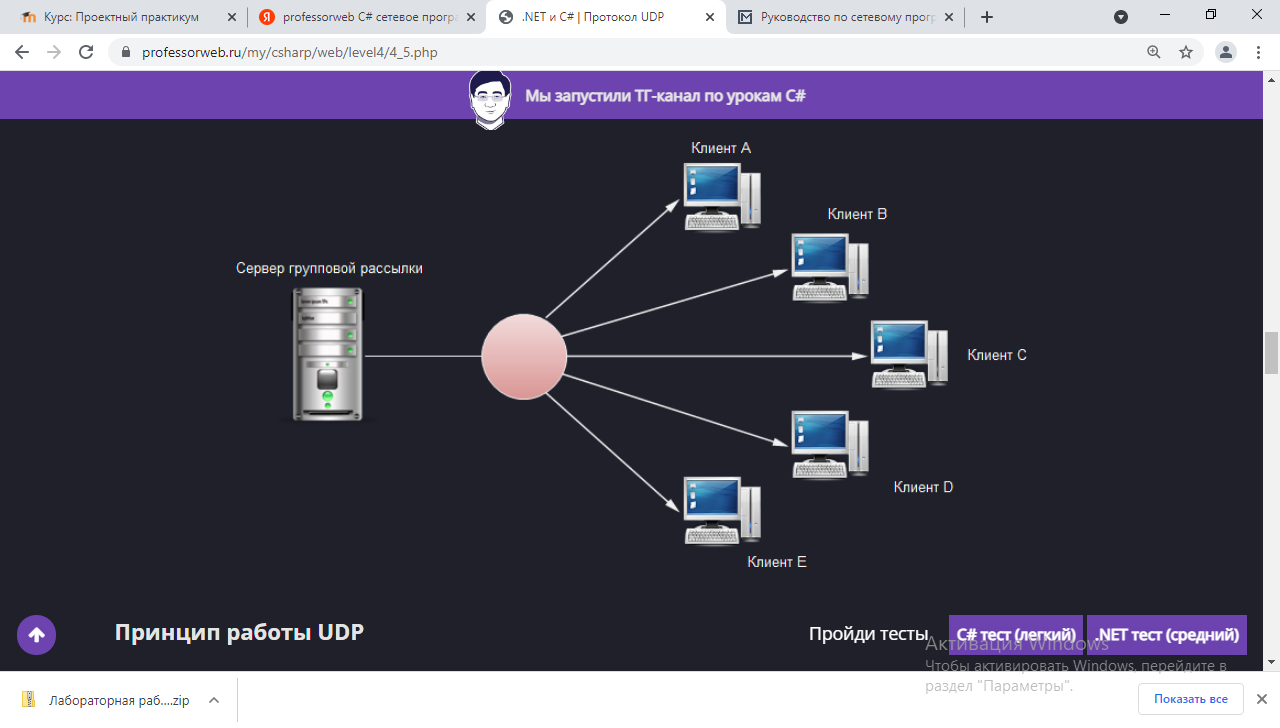
Однонаправленный IP-адрес уникально определяет хост в сети, тогда как групповой IP-адрес определяет конкретную группу адресов в сети. Широковещательные IP-адреса получаются и обрабатываются всеми хостами локальной сети или конкретной подсети.

**TTL**

Значение времени жизни, или TTL (time-to-live), позволяет установить верхний предел числа маршрутизаторов, через которые может пройти дейтаграмма. Значение TTL не дает пакетам попасть в бесконечные циклы. Оно инициализируется отправителем и уменьшается на единицу каждым маршрутизатором, обрабатывающим дейтаграмму. Когда значение TTL становится нулевым, дейтаграмма отбрасывается.

