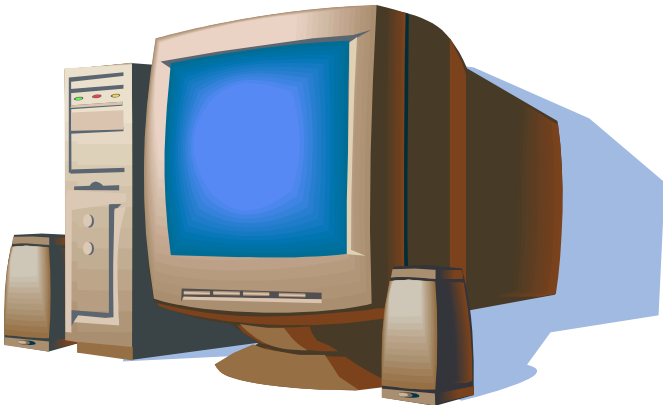
Министерство по развитию информационных технологий и коммуникаций Республики Узбекистан

Каршинский филиал ташкентского университета информационных технологий

Факультет Компьютерный инжиниринг
Кафедра Телекоммуникационный инжиниринг

ОСНОВЫ ПРОГРАММИРОВАНИЯ СЕТИ

Методические указания по выполнению лабораторных работ



Карши-2016

Составитель: асс. А.А.Азимов .

Настоящие методические указания включают в себя лабораторные работы по общему курсу дисциплины «Основы программирования сети”. Основное внимание уделено к основным приципам программирования сети с использованием Visual c#. MC в Visual Studio.
Методические указания предназначены для студентов, обучающихся по направлению бакалавриата 5350100- Телекоммуникационные технологии.

Рецензенты: к.т.н. доц.С.М.Исаев. Каршинский филиал ТУИТ, доцент кафедры Телекоммуникационный инжиниринг
к.т.н. доц.А.М.Тургунов. Каршинский филиал ТУИТ, заведующий кафедры Информационные технологии

Методические указания рекомендованы Методическим советом Каршинского филиала ТУИТ, для использования в учебном процессе.
(протокол №___ - от 2016 года)

Лабораторная работа №1

Изучения средств программированиия сети

Цель работы: Языки программирования программирвание сети и их классификация

1. Практическая часть

В этом уроке мы научимся работать с MS визуальная Студия. Эта среда предлагает множество выпусков для разработки, тестирования и развертывания приложений для веб, настольных и мобильных ПК и даже игровых консолей. С множеством испытаний и средства обеспечения совместимости, вы можете протестировать на виртуальных машинах, задействовать функции BrowserStack, чтобы увидеть, как выглядит приложение в более чем 300 браузерах, запуск отчетов совместимости для выявления закономерностей и проблем между браузерами и многое другое.

MC в Visual Studio поддерживает современные объектно-ориентированные языки:

• Разнообразие поддерживаемых языков всей продукции

• "Аякс", ASP.NET в xaml, JavaScript и динамического HTML, DHTML будет, визуальный основной, визуальный C# в Visual C++ и Visual F#, JScript

• Некоторые платформы поддерживают любой язык программирования

Для разработки сетевых приложений мы используем Visual c#. MC в Visual Studio поддерживает два основных библиотека для сетевого программирования: System.net и системы.чистая.розетки. Другие пространства имен и классы вспомогательные классы для сетевого программирования. Ниже я даю пространства, что важно для проектирования сетевого программного обеспечения.

1.1 .Чистая Framework пространства имен класса

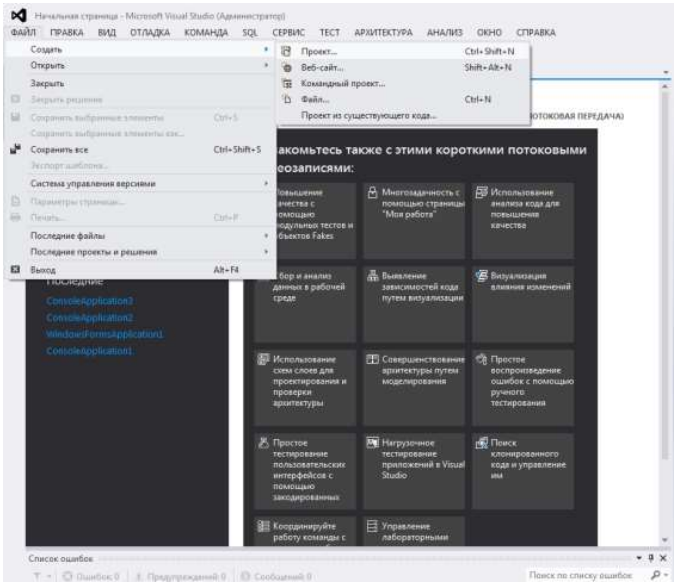
Namespace	**Description of Classes**
Microsoft.Win32	Обрабатывает события, вызываемые операционной системы и реестра управляющих классов
System	Базы .Чистые классы, определяющие часто используемые типы данных и преобразования данных
System.Collections	Определяет списки, очереди, двоичные массивы и коллекции строк

System.Data	Определяет структуру базы данных ADO.NET
ystem.Data.OleDb	Инкапсулирует OLE DB .NET .Сетевая структура базы данных
System.IO	Позволяет читать и записывать в потоки данных и файлов
System.Drawing	Обеспечивает доступ к основным графическим функциям
System.Management	Предоставляет доступ к управления Windows Инструментарий инфраструктуры (WMI) для
System.Net	Обеспечивает доступ к сети функции Windows
System.Net.Sockets	Предоставляет доступ к сокеты Windows (winsock c) интерфейс
System.Security	Обеспечивает доступ к системе разрешения безопасности среды CLR
System.Text	Представляет желании, Юникод, utf-7 и utf-8 кодировки
System.Threading	Позволяет многопоточность программирование
System.Web	Позволяет браузер и веб-сервер
System.Web.Mail	Отправка сообщений электронной почты
System.Windows.Forms	Создается Windows-приложение, использующие стандартные Windows графический интерфейс

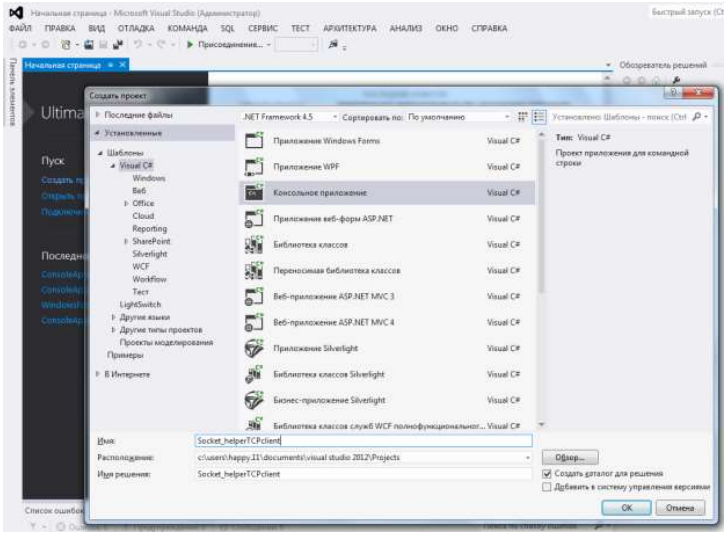
Сейчас мы научимся создавать и настраивать консольные приложения и приложения Windows для работы в сети

1.1 Изучение создания консольного приложения

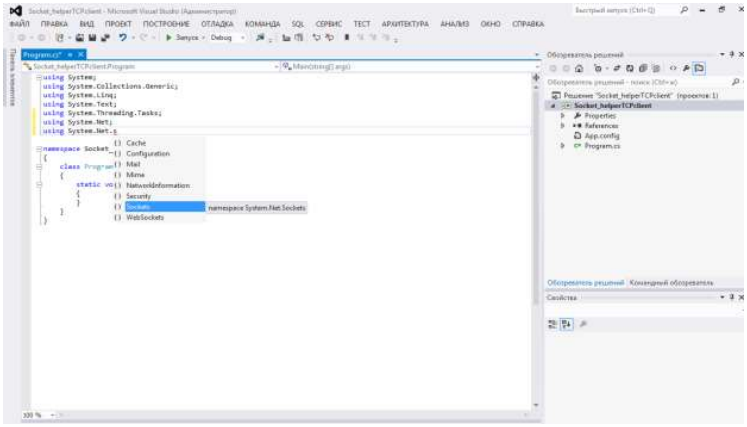
1. Создать новый проект:



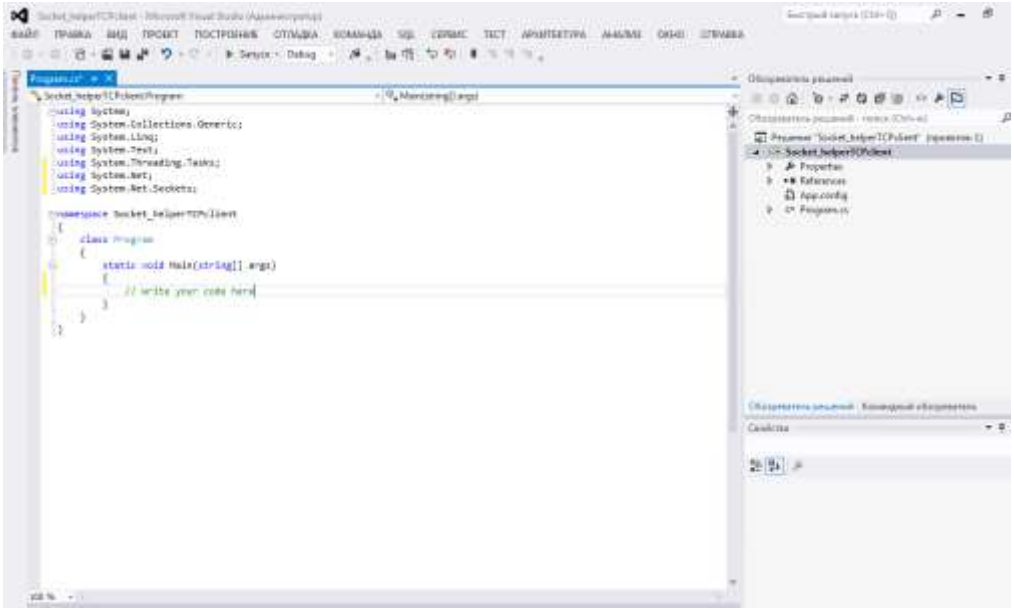
Название проекта и нажмите “ОК”:



Добавить **“**System.Net” and “System.Net.Sockets” **.**NET Framework Class Namespaces для использования сетевых функций системы

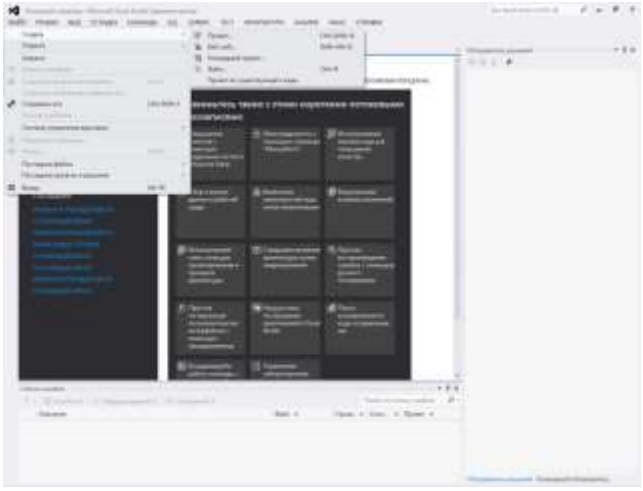


После этого, вы можете написать свой исходный код

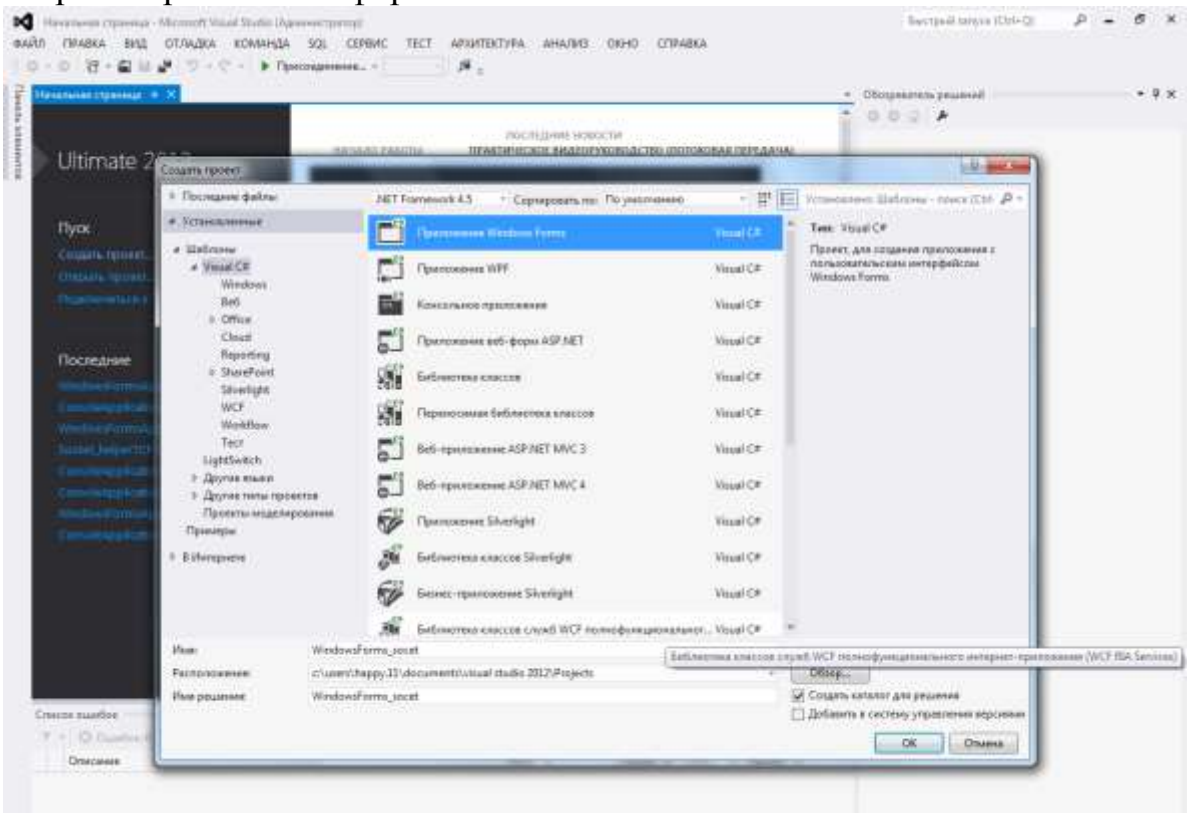


1.2 Создавание форм Windows приложение

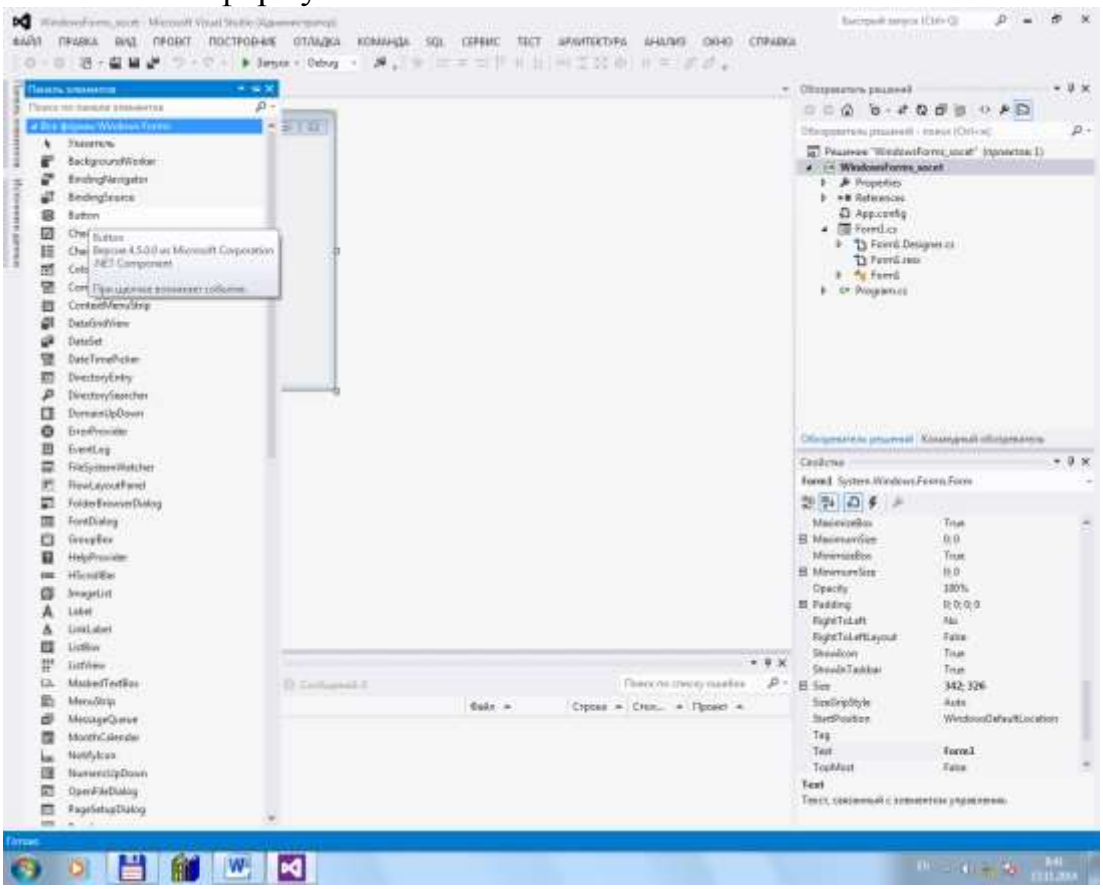
Создать новый проект:



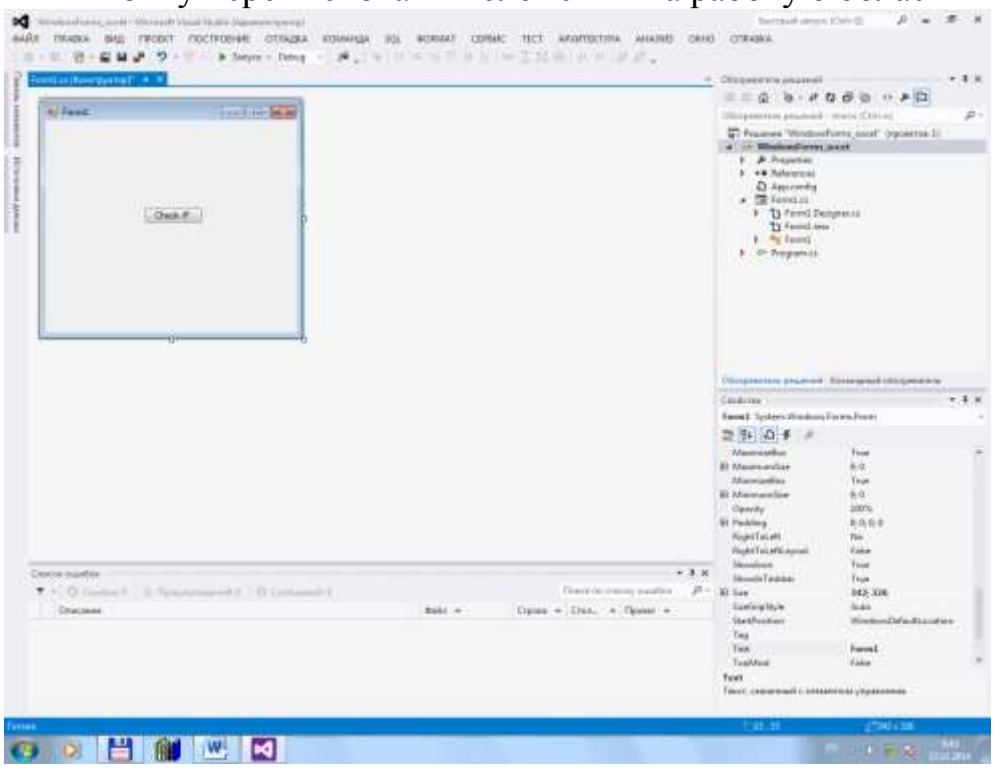
Выберите Приложение форм Windows и назовите его



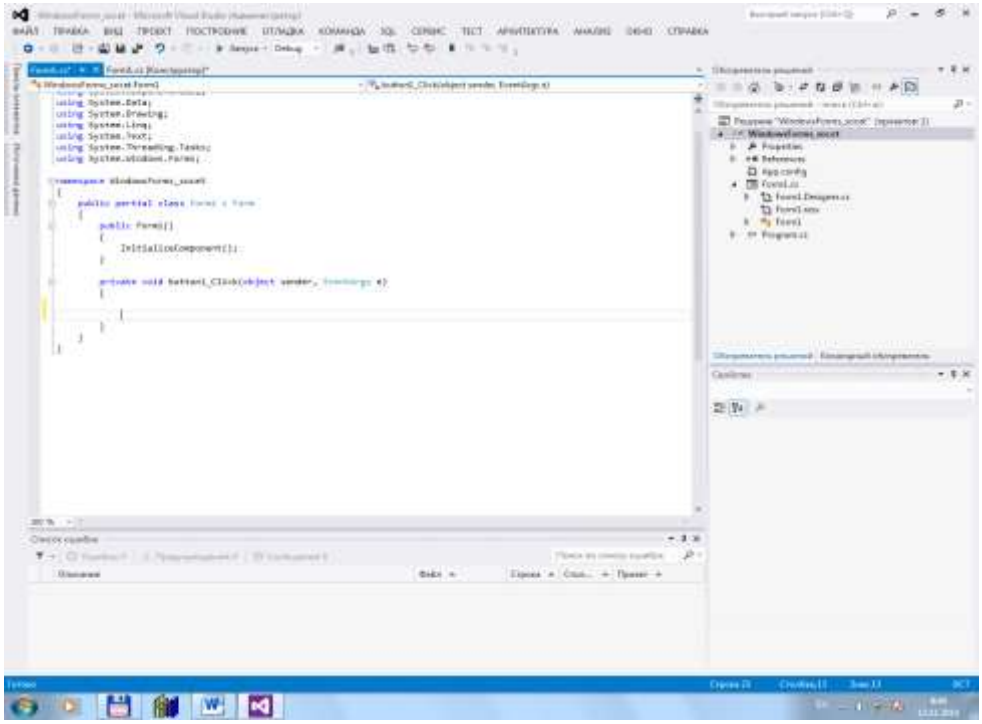
Взять элементы в форму из Toolbox



Настройки и кнопку переименовать и элементы на рабочую область



Нажмите кнопки или другие элементы на рабочей области и написать свой код



2. Вопросы

1. Что такое Сетевое программирование?
2. Интегрированная среда разработки и языки программирования.
3. .Net для программирования в C# и основное значение.
4. Современные объектно-ориентированные языки, которые поддержывает МС Visual студии?
5. Что такое Консольное приложение?
6. Что такое приложение Windows?
7. Сетевое программирование с . Net based IDEs.
8. Сетевое программирование и родной языков.

3. Теоретические сведения

Сети (и сети программирование) прошли долгий путь за последние 20 лет. В первые дни сетевых вычислений (80-е), сетевое программирование оставалось для опытных программистов, которые, как правило, построены приложения, использующие язык программирования С В (В основном) среды Unix. Теперь, сетей везде, от крупных корпораций до небольших домашних пользователей. С таким количеством компьютеров, подключенных через сети, сетевых приложений принимаются необходимостью. Существующие приложения должны включать в себя сетевые функции, чтобы оставаться конкурентоспособными на рынке, и добавив коммуникационной сети для приложений имеет важное значение. Сетевые программы используются для всего, от детской игры для современных систем корпоративной базе данных.

Мы используем языки программирования и интегрированные среды разработки для проектирования и разработки программ.

Интегрированная среда разработки (IDE) или интерактивная среда разработки является программное обеспечение, которое предоставляет широкие возможности для программистов для разработки программного обеспечения. Язь обычно состоит из редактора исходного кода, создание средств автоматизации и отладчик. Большинство современных IDE предлагают интеллектуальные функции автозавершения кода.

Некоторые Иды содержит компилятор, интерпретатор, или как, например, чистая фасоль и Eclipse; другие, например SharpDevelop и Лазаря. Граница между интегрированной среды разработки и другие части более широкой среде разработки программного обеспечения не определены. Иногда системы контроля версий и различные инструменты интегрированы для построения графического интерфейса пользователя. Многие современные среды разработки также имеют обозреватель классов, обозреватель объектов и

иерархия классов, диаграммы, предназначенные для использования в объектно-ориентированной разработки программного обеспечения.

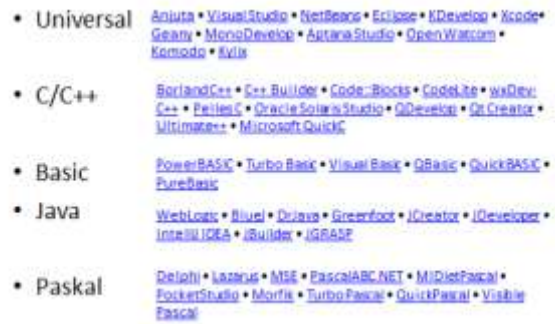
Программисты Unix могут совмещать командной строки средства posix в полную среду разработки, способных разрабатывать большие программы, такие как ядро Linux и его окружения.

Бесплатные инструменты программное обеспечение GNU (Коллекция компиляторов GNU (GCC) и, в GNU отладчик (бгд), GNU, делают) доступны на многих платформах, включая Windows.[4] разработчики, которые в командной строке пользу ориентированных инструментов можно использовать редакторы с поддержкой множества стандартных Unix и GNU построить 1 инструменты, строительство IDE с программы, такие как emacs или vim. Отображение данных отладчик предназначен для продвинутый графический передний конец для многих текстового отладчика стандартных инструментов. Некоторые программисты предпочитают управляющий make-файлов и их производных на аналогичные инструменты код, включенный в полный язь. Например, большинство доноров с использованием базы данных PostgreSQL сделать и gdb напрямую для разработки новых функций.[8] даже при построении СУБД PostgreSQL для Microsoft Windows с использованием Visual c++, Perl скрипты используются в качестве замены сделать, а не полагаться на какие-либо возможности IDE.[9] некоторые Линукс интегрированных сред разработки, таких как он пытается представить графический интерфейс для традиционных операций сборки.

На различных платформах Microsoft Windows и командная строка, инструменты для разработки используются редко. Соответственно, существует множество коммерческих и некоммерческих продуктов. Однако, каждый из них имеет различный дизайн часто создавая несовместимости. Большинство крупных производителей компиляторов для Windows по-прежнему предоставлять бесплатные копии своих инструментов командной строки, в том числе Microsoft (Visual c++, платформы SDK .Net в рамках пакета SDK, утилиты nmake), Эмбаркадеро технологий (компилятор bcc32, утилита make).

Язь всегда были популярны на Apple Macintosh и операционной системы Mac, начиная с мастерской Макинтош программиста, Турбо Паскаль, Паскаль и думаю, думаю средах с середины 1980-х годов. В настоящее время Мак ОС х программисты могут выбирать между родной среды разработки, такие как xcode и открытым исходным кодом инструменты, такие как Eclipse и netbeans. Активностити Комодо-это собственный многоязыковой среде IDE поддерживается на Mac ОС.

С появлением облачных вычислений, некоторые Иды доступны онлайн и работать в веб-браузерах; примерами этого являются онлайн на JavaScript IDE, в Codenvy, команды cloud9 IDE, в firebug и Кодинг.
Таблица IDE и языков программирования мы приводим ниже.



Сетевое программирование всегда было ключевой особенностью операционной системы Майкрософт Windows. К сожалению, вы должны были знать расширенный C или C++ программирования концепций, чтобы использовать функции сетевого программирования в Windows программ. Сейчас, правда,.Net на языках упростить задачу добавления сетевых функций для вашего приложения. Интернет .Net библиотек предоставляют многие сетевые классы, которые можно интегрировать сетевое программирование.
Как сетевой администратор, я написал множество сетевых программ с использованием C и Java языки для Windows и Unix. Сегодня управление сетью и требованиям безопасности, обусловливают необходимость общаться с сетевыми устройствами и отслеживать рабочих станций в сети. Пытаюсь быстро писать чистый код сети может быть трудно, когда вы работаете в структуре с API сокетов (особенно в winsock), и запуск Java-приложений-это часто болезненный опыт из-за медленной скорости обработки и плохой поддержки Windows.
Язык C# решил многие мои проблемы сетевого программирования, позволяя мне быстро создавать прототипы и развертывание сетевых приложений с использованием классов C#. Сочетая в C# Forms библиотека для написания графического кода с c# сокет библиотека для написания сетевого кода делает создание профессиональных сетевых приложений просто. C с сетевой# классы, используемое для того чтобы принять в день, чтобы часто писать займет всего час или меньше.
На сетевое программирование с языка C# мы используем различные IDE

.NET	MonoDevelop SharpDevelop Visual Studio Xamarin Studio

Лабораторная работа №2

Средства программирования сети и ознакомления с основными задачами

Цель работы: Языки программирования и ознакомления основными функциями. Классификация языков программирования.

1. Практически часть

Эксплорер в этом упражнении.

2) найти иконку wireshark на рабочем столе и дважды щелкните его, чтобы начать с помощью wireshark.

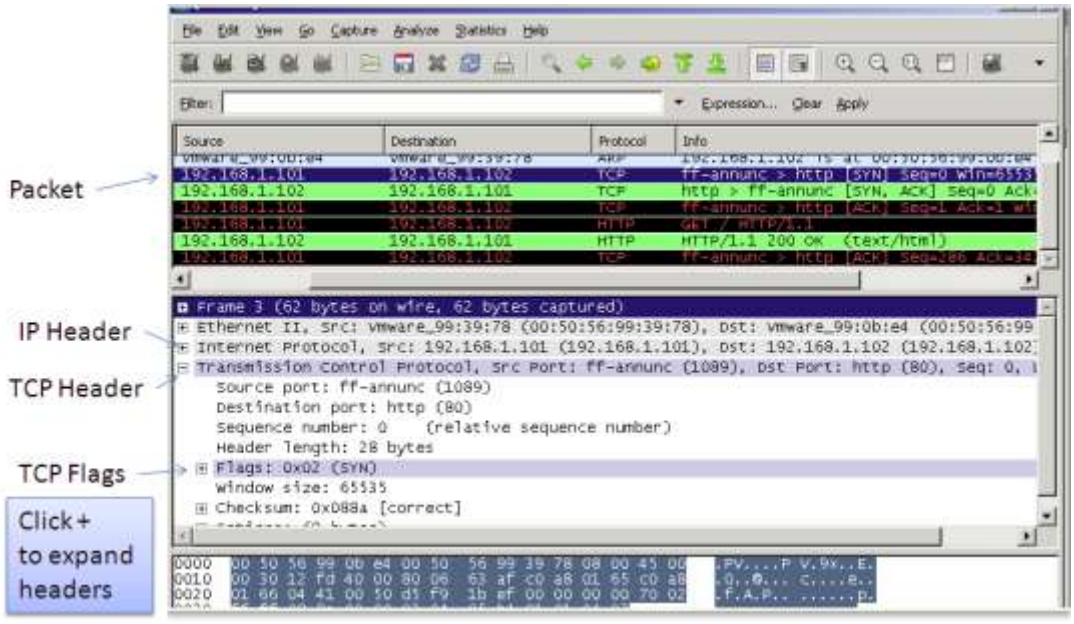
3) В меню съемки выберите Интерфейсы.

4) Нажмите на старт для VMware ускоренного PCNET АДМ адаптер

5) Вы будете подключаться к сайту печатных плат для создания сетевого трафика. В окне URL-адрес в Интернете проводнике, Тип “http://192.168.1.102” и нажмите Enter.

6) Как можно быстрее остановить захват пакетов. Вернуться в wireshark, и из меню записи выберите стоп, чтобы остановить захват пакетов. Затем, посмотрите на содержание перехваченных пакетов.

В меню wireshark выберите "анализ", а затем выберите фильтр просмотра. В фильтр Windows, найти АРП, и щелкните по нему и нажмите кнопку применить. Таким образом, Вы сможете отфильтровать все пакеты ARP. В пакете заголовки окне нажмите кнопку +, чтобы развернуть содержимое отдельных заголовков (см. рисунок ниже) и заполнить следующую таблицу.



Пакет	Источник IP	Порт Источника	IP-адрес назначения	Протокол	Флаг TCP	Последовательность

Найти первый протокол http от в 192.168.1.101 192.168.1.102. Этого пакета полезной нагрузки должен быть контент сайта. Вы можете увидеть содержимое сайта?

2. Вопросы

1. Основные функции анализатора программ
2. Опишите слои сетевого протокола в пакеты
3. Порты TCP приложения
4. Что такое wireshark?
5. Основные сетевые протоколы?
6. Что такое хорошо известные порты и почему мы их используем?

3. Теоретическое сведение

Интернет-протокола (IP) лежит в основе сетевого программирования. IP-это транспортное средство, которое перевозит данными между системами, будь то в локальной сети (LAN) или глобальной сети (WAN) среды. Хотя существуют и другие сетевые протоколы, доступные в сети Windows программист, IP обеспечивает наиболее надежную технологию для передачи данных между сетевыми устройствами, особенно если они находятся в Интернете.

Программирование с использованием IP-это зачастую сложный процесс. Есть много факторов, чтобы рассмотреть о том, как данные передаются по сети: количество клиентских и серверных устройств, Тип сети, перегрузка сети, и ошибки в сети. Поскольку все эти элементы влияют на транспортировку данных от одного устройства к другому, понимание их влияния имеет

решающее значение для вашего успеха в сетевом программировании. Часто то, что вы думаете, что происходит в ваших программах не происходит в сети. В этом уроке описан способ просмотра IP-трафика на сети, чтобы помочь вам в отладки сетевых программ.

На этом уроке также рассматривается частей IP-коммуникаций нужно Ваше понимание сетевых коммуникаций, наряду с внутренними работами двух наиболее популярных протоколов, которые используют IP: протоколом управления передачей (TCP) и протокол пользовательских Датаграмм (udp). Наконец, одним из самых запутанных вопросов в сети - это сети IP-адресации. Эта заканчивает с полезным взглянуть на системах Microsoft Windows управление IP-адресов, и как можно программно определить конфигурацию IP-адрес системы, на которой работает ваша программа.

Одна из самых больших трудностей для сетевых программистов не в состоянии видеть точно, что происходит в сети. Часто вы думаете, что вы создали идеальный клиент/сервер приложений, проводя Часы работы механику отправки и получения данных приложения между машинами, только чтобы узнать, что где-то по сети ваши данные не доходят. Сетевой анализатор может быть сетевой программист с лучшим другом в том, что происходит на проводе. Ошибки программирования часто могут быть обнаружены путем просмотра данных приложений переходят от одного устройства к другому.

Существует четыре основных функции, программа анализатор может выполнять:

- Захват и отображение сетевых пакетов;
- Отображение пакетов, хранящихся в файле;
- Захват сетевой статистики;
- Выполнить в режиме реального времени мониторинг сети.

Возможность смотреть IP-сессий и расшифровать его значение является важным навыком для сетевого программиста. Для полного понимания концепции сетевого программирования, вы должны сначала понять IP-протокол и как он перемещает данные между сетевыми устройствами. Знакомство с этой информацией может спасти вас бесчисленные часы сетевые программы диагностики, которые не ведут себя так, как вы думаете, что они должны.

Сетевые пакеты, которые вы захватываете содержать несколько информационных слоев, чтобы помочь вам транспортировать данные между двумя сетевыми устройствами. Каждый слой информации, содержащий байты организованы в заранее определенном порядке, с указанием параметров, специфичных для протокола. Большинство пакетов, которые вы

будете использовать в сетевое программирование IP будет состоять из трех отдельных слоев протокола информации, наряду с собственно передаваемых данных между сетевыми устройствами.
Рисунок 2.1 иллюстрирует иерархию протоколов, используемых для IP-пакетов.

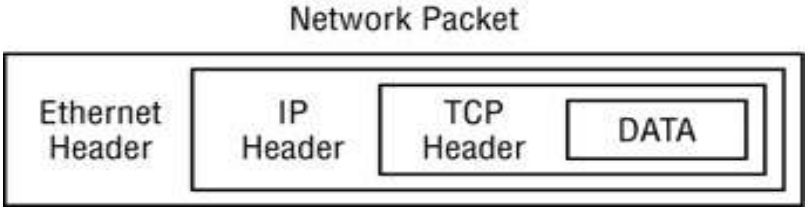


Рисунок 2.1 слоев сетевого протокола в пакеты

Первый слой сетевого пакета показано в захвате анализатор называется заголовку Ethernet. Вы можете увидеть три вида пакетов протокола Ethernet на сети: Ethernet 802.2, Ethernet и 802.3 и Ethernet версии 2.
В 802.2 Ethernet и протоколы 802.3 стандартных протоколов IEEE определенными для уровня Ethernet трафика. Ethernet-версия 2-это устаревший протокол, который не является стандартным протоколом, как таковой, но это наиболее распространенный протокол, используемый в сетях Ethernet. Почти все устройства (в том числе систем Windows) с помощью Ethernet-версия протокола 2 для передачи IP-пакетов по умолчанию.
Следующий слой протокола при взятии анализатор пакетов IP-пакетов. Пакет детали рамка показывает в разделе IP слой захвата анализатора для конкретного пакета. Как видите, IP-протокол определяет несколько дополнительных полей информации протокола Ethernet.
Значение каждого поля можно увидеть в анализаторе пакетов детали. Полей, содержащих более одного типа информации, такой как Тип сервиса, и поля флагов, могут быть расширены, чтобы показать все значения в поле.
Протокол управления передачей (TCP) добавляет информацию о подключении к пакету данных. Это позволяет программы для создания сквозного соединения между двумя сетевыми устройствами, обеспечивая последовательный путь для передачи данных. Протокол TCP гарантирует, что данные будут надежно будет поставляться с устройством назначения или что Отправитель получит признак того, что произошла сетевая ошибка.
Из-за этой особенности протокола TCP называется протоколом, ориентированным на соединения. Каждое TCP-соединение, или сессии, включает в себя некоторое количество пакетов накладные расходы, связанные с установление связи между двумя устройствами. Как только соединение установлено, данные могут быть отправлены между

устройствами без необходимости проверять за потерянные или неуместными данных.
Для сетевого программиста, это хорошая идея, чтобы иметь базовое понимание принципов TCP работает, и особенно то, как он перемещает данные между устройствами в сети.

Приложения портов TCP

В IANA определен список стандартных портов TCP назначенных для конкретных приложений. Это гарантирует, что любой хост, на котором выполняется определенное приложение будет принимать соединения на TCP-порт для этого приложения. В таблице 2.2 представлен частичный список из многих разных назначенные номера портов приложений.

Таблица 2.2 порты TCP для приложений

Описание	Порт
7	Эхо
13	Днем
17	Цитата дня
20	Протокол FTP (канал передачи данных)
21	Протокол FTP (канал управления)
22	СШ
23	Телнет
25	Протокол SMTP
37	Время
80	Протокол http

Порты с номерами от 0 до 1023 называются хорошо известные порты, потому что они относятся к конкретным, общие приложения. Если вы создаете новое приложение, старайтесь не ставить его в один из этих портов, чтобы избежать путаницы. В самом деле, если приложение уже использует порт, система не позволит вам использовать этот порт. Порты с номерами от 1024 до 65535 открыты для использования любым приложением, даже в этом широком ассортименте, это еще можно нарваться на двух приложениях, которые решили использовать тот же порт. По этой причине, множество

пользовательских приложений, предоставить пользователям выбор, определяя свой собственный номер порта.

Для сетевого программиста, TCP-порты являются важнейшими элементами. Все серверные приложения должны использовать действительный (и доступный) порту на сетевом устройстве. Это до вас, чтобы решить, как использовать назначение TCP-порт для вашего приложения. Как уже упоминалось, это всегда хорошая идея, чтобы дать вашему клиенту возможность изменить номер порта TCP приложения. Просто помните, что если номер порта сервера измененияx, клиентское приложение должно знать об этом изменении, или он никогда не будет найти приложение-сервер.

Analyzing network packets

Для сетевых программистов важно знать сетевые протоколы, форматы протокола, номера портов и другие программируемые вещи. Из-за них мы учимся анализировать сетевые пакеты. Для этой цели мы используем сетевого анализатора wireshark.

Wireshark-это анализатор протоколов, или "анализатор пакетов" приложения, используемые для диагностики, анализа, программного обеспечения и развития протокола, и образования. Это позволяет пользователю видеть весь трафик передается по сети сетевой карты в режиме promiscuous.

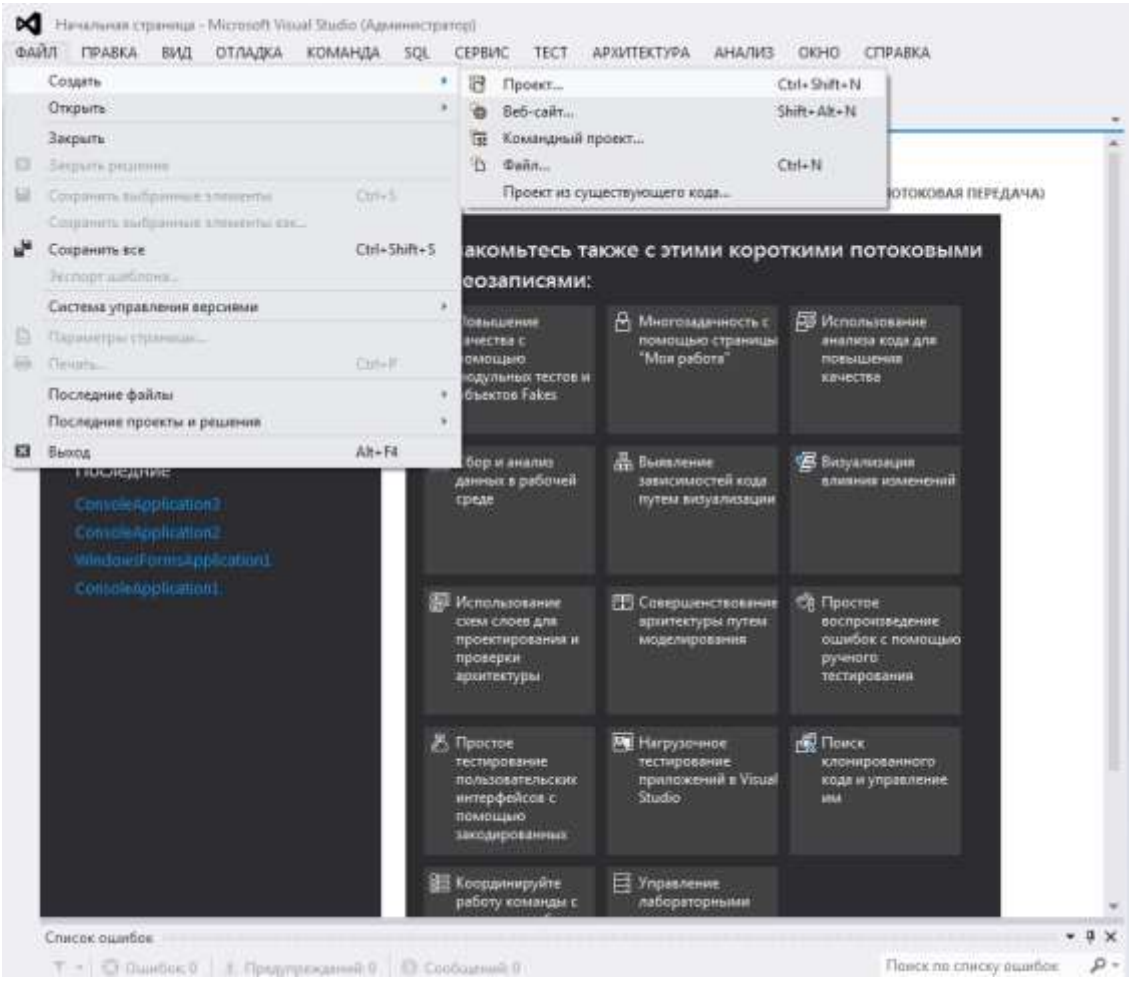
Лабораторная работа №3

Создания сетевой программы TCP клиет

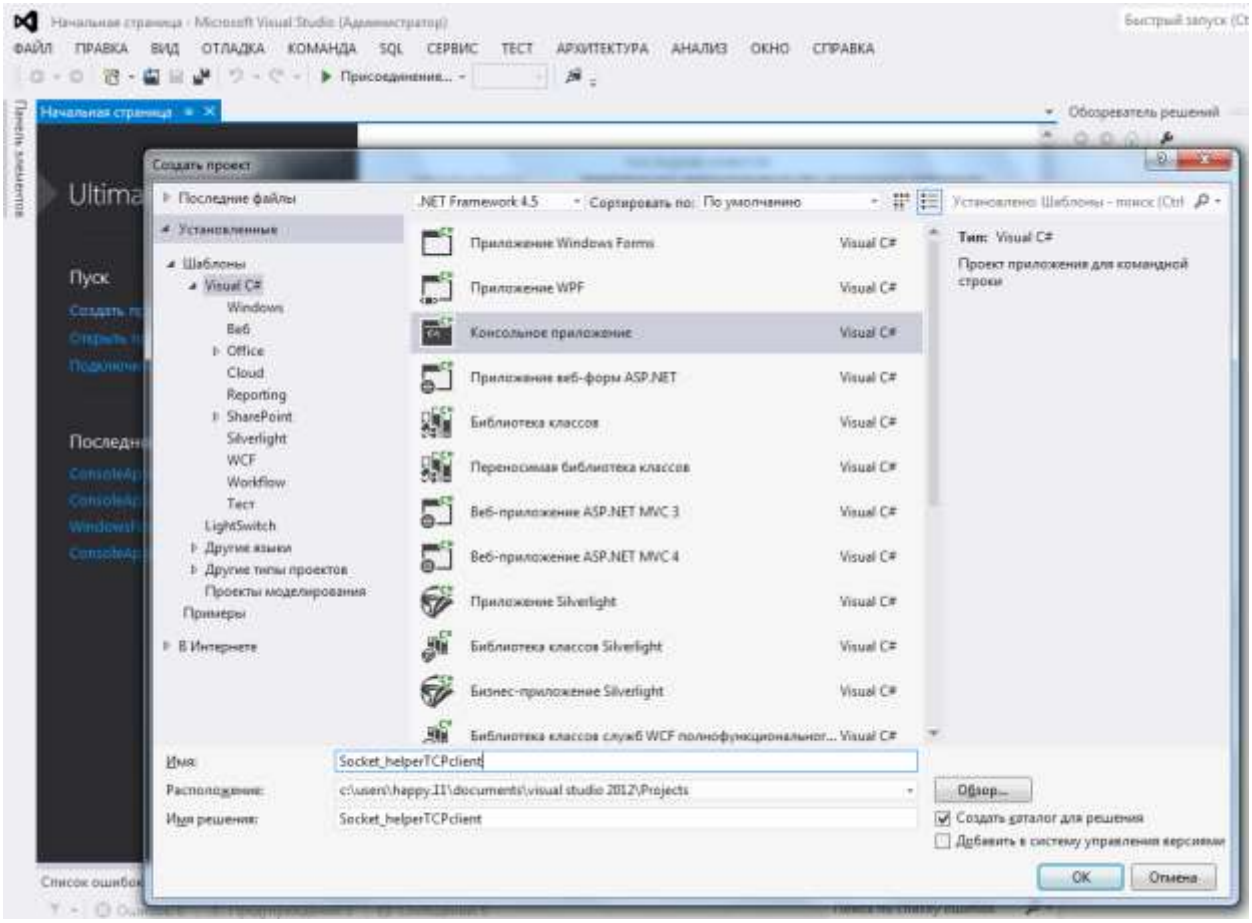
Цель занятия: На основе протокола TCP проектирования и разработать программу сетевого клиента

1. Практическая часть

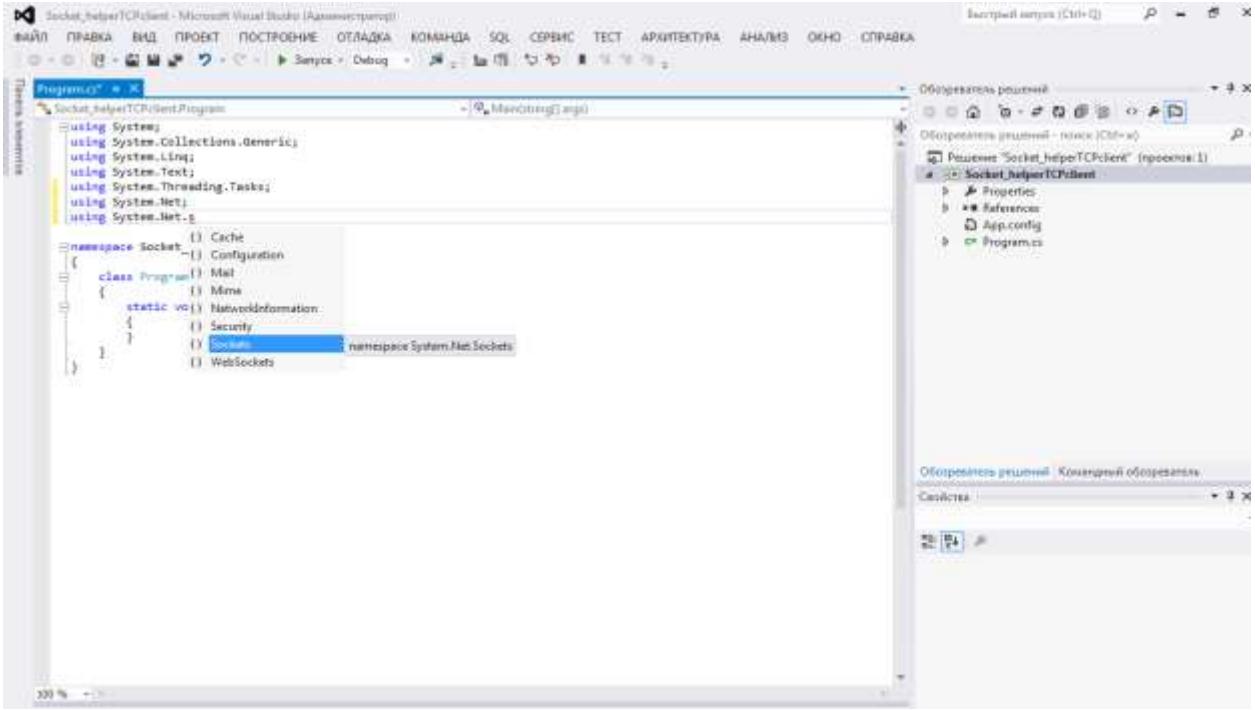
Создать новый проект:



Название проекта, например “Helper_“объект tcpclient” и нажмите “ОК”:



Добавить System.Net" and "System. Net.Sockets" .NET Framework пространства имен класса для использования сетевых функций системы



После этого, вы можете написать свой исходный код

// SocketClient.cs

```
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();
        }
    }
}
```

На этой работе вы должны:

- писать программу для Вашего варианта;
- дать краткие сведения о специальный класс и метод, который используется в вашей программе.
- Написать отчет для Вашего варианта этой работы.

2. Вопросы

1. Ориентированный на соединение классов гнездо
2. TCP и основных сетевых функций
3. Функции и методы по TCP
4. TCP, отправленных способ
5. Способ получения TCP
6. Сокеты TCP в сетевом программировании

3. Теоретическая часть

NET Framework поддерживает нормальный интерфейс гнездо для опытных сетевых программистов, но также обеспечивает упрощенный интерфейс для более легкого сетевого программирования. Упрощенный помощник гнездо классы помогают сетевым программистам создать программы сокета с более простыми заявлениями и меньше кодирования два важных преимущества для любого программиста. Этот раздел знакомит c# помощник гнездо классы.
Вот три вспомогательные классы, используемые для программирования сокета:

- Tcpclient;
- Tcplistener ;
- Udpclient.

Каждый из них предназначен для поддержки конкретной функции программирования сокетов и упростить интерфейсы, необходимые для программы для этой функции. Очевидно, что tcpclient и tcplistener с классы используются для создания программ TCP клиента и сервера; класса udpclient используется для создания udp программы.
Объект tcpclient

Методы класса tcpclient используются для создания сетевых программ клиента, которые следуют за подключение-ориентированной сетевой модели. Методы tcpclient зеркало в нормальном программирование сокетов, но многие шаги уплотняются, чтобы упростить задачу программирования.
Для начала, существует три способа создания объекта tcpclient и подключить его к удаленному узлу, и каждый метод упрощает создание сокета, чем при использовании ручного патрона методов класса.
Конструктор по умолчанию конструктор Формат создает сокет на любой доступный локальный порт по умолчанию. Затем вы можете использовать метод Connect() для подключения к указанному удаленному узлу:

```
TcpClient newclient = new TcpClient();
newclient.Connect("www.isp.net", 8000);
```

Первый оператор создает новый объект tcpclient и привязывает его к локальному адресу и порту. Второе утверждение связывает сокет с удаленным адресом узла и номер порта. Обратите внимание, что удаленный хост-адрес можно указать в качестве имени хоста. Класс tcpclient будет автоматически пытаться разрешить имя хоста в правильный IP-адрес. Что много работы уже сделано за вас!
Конкретную EndPoint объекта второй конструктор Формат идет на один шаг дальше и позволяет указать конкретный локальный объект конечную точку для использования при создании сокета:

```
IPAddress ia = Dns.GetHostByName(
Dns.GetHostName()).AddressList[0];
IPEndPoint iep = new IPEndPoint(ia, 10232);
TcpClient newclient2 = new TcpClient(iep);
newclient2.Connect("www.isp.net", 8000);
```

A specific remote host Третий Формат конструктор является наиболее распространенным. Он позволяет указать удаленный адрес хоста и порт для подключения к внутри конструктора, устраняя необходимость использования метода Connect ():

```
TcpClient newclient3 = new TcpClient("www.isp.net", 8000);
```

В один простой шаг, вы создаете новый объект объект tcpclient на случайный номер локального порта и подключить его к указанному удаленному узлу. Опять же, если вы используете имя хоста, адрес хоста, объект tcpclient будет пытаться решить ее автоматически. Создание сокета и подключение к удаленному узлу с одним заявлением, что такое система!

После создания экземпляра объект tcpclient, вы, вероятно, хотите, чтобы отправлять и получать данные с него. Метод getstream() метод используется для создания networkstream объект, который позволяет отправлять и получать байты на сокете. Если у вас есть объекте networkstream экземпляр для сокета, это совсем несложно использовать стандартный поток чтения() и write() для перемещения данных в и из розетки. Этот фрагмент кода демонстрирует назначение networkstream объект в экземпляр tcpclient и записи данных и чтения из сокета:

```
TcpClient newclient = new TcpClient("www.isp.net", 8000);

NetworkStream ns = newclient.GetStream();

byte[] outbytes = Encoding.ASCII.GetBytes("Testing");

ns.Write(outbytes, 0, outbytes.Length);

byte[] inbytes = new byte[1024];

ns.Read(inbytes, 0, inbytes.Length);

string instring = Encoding.ASCII.GetString(inbytes);

Console.WriteLine(instring);

ns.Close();

newclient.Close();
```

Лабораторная работа №4

Создания сетевой программы TCP server

Цель работы: На основе протокола TCP проектирования и разработать программу сетевого сервера

1. Прктическая чать

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

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

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

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

                while (true)
                {
                    Console.Write(userName + ": ");
                    // ввод сообщения
                    string message = Console.ReadLine();
                    message = String.Format("{0}: {1}", userName, message);
                    // преобразуем сообщение в массив байтов
```

```
            byte[] data = Encoding.Unicode.GetBytes(message);
            // отправка сообщения
            stream.Write(data, 0, data.Length);

            // получаем ответ
            data = new byte[64]; // буфер для получаемых данных
            StringBuilder builder = new StringBuilder();
            int bytes = 0;
            do
            {
               bytes = stream.Read(data, 0, data.Length);
               builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
            }
            while (stream.DataAvailable);

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

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

После отправки сообщения клиент получает сообщение с сервера.

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

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

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

namespace ConsoleServer
{
   class Program
   {
      const int port = 8888;
      static TcpListener listener;
      static void Main(string[] args)
      {
         try
         {
            listener = new TcpListener(IPAddress.Parse("127.0.0.1"), port);
            listener.Start();
            Console.WriteLine("Ожидание подключений...");

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

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

Сразу после подключения нового клиента:
TcpClient client = listener.AcceptTcpClient()
Создается объект ClientObject и новый поток, который запускает метод Process объекта ClientObject, где собственно и происходит получение и отправка сообщений. Таким образом, сервер сможет одновременно обрабатывать сразу несколько запросов.
Результаты работы программы. Один из клиентов
Единственный новый метод - метод **Connect()**, используется для соединения с удаленным сервером.

2. Вопросы

1. Что такое TCP
2. Что такое ClientObject
3. Дайте объяснение понятии tcpcrypt.
4. Дайте объяснение понятии **CLOSE-WAIT**

3. Теоретическое сведение

Механизм TCP предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета, гарантируя тем самым, в отличие от UDP, целостность передаваемых данных и уведомление отправителя о результатах передачи.
Реализации TCP обычно встроены в ядра ОС. Существуют реализации TCP, работающие в пространстве пользователя.
Когда осуществляется передача от компьютера к компьютеру через Интернет, TCP работает на верхнем уровне между двумя конечными системами, например, браузером и веб-сервером. TCP осуществляет надежную передачу потока байтов от одной программы на некотором компьютере к другой программе на другом компьютере (например, программы для электронной почты, для обмена файлами). TCP контролирует длину сообщения, скорость обмена сообщениями, сетевой трафик.

Механизм действия протоколаВ отличие от традиционной альтернативы — UDP, который может сразу же начать передачу пакетов, TCP устанавливает соединения, которые должны быть созданы перед передачей данных. TCP соединение можно разделить на 3 стадии: Установка соединения

- Передача данных
- Завершение соединения

Состояния сеанса TCP

Упрощённая диаграмма состояний TCP. Более подробно в TCP EFSM diagram (на английском языке)

Состояния сеанса TCP	
CLOSED	Начальное состояние узла. Фактически фиктивное
LISTEN	Сервер ожидает запросов установления соединения от клиента
SYN-SENT	Клиент отправил запрос серверу на установление соединения и ожидает ответа
SYN-RECEIVED	Сервер получил запрос на соединение, отправил ответный запрос и ожидает подтверждения
ESTABLISHED	Соединение установлено, идёт передача данных
FIN-WAIT-1	Одна из сторон (назовём её узел-1) завершает соединение, отправив сегмент с флагом FIN
CLOSE-WAIT	Другая сторона (узел-2) переходит в это состояние, отправив, в свою очередь сегмент ACK и продолжает одностороннюю передачу
FIN-WAIT-2	Узел-1 получает ACK, продолжает чтение и ждёт получения сегмента с флагом FIN
LAST-ACK	Узел-2 заканчивает передачу и отправляет сегмент с флагом FIN
TIME-WAIT	Узел-1 получил сегмент с флагом FIN, отправил сегмент с флагом ACK и ждёт 2*MSL секунд, перед окончательным закрытием соединения
CLOSING	Обе стороны инициировали закрытие соединения одновременно: после отправки сегмента с флагом FIN узел-1 также получает сегмент FIN, отправляет ACK и находится в ожидании сегмента ACK (подтверждения на свой запрос о разъединении)

Установка соединения

Процесс начала сеанса TCP (также называемый «рукопожатие» (англ. *handshake*)), состоит из трёх шагов.

Клиент, который намеревается установить соединение, посылает серверу сегмент с номером последовательности и флагом SYN.

- Сервер получает сегмент, запоминает номер последовательности и пытается создать сокет (буферы и управляющие структуры памяти) для обслуживания нового клиента.

- В случае успеха сервер посылает клиенту сегмент с номером последовательности и флагами SYN и ACK, и переходит в состояние SYN-RECEIVED.

- В случае неудачи сервер посылает клиенту сегмент с флагом RST.

Если клиент получает сегмент с флагом SYN, то он запоминает номер последовательности и посылает сегмент с флагом ACK.

- Если он одновременно получает и флаг ACK (что обычно и происходит), то он переходит в состояние ESTABLISHED.
- Если клиент получает сегмент с флагом RST, то он прекращает попытки соединиться.
- Если клиент не получает ответа в течение 10 секунд, то он повторяет процесс соединения заново.

Если сервер в состоянии SYN-RECEIVED получает сегмент с флагом ACK, то он переходит в состояние ESTABLISHED.

- В противном случае после тайм-аута он закрывает сокет и переходит в состояние CLOSED.

Процесс называется «трёхэтапным согласованием» (англ. *three way handshake*), так как несмотря на то что возможен процесс установления соединения с использованием четырёх сегментов (SYN в сторону сервера, ACK в сторону клиента, SYN в сторону клиента, ACK в сторону сервера), на практике для экономии времени используется три сегмента.

Пример базового 3-этапного согласования:

TCP A TCP B

1. CLOSED LISTEN
2. SYN-SENT --> <SEQ=100><CTL=SYN> --> SYN-RECEIVED
3. ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK> <-- SYN-RECEIVED
4. ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK> --> ESTABLISHED
5. ESTABLISHED <-- <SEQ=301><ACK=101><CTL=ACK> <-- ESTABLISHED

В строке 2 TCP A начинает передачу сегмента SYN, говорящего об использовании номеров последовательности, начиная со 100. В строке 3 TCP B передает SYN и подтверждение для принятого SYN в адрес TCP A. Надо отметить, что поле подтверждения показывает ожидание TCP B приёма номера последовательности 101, подтверждающего SYN с номером 100.