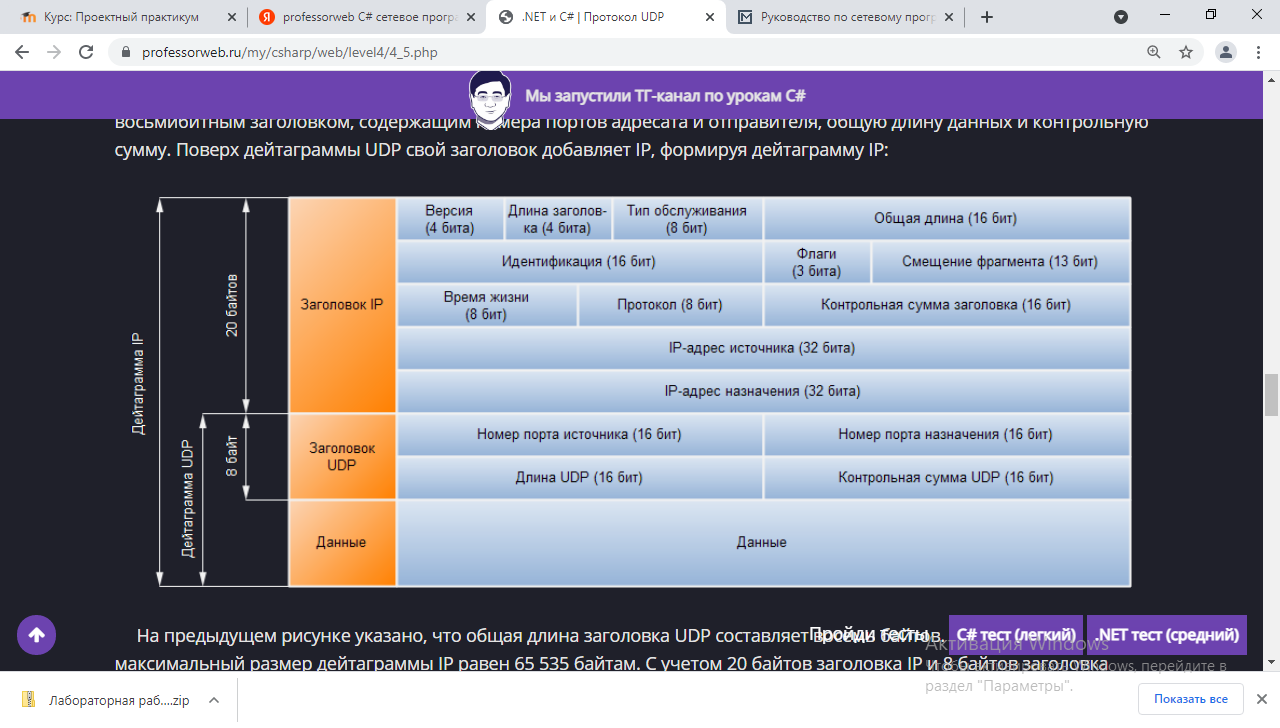
**Групповая рассылка**

Групповая рассылка — это открытый, базирующийся на стандартах, метод одновременного распространения идентичной информации нескольким пользователям. Групповая рассылка является основным средством протокола UDP, она невозможна для протокола TCP. Групповая рассылка позволяет добиться взаимодействия одного со многими, например, делает возможными такие использования, как рассылка новостей и почты нескольким получателям, интернет-радио или демонстрационные программы реального времени. Групповая рассылка не так сильно нагружает сеть, как широковещательная передача, поскольку данные отправляются сразу нескольким пользователям:



**Принцип работы UDP**

Когда приложение, базирующееся на UDP, отправляет данные другому хосту в сети, UDP дополняет их восьмибитным заголовком, содержащим номера портов адресата и отправителя, общую длину данных и контрольную сумму. Поверх дейтаграммы UDP свой заголовок добавляет IP, формируя дейтаграмму IP:



На предыдущем рисунке указано, что общая длина заголовка UDP составляет восемь байтов. Теоретически максимальный размер дейтаграммы IP равен 65 535 байтам. С учетом 20 байтов заголовка IP и 8 байтов заголовка UDP длина данных пользователя может достигать 65 507 байтов. Однако большинство программ работают с данными меньшего размера. Так, для большинства приложений по умолчанию установлена длина приблизительно 8192 байта, поскольку именно такой объем данных считывается и записывается сетевой файловой системой (NFS). Можно устанавливать размеры входного и выходного буферов.

Контрольная сумма нужна, чтобы проверить были ли данные доставлены в пункт назначения правильно или были искажены. Она охватывает как заголовок UDP, так и данные. Байт-заполнитель используется, если общее число октетов дейтаграммы нечетно. Если полученная контрольная сумма равна нулю, получатель фиксирует ошибку контрольной суммы и отбрасывает дейтаграмму. Хотя контрольная сумма является необязательным средством, ее всегда рекомендуется включать.

На следующем шаге уровень IP добавляет 20 байтов заголовка, включающего TTL, IP-адреса источника и получателя и другую информацию. Это действие называют IP-инкапсуляцией.

Как упоминалось ранее, максимальный размер пакета равен 65 507 байтам. Если пакет превышает установленный по умолчанию размер MTU, то уровень IP разбивает пакет на сегменты. Эти сегменты называются фрагментами, а процесс разбиения данных на сегменты — фрагментацией. Заголовок IP содержит всю информацию о фрагментах.

Когда приложение-отправитель "выбрасывает" дейтаграмму в сеть, она направляется по IP-адресу назначения, указанному в заголовке IP. При проходе через маршрутизатор значение времени жизни (TTL) в заголовке IP уменьшается на единицу.

Когда дейтаграмма прибывает к заданному назначению и порту, уровень IP по своему заголовку проверяет, фрагментирована ли дейтаграмма. Если это так, дейтаграмма собирается в соответствии с информацией, имеющейся в заголовке. Наконец прикладной уровень извлекает отфильтрованные данные, удаляя заголовок.