В строке 4 TCP A отвечает пустым сегментом с подтверждением ACK для сегмента SYN от TCP B; в строке 5 TCP B передает некоторые данные. Отметим, что номер подтверждения сегмента в строке 5 (ACK=101) совпадает с номером последовательности в строке 4 (SEQ=101), поскольку

ACK не занимает пространства номеров последовательности (если это сделать, придется подтверждать подтверждения — ACK для ACK).

Существуют экспериментальные расширения протокола TCP, сокращающие количество пакетов при установлении соединения, например TCP Fast Open[en]. Ранее также существовало расширение T/TCP. Для прозрачного шифрования данных предлагается использовать расширение tcpcrypt.

Лабораторная работа №5

Создания сетевой программы TCP клиент сервер

Цель занятия: Установка и разработка клиент серверных отношении на основе протоколов TCP

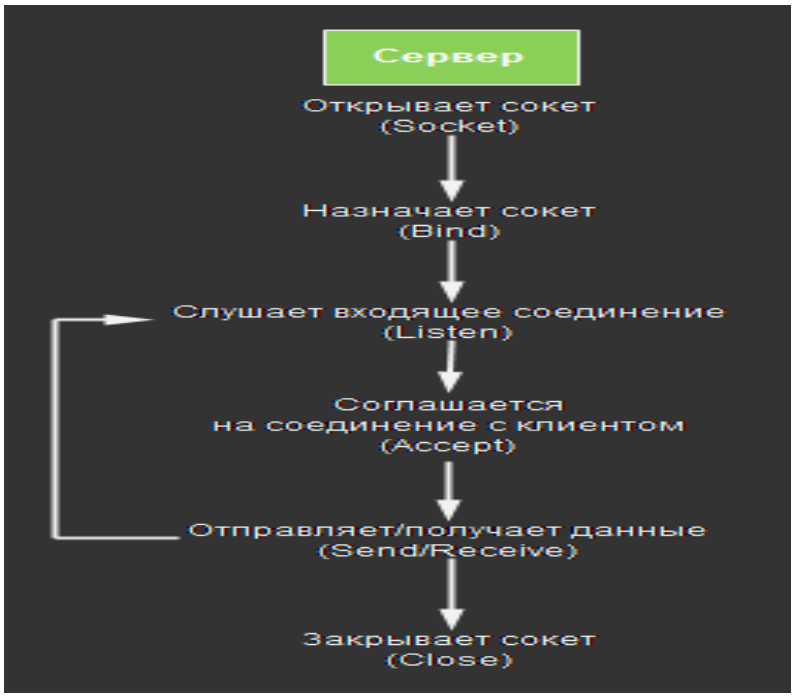
1. Практическая часть

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:



Вот полный код программы SocketServer.cs:

// SocketServer.cs

using System;

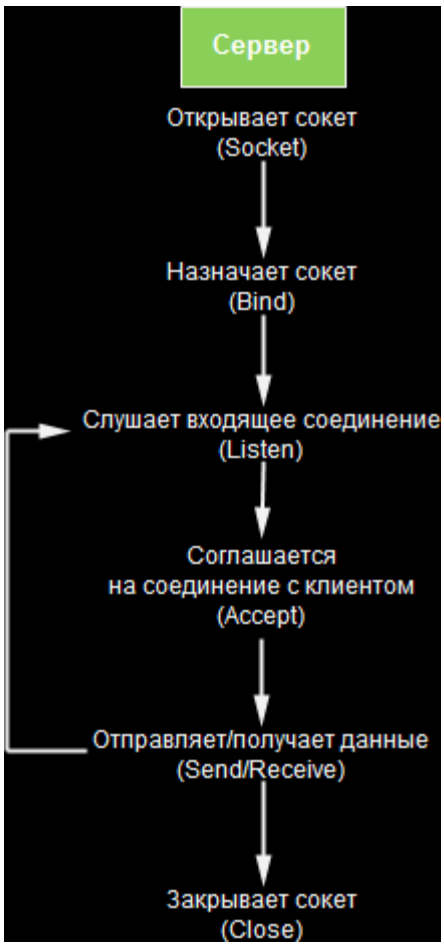
```
using System.Text;
using System.Net;
using System.Net.Sockets;

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();
      }
   }
}
```

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:



Вот полный код для SocketClient.cs и его объяснение:

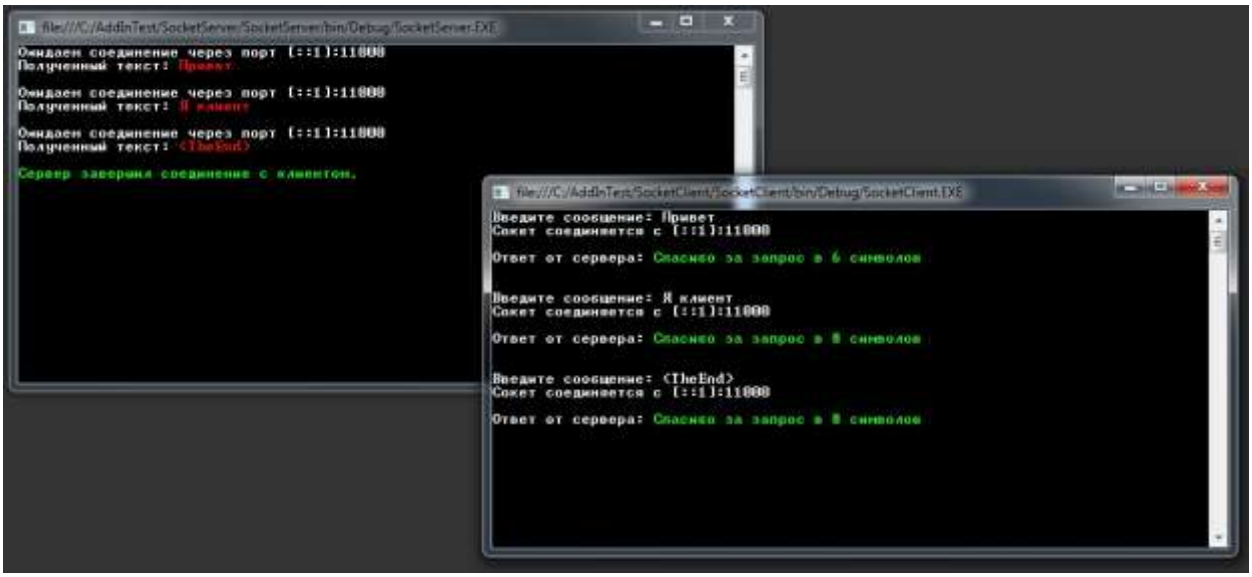
```
// SocketClient.cs
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

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();
        }
    }
}
```

На рисунке ниже показаны клиент и сервер в действии:



2. Вопросы

1. Что такое клиент серверная архитектура
2. В чем различия клиент серверных отношений на основе протоколов TCP от UDP

3. Теоретическая часть.

Клиент-серверное взаимодействие один из главных моментов в изучении дисциплины веб-дизайн. Любой сайт, размещённый в сети Интернет, основывается на связке «клиен-сервер». И это не только электронные формы. Даже простое «перелистывание» страниц некоторого сайта в Интернете – пример клиент-серверного взаимодействия, ведь странички хранятся не на вашем личном компьютере, а подгружаются извне.

Рассмотрим основные понятия, которые предполагает эта глава:

1. Клиент — достаточно широкое понятие, начиная от некоторого физического лица и заканчивая некоторой программой на компьютере (например, почтовый клиент). В нашем случае это компьютер, оснащённый специальным программным обеспечением, которое позволяет пользователю задать запрос к другой машине и получить ответ. Код, выполняемый на стороне клиента, чаще всего называют клиентским кодом. Он обеспечивает создание пользовательского интерфейса. Здесь, когда речь заходит о браузере, важную роль играет JavaScript и библиотеки расширения (в нашем случае jQuery).
2. Сервер — это компьютер, оснащённый специальным программным обеспечением, которое позволяет решить задачи предоставления пользователю доступа к некоторым услугам и ресурсам, которыми владеет и управляет данный сервер. Код, выполняемый на стороне сервера, чаще всего называют серверным кодом (серверным сценарием). Он обеспечивает обработку данных. Здесь важную роль играют серверные языки программирования. Примером таких языков являются PHP и Python.
3. Хранилище данных – это система, предназначенная для хранения данных на сервере. Чаще всего под хранилищем данным подразумевают базу данных (например, MySQL). Также хранилищем может выступать файловая система сервера.
4. Клиент-серверное взаимодействие — это обмен данными между клиентом и сервером.

Браузер не единственное программное обеспечение, которое позволяет осуществлять клиент-серверное взаимодействие. Почтовые клиенты, программы обмена мгновенными сообщениями, игровые многопользовательские приложения – пример программного обеспечения, которое осуществляет клиент-серверное взаимодействие.

Лабораторная работа №6
Создания сетевой программы UDP клиент.

Цель работы: На основе протокола UDP проектирования и разработать программу сетевого клиента

1. Практическая часть

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

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

UdpClient

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

Для создания подключения достаточно создать объект UdpClient и вызывать у него методConnect():

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Создадим примитивный консольный udp-чат:

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

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

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

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

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

```
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            sender.Close();
        }
    }

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

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

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

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

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

2. Вопросы

1. Что такое UdpClient?
2. Структура пакета это…

3. Теоретическая часть

UDP (англ. *User Datagram Protocol* — протокол пользовательских дейтаграмм) — один из ключевых элементов TCP/IP, набора сетевых протоколов для Интернета. С UDP компьютерные приложения могут посылать сообщения (в данном случае называемые датаграммами) другим хостам по IP-сети без необходимости предварительного сообщения для установки специальных каналов передачи или путей данных. Протокол был разработан Дэвидом П. Ридом в 1980 году и официально определён в RFC 768.

UDP использует простую модель передачи, без неявных «рукопожатий» для обеспечения надёжности, упорядочивания или целостности данных. Таким образом, UDP предоставляет ненадёжный сервис, и датаграммы могут прийти не по порядку, дублироваться или вовсе исчезнуть без следа. UDP подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении. Чувствительные ко времени приложения часто используют UDP, так как предпочтительнее сбросить пакеты, чем ждать задержавшиеся пакеты, что может оказаться невозможным в системах реального времени. При необходимости исправления ошибок на сетевом уровне интерфейса приложение может задействовать TCP или SCTP, разработанные для этой цели.

Природа UDP как протокола без сохранения состояния также полезна для серверов, отвечающих на небольшие запросы от огромного числа клиентов, например DNS и потоковые мультимедийные приложения вроде IPTV, Voice over IP, протоколы туннелирования IP и многие онлайн-игры.

Структура пакета UDP — минимальный ориентированный на обработку сообщений протокол транспортного уровня, задокументированный в RFC 768.UDP не предоставляет никаких гарантий доставки сообщения для вышестоящего протокола и не сохраняет состояния отправленных сообщений. По этой причине UDP иногда называют Unreliable Datagram Protocol (англ. — Ненадёжный протокол датаграмм).

UDP обеспечивает многоканальную передачу (с помощью номеров портов) и проверку целостности (с помощьюконтрольных сумм) заголовка и существенных данных. Надёжная передача в случае необходимости должна реализовываться пользовательским приложением.

Биты	0 – 15	16 - 31
0-31	Порт отправителя (Source port)	Порт получателя (Destination port)
32-63	Длина датаграммы (Length)	Контрольная сумма (Checksum)
64-...	Данные (Data)	

Заголовок UDP состоит из четырёх полей, каждое по 2 байта (16 бит). Два из них необязательны к использованию в IPv4 (розовые ячейки в таблице), в то время как в IPv6 необязателен только порт отправителя.

Порт отправителя: В этом поле указывается номер порта отправителя. Предполагается, что это значение задаёт порт, на который при необходимости будет посылаться ответ. В противном же случае, значение должно быть равным 0. Если хостом-источником является клиент, то номер порта будет, скорее всего, динамическим. Если источником является сервер, то его порт будет одним из «хорошо известных».

Порт получателя :Это поле обязательно и содержит порт получателя. Аналогично порту отправителя, если хостом-получателем является клиент, то номер порта динамический, если получатель — сервер, то это будет «хорошо известный» порт.

Длина датаграммы: Поле, задающее длину всей датаграммы (заголовка и данных) в байтах. Минимальная длина равна длине заголовка — 8 байт. Теоретически, максимальный размер поля — 65535 байт для UDP-датаграммы (8 байт на заголовок и 65527 на данные). Фактический предел для длины данных при использовании IPv4 — 65507 (помимо 8 байт на UDP-заголовок требуется ещё 20 на IP-заголовок). На практике также следует учитывать, что если длина IPv4 пакета с UDP будет превышать MTU (для Ethernet по умолчанию 1500 байт), то отправка такого пакета может вызвать его фрагментацию, что может привести к тому, что он вообще не сможет быть доставлен, если промежуточные маршрутизаторы или конечный хост не будут поддерживать фрагментированные IP пакеты. Также в RFC 791 указывается минимальная длина IP пакета 576 байт, которую должны поддерживать все участники, и рекомендуется отправлять IP пакеты большего размера только в том случае если вы уверены, что принимающая сторона может принять пакеты такого размера. Следовательно, чтобы избежать фрагментации UDP пакетов (и возможной их потери), размер данных в UDP не должен превышать: MTU — (Max IP Header Size) — (UDP Header Size) = 1500 — 60 — 8 = 1432 байт. Для того чтобы быть уверенным, что пакет будет принят любым хостом, размер

данных в UDP не должен превышать: (минимальная длина IP пакета) — (Max IP Header Size) — (UDP Header Size) = 576 — 60 — 8 = 508 байт.
В Jumbogram'max IPv6 пакеты UDP могут иметь больший размер. Максимальное значение составляет 4 294 967 295 байт (2^{32} — 1), из которых 8 байт соответствуют заголовку, а остальные 4 294 967 287 байт — данным.
Следует заметить, что большинство современных сетевых устройств отправляют и принимают пакеты IPv4 длиной до 10000 байт без их разделения на отдельные пакеты. Неофициально такие пакеты называют «Jumbo-пакетами», хотя понятие Jumbo официально относится к IPv6. Тем не менее, «Jumbo-пакеты» поддерживают не все устройства и перед организацией связи с помощью UDP/IP IPv4 посылок с длиной превышающей 1500 байт нужно проверять возможность такой связи опытным путём на конкретном оборудовании
Контрольная сумма: Поле контрольной суммы используется для проверки заголовка и данных на ошибки. Если сумма не сгенерирована передатчиком, то поле заполняется нулями. Поле не является обязательным для IPv4.

Лабораторная работа №7

Создания сетевой программы UDP server

Цель работы: На основе протокола **UDP** проектирования и разработать программу сетевого сервера

1. Практическая часть.

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 — 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.Text;
using System.Xml.Serialization;
using System.Diagnostics;
using System.Threading;

public 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();
}
```

Широковещательная передача

При выполнении этого приложения можно указать отдельный IP-адрес, по которому будет отправлен файл, но также можно задать широковещательный адрес, чтобы отправить файл всем машинам подсети или всей сети. Широковещательный адрес состоит из идентификатора подсети и единиц во всех остальных битах. Например, если хотим разослать сообщение всем хостам в диапазоне 192.168.0, то нужно использовать широковещательный адрес 192.168.0.255. Чтобы отправить сообщение всем машинам сети независимо от маски подсети, мы можем использовать адрес 255.255.255.255. При широковещательной передаче

сообщение отправляется всем машинам сети, а клиенту предстоит решить, хочет ли он обработать эти данные.

2. Вопросы

1. Файловый клиент это…
2. Псевдозаголовок для IPv4 и IPv6

3. Цель работы

Расчёт контрольной суммы

Метод для вычисления контрольной суммы определён в RFC 1071

Перед расчётом контрольной суммы, если длина UDP-сообщения в байтах нечётна, то UDP-сообщение дополняется в конце нулевым байтом (псевдозаголовок и добавочный нулевой байт не отправляются вместе с сообщением, они используются только при расчёте контрольной суммы). Поле контрольной суммы в UDP-заголовке во время расчёта контрольной суммы принимается нулевым.

Для расчёта контрольной суммы псевдозаголовок и UDP-сообщение разбивается на двухбайтные слова. Затем рассчитывается сумма всех слов в арифметике обратного кода (то есть кода, в котором отрицательное число получается из положительного инверсией всех разрядов числа и существует два нуля: 0x0000 (обозначается +0) и 0xffff (обозначается −0)). Результат записывается в соответствующее поле в UDP-заголовке.

Значение контрольной суммы, равное 0x0000 (+0 в обратном коде), зарезервировано и означает, что для посылки контрольная сумма не вычислялась. В случае, если контрольная сумма вычислялась и получилась равной 0x0000, то в поле контрольной суммы заносят значение 0xffff (-0 в обратном коде).

При получении сообщения получатель считает контрольную сумму заново (уже учитывая поле контрольной суммы), и, если в результате получится −0 (то есть 0xffff), то контрольная сумма считается сошедшейся. Если сумма не сходится (данные были повреждены при передаче, либо контрольная сумма неверно посчитана на передающей стороне), то решение о дальнейших действиях принимает принимающая сторона. Как правило, в большинстве современных устройств, работающих с UDP/IP-пакетами имеются настройки, позволяющие либо игнорировать такие пакеты, либо пропускать их на дальнейшую обработку, невзирая на неправильность контрольной суммы.

Пример расчёта контрольной суммы

Для примера рассчитаем контрольную сумму нескольких 16-битных слов: `0x398a, 0xf802, 0x14b2, 0xc281`.

Для этого можно сначала сложить попарно числа, рассматривая их как 16-разрядные беззнаковые числа с последующим приведением к дополнительному коду путём прибавления единицы к результату, если при сложении произошёл перенос в старший (17-й) разряд (то есть де-факто, этой операцией мы переводим отрицательное число из дополнительного в обратный код). Или, что равноценно, можно считать, что перенос прибавляется к младшему разряду числа.

```
0x398a + 0xf802 = 0x1318c → 0x318d (перенос в старший разряд)
0x318d + 0x14b2 = 0x0463f → 0x463f (число положительное)
0x463f + 0xc281 = 0x108c0 → 0x08c1
```

В конце выполняется инверсия всех битов получившегося числа

`0x08c1 = 0000 1000 1100 0001 → 1111 0111 0011 1110 = 0xf73e` или, иначе — `0xffff − 0x08c1 = 0xf73e`. Это и есть искомая контрольная сумма.

В документе RFC 1071 приведены и другие способы расчёта контрольной суммы, в частности, с использованием 32х-разрядной арифметики.

Псевдозаголовок для IPv4

Если UDP работает над IPv4, контрольная сумма вычисляется при помощи псевдозаголовка, который содержит некоторую информацию из заголовка IPv4. Псевдозаголовок не является настоящим IPv4-заголовком, используемым для отправления IP-пакета. В таблице приведён псевдозаголовок, используемый только для вычисления контрольной суммы.

Биты	**0 — 7**	**8 — 15**	**16 — 23**	**24 — 31**
0	Адрес источника			
32	Адрес получателя			
64	Нули	Протокол	Длина UDP	
96	Порт источника		Порт получателя	
128	Длина		Контрольная сумма	
160+	Данные			

Адреса источника и получателя берутся из IPv4-заголовка. Значения поля «Протокол» для UDP равно 17 (0x11). Поле «Длина UDP» соответствует длине заголовка и данных.

Вычисление контрольной суммы для IPv4 необязательно, если она не используется, то значение равно 0.

Псевдозаголовок для IPv6

При работе UDP над IPv6 контрольная сумма обязательна. Метод для её вычисления был опубликован в RFC 2460:

При вычислении контрольной суммы опять используется псевдозаголовок, имитирующий реальный IPv6-заголовок:

<table>
<tr><th>Биты</th><th>0 — 7</th><th>8 — 15</th><th>16 — 23</th><th>24 — 31</th></tr>
<tr><td>0</td><td colspan="4" rowspan="4">Адрес источника</td></tr>
<tr><td>32</td></tr>
<tr><td>64</td></tr>
<tr><td>96</td></tr>
<tr><td>128</td><td colspan="4" rowspan="4">Адрес получателя</td></tr>
<tr><td>160</td></tr>
<tr><td>192</td></tr>
<tr><td>224</td></tr>
<tr><td>256</td><td colspan="4">Длина UDP</td></tr>
<tr><td>288</td><td colspan="3">Нули</td><td>Следующий заголовок</td></tr>
<tr><td>320</td><td colspan="2">Порт источника</td><td colspan="2">Порт получателя</td></tr>
<tr><td>352</td><td colspan="2">Длина</td><td colspan="2">Контрольная сумма</td></tr>
<tr><td>384+</td><td colspan="4">Данные</td></tr>
</table>

Адрес источника такой же, как и в IPv6-заголовке. Адрес получателя — финальный получатель; если в IPv6-пакете не содержится заголовка маршрутизации (Routing), то это будет адрес получателя из IPv6-заголовка, в противном случае, на начальном узле, это будет адрес последнего элемента заголовка маршрутизации, а на узле-получателе — адрес получателя из IPv6-заголовка. Значение «Следующий заголовок» равно значению протокола — 17 для UDP. Длина UDP — длина UDP-заголовка и данных.

Лабораторная работа №8

Создания сетевой программы UDP клиент сервер

Цель занятия: Установка и разработка клиент серверных отношении на основе протоколов UDP

1. Практическая чать

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 A и как приемный порт для хоста B, 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.

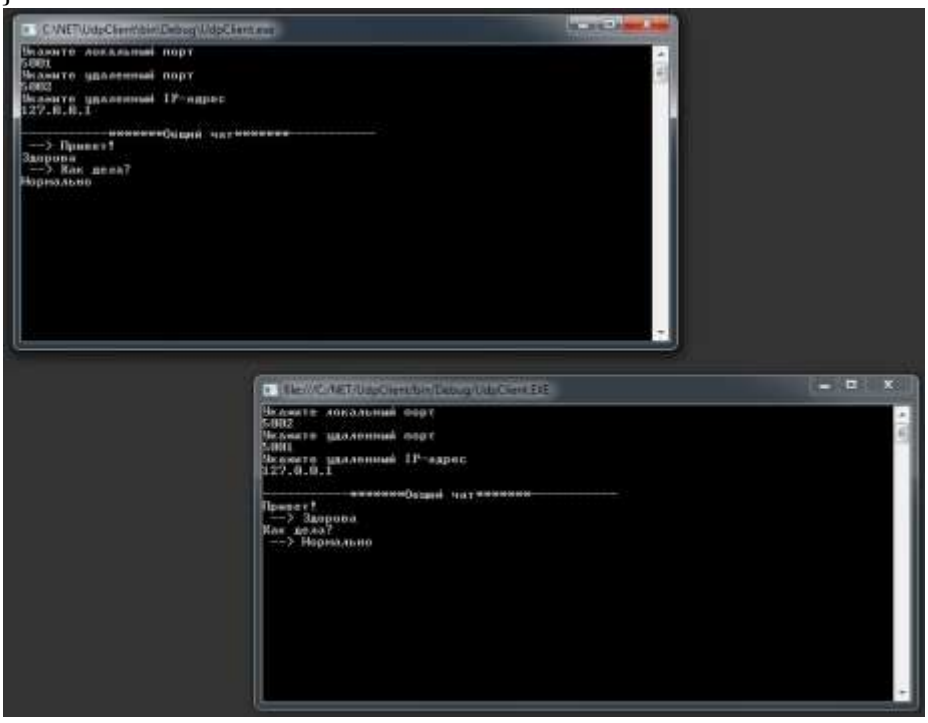
Третий логический блок в приложении принимает данные, введенные пользователем, и отправляет их указанному удаленному порту. Он выполняется на основном потоке, пока рабочий поток продолжает слушать входящие данные.

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);
            }
        }
    }
}
```



2. Теоретическая часть.

Протокол UDP

C# и .NET --- Сетевое программирование --- 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:

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
161	**SMTP**

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

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 подтвержден

Лабораторная работа №9
Использование групповых соккет в программировании сети

Цель работы: Принципы структуры клиентских программ. Принципы структуры серверных программ.

1. Практическая часть

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>.

Вот полный код программы SocketServer.cs:

```
// SocketServer.cs
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

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();
            }
        }
    }
}
```

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:

Вот полный код программы SocketClient.cs

```
// SocketClient.cs
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();
        }
    }
}
```

2. Теоретическая часть

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.

Лабораторная работа №10
Использования мультикастинговых соккетов в программировании сети

Цель работы: Создания клиент серверной, серверной и клиентской программы на основе мультикастинговых соккетов. Занчения мультикастинговых соккетов в программировании сети

1. Практическая часть

C#

```
public void JoinMulticastGroup(
    IPAddress multicastAddr
)
```

Параметры
multicastAddr

Type: System.Net.IPAddress

Многоадресная рассылка IPAddress группы нужно соединить.

Исключения

Exception	**Condition**
ObjectDisposedException	Базовый Socket был закрыт.
SocketException	Произошла ошибка при обращении к сокету. Дополнительные сведения см. в разделе "Примечания".
ArgumentException	IP-адрес не совместим с AddressFamily значение, которое определяет схему адресации сокета.

JoinMulticastGroup Метод подписывается UdpClient к группе многоадресной рассылки, используя указанный IPAddress. После вызова метода JoinMulticastGroup базовый метод Socket отправляет пакет Internet Group Management Protocol (IGMP) маршрутизатора, запросившего членство в группе многоадресной рассылки. Диапазон адресов многоадресной рассылки — от 224.0.0.0 до 239.255.255.255. При указании адреса за пределами этого диапазона или маршрутизатора, по которому производится запрос не многоадресной рассылки включен, UdpClient вызовет SocketException. Если вы получилиSocketException, использовать SocketException.ErrorCode для получения определенного кода ошибки. Получив этот код, можно ссылаться на подробное описание ошибки документации по кодам ошибок API Windows Sockets версии 2 в библиотеке MSDN.Один раз UdpClient указан с

маршрутизатора в качестве члена группы многоадресной рассылки, будут иметь возможность приема групповых датаграмм, посылаемых по указанному IPAddress.

Примечание
Необходимо создать UdpClient используя номер группового порта; в противном случае — не появится возможность приема групповых датаграмм. Не следует вызывать Connect метод до вызова метода JoinMulticastGroup метода или Receiveметод не будет работать. Необходимо входить в группу многоадресной рассылки для отправки датаграмм многоадресной IP-адрес.

Перед присоединением к группе многоадресной рассылки, убедитесь, что сокет связан с портом или конечной точки. Для этого вызова одного из конструкторов, принимающих порт или конечную точку как параметр.

Чтобы прекратить прием групповых датаграмм, вызовите DropMulticastGroup метод и предоставить IPAddress группы, из которой вы хотите отказаться.

Примечание
В случае IPv6 имеется несколько диапазонов адресов многоадресной рассылки, выбранные из. Обратитесь к описание см.

Примечание
Не удается вызвать JoinMulticastGroup на UdpClient созданных без определенного локального порта (то есть с использованием UdpClient() или UdpClient(AddressFamily) конструктор).

Примеры

В следующем примере кода показано, как для присоединения к группе многоадресной рассылки, предоставляя адрес многоадресной рассылки.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.IO;
using System.Threading;
namespace Mssc.TransportProtocols.Utilities
{
```

```
// The following Receive class is used by both the ClientOriginator and
// the ClientTarget class to receive data from one another..
public class Receive
{
  // The following static method performs the actual data
  // exchange. In particular, it performs the following tasks:
  // 1)Establishes a communication endpoint.
  // 2)Receive data through this end point on behalf of the
  // caller.
  // 3) Returns the received data in ASCII format.
  public static string ReceiveUntilStop(UdpClient c)
  {
     String strData = "";
     String Ret = "";
     ASCIIEncoding ASCII = new ASCIIEncoding();

     // Establish the communication endpoint.
     IPEndPoint endpoint = new IPEndPoint(IPAddress.IPv6Any, 50);

     while (!strData.Equals("Over"))
     {
       Byte[] data = c.Receive(ref endpoint);
       strData = ASCII.GetString(data);
       Ret += strData  + "\n";
     }
     return Ret;
  }
}

// The following Send class is used by both the ClientOriginator and
// ClientTarget classes to send data to one another.
public class Send
{
  private static char[] greetings = { 'H', 'e', 'l', 'l', 'o', ' ',
                                      'T', 'a', 'r', 'g', 'e', 't', '.' };
  private static char[] nice      = { 'H', 'a', 'v', 'e', ' ', 'a', ' ', 'n', 'i',
                                      'c', 'e', ' ', 'd', 'a', 'y', '.' };
  private static char [] eom      = { 'O', 'v', 'e', 'r'};

  private static char[] tGreetings = { 'H', 'e', 'l', 'l', 'o', ' ',
                                       'O', 'r', 'i', 'g', 'i', 'n', 'a', 't', 'o', 'r', '!' };
  private static char[] tNice  = { 'Y', 'o', 'u', ' ', 't', 'o', 'o', '.'};

  // The following static method sends data to the ClientTarget on
  // behalf of the ClientOriginator.
```

```
  public static void OriginatorSendData(UdpClient c, IPEndPoint ep)
  {
    Console.WriteLine(new string(greetings));
    c.Send(GetByteArray(greetings), greetings.Length, ep);
    Thread.Sleep(1000);
    Console.WriteLine(new String(nice));
    c.Send(GetByteArray(nice), nice.Length, ep);
    Thread.Sleep(1000);
    Console.WriteLine(new String(eom));
    c.Send(GetByteArray(eom), eom.Length, ep);
  }
  // The following static method sends data to the ClientOriginator on
  // behalf of the ClientTarget.
  public static void TargetSendData(UdpClient c, IPEndPoint ep)
  {
    Console.WriteLine(new string(tGreetings));
    c.Send(GetByteArray(tGreetings), tGreetings.Length, ep);
    Thread.Sleep(1000);
    Console.WriteLine(new String(tNice));
    c.Send(GetByteArray(tNice), tNice.Length, ep);
    Thread.Sleep(1000);
    Console.WriteLine(new String(eom));
    c.Send(GetByteArray(eom), eom.Length, ep);
  }
  // Internal utility
  private static Byte[] GetByteArray(Char[] ChArray)
  {
    Byte[] Ret = new Byte[ChArray.Length];
    for (int i = 0; i < ChArray.Length; i++)
      Ret[i] = (Byte) ChArray[i];
    return Ret;
  }
}


// The ClientTarget class is the receiver of the ClientOriginator
// messages. The StartMulticastConversation method contains the
// logic for exchanging data between the ClientTarget and its
// counterpart ClientOriginator in a multicast operation.
public class ClientTarget
{
  private static UdpClient m_ClientTarget;
  private static IPAddress m_GrpAddr;
  // The following StartMulticastConversation method connects the UDP
  // ClientTarget with the ClientOriginator.
```

```
  // It performs the following main tasks:
  // 1)Creates a UDP client to receive data on a specific port and using
  // IPv6 addresses. The port is the same one used by the ClientOriginator
  // to define its communication endpoint.
  // 2)Joins or creates a multicast group at the specified address.
  // 3)Defines the endpoint port to send data to the ClientOriginator.
  // 4)Receives data from the ClientOriginator until the end of the
  // communication.
  // 5)Sends data to the ClientOriginator.
  // Note this method is the counterpart of the
  // ClientOriginator.ConnectOriginatorAndTarget().
  public static void StartMulticastConversation()
  {
    string Ret;
    // Bind and listen on port 1000. Specify the IPv6 address family type.
    m_ClientTarget = new UdpClient(1000, AddressFamily.InterNetworkV6);
    // Join or create a multicast group
    m_GrpAddr = IPAddress.Parse("FF01::1");
    // Use the overloaded JoinMulticastGroup method.
    // Refer to the ClientOriginator method to see how to use the other
    // methods.
    m_ClientTarget.JoinMulticastGroup(m_GrpAddr);
    // Define the endpoint data port. Note that this port number
    // must match the ClientOriginator UDP port number which is the
    // port on which the ClientOriginator is receiving data.
    IPEndPoint ClientOriginatordest = new IPEndPoint(m_GrpAddr, 2000);

    // Receive data from the ClientOriginator.
    Ret = Receive.ReceiveUntilStop(m_ClientTarget);
    Console.WriteLine("\nThe ClientTarget received: " + "\n\n" + Ret + "\n");

    // Done receiving, now respond to the ClientOriginator.
    // Wait to make sure the ClientOriginator is ready to receive.
    Thread.Sleep(2000);
    Console.WriteLine("\nThe ClientTarget sent:\n");
    Send.TargetSendData(m_ClientTarget, ClientOriginatordest);
    // Exit the multicast conversation.
    m_ClientTarget.DropMulticastGroup(m_GrpAddr);
  }
}
// The following ClientOriginator class starts the multicast conversation
// with the ClientTarget class..
// It performs the following main tasks:
// 1)Creates a socket and binds it to the port on which to communicate.
// 2)Specifies that the connection must use an IPv6 address.
```

```
// 3)Joins or create a multicast group.
//   Note that the multicast address ranges to use are specified
//   in the RFC#2375.
// 4)Defines the endpoint to send the data to and starts the
// client target (ClientTarget) thread.
public class ClientOriginator
{
  private static UdpClient clientOriginator;
  private static IPAddress m_GrpAddr;
  private static IPEndPoint m_ClientTargetdest;
  private static Thread m_t;
  // The ConnectOriginatorAndTarget method connects the
  // ClientOriginator with the ClientTarget.
  // It performs the following main tasks:
  // 1)Creates a UDP client to receive data on a specific port
  //   using IPv6 addresses.
  // 2)Joins or create a multicast group at the specified address.
  // 3)Defines the endpoint port to send data to on the ClientTarget.
  // 4)Starts the ClientTarget thread that also creates the ClientTarget object.
  // Note this method is the counterpart of the
  // ClientTarget.StartMulticastConversation().
  public static bool ConnectOriginatorAndTarget()
  {
    try
    {
      // Bind and listen on port 2000. This constructor creates a socket
      // and binds it to the port on which to receive data. The family
      // parameter specifies that this connection uses an IPv6 address.
      clientOriginator = new UdpClient(2000, AddressFamily.InterNetworkV6);

      // Join or create a multicast group. The multicast address ranges
      // to use are specified in RFC#2375. You are free to use
      // different addresses.
      // Transform the string address into the internal format.
      m_GrpAddr = IPAddress.Parse("FF01::1");
      // Display the multicast address used.
      Console.WriteLine("Multicast Address: [" + m_GrpAddr.ToString() + "]");
      // Exercise the use of the IPv6MulticastOption.
      Console.WriteLine("Instantiate IPv6MulticastOption(IPAddress)");
      // Instantiate IPv6MulticastOption using one of the
      // overloaded constructors.
      IPv6MulticastOption ipv6MulticastOption = new
IPv6MulticastOption(m_GrpAddr);
      // Store the IPAdress multicast options.
      IPAddress group =  ipv6MulticastOption.Group;
```

```
        long interfaceIndex = ipv6MulticastOption.InterfaceIndex;
        // Display IPv6MulticastOption properties.
        Console.WriteLine("IPv6MulticastOption.Group: [" + group  + "]");
        Console.WriteLine("IPv6MulticastOption.InterfaceIndex: [" + interfaceIndex
+ "]");
        // Instantiate IPv6MulticastOption using another
        // overloaded constructor.
        IPv6MulticastOption ipv6MulticastOption2 = new
IPv6MulticastOption(group, interfaceIndex);
        // Store the IPAdress multicast options.
        group =  ipv6MulticastOption2.Group;
        interfaceIndex = ipv6MulticastOption2.InterfaceIndex;
        // Display the IPv6MulticastOption2 properties.
        Console.WriteLine("IPv6MulticastOption.Group: [" + group  + "]");
        Console.WriteLine("IPv6MulticastOption.InterfaceIndex: [" + interfaceIndex
+ "]");
        // Join the specified multicast group using one of the
        // JoinMulticastGroup overloaded methods.
        clientOriginator.JoinMulticastGroup((int)interfaceIndex, group);
        // Define the endpoint data port. Note that this port number
        // must match the ClientTarget UDP port number which is the
        // port on which the ClientTarget is receiving data.
        m_ClientTargetdest = new IPEndPoint(m_GrpAddr, 1000);
        // Start the ClientTarget thread so it is ready to receive.
        m_t = new Thread(new
ThreadStart(ClientTarget.StartMulticastConversation));
        m_t.Start();
        // Make sure that the thread has started.
        Thread.Sleep(2000);
        return true;
      }
      catch (Exception e)
      {
        Console.WriteLine("[ClientOriginator.ConnectClients] Exception: " +
e.ToString());
        return false;
      }
    }

    // The SendAndReceive performs the data exchange
    // between the ClientOriginator and the ClientTarget classes.
    public static string SendAndReceive()
    {
      string Ret = "";
      // Send data to ClientTarget.
```

```
    Console.WriteLine("\nThe ClientOriginator sent:\n");
    Send.OriginatorSendData(clientOriginator, m_ClientTargetdest);
    // Receive data from ClientTarget
    Ret = Receive.ReceiveUntilStop(clientOriginator);
    // Stop the ClientTarget thread
    m_t.Abort();
    // Abandon the multicast group.
    clientOriginator.DropMulticastGroup(m_GrpAddr);
    return Ret;
  }
  //This is the console application entry point.
  public static void Main()
  {
    // Join the multicast group.
    if (ConnectOriginatorAndTarget())
    {
      // Perform a multicast conversation with the ClientTarget.
      string Ret = SendAndReceive();
      Console.WriteLine("\nThe ClientOriginator received: " + "\n\n" + Ret);
    }
    else
    {
      Console.WriteLine("Unable to Join the multicast group");
    }
  }
}
}
```

2. Теоретическая часть

Ряд приложений, например, дистанционное обучение, рассылка почты, радио, видео по запросу, видеоконференцсвязь, поддерживают мультивещание. В одноадресной сети с каждым получателем устанавливается индивидуальное соединение даже при потреблении одного ресурса по общему маршруту. В многоадресной рассылке источник посылает единственный экземпляр данных по общему маршруту тем получателям, кто подписался на рассылку. Преимущество этого подхода: добавление новых пользователей не влечет за собой необходимость увеличения пропускной способности сети по общему маршруту до потребителей услуги. Соответственно, снижается нагрузка и на промежуточное оборудование.

При запуске на сервере приложения с поддержкой мультивещания, оно посылает в сеть уведомление, что соответствующая группа доступна для присоединения. Клиент, который хочет присоединиться к рассылке посылает уведомление об этом. Все промежуточные маршрутизаторы записывают, что за соответствующим маршрутом находится клиент соответствующей

мультикастной группы. Поскольку состав группы со временем может меняться, вновь появившиеся и выбывшие члены группы динамически учитываются в построении путей маршрутизации. В локальной сети управлением мультикастными группами обычно занимаетсяIGMP. Существует несколько алгоритмов и протоколов для построения мультивещательного дерева и опроса участников.

Для мультикастных групп зарезервированы адреса как на канальном, так и на сетевом уровнях.

Чтобы технология работала, она должна поддерживаться сервером, клиентом и всеми промежуточными маршрутизаторами. Чтобы коммутаторы посылали пакеты только нужным получателям, они должны поддерживатьIGMP snooping (у Cisco есть своя реализация — CGMP), иначе пакеты рассылаются широковещательно. Также нужно иметь в виду, что мультикаст может блокироваться межсетевыми экранами.

Мультивещание в интернете

В 1995 году была создана международная магистральная сеть для обмена мультивещательным трафикомMbone. На её основе с 1997 до 2008 работала система виртуальных комнат для видео-конференций. Российская точка обмена мультикаст-трафиком Multicast Internet Exchange (Multicast-IX) создана на базе Московского Internet Exchange в 2002 году.

Существуют уникальные глобальные мультикастовые группы, принадлежащие соответствующим компаниям. Обычное физическое лицо не может вещать в интернете с использованием частных мультикастных групп. Кроме того, многие интернет-провайдеры не имеют мультикаст-связности либо запрещают её. Для преодоления участков, не поддерживающих мультивещание, может применяться туннелирование.

Мультивещание, многоадресное вещание (англ. *multicast* — групповая передача) — форма широковещания, при которой адресом назначения сетевого пакета является мультикастная группа (один ко многим). Существует мультивещание на канальном, сетевом и прикладном уровнях. Мультивещание не следует путать с технологией передачи на физическом уровне point-to-multipoint communication.

Лабораторная работа №11
Программирование параллельных действии в сети

Цель работы: Понятия параллельных действий. Программирование параллельных действии в сети

1. Практическая часть

Специальные сетевого адреса используются для поддержки широковещательные сообщения UDP на основе IP сетях. Следующее обсуждение использует семейство адресов версии 4 протокола IP, используемое в Интернете в качестве примера.

Адреса IP версии 4 использует 32 бита, чтобы указать сетевой адрес. Для адресов C# класса с помощью netmask 255.255.255.0, эти биты разделяются на 4 октета. Выражанный в десятичном числе, 4 формируют знакомую октета нотации ставить точки- квартета, например 192.168.100.2. Первые 2 192,168 октета (в данном примере) формируют номер сети, третий октет (100) определяют подсеть и конечный октет (2) идентификатор основного приложения.

Установка все биты IP-адрес, 255.255.255.255, ограниченный или формы адрес широковещательной рассылки. Отправляет датаграмму UDP к этому адресу доставляет сообщение любому узлу в сегменте локальной сети. Поскольку сообщения, отправляемые по этому адресу, только основные приложения никогда front маршрутизаторов на сегмент сети, получают широковещательное сообщение.

Широковещательные можно непосредственно к определенным фрагментам сети, задав все биты идентификатора основного приложения. Например, чтобы отправить широковещательной ко всем основным приложениям в сети IP-адресами, начиная с указанной 192.168.1, используйте адрес 192.168.1.255.

В следующем примере кода используется UdpClient для прослушивания датаграмм UDP, отправляемых непосредственно широковещательному адресу 192.168.1.255 на порту 11.000. Клиент получает строку сообщения и записывает сообщение на консоль.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
public class UDPListener
{
   private const int listenPort = 11000;
   private static void StartListener()
   {
      bool done = false;
```

```
      UdpClient listener = new UdpClient(listenPort);
      IPEndPoint groupEP = new IPEndPoint(IPAddress.Any,listenPort);
      try
      {
         while (!done)
         {
            Console.WriteLine("Waiting for broadcast");
            byte[] bytes = listener.Receive( ref groupEP);
            Console.WriteLine("Received broadcast from {0} :\n {1}\n",
               groupEP.ToString(),
               Encoding.ASCII.GetString(bytes,0,bytes.Length));
         }

      }
      catch (Exception e)
      {
         Console.WriteLine(e.ToString());
      }
      finally
      {
         listener.Close();
      }
   }

   public static int Main()
   {
      StartListener();
      return 0;
   }
}
```

В следующем примере кода используется UdpClient отправить датаграммы UDP значение прямой широковещательному адресу 192.168.1.255, используя порт 11.000. Клиент отправляет строку сообщения, указанную в командной строке.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class Program
{
   static void Main(string[] args)
   {
      Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
```

```
        ProtocolType.Udp);
      IPAddress broadcast = IPAddress.Parse("192.168.1.255");
      byte[] sendbuf = Encoding.ASCII.GetBytes(args[0]);
      IPEndPoint ep = new IPEndPoint(broadcast, 11000);
      s.SendTo(sendbuf, ep);
      Console.WriteLine("Message sent to the broadcast address");
   }
}
```

2. Теоретическая часть

Мультипроцессоры и *компьютеры параллельного действия* получили широкое распространение, поэтому материал, связанный с архитектурами параллельного действия, был полностью переделан и значительно расширен. В этой книге мы затрагиваем широкий диапазон тем от мультипроцессоров до кластеров рабочих станций.

В этой главе мы изложим основные принципы разработки *компьютеров параллельного действия* и рассмотрим различные примеры. Все эти машины состоят из элементов процессора и элементов памяти. Отличаются они друг от друга количеством элементов, их типом и способом взаимодействия между элементами.

В следующих разделах мы рассмотрим некоторые вопросы разработки *компьютеров параллельного действия*. Мы начнем с информационных моделей и сетей межсоединений, затем рассмотрим вопросы, связанные с производительностью и программным обеспечением, и, наконец, перейдем к классификации архитектур компьютеров параллельного действия.

Чтобы решать более сложные задачи, разработчики обращаются к *компьютерам параллельного действия*. Невозможно построить компьютер с одним процессором и временем цикла в 0 001 не, но зато можно построить компьютер с 1000 процессорами, время цикла каждого из которых составляет 1 не. И хотя во втором случае мы используем процессоры, которые работают с более низкой скоростью, общая производительность теоретически должна быть такой же.

Закон Амдала ограничивает потенциальный коэффициент ускорения, достижимый в *компьютере параллельного действия*. Вычислите как функцию от f максимально возможный коэффициент ускорения, если число процессоров стремится к бесконечности.

Многое можно сказать о программном обеспечении для параллельных компьютеров, но сейчас мы должны вернуться к основной теме данной главы - архитектуре *компьютеров параллельного действия*. Было предложено и построено множество различных видов параллельных компьютеров, поэтому хотелось бы узнать, можно ли их как-либо категоризировать. К сожалению, хорошей классификации компьютеров параллельного действия до сих пор не существует. Чаще всего используется классификация Флин-на (Flynn), но даже она является очень грубым приближением.

Здесь существует несколько вариантов. Некоторые *компьютеры параллельного действия* одновременно выполняют несколько независимых задач. Эти задачи никак не связаны друг с другом и не взаимодействуют. Типичный пример - компьютер, содержащий от 8 до 64 процессоров, представляющий собой большую систему UNIX с разделением времени, с которой могут работать тысячи пользователей. В эту категорию попадают системы обработки транзакций, которые используются в банках (например, банковские автоматы), на авиалиниях (например, системы резервирования) и в больших web - серверах. Сюда же относятся независимые прогоны моделирующих программ, при которых используется несколько наборов параметров.

Если эта цель не достигнута, то никакого смысла в построении *компьютера параллельного действия* нет. Более того, эта цель должна быть достигнута при наименьших затратах.

В следующих разделах мы рассмотрим некоторые вопросы разработки компьютеров параллельного действия. Мы начнем с информационных моделей и сетей межсоединений, затем рассмотрим вопросы, связанные с производительностью и программным обеспечением, и, наконец, перейдем к классификации архитектур*компьютеров параллельного действия*.

Многое можно сказать о программном обеспечении для параллельных компьютеров, но сейчас мы должны вернуться к основной теме данной главы - архитектуре компьютеров параллельного действия. Было предложено и построено множество различных видов параллельных компьютеров, поэтому хотелось бы узнать, можно ли их как-либо категоризировать. К сожалению, хорошей классификации *компьютеров параллельного действия* до сих пор не существует. Чаще всего используется классификация Флин-на (Flynn), но даже она является очень грубым приближением.

Метрика аппаратного обеспечения показывает, на что способно аппаратное обеспечение. Но пользователей интересует совсем другое. Они хотят знать, насколько быстрее будут работать их программы на *компьютере параллельного действия* по сравнению с однопроцессорным компьютером. Для них ключевым показателем является коэффициент ускорения: насколько быстрее работает программа в п-про-цессорной системе по сравнению с 1-процессорной системой. Здесь мы видим несколько разных параллельных программ, которые работают на мультикомпьютере, состоящем из 64 процессоров Pentium Pro. Лишь немногие программы достигают совершенного повышения скорости, но есть достаточное число программ, которые приближаются к идеалу.

Лабораторная работа № 12
Программирования коммутации в сети

Цель работы: Понятия коммутации и принципы коммутации данных. Программирования коммутации в сети

1. Практическая часть

IPEndPoint

Похожа на структуру sockaddr_in с Unix .Framework использует объект IPEndPoint представлять определенный IP-адрес/порт сочетание. Объект IPEndPoint используется для привязки розеток для локальных адресов, или при подключении розетки к удаленным адресам. Мы будем сначала изучить все кусочки IPEndPoint, а затем посмотреть на программу, которая ставит его на работу.

Два конструкторы используются для создания IPEndPoint экземпляры:

- IPEndPoint(длинный адрес, порт инт);
- IPEndPoint(адрес IP-адрес, порт инт).

Оба конструктора используют два параметра: IP-адрес, значение, представлено как длинное значение или объект, IP-адрес, и номер порта целое число. Как вы можете догадаться, наиболее распространенных конструктор форме Адрес.
Класс SocketAddress особый класс в пространстве имен system.net . Он представляет собой сериализованный версии объекта IPEndPoint. Этот класс может использоваться для хранения IPEndPoint экземпляр, который затем может быть воссоздан с помощью IPEndPoint.Метод Create (). Формат класса SocketAddress выглядит следующим образом:
- 1 байт представляет AddressFamily объекта.
- 1 байт-это размер объекта.
- 2 байта представляют номер порта объекта.
- Оставшиеся байты представляют собой IP-адрес объекта.

В дополнение к методам, класса IPEndPoint также содержит три свойства, которые могут быть получены из экземпляра:
Адрес - получает или задает IP-адрес собственность
AddressFamily - получает IP-адрес семьи
Порт - получает или задает TCP-или udp-порт

Каждое из этих свойств может быть использован с IPEndPoint экземпляр, чтобы получить информацию об отдельных частях объекта IPEndPoint. Адрес и свойствами порт также может быть использован для установки отдельных значений в пределах существующего объекта IPEndPoint.

Есть также два поля, которые могут быть использованы с объектом IPEndPoint получить доступные диапазоны портов из системы:

MaxPort - максимальное значение, которое можно присвоить номер порта
MinPort - минимальное значение, которое можно присвоить номер порта

Пример программы IPEndPoint

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

namespace ConsoleApplication3
{
   class Program
   {
      static void Main(string[] args)
      {
IPAddress test1 = IPAddress.Parse("192.168.1.1");
IPEndPoint ie = new IPEndPoint(test1, 8000);
Console.WriteLine("The IPEndPoint is: {0}", ie.ToString());
Console.WriteLine("The AddressFamily is: {0}", ie.AddressFamily);
Console.WriteLine("The address is: {0}, and the A port is: {1}\n",
ie.Address, ie.Port);
Console.WriteLine("The min port number is: {0}", IPEndPoint.MinPort);
Console.WriteLine("The max port number is: {0}\n", IPEndPoint.MaxPort);
ie.Port = 80;
Console.WriteLine("The changed IPEndPoint value A is: {0}",
ie.ToString());
SocketAddress sa = ie.Serialize();
Console.WriteLine("The SocketAddress is: {0}", sa.ToString());
      }
   }
```

В IPEndPointSample.программы CS демонстрирует несколько важных особенностей IPEndPoint. Обратите внимание, что вы можете отобразить полный объект IPEndPoint как одну строку, или вы можете извлечь отдельные части объекта:

```
Console.WriteLine("The IPEndPoint is: {0}",
```

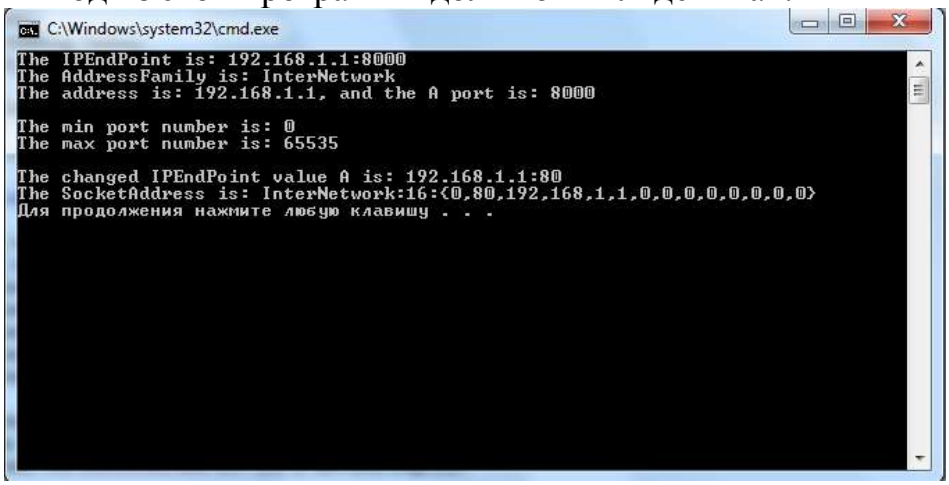
```
ie.ToString());
Console.WriteLine("The AddressFamily is: {0}",
ie.AddressFamily);
Console.WriteLine("The address is: {0}, and the A
port is: {1}\n", ie.Address, ie.Port);
```

Программа также демонстрирует, как изменить порт стоимость объекта IPEndPoint индивидуально, используя свойство Port:

т. е. Порт = 80;

Это позволит вам изменять адрес и порт значения в объект, не создавая новый объект.

Выход из этой программы должно выглядеть так:



2. Вопросы

1. Описать IP-адресации?
2. Класс функций IP-адрес
3. Функции класса IPEndPoint
4. Функции ДНС класс
5. Описать методы класса IP-адрес
6. Описать методы класса IPEndPoint

3. Теоретическая часть

Под коммутацией данных понимается их передача, при которой канал передачи данных может использоваться попеременно для обмена информацией между различными пунктами информационной сети

Коммутация основана на использовании маршрутизации, определяющей путь, по которому в соответствии с адресом назначения передаются данные. Осуществляется коммутация функциональными блоками всех систем информационной сети.

Коммутация является основой технологии сети с маршрутизацией данных. В зависимости от задач, поставленных перед коммуникационной сетью, используют несколько методов коммутации. Каждый из них определяется различными штабелями уровней области Взаимодействия Открытых Систем

(ВОС).У каждого из методов коммутации имеется своя область применения, обусловленная его особенностями. Отсюда следует целесообразность сочетания разных методов коммутации на сетях, объединяющих большое число абонентов с отличающимися друг от друга величинами нагрузки, характером ее распределения во времени, объемами сообщений, используемой оконечной аппаратурой. На таких сетях при небольшой средней нагрузке и передаче сообщений большими массивами в небольшое число адресов доля потери времени на установление соединения сравнительно невелика, и предпочтительнее использовать систему с коммутацией каналов. При передаче же многоадресных сообщений, необходимости обеспечения приоритетности сообщениям высокой категории срочности и при большой загрузке абонентских установок более эффективно использовать систему с коммутацией сообщений. При передаче коротких сообщений в интерактивном (диалоговом) режиме наиболее целесообразно использовать коммутацию пакетов.

Выбор методов коммутации - достаточно сложная оптимизационная задача. Она решается исходя из требований к транспортной сети, которые в свою очередь определяются особенностями графика, классом пользователей и показателями качества их обслуживания.

В коммутации блоков данных участвуют N нижних уровней взаимодействующих друг с другом абонентских систем или административных систем, а также расположенных между ними ретрансляционных систем. В зависимости от метода коммутации, число уровней N изменяется от одного до семи.В коммутации блоков данных участвуют N нижних уровней взаимодействующих друг с другом абонентских систем или административных систем, а также расположенных между ними ретрансляционных систем. В зависимости от метода коммутации, число уровней N изменяется от одного до семи. В коммутации блоков данных участвуют N нижних уровней взаимодействующих друг с другом абонентских систем или административных систем, а также расположенных между ними ретрансляционных систем. В зависимости от метода коммутации, число уровней N изменяется от одного до семи.

Лабораторная работа №13

Проектирования и программирования IP коммутаторов

Цель работы: Принципы проектирования и программирования IP коммутаторов

2. Практическая часть

Для рисования ПК выбираем в конечных устройства настольный компьютер и, удерживая **Ctrl**, (так быстрее) нажмите 1 раз на ПК а затем рисуйте нужное кол-во ПК, щелкая мышкой (рис. 1). Этим приемом вы сможете за один раз нарисовать сразу 4 ПК.

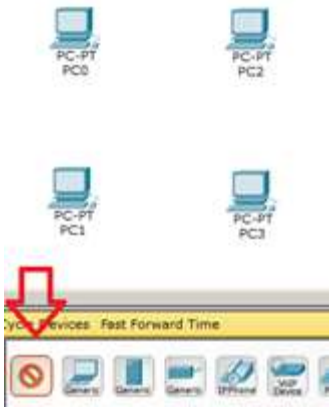


Рис. 1. Выбор устройств, удерживая Ctrl

Устанавливаем коммутатор и, удерживая Ctrl, создаем подключение прямым кабелем, выбирая порты коммутатора. После инициализации портов все лампы загорятся зеленым. На схему будет две подсети (рис. 2) .

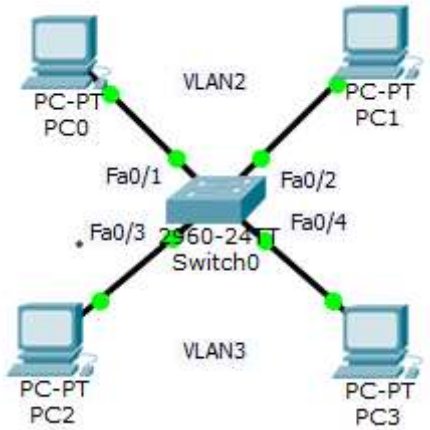


Рис.2. Две подсети: VLAN2 и VLAN3

Примечание

Имя VLAN1 используется по умолчанию, его лучше в нашем примере не использовать.

На коммутаторе набираем команду **en** и входим в привилегированный режим. Затем набираем команду **conf t** для входа в режим глобального конфигурирования. Если подвести курсор мыши к портам коммутатора, то вы увидите какие порты в каком сегменте задействованы. Для VLAN3 – это Fa0/3 и Fa0/4 (предположим, что это будет бухгалтерия - buh) и для VLAN2 – это Fa0/1 и Fa0/2 (предположим, что это будет склад – sklad). Сначала будем конфигурировать второй сегмент сети VLAN2 (sklad) – рис. 3

```
Switch#
Switch#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
Switch(config)#vlan 2
Switch(config-vlan)#name sklad
```

Рис. 3. VLAN2 получает имя sklad

В виртуальной сети VLAN2 настраиваем порты коммутатора Fa0/1 и Fa0/2 как access порты, т.е. порты для подключения пользователей (рис. 4).

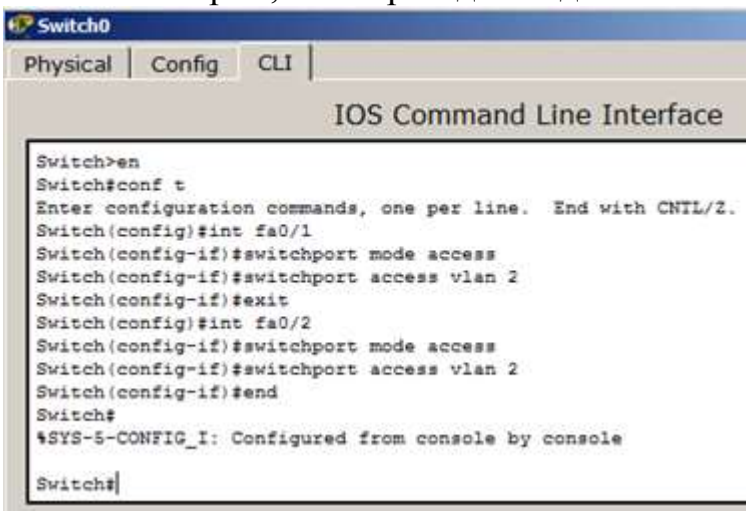


Рис. 4. Указываем порты коммутатора для подключения пользователей

Теперь командой **show vlan** можно проверить результат (рис. 5.).

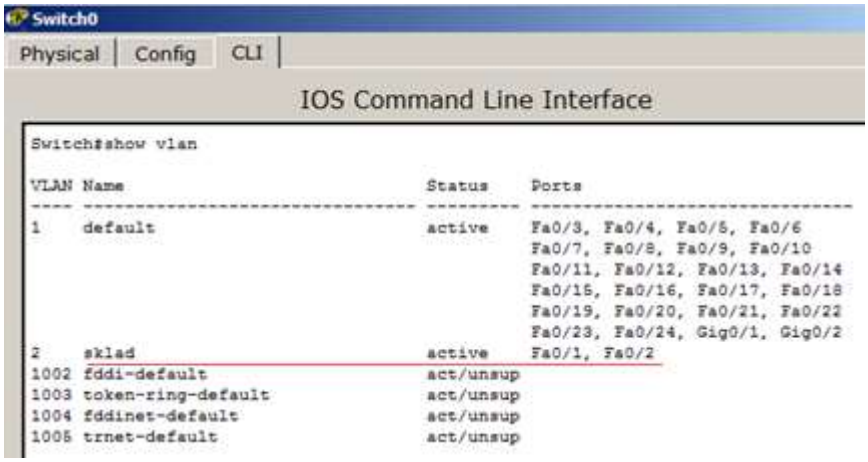


Рис. 6. Подсеть VLAN2 склад настроена
Далее работаем с VLAN3 (рис.).