**Недостатки UDP**

По сравнению с TCP UDP имеет следующие недостатки:

* Отсутствие сигналов квитирования. Перед отправкой пакета UDP, отправляющая сторона не обменивается с получающей стороной квитирующими сигналами. Следовательно, у отправителя нет способа узнать, достигла ли дейтаграмма конечной системы. В результате UDP не может гарантировать, что данные будут действительно доставлены адресату (например, если не работает конечная система или сеть).
* Напротив, протокол TCP ориентирован на установление соединений и обеспечивает взаимодействие между подключенными к сети хостами, используя пакеты. В TCP применяются сигналы квитирования, позволяющие проверить успешность транспортировки данных.
* Использование сессий. Ориентированность TCP на соединения поддерживается сеансами между хостами. TCP использует идентификатор сеанса, позволяющий отслеживать соединения между двумя хостами. UDP не имеет поддержки сеансов из-за своей природы, не ориентированной на соединения.
* Надежность. UDP не гарантирует, что адресату будет доставлена только одна копия данных. Чтобы отправить конечной системе большой объем данных, UDP разбивает его на небольшие части. UDP не гарантирует, что эти части будут доставлены по назначению в том же порядке, в каком они создавались в источнике. Напротив, TCP вместе с номерами портов использует порядковые номера и регулярно отправляемые подтверждения, гарантирующие упорядоченную доставку данных.
* Безопасность. TCP более защищен, чем UDP. Во многих организациях брандмауэры и маршрутизаторы не пропускают пакеты UDP. Это связано с тем, что хакеры могут воспользоваться портами UDP, не устанавливая явных соединений.
* Управление потоком. В UDP управление потоком отсутствует, в результате плохо спроектированное UDP-приложение может захватить значительную часть пропускной способности сети.

**Преимущества UDP**

По сравнению с TCP UDP имеет следующие преимущества:

* Нет установки соединения. UDP является протоколом без организации соединений, поэтому он освобождает от накладных расходов, связанных с установкой соединений. Поскольку UDP не пользуется сигналами квитирования, то задержек, вызванных установкой соединений, также удается избежать. Именно поэтому DNS отдает предпочтение UDP перед TCP — DNS работала бы гораздо медленнее, если бы она выполнялась через TCP.
* Скорость. UDP работает быстрее TCP. По этой причине многие приложения предпочитают не TCP, a UDP. Те же средства, которые делают TCP более устойчивым (например сигналы квитирования), замедляют его работу.
* Топологическое разнообразие. UDP поддерживает взаимодействия "один с одним" и "один с многими", в то время как TCP поддерживает лишь взаимодействие "один с одним".
* Накладные расходы. Работа с TCP означает повышенные накладные расходы, издержки, налагаемые UDP, существенно ниже. TCP по сравнению с UDP использует значительно больше ресурсов операционной системы, и, как следствие, в таких средах, где серверы одновременно обслуживают многих клиентов, широко используют UDP.
* Размер заголовка. Для каждого пакета заголовок UDP имеет длину всего лишь восемь байтов, в то время как TCP имеет 20-байтовые заголовки, и поэтому UDP потребляет меньше пропускной способности сети.

**Класс UdpClient**

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

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

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

Создание экземпляра класса UdpClient

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

Простейший способ состоит в вызове конструктора по умолчанию (без передачи параметров). Когда экземпляр класса создается так, нужно или вызвать метод Connect() и установить соединение, или задать информацию о соединении при отправке данных.

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

// Создаем UdpClient, используя номер порта

try

{

UdpClient udpClient = new UdpClient(5001);

}

catch (ArgumentOutOfRangeException ex)

{

Console.WriteLine("Некорректный номер порта");

}

catch (SocketException ex)

{

Console.WriteLine("Порт уже используется");

}

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

Если конструктору передается пустой объект IPEndPoint, порождается исключение ArgumentNullException:

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

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 5001);

// Создаем UdpClient, используя экземпляр IPEndPoint

try

{

UdpClient udpClient = new UdpClient(ipEndPoint);

}

catch (ArgumentNullException ex)

{

Console.WriteLine(ex.ToString());

}

Последний способ создания экземпляра класса UdpClient состоит в передаче конструктору имени хоста и номера порта. В этом случае конструктор инициализируется именем хоста и номером порта удаленного хоста. Это позволяет исключить шаг с вызовом метода Connect(), поскольку он вызывается из конструктора (как можно видеть, если вызвать ILDasm и исследовать IL-код конструктора).