```
Switch#
Switch#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
Switch(config-vlan)#vlan 3
Switch(config-vlan)#name buh
Switch(config-vlan)#exit
Switch(config)#
```

Рис. VLAN3 получает имя buh
В виртуальной сети VLAN3 настраиваем порты коммутатора Fa0/3 и Fa0/4 как **access** порты, т.е. порты для подключения пользователей, затем командой **show vlan** можно проверить и убедиться, что мы создали в сети 2 сегмента на разные порты коммутатора (рис. 8).

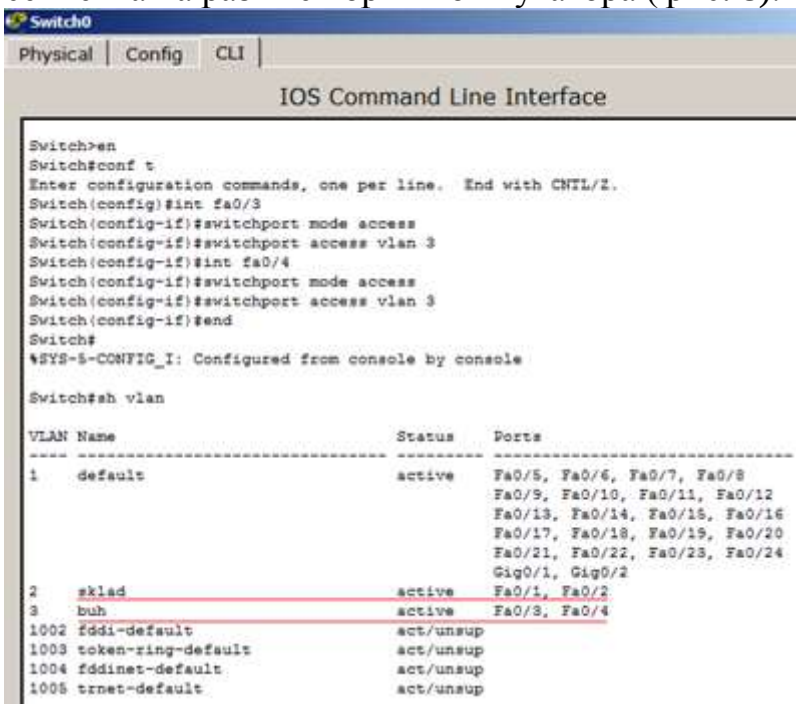


Рис. 8. Мы настроили VLAN2 и VLAN3

Настраиваем IP адреса компьютеров – для VLAN2 из сети 192.168.2.0, а для VLAN3 из сети 192.168.3.0 (рис. 9).

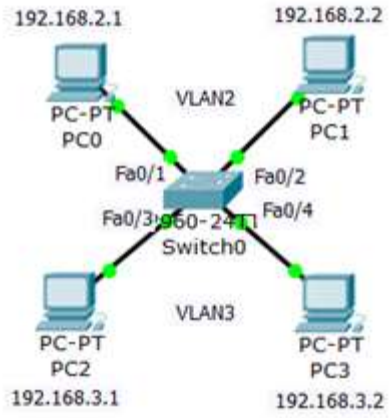


Рис. 9. Настраиваем IP адреса компьютеров

Проверяем связь ПК в пределах VLAN и отсутствие связи между VLAN2 и VLAN3 (рис. 10).

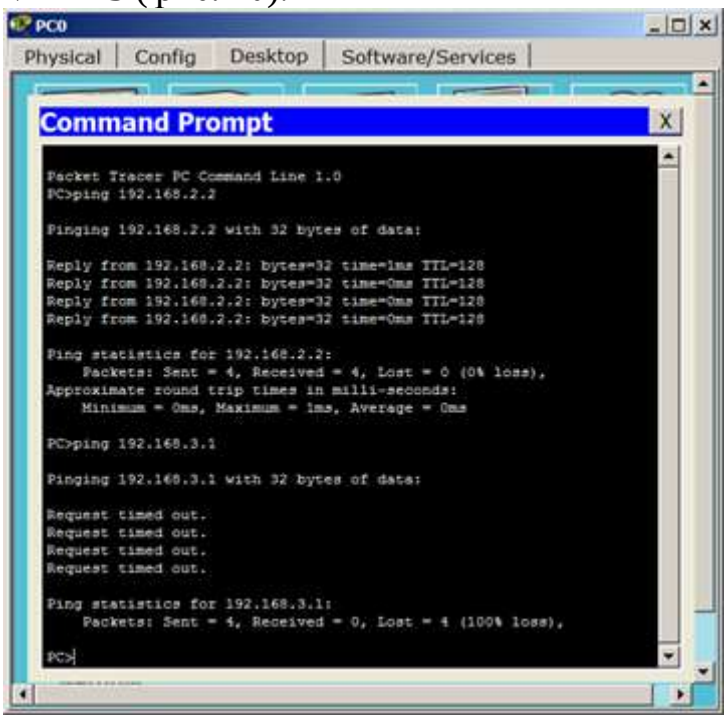


Рис. 10. Все работает так, как было задумано

Итак, на компьютере ПК0 мы убедились, что компьютер в своем сегменте видит ПК, а в другом сегменте – нет.

Практическая работа 2 Настройка сети на коммутаторе 2960

В данной работе рассматривается настройка *VLAN* на коммутаторе фирмы Cisco в программе CPT. Мы уже делали подобную работу. Но здесь мы не только закрепим пройденное, но и узнаем ряд новых команд Cisco *IOS*.

Создайте *сеть*, *топология* которой представлена на рис. 11.

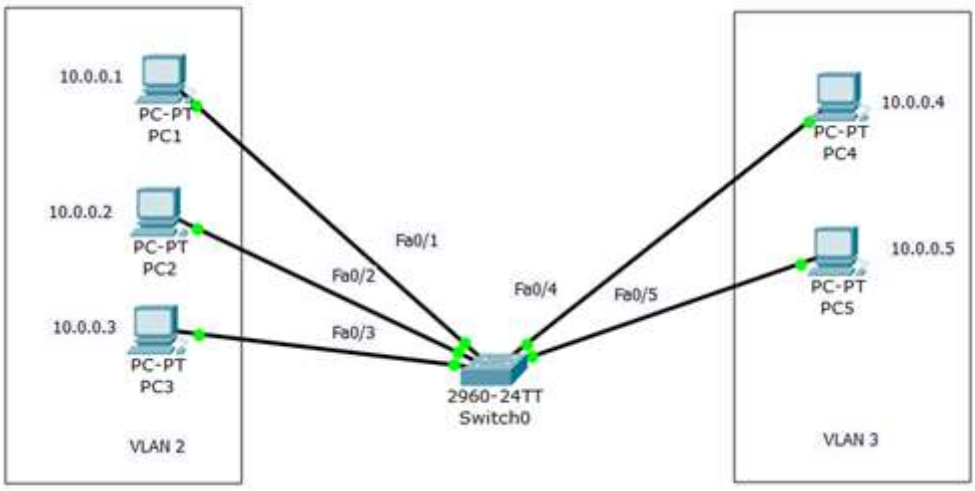


Рис. 11. Схема сети с одним коммутатором

Задача данной работы является создание 2х независимых групп компьютеров: ПК1-ПК3 должны быть доступны только друг для друга, а вторая независимая *группа* - компьютеры ПК4 и ПК5.

Настройка коммутатора

Первоначально сформируем VLAN2. Дважды щелкните левой кнопкой мыши по коммутатору. В открывшемся окне перейдите на вкладку **CLI**. Вы увидите окно консоли. Нажмите на клавишу **Enter** для того, чтобы приступить к вводу команд. Перейдем в привилегированный режим, выполнив команду **enable**:

Switch>en

По умолчанию все ПК объединены в VLAN1. Для реализации сети, которую мы запланировали, создадим на коммутаторе еще два VLAN (2 и 3). Для этого в привилегированном режиме выполните следующую команду для перехода в режим конфигурации:

Switch#conf t

Теперь вводим команду **VLAN 2**. Данной командой вы создадите на коммутатореVLANс номером 2. Указатель ввода Switch (config)# изменится на Switch (config-vlan)# это свидетельствует о том, что вы конфигурируете уже не весь коммутатор в целом, а только отдельныйVLAN, в данном случаеVLANномер 2 (рис. 12).

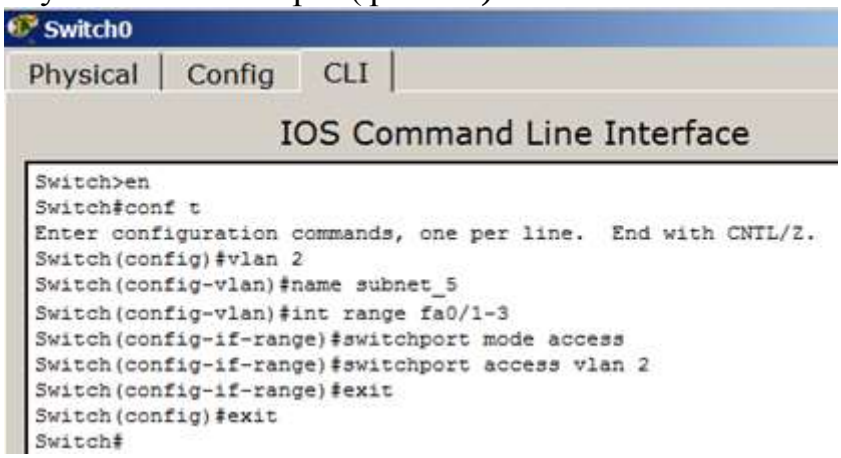


Рис. 12. Листинг команд для формирования VLAN2

Примечание

Командой VLAN2, мы создаем на коммутаторе новый VLAN с номером 2. Команда **name subnet_5** присваивает имя subnet_5 виртуальной сети номер 2. Выполняя команду **interface range fast Ethernet 0/1-3** мы переходим к конфигурированию интерфейсов fastEthernet 0/1, fastEthernet 0/2 и fastEthernet 0/3 коммутатора. Слово range в данной команде, указывает на то, что мы будем конфигурировать не один порт, а диапазон портов. Команда **switch port mode access** конфигурирует выбранный порт коммутатора, как порт доступа (access порт). Команда **switch port access vlan 2** указывает, что данный порт является портом доступа для VLAN номер 2.

Выйдите из режима конфигурирования, дважды набрав команду **exit** и просмотрите результат конфигурирования (рис. 13), выполнив команду **sh vl br**. Как видим, на коммутаторе появился VLAN с номером 2 и именем subnet_5, портами доступа которого являются fastEthernet 0/1, fastEthernet 0/2 и fastEthernet 0/3.

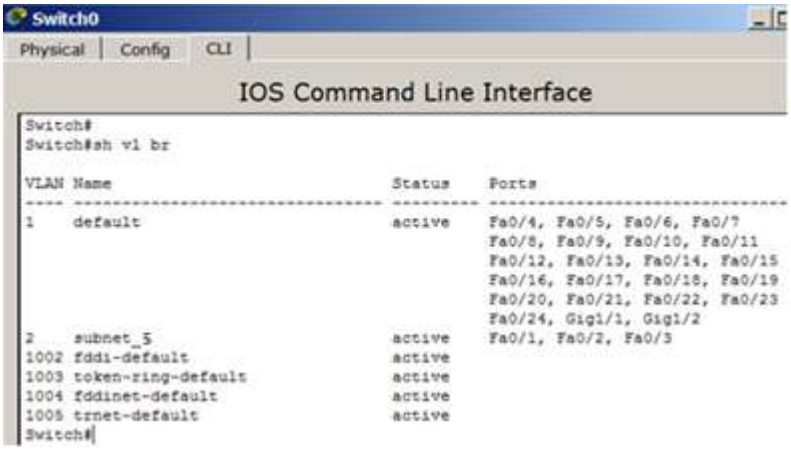


Рис. 13. Просмотр информации о VLAN на коммутаторе

Примечание

Команда **shvlbr** выводит информацию о существующих на коммутаторе VLAN-ах. В результате выполнения команды на экране появится: **номера VLAN** (первый столбец), **название VLAN** (второй столбец), **состояние VLAN** (работает он или нет) – третий столбец, **порты**, принадлежащие к данному VLAN (четвертый столбец).

Далее аналогичным образом создадим **VLAN 3** с именем **subnet_6** и сделаем его портами доступа интерфейсы fastEthernet 0/4 и fastEthernet 0/5. Результат показан на рис. 14.

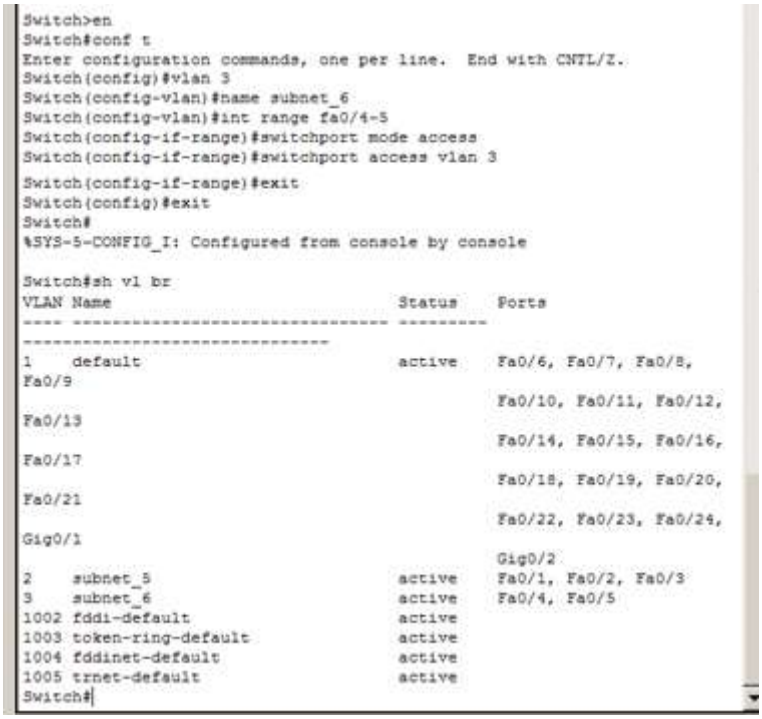


Рис. 14. Результат – настройка на коммутатореVLAN2 и VLAN3

Проверка результатов работы

Сеть настроена и нужно ее протестировать. Результат положителен, если в пределах своей VLAN компьютеры доступны, а компьютеры из разных VLAN не доступны (рис.15). У нас все пять компьютеров находя в одной сети 10.0.0.0/8, но они находятся в разных виртуальных локальных сетях.

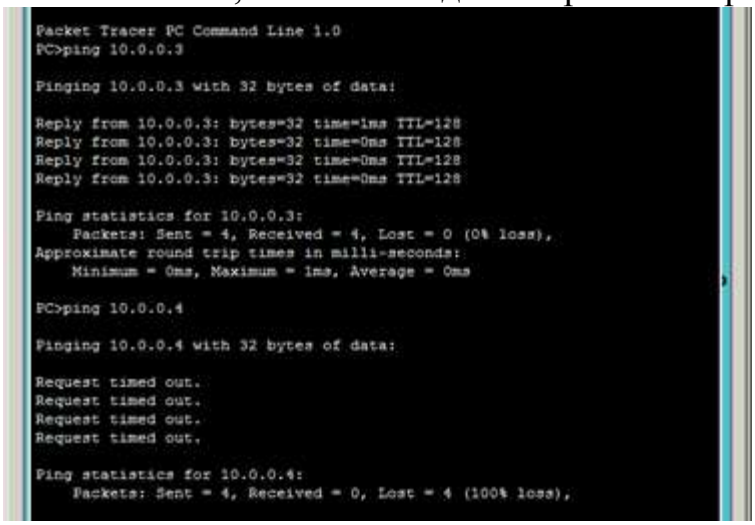


Рис. 15. Пинг с PC1 на PC3 и PC4

2. Теортическая часть:

Сетевой коммутатор (жарг. **свитч** от англ. *switch* — переключатель) — устройство, предназначенное для соединения нескольких узлов компьютерной сети в пределах одного или

нескольких сегментов сети. Коммутатор работает на канальном (втором) уровне модели OSI. Коммутаторы были разработаны с использованием мостовых технологий и часто рассматриваются как много портовые мосты. Для соединения нескольких сетей на основе сетевого уровня служат маршрутизаторы (3 уровень OSI).

В отличие от концентратора (1 уровень OSI), который распространяет трафик от одного подключённого устройства ко всем остальным, коммутатор передаёт данные только непосредственно получателю (исключение составляет широковещательный трафик всем узлам сети и трафик для устройств, для которых неизвестен исходящий порт коммутатора). Это повышает производительность и безопасность сети, избавляя остальные сегменты сети от необходимости (и возможности) обрабатывать данные, которые им не предназначались.

Далее в этой статье рассматриваются исключительно коммутаторы для технологии Ethernet.

Принцип работы коммутатора

Коммутатор хранит в памяти (т.н. ассоциативной памяти) таблицу коммутации, в которой указывается соответствие MAC-адреса узла порту **коммутатора**. При включении коммутатора эта таблица пуста, и он работает в режиме обучения. В этом режиме поступающие на какой-либо порт данные передаются на все остальные порты коммутатора. При этом коммутатор анализирует фреймы (кадры) и, определив MAC-адрес хоста-отправителя, заносит его в таблицу на некоторое время. Впоследствии, если на один из портов коммутатора поступит кадр, предназначенный для хоста, MAC-адрес которого уже есть в таблице, то этот кадр будет передан только через порт, указанный в таблице. Если MAC-адрес хоста-получателя не ассоциирован с каким-либо портом коммутатора, то кадр будет отправлен на все порты, за исключением того порта, с которого он был получен. Со временем коммутатор строит таблицу для всех активных MAC-адресов, в результате трафик локализуется.

Стоит отметить малую латентность (задержку) и высокую скорость пересылки на каждом порту интерфейса.

Режимы коммутации

Существует три способа коммутации. Каждый из них — это комбинация таких параметров, как время ожидания и надёжность передачи.

1. С промежуточным хранением (Store and Forward). Коммутатор читает всю информацию в кадре, проверяет его на отсутствие ошибок, выбирает порт коммутации и после этого посылает в него кадр.
2. Сквозной (cut-through). Коммутатор считывает в кадре только адрес назначения и после выполняет коммутацию. Этот режим уменьшает задержки при передаче, но в нём нет метода обнаружения ошибок.
3. Бесфрагментный (fragment-free) или *гибридный*. Этот режим является модификацией сквозного режима. Передача осуществляется после фильтрации фрагментов коллизий (первые 64 байта кадра анализируются на

наличие ошибки и при её отсутствии кадр обрабатывается в сквозном режиме).

Задержка, связанная с «принятием коммутатором решения», добавляется к времени, которое требуется кадру для входа на порт коммутатора и выхода с него, и вместе с ним определяет общую задержку коммутатора.

Симметричная и асимметричная коммутация

Свойство симметрии при коммутации позволяет дать характеристику коммутатора с точки зрения ширины полосы пропускания для каждого его порта. Симметричный коммутатор обеспечивает коммутируемые соединения между портами с одинаковой шириной полосы пропускания, например, когда все порты имеют ширину пропускания 10 Мб/с или 100 Мб/с.

Асимметричный коммутатор обеспечивает коммутируемые соединения между портами с различной шириной полосы пропускания, например, в случаях комбинации портов с шириной полосы пропускания 10 Мб/с или 100 Мб/с и 1000 Мб/с.

Асимметричная коммутация используется в случае наличия больших сетевых потоков типа клиент-сервер, когда многочисленные пользователи обмениваются информацией с сервером одновременно, что требует большей ширины пропускания для того порта коммутатора, к которому подсоединён сервер, с целью предотвращения переполнения на этом порте. Для того, чтобы направить поток данных с порта 100 Мб/с на порт 10 Мб/с без опасности переполнения на последнем, асимметричный коммутатор должен иметь буфер памяти. Асимметричный коммутатор также необходим для обеспечения большей ширины полосы пропускания каналов между коммутаторами, осуществляемых через вертикальные кросс-соединения, или каналов между сегментами магистрали.

Буфер памяти

Для временного хранения фреймов и последующей их отправки по нужному адресу, коммутатор может использовать буферизацию. Буферизация может быть также использована в том случае, когда порт пункта назначения занят. Буфером называется область памяти, в которой коммутатор хранит передаваемые данные.

Буфер памяти может использовать два метода хранения и отправки фреймов: буферизация по портам и буферизация с общей памятью. При буферизации по портам пакеты хранятся в очередях (queue), которые связаны с отдельными входными портами. Пакет передаётся на выходной порт только тогда, когда все фреймы, находившиеся впереди него в очереди, были успешно переданы. При этом возможна ситуация, когда один фрейм задерживает всю очередь из-за занятости порта его пункта назначения. Эта задержка может происходить даже в том случае, когда остальные фреймы могут быть переданы на открытые порты их пунктов назначения.

При буферизации в общей памяти все фреймы хранятся в общем буфере памяти, который используется всеми портами коммутатора. Количество

памяти, отводимой порту, определяется требуемым ему количеством. Такой метод называется динамическим распределением буферной памяти. После этого фреймы, находившиеся в буфере, динамически распределяются по выходным портам. Это позволяет получить фрейм на одном порте и отправить его с другого порта, не устанавливая его в очередь.

Коммутатор поддерживает карту портов, в которые требуется отправить фреймы. Очистка этой карты происходит только после того, как фрейм успешно отправлен. Поскольку память буфера является общей, размер фрейма ограничивается всем размером буфера, а не долей, предназначенной для конкретного порта. Это означает, что крупные фреймы могут быть переданы с меньшими потерями, что особенно важно при асимметричной коммутации, то есть когда порт с шириной полосы пропускания 100 Мб/с должен отправлять пакеты на порт 10 Мб/с.

Возможности и разновидности коммутаторов

Коммутаторы подразделяются на управляемые и неуправляемые (наиболее простые).

Более сложные коммутаторы позволяют управлять коммутацией на сетевом (третьем) уровне модели OSI. Обычно их именуют соответственно, например «Layer 3 Switch» или сокращенно «L3 Switch». Управление коммутатором может осуществляться посредством Web-интерфейса, интерфейса командной строки (CLI), протокола **SNMP**, **RMONи** т. п. Многие управляемые коммутаторы позволяют настраивать дополнительные функции: VLAN, QoS, агрегирование,зеркалирование. Многие коммутаторы уровня доступа обладают такими расширенными возможностями, как сегментация трафика между портами, контроль трафика на предмет штормов, обнаружение петель, ограничение количества изучаемых mac-адресов, ограничение входящей/исходящей скорости на портах, функции списков доступа и т.п. Сложные коммутаторы можно объединять в одно логическое устройство — стек — с целью увеличения числа портов. Например, можно объединить 4 коммутатора с 24 портами и получить логический коммутатор с 90 ((4*24)-6=90) портами либо с 96 портами (если для стекирования используются специальные порты).

Лабораторная работа № 14

Проектирования и программирования IP маршрутизации

Цель работы:

- Понятия маршрутизации и принципы маршрутизации данных. Программирования маршрутизации в сети.
- Принципы проектирования и программирования IP маршрутизации

1. Практическая часть

Настраиваем связь двух сетей через маршрутизатор

Построим такую *сеть* (Рис.1).

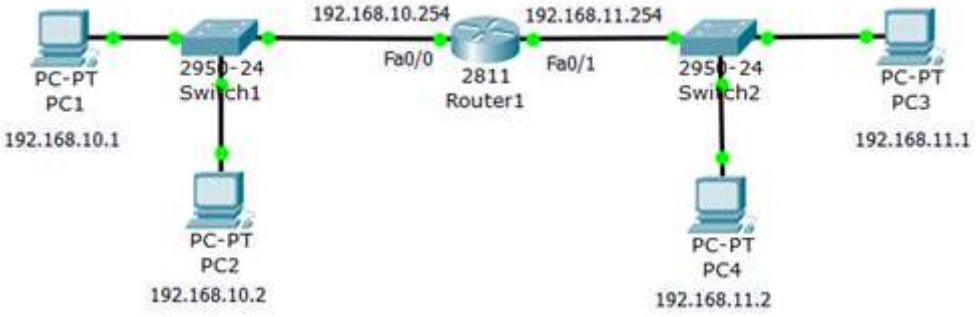


Рис.1. Постановка задачи

Наша цель – настроить *связь* двух сетей через *маршрутизатор* (роутер).

Шаг 1. Настройка ПК

Настраиваем компьютеры подсети 192.168.10.0 (Рис.2).

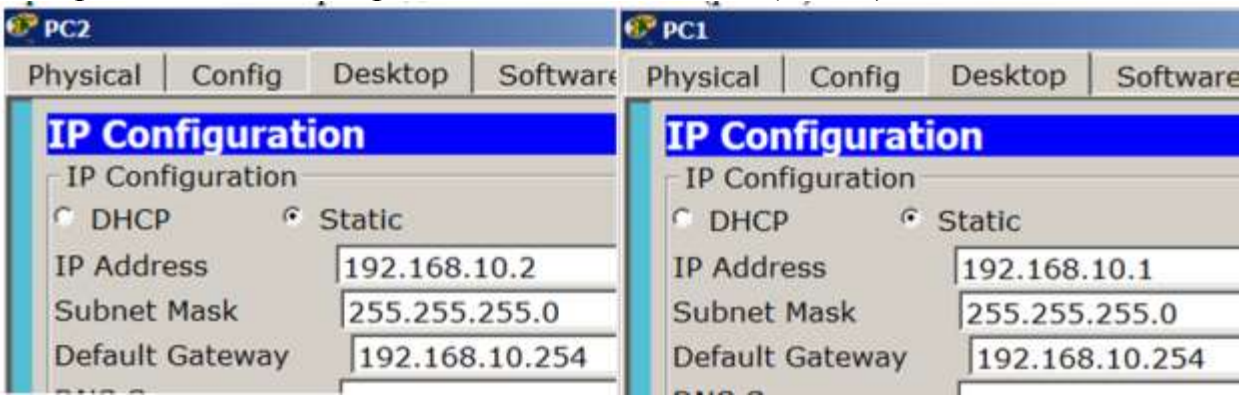


Рис.2. Настраиваем компьютеры подсети 192.168.10.0

Настраиваем компьютеры подсети 192.168.11.0 (Рис.3).

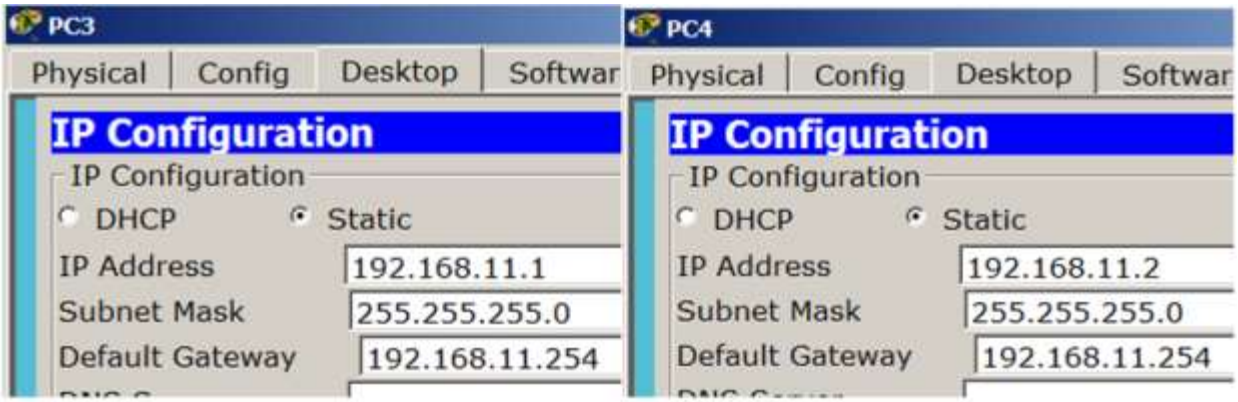


Рис.3. Настраиваем компьютеры подсети 192.168.11.0

Шаг 2. Настройка роутера (маршрутизатора)

Настраиваем роутер (маршрутизатор)как шлюз 192.168.10.254 для первой сети на интерфейсе Fa0/0 (Рис.4).

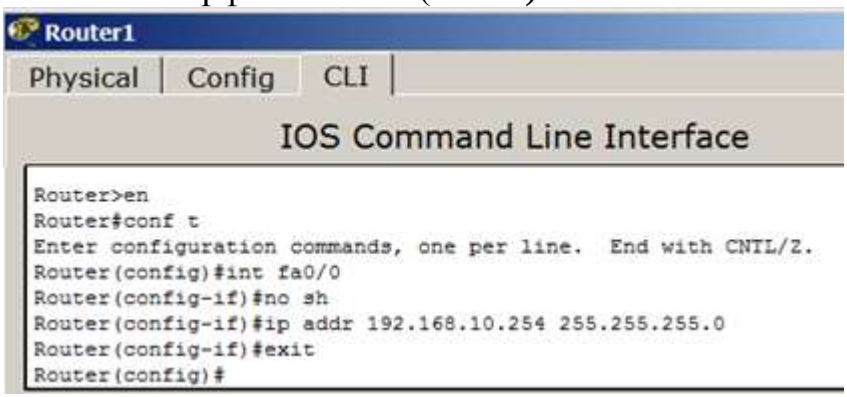


Рис.4. Окно ввода команд

Примечание

Здесь описаны следующие команды: привилегированный режим, режим конфигурирования, заходим на интерфейс, включаем этот интерфейс, задаем IPадрес и маску порта, выходим.

Аналогично настраиваем роутер как шлюз 192.168.11.254 для второй сети на интерфейсе Fa0/1 (Рис.5).

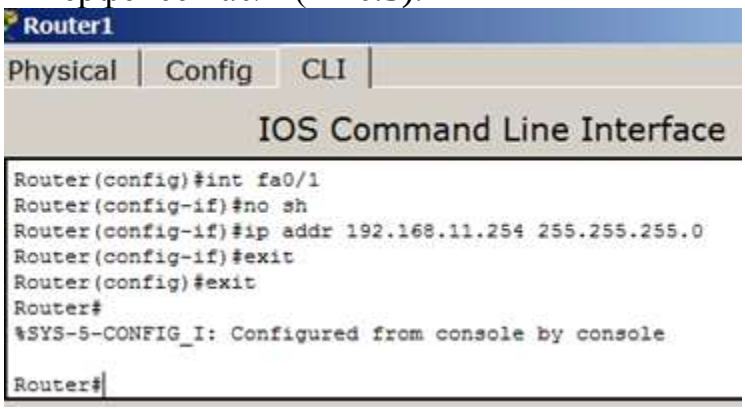


Рис.5. Настраиваем R1как шлюз 192.168.11.254 для второй сети

Шаг 3. Проверка связи сетей

Проверяем таблицу маршрутизации командой **show ip route** (Рис.6).

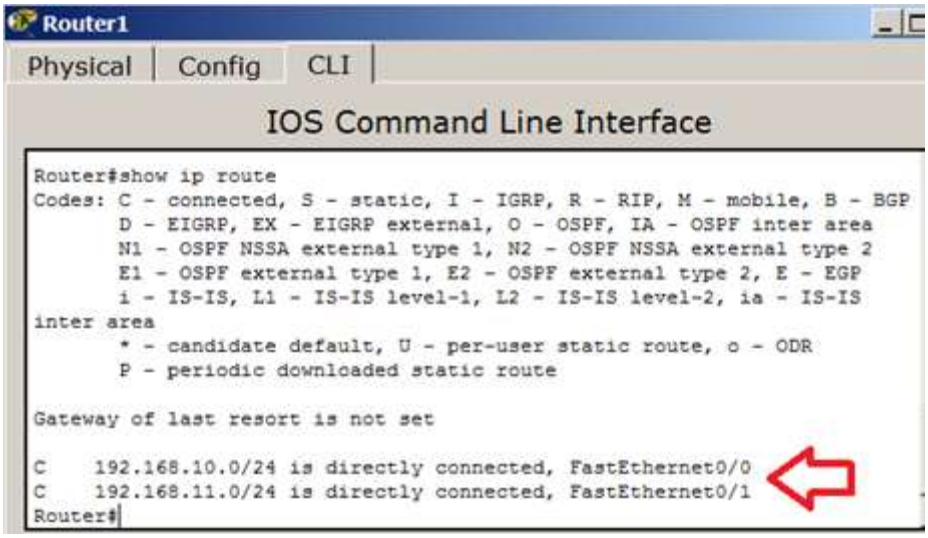


Рис.6. Проверяем таблицу маршрутизации роутера R1

У нас роутер обслуживает две сети. Проверяем связь роутера и ПК (Рис.7).

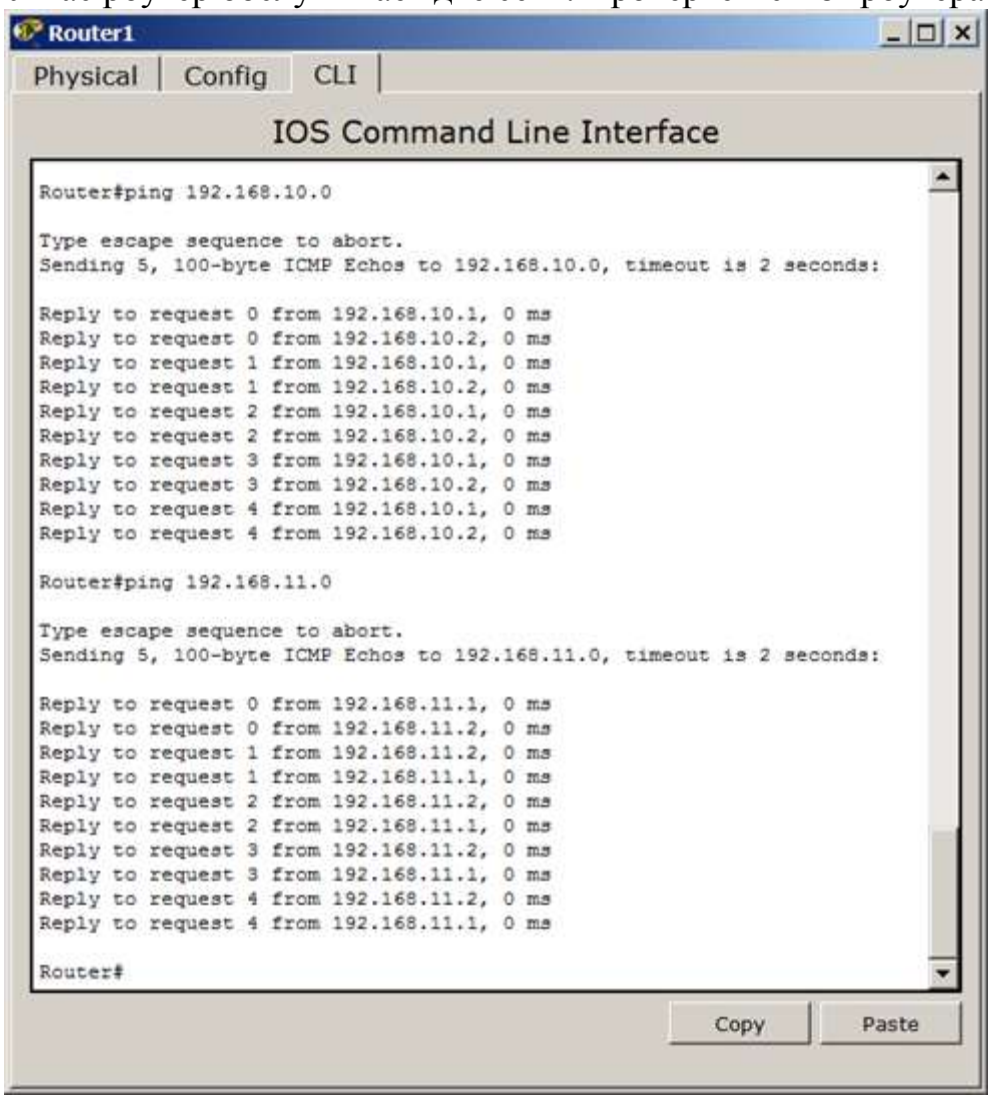


Рис.7. Связь роутера со всеми ПК есть

Проверяем связь роутера с подсетями (Рис.8).

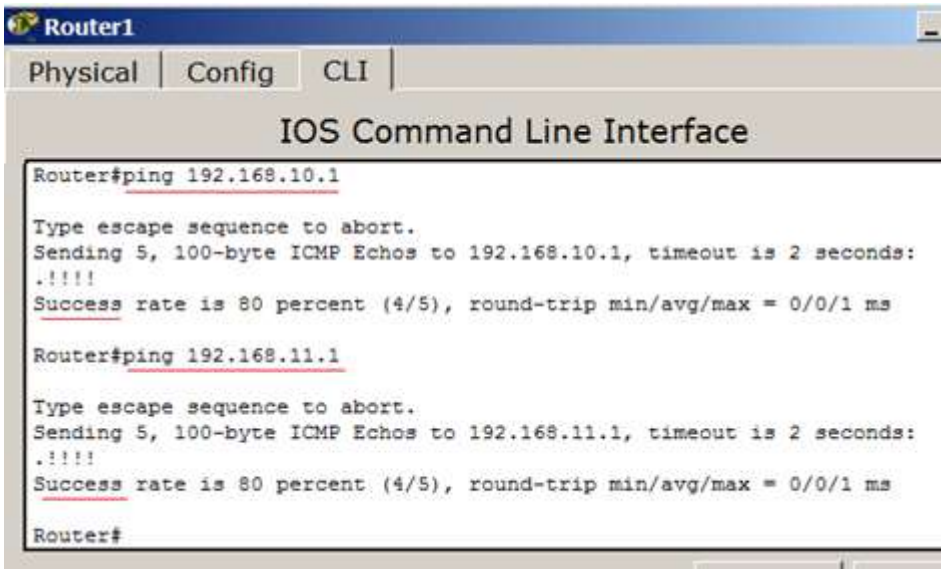


Рис.8. Проверяем связь роутера с подсетями

Примечание

Команда ping посылает ICMP эхо-пакеты для верификациисоединения. В приведённом выше примере время прохождения одного эхо-пакета превысило заданное, о чём свидетельствует точка (.) в выведенной информации, а четыре пакета прошли успешно, о чём говорит восклицательный знак (!).

Проверим также связь ПК из разных сетей между собой (Рис.9).

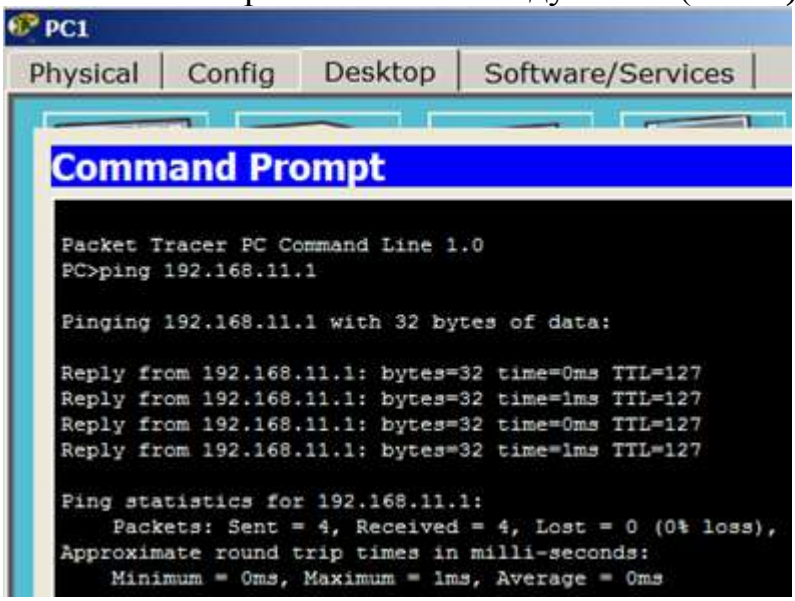


Рис.9. Проверка связи PC1иPC3

Примечание

Как выглядит порт маршрутизатора физически показано на Рис.10. Как видите, в него вставляется кабель с разъемом RJ-45.

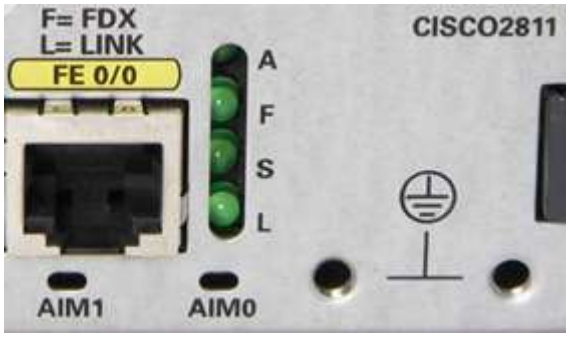


Рис.10. Ethernetport 0/0 маршрутизатора CISCO 2811

Задание 1. Настройка статической маршрутизации на оборудовании Cisco

Схема сети показана на Рис.11.

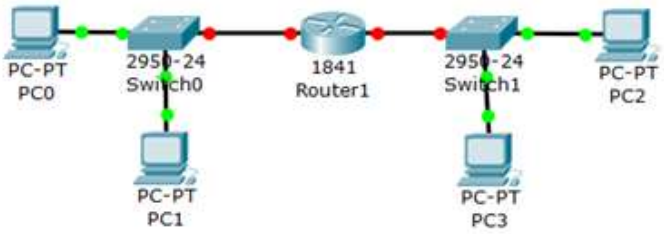


Рис.11. Схема сети

Студент должен:

1. Выполнить весь пример по настройке связи двух сетей
2. Покажите преподавателю Шаг 1. Настройку ПК
3. Покажите преподавателю Шаг 2. Настройку роутера (маршрутизатора)
4. Покажите преподавателю Шаг 3. Проверку связи сетей
5. Какой протокол следит за тем, чтобы в сети не было повторения IP адресов? (ARP)
6. Как шлюз по умолчанию для узлов сети связан с портами маршрутизатора?

В процессе выполнения задания необходимо:

1. Задать IP адреса сетевым интерфейсам маршрутизаторов, интерфейсам управления коммутаторов и сетевым интерфейсам локальных компьютеров;
2. Установить связь на физическом и канальном уровнях между соседними маршрутизаторами по последовательному сетевому интерфейсу;
3. Добиться возможности пересылки данных по протоколу IP между соседними объектами сети (C1-S1, C1-R1, S1-R1, R1-R2, R2-S2, R2-C2, и т.д.);
4. Настроить на маршрутизаторе R2 статические маршруты к сетям локальных компьютеров C1, C3
5. Настроить на маршрутизаторах R1, R3 маршруты "по умолчанию" к сетям локальных компьютеров C2-C3 и C1-C2 соответственно;

6. Добиться возможности пересылки данных по протоколу IP между любыми объектами сети (ping);
7. Переключившись в "Режим симуляции" рассмотреть и пояснить процесс обмена данными по протоколу ICMP между устройствами (выполнив команду Ping с одного компьютера на другой), пояснить роль протокола ARP в этом процессе

Специальные термины и понятия

Маршрутизатором (шлюзом), называется *узел сети* с несколькими IP-интерфейсами (содержащими свой MAC-*адрес* и IP-*адрес*), подключенными к разным IP-сетям, осуществляющий на основе решения задачи маршрутизации перенаправление дейтаграмм из одной сети в другую для доставки от отправителя к получателю. Как уже отмечалось, ***динамическая маршрутизация*** — это процесс протокола маршрутизации, определяющий взаимодействие устройства с соседними маршрутизаторами. *Маршрутизатор*будет обновлять сведения о каждой подключенной к нему сети. Если в сети произойдет изменение, протокол динамической маршрутизации автоматически информирует об изменении все маршрутизаторы. Если же используется ***статическая маршрутизация,*** обновить таблицы маршрутизации на всех устройствах придется системному администратору. Статическая*маршрутизация* позволяет сократить объем таблиц маршрутизации в конечных узлах и маршрутизаторах за счет использования в качестве номера сети назначения так называемого ***маршрута по умолчанию*** – *default* (0.0.0.0), который обычно занимает в таблице маршрутизации последнюю строку. Если в таблице маршрутизации есть такая *запись*, то все пакеты с номерами сетей, которые отсутствуют в таблице маршрутизации, передаются маршрутизатору, указанному в строке *default*.

Новый термин

Шлюз по умолчанию (defaultgateway) - адрес маршрутизатора, на который отправляется трафик для которого не нашлось отдельных записей в таблице маршрутизации. Для устройств, подключенных к одному маршрутизатору (как правило, это рабочие станции) использование шлюза по умолчанию — единственная форма маршрутизации.

Доступность компьютера проверяется при помощи посылки контрольного диагностического сообщения по протоколу ***ICMP***(*Internet Control* Message *Protocol*), по которому любая оконечная станция должна выдать эхо-ответ узлу, отправившему такое сообщение. В сетях на основе *TCP/IP* для проверки соединений обычно используется ***утилита ping***. Эта *программа* отправляет запросы (*ICMP* Echo-Request) протокола *ICMP* узлу сети с указанным IP-адресом. Получив этот *запрос*, исследуемый узел должен послать пакет с ответом (*ICMP* Echo-Reply). Первый узел фиксирует поступающие ответы. ***Время между отправкой запроса и получением ответа*** (*RTT*, от англ. Round Trip *Time*) позволяет определять двусторонние задержки (*RTT*) по маршруту и частоту потери пакетов, то есть косвенно определить загруженность

каналов передачи данных и промежуточных устройств. *Метрика* — числовой коэффициент, влияющий на выбор маршрута в компьютерных сетях. Как правило, определяется количеством "хопов" (ретрансляционных переходов) до сети назначения или параметрами канала связи. Чем *метрика* меньше, тем *маршрут* приоритетнее.***Петля маршрутизации*** — явление, возникающее, когда *маршрутизатор* отсылает пакет на неверный *адрес* назначения. Получивший такой пакет *маршрутизатор* возвращает его обратно. Таким образом получается петля. Для борьбы с подобными петлями в *TCP/IP*предусмотрен механизм *TTL*. Протоколы маршрутизации так же предлагают свои способы борьбы с петлями.

Практическая работа 2-2. Настройка трех сетей с WEB сервером. Понятие маршрута по умолчанию

Схема у нас будет следующая:два коммутатора 2950-24, два ПК в сети 192.168.10.0 с маской 255.255.255.0. *Сервер* и *компьютер* в сети 192.168.20.0 с маской 255.255.255.0. *Сеть* между маршрутизаторами (марки 1841) 192.168.1.0 с маской 255.255.255.252. Компьютеры из сети 192.168.10.0 должны достучаться к DNSсерверу в сети 192.168.20.0 (Рис.12).

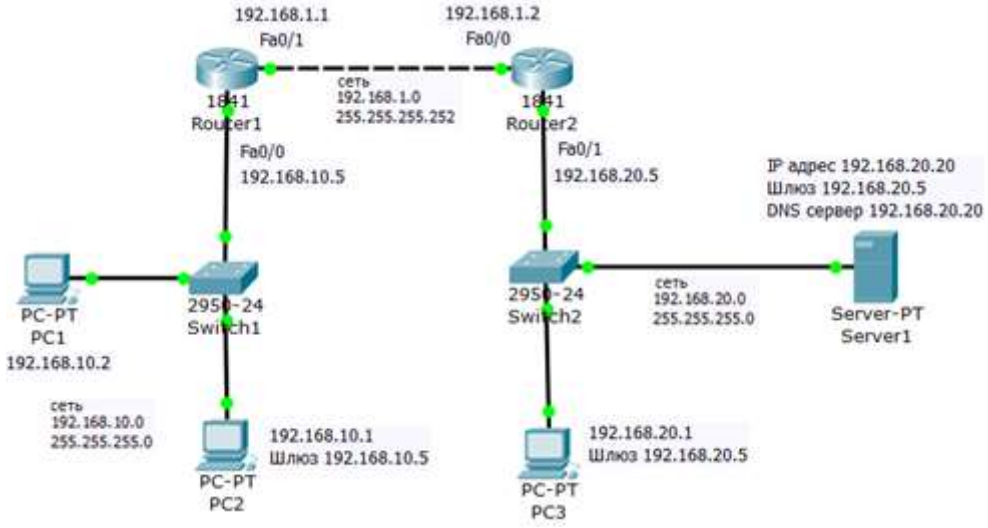


Рис.12. Постановка задачи

Сеть у нас не сложная, ПК в ней немного, поэтому будем использовать не динамическую, а статическую маршрутизацию.

Настройки сетевых интерфейсов роутеров