**Определение информации о соединении**

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

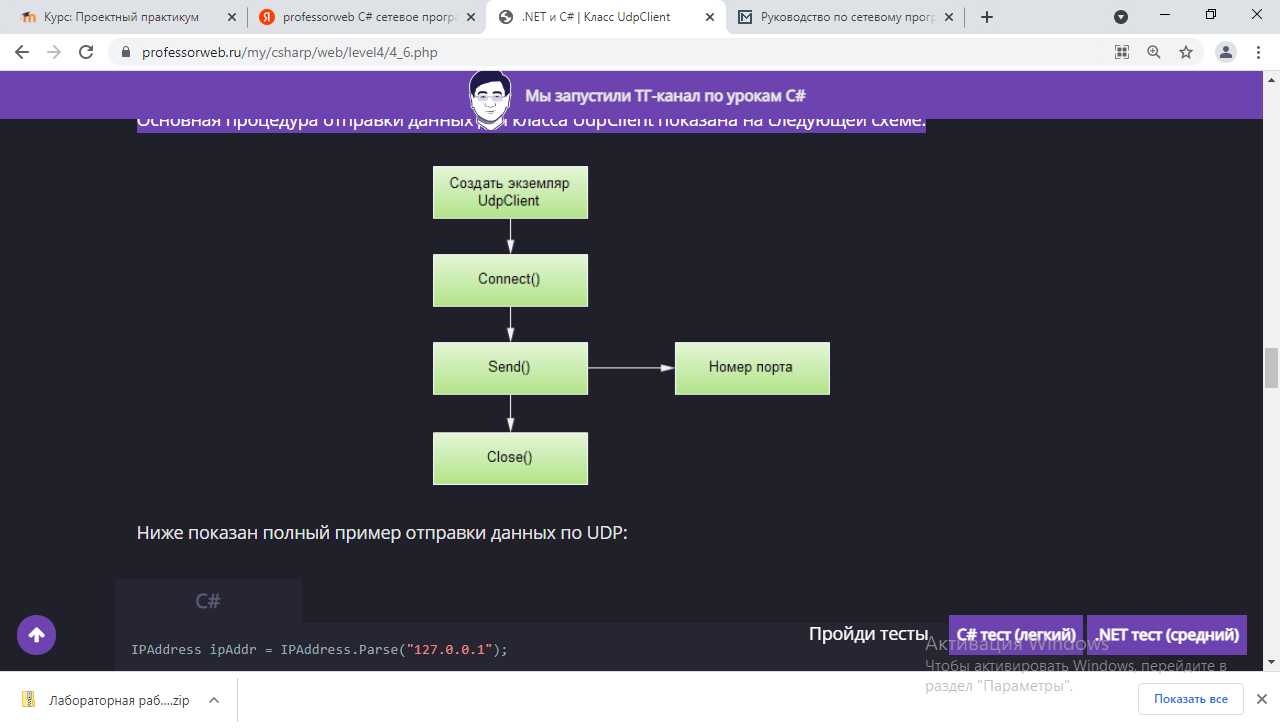
Эту информацию можно указать в любом из следующих трех мест: как мы уже видели, ее можно задать в конструкторе UdpClient, можно явно вызвать метод Connect() класса UdpClient или ее можно включить в метод Send() при фактической передаче данных.

Существуют три перегруженных метода Connect(): использующий объект IPEndPoint; устанавливающий соединение, используя IP-адрес и номер порта удаленного хоста; использующий имя DNS или машины и номер порта удаленного хоста.

**Отправка данных через объект UdpClient**

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

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



Ниже показан полный пример отправки данных по UDP:

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

try

{

// Создаем UdpClient

UdpClient udpClient = new UdpClient();

// Соединяемся с удаленным хостом

udpClient.Connect(ipAddr, 5001);

// Отправка простого сообщения

byte[] bytes = Encoding.UTF8.GetBytes("Test");

udpClient.Send(bytes, bytes.Length);

// Закрываем соединение

udpClient.Close();

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

**Получение данных с использованием объекта UdpClient**

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

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

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

* Создать экземпляр класса UdpClient.
* Получить данные (Receive()).
* Закрыть UdpClient (Close())

Все три шага показаны в следующем примере:

try

{

// Создаем UdpClient

UdpClient udpClient = new UdpClient(5001);

// Создаем переменную IPEndPoint, чтобы передать ссылку на нее в Receive()

IPEndPoint RemoteIPEndPoint = null;

// Получение данных

byte[] bytes = udpClient.Receive(ref RemoteIPEndPoint);

string results = Encoding.UTF8.GetString(bytes);

// Закрываем соединение

udpClient.Close();

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

**Методы групповой рассылки**

Давайте рассмотрим методы класса UdpClient, которые будут использоваться при групповой рассылке.

**Метод JoinMulticastGroup()**

Этот метод позволяет присоединиться к группе. С этим методом объект UdpClient может получать групповые дейтаграммы, рассылаемые по указанному IP-адресу. Групповая рассылка позволяет доставлять данные по нескольким назначениям. Она характерна для протокола UDP и широко распространена в мире Интернета.

Метод JoinMulticastGroup() позволяет присоединиться к групповому IP-адресу. Существуют две перегруженные версии этого метода. Первая принимает только объект IPAddress, представляющий IP-адрес группы, к которой мы хотим присоединиться. Объект UdpClient будет получать любые дейтаграммы, рассылаемые по этому IP-адресу:

// Создаем UdpClient

UdpClient udpClient = new UdpClient();

// IP-адрес группы

IPAddress multicastIP = IPAddress.Parse("96.96.96.96");

try

{

// Присоединяемся к группе

udpClient.JoinMulticastGroup(multicastIP);

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

Второй перегруженный метод принимает групповой IP-адрес со значением TTL в поле int:

udpClient.JoinMulticastGroup(multicastIP, 30);

**Метод DropMulticastGroup()**

Этот метод можно использовать, чтобы отсоединить объект UdpClient от группы. Этот метод принимает один параметр — IP-адрес группы, которую этот клиент должен оставить:

// Отсоединяем группу

udpClient.DropMulticastGroup(multicastIP);

**Задание 1**: Реализовать ниже приведенный пример чата обмена текстовыми сообщениями на протоколе UDP:

**Пример приложения UDP**

Чтобы проиллюстрировать работу с UDP, разработаем простое диалоговое приложение на языке C#, использующее класс UdpClient. Диалоговое приложение использует отдельный поток, чтобы слушать сообщения от удаленных хостов.