Будем настраивать связь роутеров через порты Fa0/1для R1 и Fa0/0 для R2. Настраиваем Router1 исходя из постановки задачи о том, что сеть между маршрутизаторами192.168.1.0 с маской 255.255.255.252. Поэтому порту Fa0/1 присвоим IP адрес 192.168.1.1 (Рис.13).

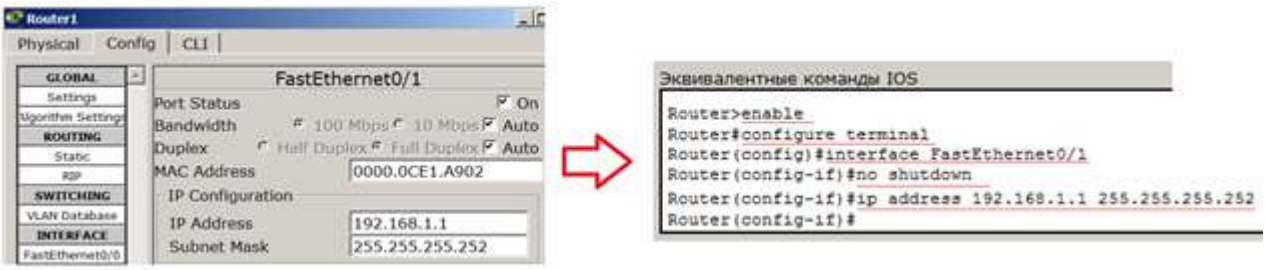


Рис.13. Настраиваем порт 0/1для маршрутизатораR1

Важно

При конфигурировании через webинтерфейс обязательно установите флажок **On** (Вкл.), что эквивалентно команде **nosh**.

Примечание

Как вариант, все параметры маршрутизатор можно настроить из командной строки на вкладке CLI следующими командами: **enable**(включаем привилегированный режим), **config terminal** (входим в режим конфигурации), **interface fastethernet0/1** (настраиваем интерфейс 100 мб Ethernet 0/1), **ip address 192.168.1.1 255.255.255.252** (прописываем ip адрес интерфейса и маску сети маршрутизатора), **no shutdown** (включаем интерфейс - по умолчанию все выключено), **exit** (выходим из режима конфигурирования интерфейса), **end** (закончили редактирование), **write** (сохранили конфигурацию).

Аналогично настраиваем Router2 исходя из постановки задачи о том, что сеть между маршрутизаторами192.168.1.0 с маской 255.255.255.252. Порту Fa0/0 присвоим IP адрес 192.168.1.2 (Рис.14).

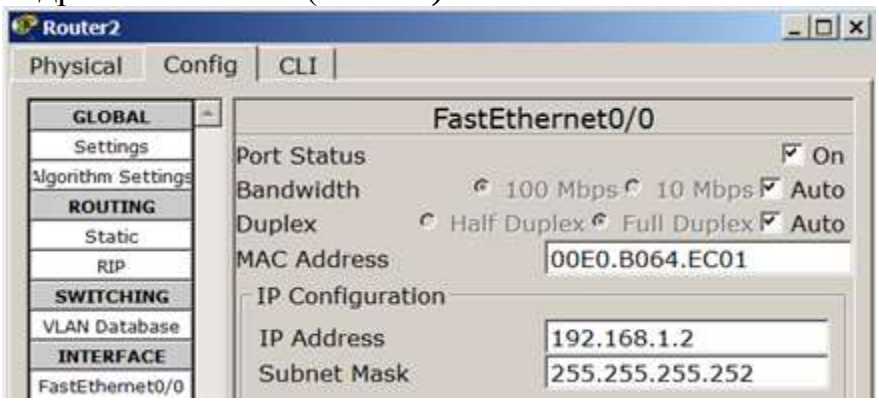


Рис.14. Конфигурируем R2

Примечание

При конфигурировании роутера из командной строки можно использовать сокращенную форму записи команд: **en** (включаем расширенный режим). **conf t** (входим в режим конфигурации). **int fa0/0** (настраиваем интерфейс 100 мб. Ethernet 0/0). **Ip addr192.168.1.2 255.255.255.252** (прописываем ip адрес интерфейса и маску сети). **No shut** (включаем интерфейс - по умолчанию он выключен). **exit** (выходим из режима конфигурирования интерфейса). **end** (заканчиваем редактирование). **wr**(сохраняем конфигурацию).

В итоге после настройки маршрутизаторов на портах загораются зеленые маркеры, то есть, связь между ними есть. Сеть между маршрутизаторами работает, но маршрутизации пока нет, то есть, из одной сети в другую попасть нельзя.

Настройка связи маршрутизаторов с подсетями (настройка шлюзов)

Настроим порт Fa0/0 маршрутизатора R1 на работу с сетью 192.168.10.0 (Рис.15).

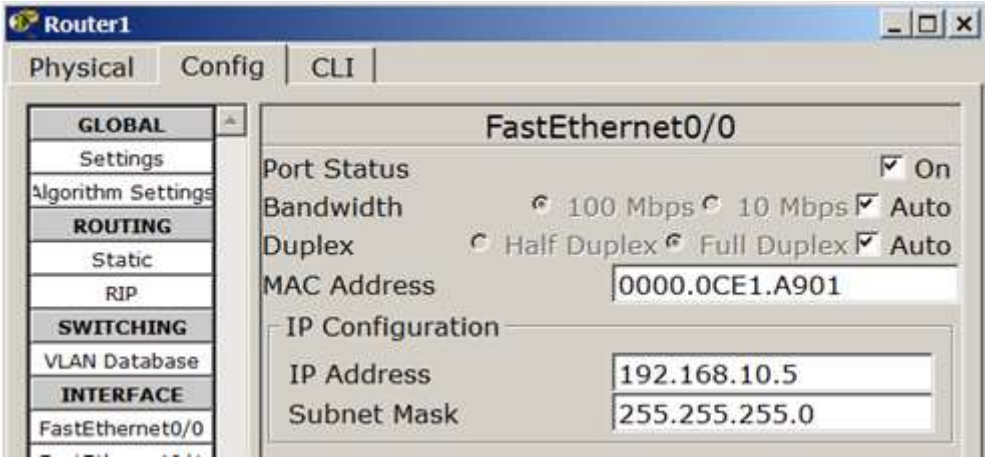


Рис.15. Настроим портFa0/0 маршрутизатораR1 на работу с сетью 192.168.10.0

Аналогично порт Fa0/1 маршрутизатора R2 настроим на работу с сетью 192.168.20.0 (Рис.16).

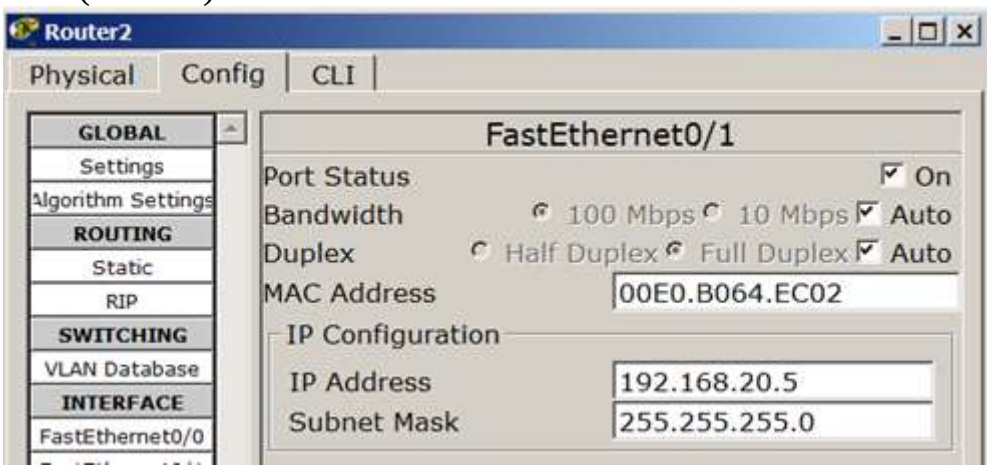


Рис.16. Порт Fa0/1 маршрутизатораR2 настроим на работу с сетью 192.168.20.0

Как теперь видно по маркерам – сеть поднялась (Up), то есть все индикаторы горят зеленым цветом.

Настройка PC1 и PC2

Продолжим работу и настроим компьютеры в сети 192.168.10.0, то есть, нужно задать IP компьютеров, маску сети и основной шлюз. По исходным условиям задачи у нас слева пара компьютеров в сети 192.168.10.0 с маской 255.255.255.0 (Рис.17).

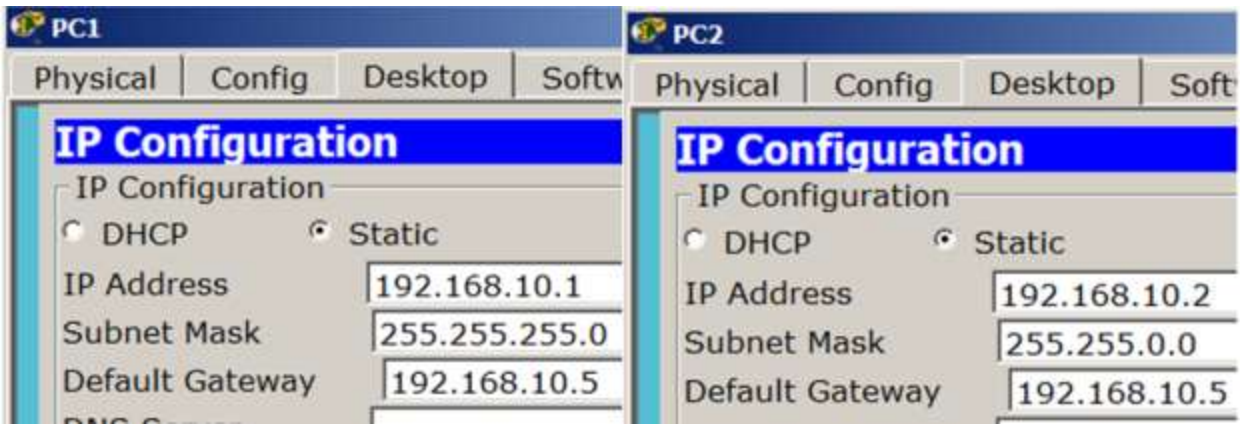


Рис.17. Настраиваем PC1 и PC2

Новый термин

Основной шлюз (Default Gateway) – это адрес, куда компьютер отправляет пакет, если не знает, куда его отправить. Например, при попытке узла Б отправить данные узлу А. в отсутствие конкретного адреса к узлу А, узел Б направляет трафик TCP/IP, предназначенный для узла А, своему основному шлюзу.

Настройка сервера и PC3

Далее нужно настроитьPC3 и сервер в сети 192.168.20.0 (Рис.18 и Рис.19).

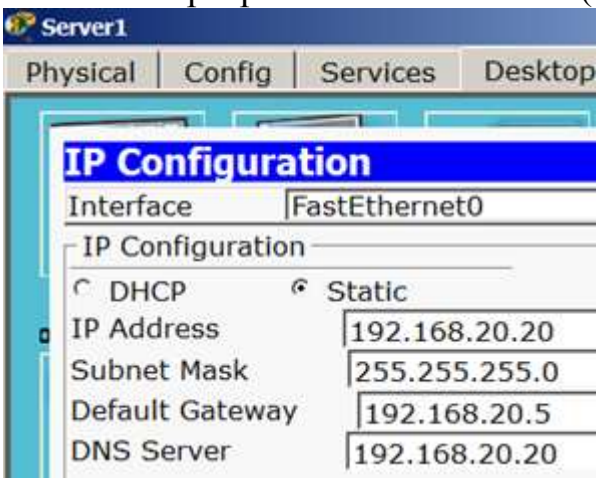


Рис.18. Настройка сервера

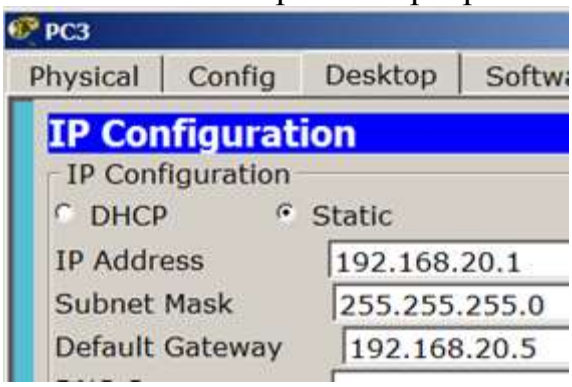


Рис.19. НастраиваемPC2

Настройка маршрутизации на маршрутизаторах (маршрута по умолчанию)

Можете пропинговать сети и убедиться в том, что ситуация такая: запросы из сети …10.0 в сеть…20.0 проходят, а ответов – нет. Поэтому надо прописать на маршрутизаторах маршруты по умолчанию. Вспомним, что порту Fa0/1 мы присвоили IP адрес 192.168.1.1, а порту Fa0/0 – адрес 192.168.1.2. Поэтому на маршрутизаторе R1 для порта Fa0/1 с IP адресом 192.168.1.1 следует выполнить такие команды (Рис.20).

Рис.20. Прописываем маршрут по умолчанию на R1

Примечание

Запись означает, что все запросы, для которых не прописаны маршруты, R1 посылает на 192.168.1.2, то есть, на R2.

Для R2 поступаем аналогично (Рис.21).

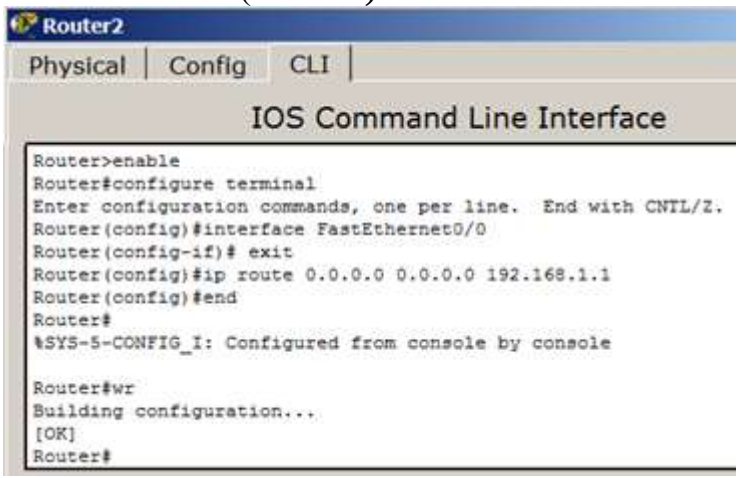


Рис.21. Прописываем маршрут по умолчанию на R2

Примечание

Запись означает, что все запросы, для которых не прописаны маршруты, R2 отправляет на 192.168.1.1, то есть, на R1.

Проверяем работу сети

После настройки роутеров можно протестировать сеть, для этого нужно пропинговать компьютерами из одной сети — компьютеры из другой сети (Рис.22).

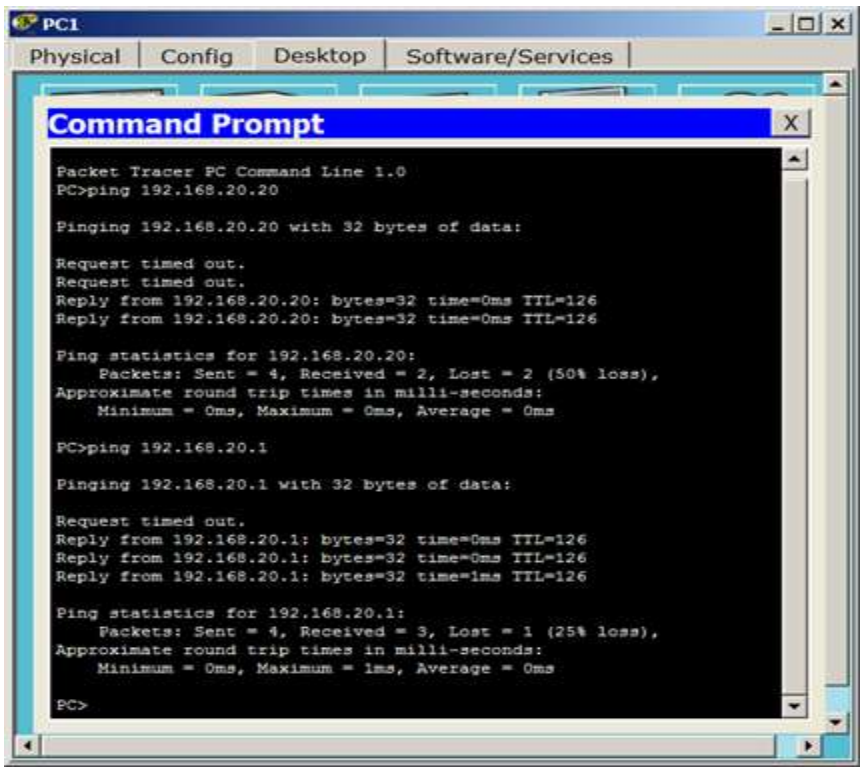


Рис.22. Связь не идеальная, но есть

Чтобы убедиться наверняка, давайте посмотрим, как идут пакеты по узлам сети и для этого воспользуемся командой **tracert 192.168.20.20** (Рис.23).

Примечание

Tracert — команда, предназначенная для определения маршрутов следования данных в сетях TCP/IP.

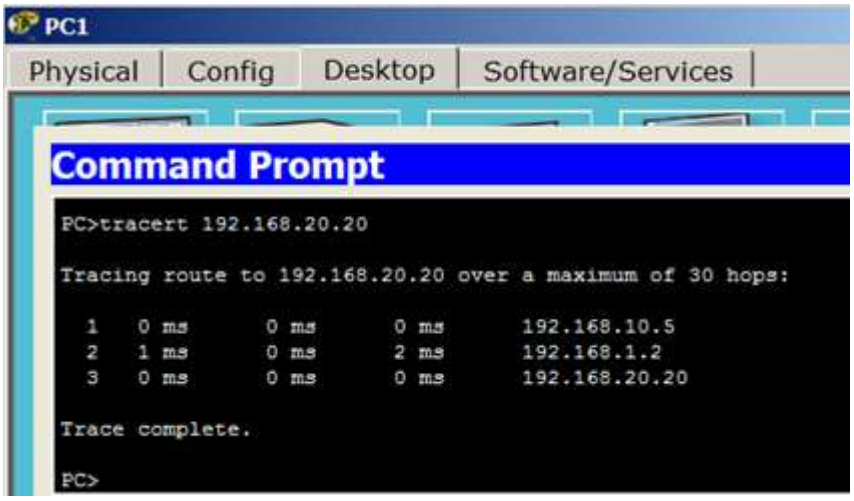


Рис.23. Наблюдаем как идут пакеты между сегментами сети от PC1 на сервер

Как видно из скриншота пакеты сначала уходят на адрес 192.168.10.5 (R1–порт Fa0/0), далее на адрес 192.168.1.2 (R2 – порт Fa0/0), а дальше приходит на сервер 192.168.20.20 — все верно!

Примечание

Web страниц на сервере мы не создавали, но они там существуют изначально, по умолчанию. Запустите Web Browser и убедитесь в этом самостоятельно (Рис.24).

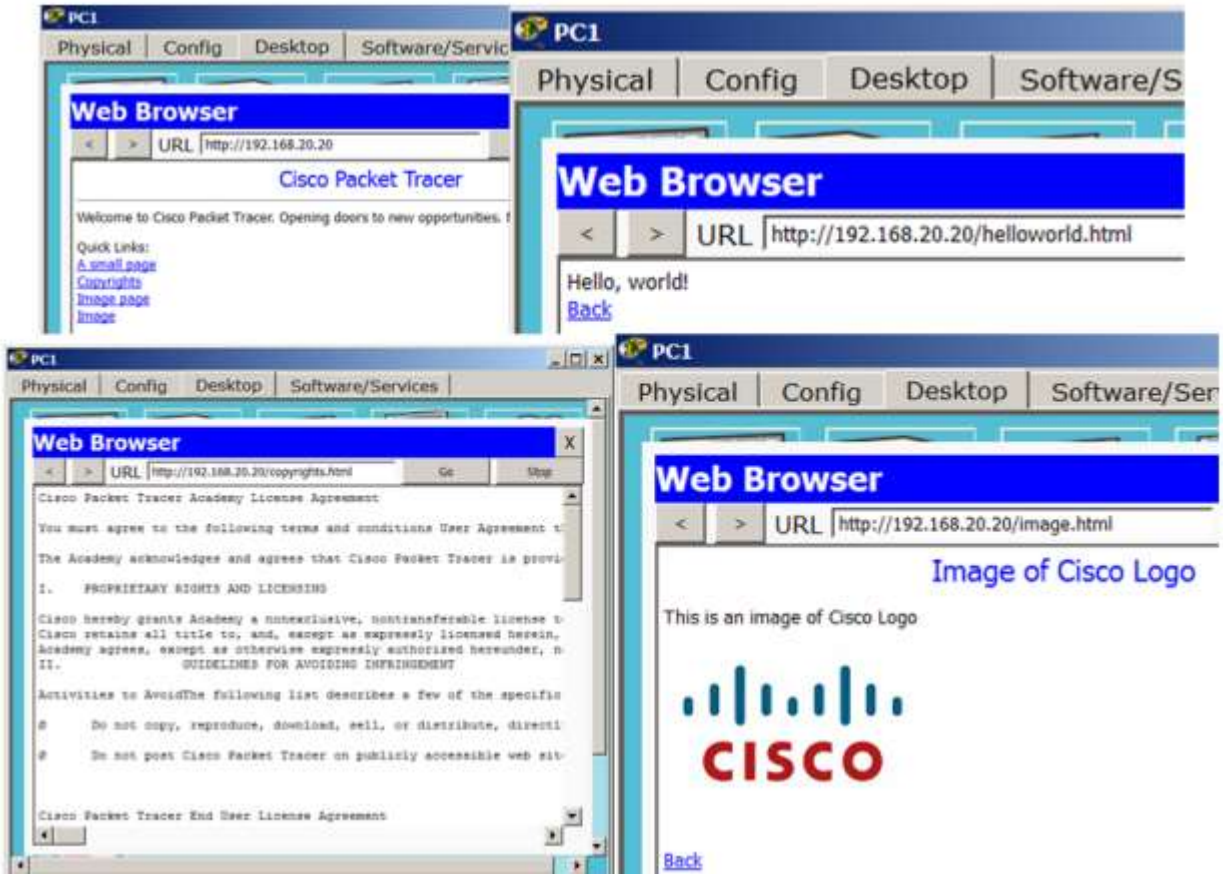


Рис.24. На сервере работает служба HTTP

Практическая работа 3. Сеть на двух маршрутизаторах

Далее мы изучим статическую маршрутизацию в локальных сетях, рассмотрев этот вопрос на двух практических примерах.

Схема сети для настройки статической маршрутизации приведена на Рис.25.

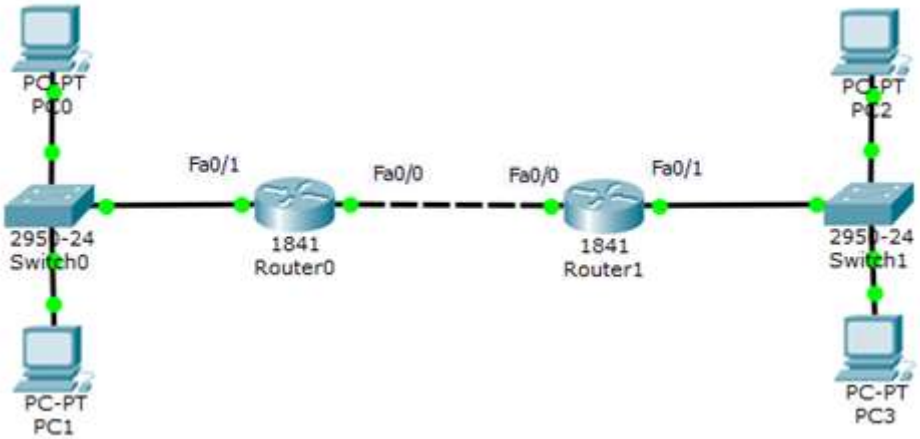


Рис.25. Схема сети

Если сейчас командой show ip route посмотреть таблицу маршрутизации на R0 и R1, то увидим следующее (Рис.26 и Рис.27).

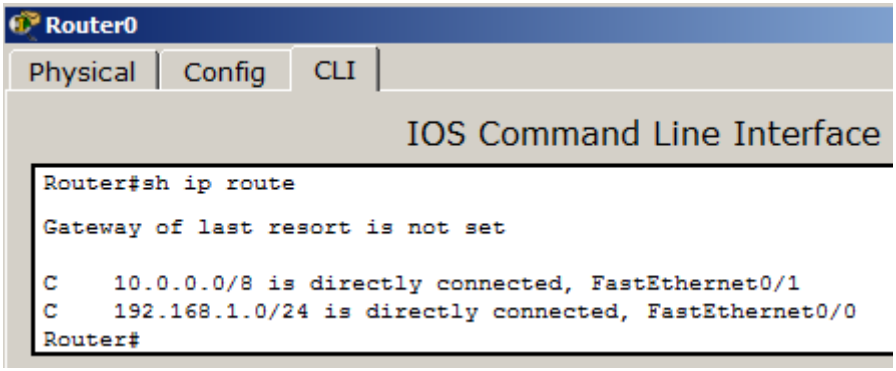


Рис.26. Таблица маршрутизации на 1-ммаршрутизаторе

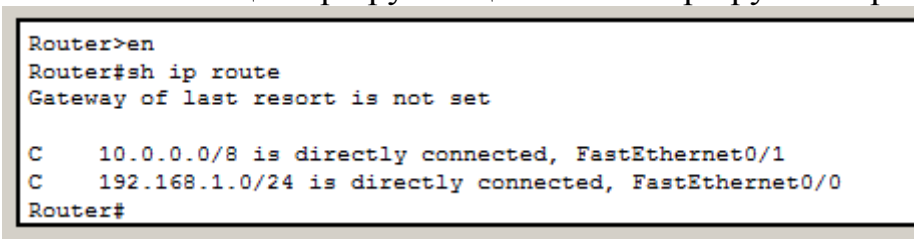


Рис.27. Таблица маршрутизации на 2-ммаршрутизаторе

Мы видим, что в данный момент в нашей таблице есть только сети, подключенные напрямую. R0 не знает *сеть* 10.1.2.0, а R1 не знает *сеть* 10.1.1.0. Поэтому, чтобы настроить маршрутизацию, следует добавим эти маршруты в таблицы маршрутизаторов:

R0 (config)#ip route 10.1.2.0 255.255.255.0 192.168.1.2

R1 (config)#ip route 10.1.1.0 255.255.255.0 192.168.1.1

Теперь снова выведем таблицы маршрутизации наших устройств (Рис.28).

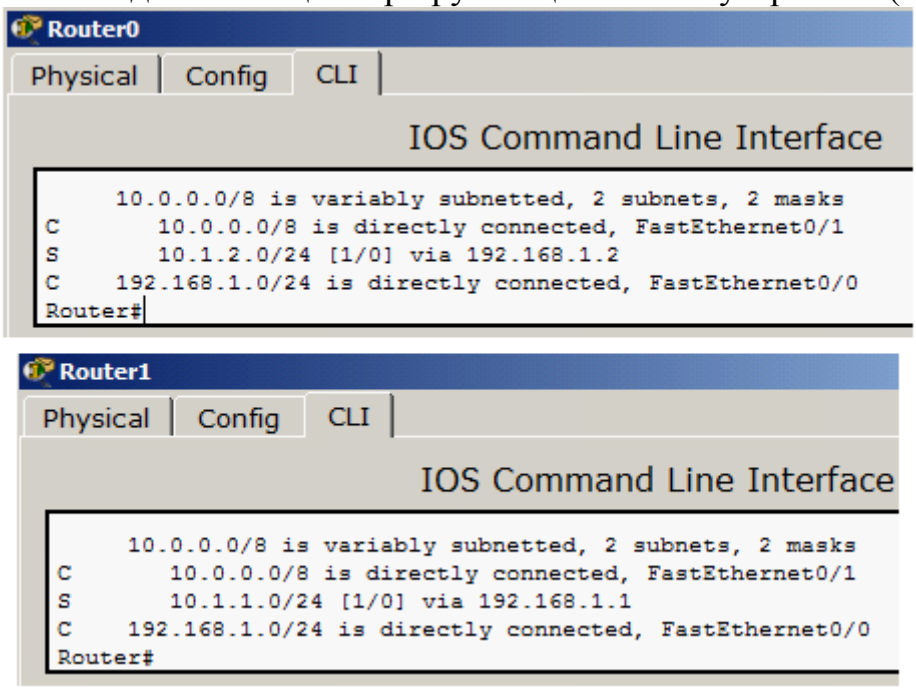


Рис.28. Маршрутизация настроена

Теперь 1-й *маршрутизатор* знает, что пакеты, направляемые в *подсеть* 10.1.2.0 можно переслать маршрутизатору с ip адресом 192.168.1.2,

а 2-й *маршрутизатор* знает, что пакеты, направляемые в *подсеть* 10.1.1.0 можно переслать маршрутизатору с ip адресом 192.168.1.1. Проверяем *связь* ПК из разных сетей (Рис.29).

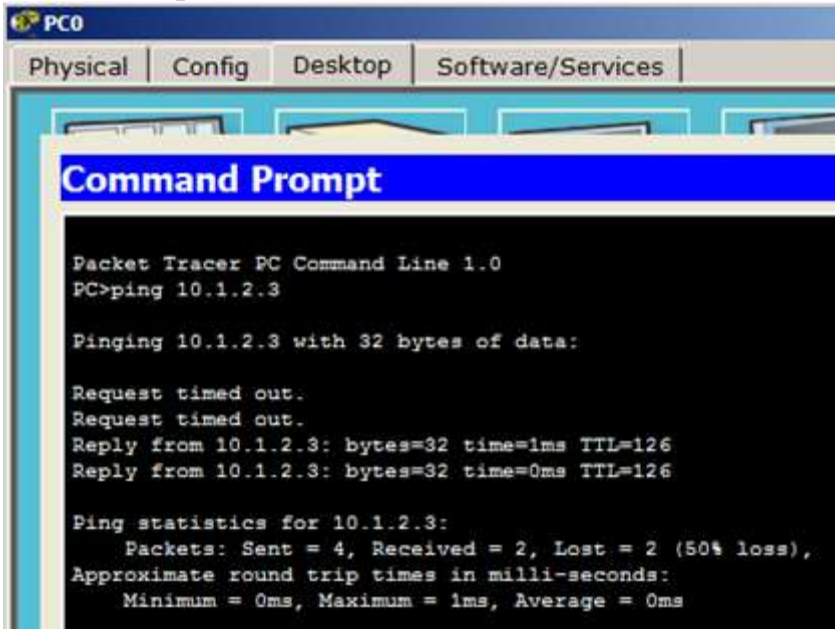


Рис.29. Статическая маршрутизация настроена – PC0 может общаться с PC3

Статическая маршрутизация для пяти сетей и роутеров с тремя портами
В этом примере мы соберем и настроим следующую схему сети (Рис.30).
Схема сети
На данной схеме имеется пять сетей: 192.168.1.0, 172.20.20.0, 192.168.100.0, 10.10.10.0 и 192.168.2.0. В качестве шлюза по умолчанию у каждого компьютера указан интерфейс маршрутизатора, к которому он подключен. Маска у всех ПК одна - 255.255.255.0. Маска маршрутизаторов для каждого порта своя: Fa0/0 -255.255.255.0, Fa0/1 - 255.255.0.0, Fa1/0 - 255.255.255.252.
На данной схеме имеется пять сетей: 192.168.1.0, 172.20.20.0, 192.168.100.0, 10.10.10.0 и 192.168.2.0. В качестве шлюза по умолчанию у каждого компьютера указан интерфейс маршрутизатора, к которому он подключен. Маска у всех ПК одна - 255.255.255.0. Маска маршрутизаторов для каждого порта своя: Fa0/0 -255.255.255.0, Fa0/1 - 255.255.0.0, Fa1/0 - 255.255.255.252.

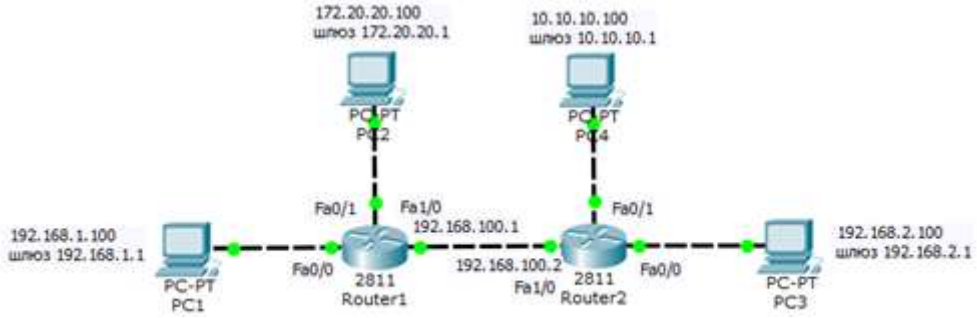


Рис.30. Связь сетей посредством маршрутизаторов

Далее соединим маршрутизаторы между собой нам потребуется добавить к маршрутизатору интерфейсную плату NM-1FE-TX (NM – Network module,

1FE – содержит один порт FastEthernet, TX – поддерживает 10/100MBase-TX). Чтобы это сделать перейдите к окну конфигурации маршрутизатора0, выключите его, щелкнув на кнопке питания. После этого перетяните интерфейсную плату NM-1FE-TX в разъем маршрутизатора (Рис.31). После того как карта добавлена, еще раз щелкните по тумблеру маршрутизатора, чтобы включить его. Повторите аналогичные действия со вторым маршрутизатором.

Рис.31. Вставляем интерфейсную плату в маршрутизатор

Постановка задачи

Нам требуется произвести необходимые настройки для того, чтобы все ПК могли общаться друг с другом, то есть, необходимо обеспечить доступность компьютеров из разных сетей между собой.

Настройка маршрутизации (маршрута по умолчанию)

В настоящий момент если мы отправим с компьютера PC1 с IP адресом 192.168.1.100 пакет на интерфейс Fa1/0 с IP адресом 192.168.100.2 маршрутизатора R2, то ICMP пакет слева дойдет до этого маршрутизатора, но при отправке ICMP пакетов в обратном направлении с адреса 192.168.100.2 на адрес 192.168.1.100 возникнет проблема. Дело в том, что маршрутизатор R2 не имеет в своей таблице маршрутизации информации о сети 172.20.20.0, так как шлюз по умолчанию мы еще не прописывали и маршрутизатор R2 не знает, куда отправлять ответы на запрос. В небольших сетях самым простым способом настроить маршрутизацию, является добавление маршрута по умолчанию. Для того чтобы это сделать выполните на маршрутизаторе R1 в режиме конфигурирования следующие команды (Рис.32).

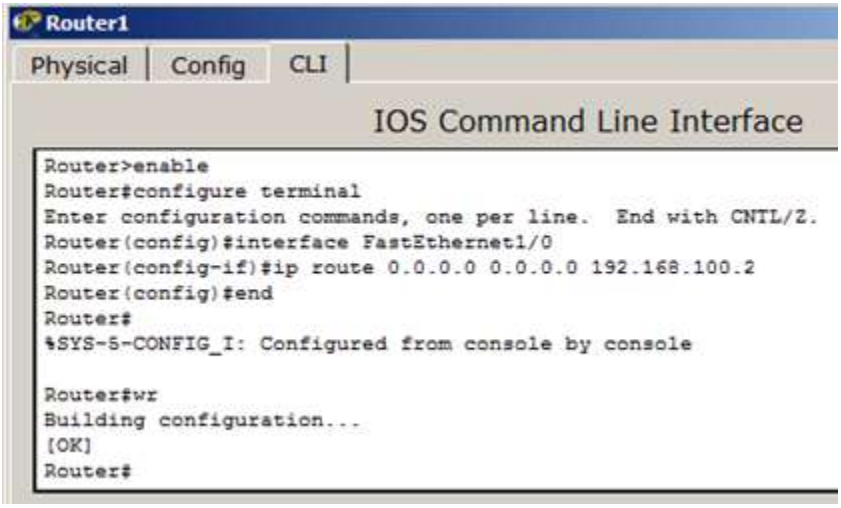


Рис.32. Настройка маршрута по умолчанию на R1

Примечание

В этих командах первая группа цифр 0.0.0.0 обозначают IP адрес сети назначения, следующая группа цифр 0.0.0.0 обозначает её маску, а последние цифры – 192.168.100.2 это IP адрес интерфейса, на который необходимо передать пакеты, чтобы попасть в данную сеть. Если мы указываем в качестве адреса сети 0.0.0.0 с маской 0.0.0.0, то данный маршрут становится маршрутом по умолчанию, и все пакеты, адреса назначения которых, прямо не указаны в таблице маршрутизации будут отправлены на него.

На правом маршрутизаторе R2 поступаем аналогично (Рис.33).

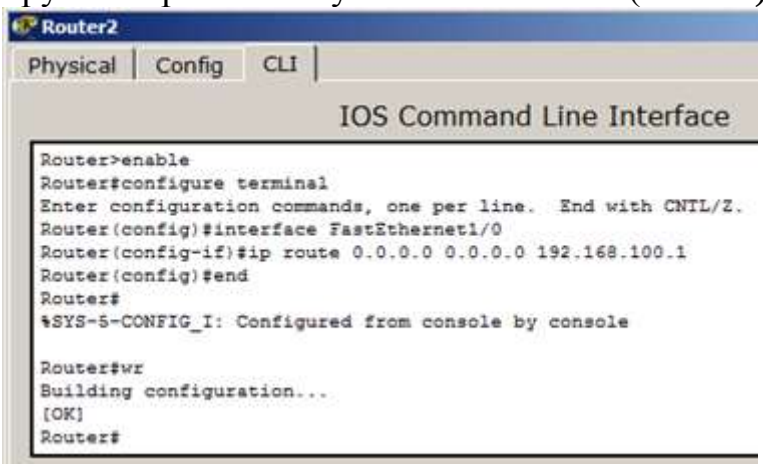


Рис.33. Настройка маршрута по умолчанию на R2

Отправим с компьютера PC1 с IP адресом 192.168.1.100 пакет на интерфейс Fa1/0 с IP адресом 192.168.100.2 маршрутизатора R2 и посмотрим, что изменилось (Рис.34).

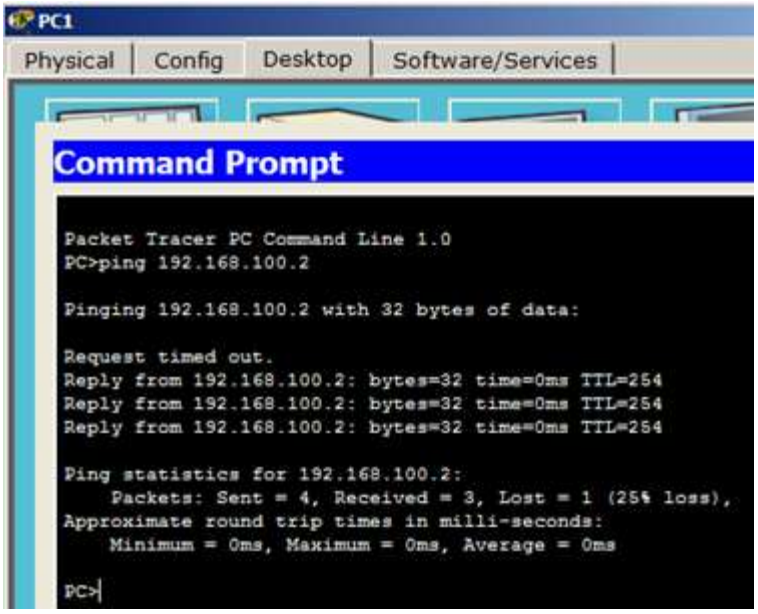


Рис.34. С компьютера PC1 с IP адресом 192.168.1.100 успешно пингуем интерфейс Fa1/0 с IP адресом 192.168.100.2 маршрутизатора R2

Резюме

Допустим, мы хотим пропинговать с компьютера PC1 с адресом 192.168.1.100 (из левой сети) компьютер PC4 с IP адресом 10.10.10.100 (из правой сети). В качестве шлюза по умолчанию на компьютере с адресом 192.168.1.100 установлен адрес 192.168.1.1интерфейса Fa0/0 маршрутизатора R1. Сначала компьютер будет искать в свой таблице маршрутизации адрес 10.10.10.100, а после того, как он его не найдет, ICMP пакеты будут отправлены на адрес по умолчанию, то есть на интерфейс маршрутизатора R1 с адресом 192.168.1.1 (порт Fa0/0). Получив пакет, маршрутизатор R1 просмотрит адрес его назначения – 10.10.10.100 и также попытается обнаружить его в свой таблице маршрутизации. Когда он не обнаружит и его, пакет будет отправлен на интерфейс Fa1/0, с адресом 192.168.100.2 маршрутизатора R2. Маршрутизатор R2 попробует обнаружить в свой таблице маршрутизации маршрут к адресу 10.10.10.100. Когда это не увенчается успехом, маршрутизатор будет искать маршрут к сети 10.0.0.0. Информация о данной сети содержится в таблице маршрутизации, и маршрутизатор знает, что для того чтобы попасть в данную сеть необходимо отправить пакеты на интерфейс FastEthernet0/1, непосредственно к которому подключена данная сеть. Так как в нашем примере вся сеть 10.0.0.0, представляет из себя всего 1 компьютер, то пакеты сразу же попадают в место назначения, то есть, на компьютер с IP адресом 10.10.10.100. При отсылке ответных ICMP пакетов, все происходит аналогичным образом. Однако, не всегда можно обойтись указанием только маршрутов по умолчанию. В более сложных сетевых конфигурациях может потребоваться прописывать маршрут для каждой из сетей в отдельности. Это будет непросто. Поэтому в больших сетях обычно используют не статическую, а динамическую маршрутизацию.

1. Теоретическая часть

Маршрутизация (англ. *Routing*) — процесс определения маршрута следования информации в сетях связи.

Маршруты могут задаваться административно (статические маршруты), либо вычисляться с помощью алгоритмов маршрутизации, базируясь на информации отопологии и состоянии сети, полученной с помощью протоколов маршрутизации(динамические маршруты).

Статическими маршрутами могут быть:

маршруты, не изменяющиеся во времени;

маршруты, изменяющиеся по расписанию;

Маршрутизация в компьютерных сетях выполняется специальными программно-аппаратными средствами — маршрутизаторами; в простых конфигурациях может выполняться и компьютерами общего назначения, соответственно настроенными.

Маршрутизируемые протоколы

Протокол маршрутизации может работать только с пакетами, принадлежащими к одному из маршрутизируемых протоколов, например, IP, IPX или Xerox Network System,AppleTalk. Маршрутизируемые протоколы определяют формат пакетов (заголовков), важнейшей информацией из которых для маршрутизации является адрес назначения. Протоколы, не поддерживающие маршрутизацию, могут передаваться между сетями с помощью туннелей. Подобные возможности обычно предоставляют программные маршрутизаторы и некоторые модели аппаратных маршрутизаторов.

Программная и аппаратная маршрутизация

Первые маршрутизаторы представляли собой специализированное ПО, обрабатывающее приходящие IP-пакеты специфичным образом. Это ПО работало на компьютерах, у которых было несколько сетевых интерфейсов, входящих в состав различных сетей (между которыми осуществляется маршрутизация). В дальнейшем появилисьмаршрутизаторы в форме специализированных устройств. Компьютеры с маршрутизирующим ПО называют*программные маршрутизаторы*, оборудование — *аппаратные маршрутизаторы*.

В современных аппаратных маршрутизаторах для построения таблиц маршрутизации используется специализированное ПО («прошивка»), для обработки же IP-пакетов используется коммутационная матрица (или другая технология аппаратной коммутации), расширенная фильтрами адресов в заголовке IP-пакета.

Аппаратная маршрутизация

Выделяют два типа аппаратной маршрутизации: со статическими шаблонами потоков и с динамически адаптируемыми таблицами.

Статические шаблоны подразумевают разделение всех входящих в маршрутизатор IP-пакетов на виртуальные потоки; каждый поток характеризуется набором признаков для пакета такие как: *IP-адресами отправителя/получателя*, *TCP/UDP-порт отправителя/получателя* (в

случае поддержки маршрутизации на основании информации 4 уровня), *порт*, через который пришёл пакет.

Оптимизация маршрутизации при этом строится на идее, что все пакеты с одинаковыми признаками должны обрабатываться одинаково (по одинаковым правилам), при этом признаки проверяются только для первого пакета в потоке (при появлении пакета с набором признаков, не укладывающимся в существующие потоки, создаётся новый поток), по результатам анализа этого пакета формируется статический шаблон, который и используется для определения правил коммутации приходящих пакетов (внутри потока). Обычно время хранения неиспользующегося шаблона ограничено (для освобождения ресурсов маршрутизатора). Ключевым недостатком подобной схемы является инерционность по отношению к изменению таблицы маршрутизации (в случае существующего потока изменение правил маршрутизации пакетов не будет «замечено» до момента удаления шаблона).

Динамически адаптируемые таблицы используют правила маршрутизации «напрямую», используя маску и номер сети из таблицы маршрутизации для проверки пакета и определения порта, на который нужно передать пакет. При этом изменения в таблице маршрутизации (в результате работы, например, протоколов маршрутизации/резервирования) сразу же влияют на обработку всех новопришедших пакетов. Динамически адаптируемые таблицы также позволяют легко реализовывать быструю (аппаратную) проверку списков доступа.

Программная маршрутизация

Программная маршрутизация выполняется либо специализированным ПО маршрутизаторов (в случае, когда аппаратные методы не могут быть использованы, например, в случае организации туннелей), либо программным обеспечением на компьютере. В общем случае, любой компьютер осуществляет маршрутизацию своих собственных исходящих пакетов (как минимум, для разделения пакетов, отправляемых на шлюз по умолчанию и пакетов, предназначенных узлам в локальном сегменте сети). Для маршрутизации *чужих* IP-пакетов, а также построения таблиц маршрутизации используется различное ПО:

Сервис RRAS (англ. *routing and remote access service*) в Windows Server

Демоны routed, gated, quagga в Unix-подобных операционных системах (Linux, FreeBSD и т. д.)

Маршрутизация в сети Интернет

Маршрутизация в сети Интернет основана на протоколах TCP/IP.

Передача информации осуществляется с помощью IP-пакетов, заголовок каждого IP-пакета содержит IP-адреса получателя и отправителя пакета. Каждый пакет обрабатывается маршрутизатором в соответствии с его таблицей маршрутизации. Таблица, в свою очередь, содержит информацию, компьютеру с каким адресом направлять пакеты с тем или иным диапазоном адресов. Например, все пакеты определённого диапазона

могут направляться другому маршрутизатору, который «отвечает» за этот сегмент.

В ряде случаев маршрутизатор может преобразовывать заголовок пакета, заменяя адреса отправителя и/или получателя пакета. В частности, это происходит при взаимодействии локальной сети (имеющей свои адреса) с глобальной сетью Интернет. В этом случае локальная сеть может быть видна извне по одному глобальному IP-адресу. Для того, чтобы маршрутизатор мог направлять пакеты с одним глобальным адресом тем или иным получателям в локальной сети, используется таблица NAT, где помимо IP-адресов указываются порты, идентифицирующие приложения, устанавливающие соединение. При этом номера портов указаны не в заголовке IP-пакета, а в заголовке сегмента TCP либо UDP (сегменты инкапсулируются в поле данных IP-пакетов). Это позволяет осуществлять взаимно-однозначную идентификацию получателя и отправителя в тех случаях, когда за одним глобальным адресом находится множество компьютеров локальных сетей.
Пример таблицы NAT

Глобальный адрес	**Локальный адрес**
209.165.200.226:1444	192.168.1.15:1444
209.165.200.226:1445	192.168.1.26:1444

Таблица маршрутизации — электронная таблица (файл) или база данных, хранящаяся на маршрутизаторе или сетевом компьютере, которая описывает соответствие между адресами назначения и интерфейсами, через которые следует отправить пакет данных до следующего маршрутизатора. Является простейшей формой *правил маршрутизации*.

Таблица маршрутизации обычно содержит:

адрес сети или узла назначения, либо указание, что маршрут является *маршрутом по умолчанию*

маску сети назначения (для IPv4-сетей маска /32 (255.255.255.255) позволяет указать единичный узел сети)

шлюз, обозначающий адрес маршрутизатора в сети, на который необходимо отправить пакет, следующий до указанного адреса назначения

интерфейс, через который доступен шлюз (в зависимости от системы, это может быть порядковый номер, GUIDили символьное имя устройства; интерфейс может быть отличен от шлюза, если шлюз доступен через дополнительное сетевое устройство, например, сетевую карту)

метрику — числовой показатель, задающий предпочтительность маршрута. Чем меньше число, тем более предпочтителен маршрут (интуитивно представляется как расстояние).

В таблице может быть один, а в некоторых операционных системах и несколько шлюзов по умолчанию. Такой шлюз используется в сетях, для которых нет более конкретных маршрутов в *таблице маршрутизации*.

==========
==========