Это приложение разделено на три логические части. В первой части пользователю предлагается ввести информацию о локальных и удаленных портах и удаленном IP-адресе, которые он хочет использовать. Например, порт 5001 используется как отправляющий порт для хоста А и как приемный порт для хоста В, для порта 5002 все наоборот.

Во второй части приложение слушает входящие данные от удаленного хоста. Как обсуждалось ранее, метод Receive() проверяет наличие входящих дейтаграмм и блокирует поток, пока от удаленного хоста не поступит сообщение. Чтобы отделить этот процесс от основной последовательности действий, создается новый поток.

Третий логический блок в приложении принимает данные, введенные пользователем, и отправляет их указанному удаленному порту. Он выполняется на основном потоке, пока рабочий поток продолжает слушать входящие данные.

Ниже приведен полный код программы, реализующей простой чат:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

namespace UdpSample

{

class Chat

{

private static IPAddress remoteIPAddress;

private static int remotePort;

private static int localPort;

[STAThread]

static void Main(string[] args)

{

try

{

// Получаем данные, необходимые для соединения

Console.WriteLine("Укажите локальный порт");

localPort = Convert.ToInt16(Console.ReadLine());

Console.WriteLine("Укажите удаленный порт");

remotePort = Convert.ToInt16(Console.ReadLine());

Console.WriteLine("Укажите удаленный IP-адрес");

remoteIPAddress = IPAddress.Parse(Console.ReadLine());

// Создаем поток для прослушивания

Thread tRec = new Thread(new ThreadStart(Receiver));

tRec.Start();

while (true)

{

Send(Console.ReadLine());

}

}

catch (Exception ex)

{

Console.WriteLine("Возникло исключение: " + ex.ToString() + "\n " + ex.Message);

}

}

private static void Send(string datagram)

{

// Создаем UdpClient

UdpClient sender = new UdpClient();

// Создаем endPoint по информации об удаленном хосте

IPEndPoint endPoint = new IPEndPoint(remoteIPAddress, remotePort);

try

{

// Преобразуем данные в массив байтов

byte[] bytes = Encoding.UTF8.GetBytes(datagram);

// Отправляем данные

sender.Send(bytes, bytes.Length, endPoint);

}

catch (Exception ex)

{

Console.WriteLine("Возникло исключение: " + ex.ToString() + "\n " + ex.Message);

}

finally

{

// Закрыть соединение

sender.Close();

}

}

public static void Receiver()

{

// Создаем UdpClient для чтения входящих данных

UdpClient receivingUdpClient = new UdpClient(localPort);

IPEndPoint RemoteIpEndPoint = null;

try

{

Console.WriteLine(

"\n-----------\*\*\*\*\*\*\*Общий чат\*\*\*\*\*\*\*-----------");

while (true)

{

// Ожидание дейтаграммы

byte[] receiveBytes = receivingUdpClient.Receive(

ref RemoteIpEndPoint);

// Преобразуем и отображаем данные

string returnData = Encoding.UTF8.GetString(receiveBytes);

Console.WriteLine(" --> " + returnData.ToString());

}

}

catch (Exception ex)

{

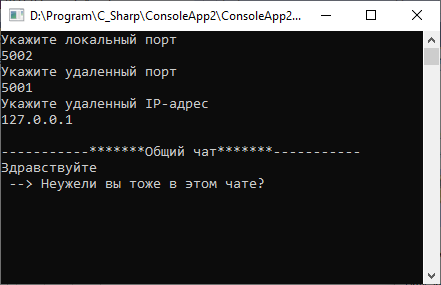
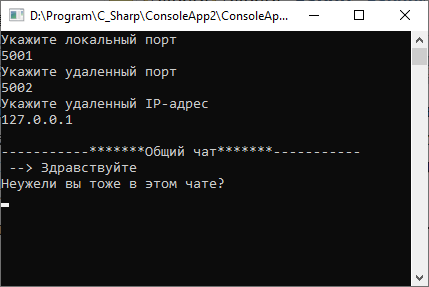
Console.WriteLine("Возникло исключение: " + ex.ToString() + "\n " + ex.Message);

}

}

}

}



**Задание 2**: Реализовать ниже приведенный пример с передачей файлов по сети с помощью протокола UDP:

**Передача файлов по UDP**

Мы узнали, как пользоваться классом UdpClient для отправки и получения дейтаграмм, и, опираясь на один и тот же принцип, создали приложение интерактивной переписки. Теперь научимся пользоваться классом UdpClient для передачи файла и сериализованного объекта.

Программы отправителя и получателя разделены на две логические части. В первой части отправитель посылает получателю (или получателям) информацию о файле (а именно расширение и размер файла) как сериализованный объект, а во второй части отправляется сам файл. В получателе первая часть принимает сериализованный объект с соответствующей информацией, а вторая часть создает файл на машине получателя. Чтобы сделать приложение более интересным, откроем сохраненный файл соответствующей программой (например, .doc-файл можно открыть программой Microsoft Word, а .html-файл — браузером).

**Файловый сервер**

Файловый сервер — это простое консольное приложение, реализованное в классе UdpFileServer. В этом классе есть вложенный класс FileDetails, содержащий информацию о файле — тип и размер файла. Начнем с импорта необходимых пространств имен и объявления полей класса. В классе есть пять закрытых полей: экземпляр класса FileDetails, объект UdpClient, а также информация о соединении с удаленным клиентом и объект FileStream для считывания файла, который отправляется клиенту:

using System;

using System.IO;

using System.Net;

using System.Net.Sockets;

using System.Threading;

using System.Xml.Serialization;

class UdpFileServer

{

// Информация о файле (требуется для получателя)

[Serializable]

public class FileDetails

{

public string FILETYPE = "";

public long FILESIZE = 0;

}

private static FileDetails fileDet = new FileDetails();

// Поля, связанные с UdpClient

private static IPAddress remoteIPAddress;

private const int remotePort = 5002;

private static UdpClient sender = new UdpClient();

private static IPEndPoint endPoint;

// Filestream object

private static FileStream fs;

Итак, мы дошли до метода Main() сервера. В этом методе приглашаем пользователя ввести удаленный IP-адрес, по которому нужно отправить файл, путь и имя отправляемого файла. Открываем этот файл в объекте FileStream и определяем его длину. Если она больше максимально допустимой длины, равной 8192 байтам, закрываем UdpClient и FileStream и выходим из приложения. Иначе отправляем информацию о файле, выжидаем две секунды, вызвав метод Thread.Sleep(), и отправляем сам файл:

[STAThread]

static void Main(string[] args)

{

try

{

// Получаем удаленный IP-адрес и создаем IPEndPoint

Console.WriteLine("Введите удаленный IP-адрес");

remoteIPAddress = IPAddress.Parse(Console.ReadLine().ToString());//"127.0.0.1");

endPoint = new IPEndPoint(remoteIPAddress, remotePort);

// Получаем путь файла и его размер (должен быть меньше 8kb)

Console.WriteLine("Введите путь к файлу и его имя");

fs = new FileStream(@Console.ReadLine().ToString(), FileMode.Open, FileAccess.Read);

if (fs.Length > 8192)

{

Console.Write("Файл должен весить меньше 8кБ");

sender.Close();

fs.Close();

return;

}

// Отправляем информацию о файле

SendFileInfo();

// Ждем 2 секунды

Thread.Sleep(2000);

// Отправляем сам файл

SendFile();

Console.ReadLine();

}

catch (Exception eR)

{

Console.WriteLine(eR.ToString());

}

}

Метод SendFileInfo() заполняет поля объекта FileDetails, а затем сериализует объект в MemoryStream, используя объект XmlSerializer. Этот объект считывается в массив байтов и передается методу Send() класса UdpClient, который отправляет информацию о файле клиенту:

public static void SendFileInfo()

{

// Получаем тип и расширение файла

fileDet.FILETYPE = fs.Name.Substring((int)fs.Name.Length - 3, 3);

// Получаем длину файла

fileDet.FILESIZE = fs.Length;

XmlSerializer fileSerializer = new XmlSerializer(typeof(FileDetails));

MemoryStream stream = new MemoryStream();

// Сериализуем объект

fileSerializer.Serialize(stream, fileDet);

// Считываем поток в байты

stream.Position = 0;

Byte[] bytes = new Byte[stream.Length];

stream.Read(bytes, 0, Convert.ToInt32(stream.Length));

Console.WriteLine("Отправка деталей файла...");

// Отправляем информацию о файле

sender.Send(bytes, bytes.Length, endPoint);

stream.Close();

}

Метод SendFile() просто считывает содержимое файла из FileStream в массив байтов и отправляет его клиенту:

private static void SendFile()

{

// Создаем файловый поток и переводим его в байты

Byte[] bytes = new Byte[fs.Length];

fs.Read(bytes, 0, bytes.Length);

Console.WriteLine("Отправка файла размером " + fs.Length + " байт");

try

{

// Отправляем файл

sender.Send(bytes, bytes.Length, endPoint);

}

catch (Exception eR)

{

Console.WriteLine(eR.ToString());

}

finally

{

// Закрываем соединение и очищаем поток

fs.Close();

sender.Close();

}

Console.WriteLine("Файл успешно отправлен.");

Console.Read();

}

**Файловый клиент**

Файловый клиент или приемник файла — тоже консольное приложение, реализованное в классе UdpFileClient. Здесь также начинаем с импорта необходимых пространств имен и объявления полей класса:

class UdpFileClient

{

// Детали файла

[Serializable]

public class FileDetails

{

public string FILETYPE = "";

public long FILESIZE = 0;

}

private static FileDetails fileDet;

// Поля, связанные с UdpClient

private static int localPort = 5002;

private static UdpClient receivingUdpClient = new UdpClient(localPort);

private static IPEndPoint RemoteIpEndPoint = null;

private static FileStream fs;

private static Byte[] receiveBytes = new Byte[0];

Метод Main() этого приложения только вызывает два метода, чтобы получить, соответственно, информацию о файле и сам файл:

[STAThread]

static void Main(string[] args)

{

// Получаем информацию о файле

GetFileDetails();

// Получаем файл

ReceiveFile();

}

Метод GetFileDetails() вызывает метод Receive() объекта UdpClient. Тот в свою очередь получает от сервера сериализованный объект FileDetails, который сохраняется в объекте MemoryStream. Для десериализации этого потока в объект FileDetails используем объект XmlSerializer и отображаем полученную информацию на консоли:

private static void GetFileDetails()

{

try

{

Console.WriteLine("-----------\*\*\*\*\*\*\*Ожидание информации о файле\*\*\*\*\*\*\*-----------");

// Получаем информацию о файле

receiveBytes = receivingUdpClient.Receive(ref RemoteIpEndPoint);

Console.WriteLine("----Информация о файле получена!");

XmlSerializer fileSerializer = new XmlSerializer(typeof(FileDetails));

MemoryStream stream1 = new MemoryStream();

// Считываем информацию о файле

stream1.Write(receiveBytes, 0, receiveBytes.Length);

stream1.Position = 0;

// Вызываем метод Deserialize

fileDet = (FileDetails)fileSerializer.Deserialize(stream1);

Console.WriteLine("Получен файл типа ." + fileDet.FILETYPE +

" имеющий размер " + fileDet.FILESIZE.ToString() + " байт");

}

catch (Exception eR)

{

Console.WriteLine(eR.ToString());

}

}

Метод ReceiveFile() получает файл от сервера и сохраняет его на диске под именем temp, добавляя расширение, извлеченное из объекта FileDetails. Затем вызываем статический метод Process.Start() и открываем документ связанной с расширением программой:

public static void ReceiveFile()

{

try

{

Console.WriteLine("-----------\*\*\*\*\*\*\*Ожидайте получение файла\*\*\*\*\*\*\*-----------");

// Получаем файл

receiveBytes = receivingUdpClient.Receive(ref RemoteIpEndPoint);

// Преобразуем и отображаем данные

Console.WriteLine("----Файл получен...Сохраняем...");

// Создаем временный файл с полученным расширением

fs = new FileStream("temp." + fileDet.FILETYPE, FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite);

fs.Write(receiveBytes, 0, receiveBytes.Length);

Console.WriteLine("----Файл сохранен...");

Console.WriteLine("-------Открытие файла------");

// Открываем файл связанной с ним программой

Process.Start(fs.Name);

}

catch (Exception eR)

{

Console.WriteLine(eR.ToString());

}

finally

{

fs.Close();

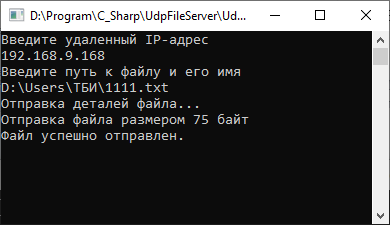
receivingUdpClient.Close();

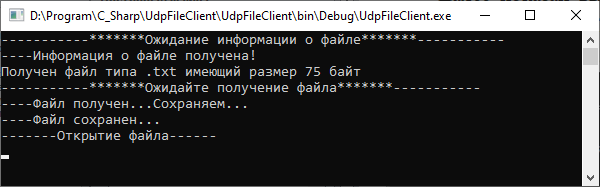
Console.Read();

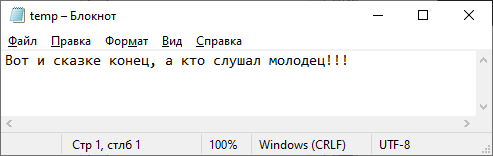
}

}

Далее показан вывод на консоль на клиенте и сервере, полученный после запуска этой программы:







**Высокоуровневые протоколы, основанные на UDP**

Протокол UDP полезен там, где порядок доставки не имеет значения и его не нужно обеспечивать. Поскольку отправителю неизвестно, какое назначение активно, он использует номер порта, чтобы указать тип обслуживания, требуемый от удаленного хоста.

В современном мире Интернета существуют многочисленные приложения, использующие UDP-сервисы. Большинство пользователей не знают, что UDP является предпочтительным транспортным протоколом для DCOM. Не ориентированная на соединения природа UDP позволяет DCOM выполнять определенную оптимизацию, объединяя численные низкоуровневые пакеты подтверждений с реальными данными и сообщениями тестового опроса (pinging). Этот подход действительно минимизирует число циклов сетевого обмена и, следовательно, повышает скорость и производительность. Сетевые программы компании Microsoft используют UDP для входа в систему, просмотра и разрешения имен.

Вот некоторые высокоуровневые протоколы, базирующиеся на UDP:

* RTP (Real-time Protocol) - средства информации реального времени
* NFS (Network File System) - удаленный файловый сервер
* SNMP (Simple Network Management Protocol) - сетевое управление
* RIP (Routing Information Protocol) - Протокол маршрутизации
* DNS (Domain Name Service) - разрешение имен хостов
* TFTP (Trivial File Transfer Protocol) - приложение передачи файлов
* RFC (Remote Procedure Call) - типичная модель клиент-сервер
* LDAP (Lightweight Directory Access Protocol) - служба каталогов

**Real-time Protocol (RTP)**

RTP — это протокол прикладного уровня, предназначенный для доставки такой информации реального времени, как аудиовизуальная информация, через частные или общественные сети с одиночными или групповыми IP-адресами. Например, Microsoft NetMeeting использует RTP для рассылки информации реального времени через Интернет.

Приложение, базирующееся на RTP, реализует RTP, используя протокол UDP и добавляя некоторую дополнительную функциональность, обеспечивающую ведение порядковых номеров, идентификацию нагрузки в сети, идентификацию источника и создание меток времени.

**Network File System (NFS)**

Еще одним популярным приложением, использующим UDP, является Network Fife System, которая обеспечивает прозрачный доступ к файлам и файловым системам через сеть. Преимущество NFS перед FTP (File Transfer Protocol) заключено в прозрачном доступе к файлам: NFS может обращаться к части файла, на которую ссылается приложение или процесс.

NFS построена на Sun RPC и для выполнения операций с файлами использует зарезервированный порт 2049.

**Simple Network Management Protocol (SNMP)**

Как следует из названия, Simple Network Management Protocol (SNMP) — это протокол управления сетью, широко используемый в сетях. SNMP позволяет администраторам наблюдать за удаленными хостами и шлюзами в сети и управлять ими. Служба SNMP может обрабатывать один или несколько запросов от хоста. Этот протокол организует взаимодействие между программой управления, выполняемой администратором, и агентом сетевого управления, выполняемой на хосте. SNMP использует порты 161 и 162 для менеджера и агента соответственно.

**Domain Name Service (DNS)**

Domain Name Service — это распределенная база данных, используемая приложениями TCP/IP для установления соответствия между именами хостов и IP-адресами. Для приложений DNS предпочтителен протокол UDP, использующий порт 53 для отправки запросов DNS на сервер имен.

**Trivial File Transfer Protocol (TFTP)**

TFTP — еще один протокол прикладного уровня, полезный для передачи файлов между удаленными хостами. Этот протокол предназначен для использования на самозагружаемых бездисковых системах. Для действий при передаче файлов используется UDP-порт 69.

**Задание 3**

Проанализировать приложения реализованных в архивах (источники позаимствованы из источника [2]):

* udplocal1csharp\_vs11

Исходник написан на языке программирования C#, первая часть исходника предназначена для ознакомления с высокоуровневым классом UdpClient, не отвлекаясь на другие детали. Для отправки и извлечения сообщений используются блокирующие методы. Небольшие размеры отправляемых сообщений лишь на короткое время блокируют программу. Для приема сообщений не обойтись без дополнительного потока, иначе блокирование приложения может продолжаться все время работы программы.

**Примечания**: Исходник состоит из двух приложений: первая отправляет сообщение, вторая извлекает сообщение. Для тестирования исходного кода необходимо разрешить программу извлечения сообщений в брандмауэре операционной системы.

* udplocal2csharp\_vs11

Исходник на языке C#, для тестирования и изучения протокола UDP. В отличие от первой части в данной приложении получатель и отправитель в одном пакете. Тестовое приложение рассчитано на передачу сообщений по локальной сети. Для взаимодействия двух программ на одном компьютере необходимо изменить ip хост-адрес одной на 127.0.0.2. У другой же программы изменить удаленный ip адрес на 127.0.0.2. Только в этом случае программы, работающие по UDP, смогут общаться на одной машине.

Это связано с тем, что невозможно запустить две прослушивающие (извлекающие сообщения) программы с одинаковыми ip адресами и одинаковыми портами. На разных машинах программы будут работать нормально, и отсылать, получать сообщения без проблем. Протокол UDP позволяет работать по сети без установки соединения, что и демонстрируют тестовые программы.

**Блокирующие методы в отдельных потоках**

Работа сетевого протокола без установления соединений и без проверки отсылаемых и входных данных дают существенный выигрыш в скорости. Хотя использованы блокирующие методы, но при использовании их в отдельном потоке главный полностью владеет интерфейсом программы и не блокируется.

**Задание 5**

Реализовать с применением технологии WPF чат обмена текстовыми сообщениями и файлами работающие на протоколе UDP

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/level4/4_5.php> (дата обращения 10.04.2021)
4. URL: <https://professorweb.ru/my/csharp/web/level4/4_6.php> (дата обращения 10.04.2021)
5. URL: <https://professorweb.ru/my/csharp/web/level4/4_7.php> (дата обращения 10.04.2021)
6. URL: <https://professorweb.ru/my/csharp/web/level4/4_8.php> (дата обращения 10.04.2021)
7. URL: <https://professorweb.ru/my/csharp/web/level4/4_9.php> (дата обращения 10.04.2021)