```
Interface List
0x1 ........................... MS TCP Loopback interface
0x2 ...00 14 2a 8b a1 b5 ...... NVIDIA nForce Networking Controller
0x3 ...00 50 56 c0 00 01 ...... VMware Virtual Ethernet Adapter for VMnet1
0xd0005 ...00 53 45 00 00 00 ...... WAN (PPP/SLIP) Interface
===========================================================================
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
          0.0.0.0          0.0.0.0    89.223.67.129   89.223.67.131       20
      60.48.85.155  255.255.255.255    89.223.67.129   89.223.67.131       20
      60.48.105.1  255.255.255.255    89.223.67.129   89.223.67.131       20
     60.48.172.103  255.255.255.255    89.223.67.129   89.223.67.131       20
     60.48.203.116  255.255.255.255    89.223.67.129   89.223.67.131       20
      60.49.71.132  255.255.255.255    89.223.67.129   89.223.67.131       20
     66.36.138.228  255.255.255.255    89.223.67.129   89.223.67.131       20
     66.36.152.228  255.255.255.255    89.223.67.129   89.223.67.131       20
    74.108.102.130  255.255.255.255    89.223.67.129   89.223.67.131       20
     89.223.67.128  255.255.255.192    89.223.67.131   89.223.67.131       20
     89.223.67.131  255.255.255.255        127.0.0.1       127.0.0.1       20
    89.255.255.255  255.255.255.255    89.223.67.131   89.223.67.131       20
          127.0.0.0        255.0.0.0        127.0.0.1       127.0.0.1       1
    164.77.239.153  255.255.255.255    89.223.67.129   89.223.67.131       20
      192.168.23.0    255.255.255.0     192.168.23.1    192.168.23.1       20
      192.168.23.1  255.255.255.255        127.0.0.1       127.0.0.1       20
    192.168.23.255  255.255.255.255     192.168.23.1    192.168.23.1       20
     192.168.192.0    255.255.255.0  192.168.192.251  192.168.192.251      1
   192.168.192.251  255.255.255.255        127.0.0.1       127.0.0.1       50
   192.168.192.255  255.255.255.255  192.168.192.251  192.168.192.251      50
    212.113.96.250  255.255.255.255    89.223.67.129   89.223.67.131       20
    219.95.153.243  255.255.255.255    89.223.67.129   89.223.67.131       20
          224.0.0.0        240.0.0.0    89.223.67.131   89.223.67.131       20
          224.0.0.0        240.0.0.0     192.168.23.1    192.168.23.1       20
          224.0.0.0        240.0.0.0  192.168.192.251  192.168.192.251      50
  255.255.255.255  255.255.255.255    89.223.67.131   89.223.67.131       1
  255.255.255.255  255.255.255.255     192.168.23.1    192.168.23.1       1
  255.255.255.255  255.255.255.255  192.168.192.251  192.168.192.251      1
Default Gateway:     89.223.67.129
===========================================================================
===========================================================================
```

Пример таблицы маршрутизации при четырёх интерфейсах (loopback, две сетевые карты, VPN-соединение)

Типы записей в таблице маршрутизации:

маршрут до сети
маршрут до компьютера
маршрут по умолчанию

Расщепление горизонта (англ. *split horizon*) — метод предотвращения петель маршрутизации, вызванных медленной сходимостью дистанционно-векторных протоколов маршрутизации. Может применяться вместе с отправлением обратного маршрута.

Правило расщеплённого горизонта говорит, что маршрутизатор не должен распространять информацию о сети через интерфейс, на который прибыло обновление.

Возьмём для примера три маршрутизатора - R1, R2, R3. R1 анонсирует R2 некую сеть, R2 принимает информацию и обновляет свою таблицу маршрутизации, после чего пересылает обновлённую информацию только к R3, не затрагивая R1 т.к. именно от R1 пришёл анонс некой сети.

Расщепление горизонта не позволяет распространять неверную информацию о маршрутизации и уменьшает объём передаваемых служебных сообщений. Применяется в дистанционно-векторных протоколах (таких как RIPv1, RIPv2, IGRP, EIGRP и др.).

Лабораторная работа №15

Программирование с использованием Message Queuing

Цель работы:
- Обзор Message Queuing
- Изучения Архитектура Message Queuing
- Программирование с использованием Message Queuing

1. Практическая часть

Теперь, когда архитектура Message Queuing известна, можно приступать к программированию. В последующих разделах будет показано, как создавать и управлять очередями, а также как отправлять и принимать сообщения.

Создание очереди сообщений

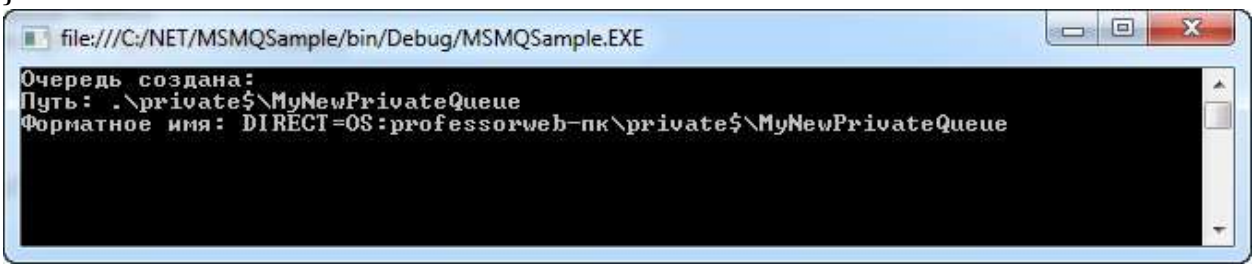
Вы уже видели, как создаются очереди сообщений утилитой Computer Management. Но очереди сообщений могут быть созданы и программно вызовом метода Create() класса **MessageQueue**. Методу Create() должен быть передан путь к новой очереди. Этот путь состоит из имени хоста, где расположена очередь, и имени очереди.

В следующем примере очередь MyNewPrivateQueue создается на локальном хосте. Чтобы создать частную очередь, путь должен включать Private$; например, \Private$\MyNewPrivateQueue.

После вызова метода Create() свойства очереди могут быть изменены. Например, используя свойство Label, установим метку очереди в Demo Queue. В примере программы путь очереди и форматное имя выводятся на консоль. Форматное имя создается автоматически с UUID, который может применяться для доступа к очереди без указания имени сервера:

```
using System;
using System.Messaging;

namespace MSMQSample
{
   class Program
   {
      static void Main(string[] args)
      {
         using (var queue = MessageQueue.Create(".\\private$\\MyNewPrivateQueue"))
         {
```

```
            queue.Label = "Demo Queue";
            Console.WriteLine("Очередь создана:");
            Console.WriteLine("Путь: {0}", queue.Path);
            Console.WriteLine("Форматное имя: {0}", queue.FormatName);
        }
        Console.ReadLine();
    }
  }
}
```



Для создания очереди необходимы административные привилегии. Обычно нельзя рассчитывать, что пользователь приложения будет иметь их. Именно поэтому очереди обычно создаются программой установки.

Нахождение очереди

Для идентификации очередей могут использоваться путевое имя и форматное имя. При поиске очереди необходимо делать различия между общедоступными и частными очередями. Общедоступные очереди публикуются в Active Directory. Для таких очередей не обязательно знать систему, на которой они расположены. Частные очереди могут быть найдены только в случае, если известно имя системы, на которой расположена очередь:

```
using System;
using System.Messaging;

namespace MSMQSample
{
  class Program
  {
    static void Main(string[] args)
    {
      foreach (var queue in MessageQueue.GetPrivateQueuesByMachine(Environment.MachineName))
        Console.WriteLine("Очередь: {0}\n", queue.Path);

      Console.ReadLine();
    }
```

```
    }
}
```

Общедоступные очереди в домене Active Directory можно искать по метке очереди, категории или форматного имени. Можно также получить все очереди, имеющиеся на машине. В классе MessageQueue предусмотрены статические методы для поиска: *GetPublicQueuesByLabel(), GetPublicQueuesByCategory()* и *GetPublicQueuesByMachine()*. Метод GetPublicQueues() возвращает массив всех общедоступных очередей в домене.

Открытие известных очередей

Если имя очереди известно, то искать ее не обязательно. Очереди могут открываться с использованием пути или форматного имени. И то, и другое может быть установлено конструктором класса MessageQueue.

Путевое имя

Путь указывает имя машины и имя очереди для ее открытия. В следующем примере кода открывается очередь MyNewPrivateQueue на локальном хосте. Чтобы удостовериться в существовании очереди, применяется статический метод MessageQueue.Exists():

```
if (MessageQueue.Exists(@".\private$\MyNewPrivateQueue"))
{
    var queue = new MessageQueue(@".\private$\MyNewPrivateQueue");
}
else
{
    Console.WriteLine("Очередь не существует");
}
```

В зависимости от типа очереди при открытии очередей должны использоваться разные идентификаторы. В следующей таблице показан синтаксис идентификаторов для очередей разных типов:

Синтаксис идентификаторов для очередей разных типов

Тип очереди	**Синтаксис**
Общедоступная очередь	ИмяКомпьютера\ИмяОчереди
Частная очередь	ИмяКомпьютера\Private$\ИмяОчереди

Журнальная очередь	ИмяКомпьютера\ИмяОчереди\Journal$
Журнальная очередь машины	ИмяКомпьютера\Journal$
Очередь “мертвых писем” машины	ИмяКомпьютера\DeadLetter$
Транзакционная очередь “мертвых писем” машины	ИмяКомпьютера\XactDeadLetter$

Если для открытия общедоступной очереди используется путевое имя, необходимо передавать имя машины. Если имя машины не известно, вместо него может быть указано форматное имя. Путевое имя частной очереди может применяться только на локальных системах. Форматное имя должно использоваться для удаленного доступа к частным очередям.

Форматное имя

Вместо использования путевого имени для открытия очереди можно применять форматное имя. Форматное имя применяется для поиска очереди в Active Directory, чтобы получить хост, на котором расположена очередь. В автономной среде, где очередь во время отправки сообщения недоступна, необходимо использовать форматное имя:

```
MessageQueue queue = new MessageQueue(
    @"FormatName:PUBLIC=09816AFF-3608-4c5d-B892-69754BA151FF");
```

Форматное имя используется по-разному. С его помощью можно открывать частные очереди и указывать нужный протокол.

- Для доступа к частной очереди строка, передаваемая конструктору, выглядит так:*FormatName:PRIVATE=MachineGUID\QueueNumber*. Номер QueueNumber для частных очередей генерируется при их создании. Номера очередей можно просмотреть в каталоге <windows>\System32\msmq\storage\lqs.
- В *FormatName:DIRECT=Protocol:MachineAddress\QueueName* можно указать протокол, который должен использоваться для отправки сообщений. Протокол HTTP поддерживается, начиная с версии Message Queuing 3.0.
- *FormatName:DIRECT=OS:MachineName\QueueName* — другой способ задания очереди с применением форматного имени. Таким образом, протокол указывать не нужно, а только имя машины и форматное имя.

Отправка сообщения

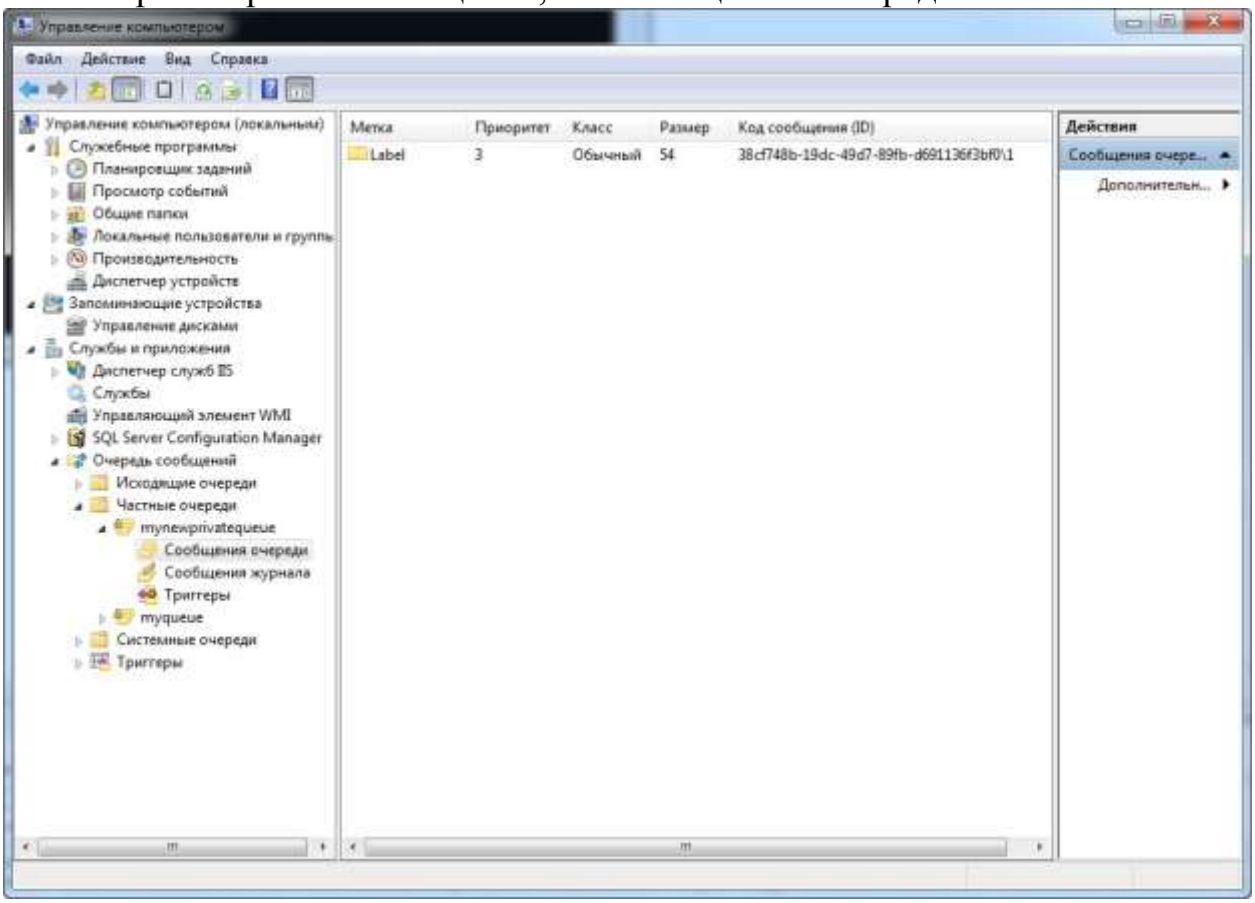
Для отправки сообщения в очередь используется метод Send() класса MessageQueue. Объект, переданный в качестве аргумента методу Send(), сериализуется в ассоциированную очередь. Метод Send() перегружен так, что можно передавать метку и объект MessageQueueTransaction.

В приведенном ниже примере кода сначала выполняется проверка, существует ли очередь. Если очередь не существует, она создается. Затем

очередь открывается и с помощью метода Send() в очередь отправляется сообщение “Sample Message”. Путевое имя специфицирует точку вместо имени сервера, что означает локальную систему. Путевые имена частных очередей работают только локально:

```
try
{
    if (!MessageQueue.Exists(@".\private$\MyNewPrivateQueue"))
    {
        MessageQueue.Create(@".\private$\MyNewPrivateQueue");
    }

    var queue = new MessageQueue(@".\private$\MyNewPrivateQueue");
    queue.Send("Sample Message", "Label");
}
catch (MessageQueueException ex)
{
    Console.WriteLine(ex.Message);
}
```

На рисунке ниже показано окно оснастки Computer Management, в котором можно просматривать сообщения, появляющиеся в очереди:



Открыв сообщение и выбрав в диалоговом окне вкладку Body (Тело), можно увидеть, что сообщение сформатировано с использованием XML. Способ форматирования сообщения — функция форматировщика, ассоциированного с очередью сообщений

.

Форматировщик сообщений

Формат, в котором передаются сообщения в очередь, зависит от используемого форматировщика. Класс MessageQueue имеет свойство **Formatter**, через которое очереди может быть назначен объект-форматировщик.

Форматировщик по умолчанию — XmlMessageFormatter — форматирует сообщение в синтаксисе XML, как показано в предыдущем примере. Форматировщик сообщений реализует интерфейс *IMessageFormatter*. В пространстве имен System.Messaging доступны три форматировщика сообщений:

XmlMessageFormatter

Форматировщик по умолчанию. Он сериализует объекты, используя XML.

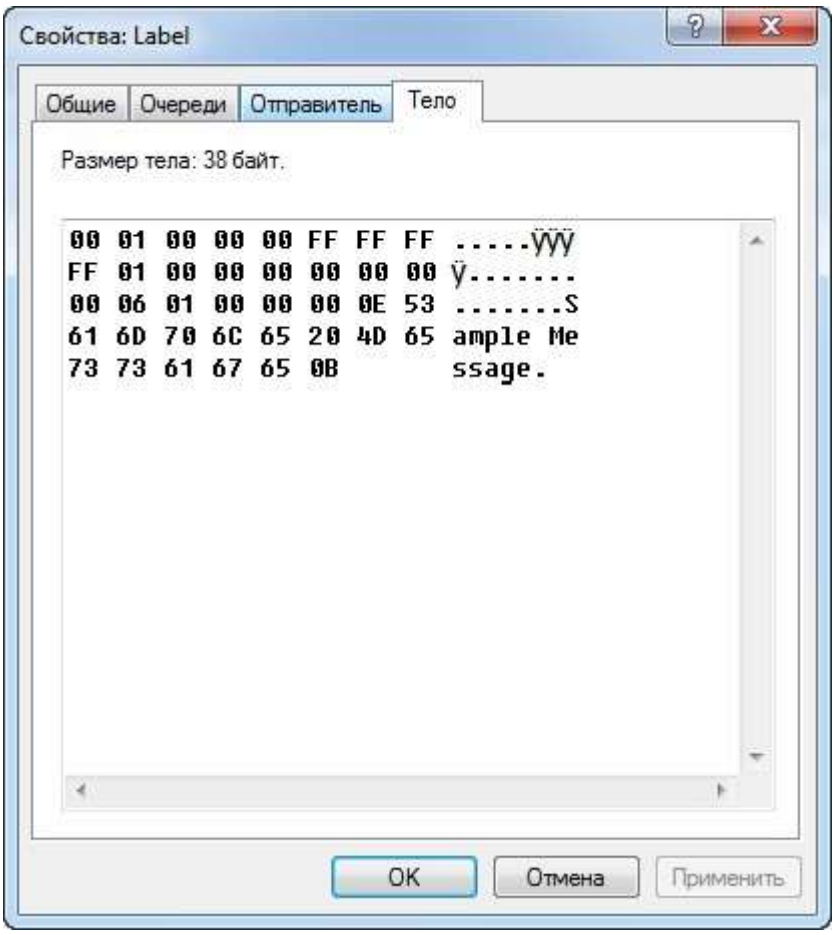
BinaryMessageFormatter

С помощью форматировщика BinaryMessageFormatter сообщения сериализуются в двоичный формат. Эти сообщения короче, чем сформатированные с применением XML.

ActiveXMessageFormatter

Двоичный форматировщик, так что сообщения могут быть прочитаны и записаны объектами COM. Используя этот форматировщик, можно записывать сообщения в очередь с помощью классов .NET и читать их оттуда объектами COM, и наоборот.

Простое сообщение, показанное на выше в формате XML, на рисунке ниже сформатировано с помощью BinaryMessageFormatter:



Отправка сложных сообщений

Вместо строк методу Send() класса MessageQueue можно передавать объекты. Тип класса таких объектов должен соответствовать определенным требованиям, но они зависят от форматировщика.

Для двоичного форматировщика класс должен быть сериализуемым и снабжен атрибутом [Serializable]. При сериализации исполняющей системой .NET сериализуются все поля (включая приватные). Специальная сериализация может быть определена за счет реализации интерфейса ISerializable.

Сериализация XML происходит с помощью XML-форматировщика. При сериализации XML сериализуются все общедоступные поля и свойства. На сериализацию XML могут оказывать влияние атрибуты из пространства имен System.Xml.Serialization.

Прием сообщений

Для чтения сообщений можно использовать класс MessageQueue. Метод *Receive()* читает единственное сообщение и удаляет его из очереди. Если сообщения отправлены с разными приоритетами, читается сообщение с наивысшим приоритетом. Чтение сообщений с одинаковым приоритетом не обеспечивает поступление сообщений в порядке их отправки, потому что порядок сообщений в сети не гарантируется. Для получения гарантированного порядка применяйте транзакционные очереди сообщений.

В следующем примере сообщение читается из частной очереди MyNewPrivateQueue. Ранее в сообщение была передана простая строка. При

чтении сообщения с использованием XmlMessageFormatter необходимо передавать типы прочитанных объектов конструктору форматировщика. В данном примере тип System.String передается в массив аргументов конструктора XmlMessageFormatter. Этот конструктор принимает либо массив String с типами в виде строк, либо массив объектов Type. Сообщение читается методом Receive() и затем тело сообщения выводится на консоль:

```
var queue = new MessageQueue(@".\private$\MyNewPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new String[] { "System.String" });
Message message = queue.Receive();
Console.WriteLine(message.Body);
```

Метод Receive() ведет себя синхронно и ожидает появления сообщения в очереди, если на момент его вызова там было пусто.

Перечисление сообщений

Вместо чтения сообщений друг за другом с помощью метода Receive() можно применить перечислитель для прохождения сразу по всем сообщениям в очереди. Класс MessageQueue реализует интерфейс IEnumerable, и потому может быть использован в операторе foreach или LINQ. Здесь сообщения не удаляются из очереди, вы только заглядываете в каждое сообщение для извлечения содержимого:

```
var queue = new MessageQueue(@".\private$\MyNewPrivateQueue");
    queue.Formatter = new BinaryMessageFormatter();
foreach (Message message in queue)
    Console.WriteLine(message.Body);
```

Транзакционные очереди

В случае восстанавливаемых сообщений нет никакой гарантии их доставки в том порядке, в котором они были отправлены, а также в том, что доставка будет однократной. Сбои в сети могут привести к многократной доставке одних и тех же сообщений; это случается и тогда, когда отправитель и получатель имеют несколько установленных протоколов, используемых Message Queuing.

Транзакционные очереди должны применяться, когда необходимы следующие гарантии:

- сообщения должны появляться в том порядке, в котором были отправлены;
- сообщения должны появляться по одному разу.

В случае транзакционных очередей одна транзакция не охватывает отправку и прием сообщений. Природа Message Queuing такова, что между отправкой и получением сообщения может пройти довольно длительное время. В отличие от этого, транзакции должны быть кратковременными. В Message Queuing первая транзакция используется для отправки сообщения в очередь, вторая — для передачи его по сети и третья — для получения сообщения.

В следующем примере демонстрируется создание транзакционной очереди сообщений, а также отправка сообщения с использованием транзакции.

Транзакционная очередь сообщений создается передачей true во втором параметре методу MessageQueue.Create():

```
using System;
using System.Messaging;

namespace MSMQSample
{
   class Program
   {
      static void Main(string[] args)
      {
         if (!MessageQueue.Exists(@".\MyTransactionalQueue"))
         {
            MessageQueue.Create(@".\MyTransactionalQueue", true);
         }

         var queue = new MessageQueue(@".\MyTransactionalQueue");
         var transaction = new MessageQueueTransaction();

         try
         {
            transaction.Begin();
            queue.Send("a", transaction);
            queue.Send("b", transaction);
            queue.Send("c", transaction);
            transaction.Commit();
         }
         catch
         {
            transaction.Abort();
         }
      }
   }
}
```

Если в очередь необходимо записывать сразу несколько сообщений в пределах одной транзакции, то для этого придется создать экземпляр объекта **MessageQueueTransaction** и вызвать его метод *Begin()*. По завершении отправки всех сообщений, относящихся к транзакции, следует вызвать метод *Commit()* того же объекта MessageQueueTransaction. Для отмены транзакции (не оставляя никаких сообщений в очереди) должен быть вызван метод *Abort()*, что и делается в блоке catch.

2. теоретическая часть.

В пространстве имен **System.Messaging** определены классы, которые позволяют выполнять чтение и запись сообщений с использованием такого

предлагаемого в составе операционной системы Windows средства для организации сообщений, как ***Message Queuing***.
Обмен сообщениями может применяться в сценарии автономной работы, в котором не требуется, чтобы клиент и сервер обязательно запускались в одно и то же время.
Прежде чем углубляться в детали программирования с использованием Message Queuing, в данной статье я предлагаю ознакомиться с основными концепциями организации очередей сообщений и сравнить их с концепциями синхронного и асинхронного программирования.
При синхронном программировании, когда вызывается метод, то вызвавший его код должен ожидать, пока метод не завершит свою работу. При асинхронном программировании вызывающий поток запускает метод и параллельно продолжает свою работу. Асинхронное программирование основано на применении делегатов, библиотек классов, которые уже поддерживают асинхронные методы (например, прокси-классы веб-служб и классы из пространств System.Net и System.IO), либо специальных потоков. Как при синхронном, так и при асинхронном программировании клиент и сервер должны работать в одно и то же время.
Хотя Message Queuing работает асинхронно, поскольку клиент (отправитель) не ожидает прочтения получателем (сервером) оправленных ему данных, между Message Queuing и асинхронным программированием существует принципиальная разница: Message Queuing может выполняться в автономной (отключенной) среде. На момент отправки данных их получатель может быть отключен. Позднее, когда получатель подключится, он получит данные без вмешательства отправляющего приложения.
Подключенное и отключенное программирование можно сравнить с разговором по телефону и отправкой почтовых сообщений. При разговоре с кем-либо по телефону оба участника должны быть подключены одновременно - это синхронная коммуникация. В случае обмена электронной почтой отправитель не знает, когда его послание будет прочитано. Люди, использующие эту технологию, работают в автономном (отключенном) режиме.
Конечно, может случиться так, что почта не будет прочитана никогда, а просто проигнорирована. Такова природа отключенных коммуникаций. Чтобы избежать этой проблемы, можно запросить ответ или подтверждение факта прочтения письма. Если ответ не придет в течение определенного времени, возможно, придется как-то справляться с таким "исключением". Все это также возможно в Message Queuing.
Message Queuing, по сути, можно считать технологией для обмена электронными сообщениями между приложениями, а не людьми. Она обладает множеством функциональных возможностей, которые в других службах обмена сообщениями не доступны: гарантированием доставки, применением транзакций, получением подтверждений, экспресс-режимом, использующим память, и т.д. Message Queuing предлагает массу полезных средств для коммуникаций между приложениями.

С помощью Message Queuing можно отправлять, принимать и маршрутизировать сообщения в подключенной и отключенной (автономной) среде. На рисунке ниже показан очень простой способ использования сообщений. Отправитель посылает сообщения в очередь сообщений, а получатель принимает их из этой очереди:

Одной из ситуаций, в которых удобно применять Message Queuing — это когда клиентское приложение часто отключается от сети (например, у коммивояжера, навещающего за-казчиков на местах). Коммивояжер может вводить данные заказа непосредственно у заказчика. Приложение ставит сообщение о каждом заказе в очередь сообщений, находящуюся на клиентской системе. Как только коммивояжер возвращается в свой офис, заказ автоматически передается из очереди сообщений клиентской системы в очередь сообщений целевой системы, где и обрабатывается.

Помимо портативного компьютера, коммивояжер может использовать устройство Pocket Windows, где также доступно Message Queuing.

Технология Message Queuing может быть полезна и в подключенной среде. Представьте сайт электронной коммерции (показан на рисунке ниже), где в определенные периоды времени сервер полностью загружен обработкой заказов, например, в ранний вечер и в выходные, при этом по ночам нагрузка значительно уменьшается. Решение проблемы может состоять в приобретении более быстрого сервера или в добавлении дополнительных серверов к системе, чтобы они справлялись с пиковыми нагрузками.

Однако существует более дешевое решение: сгладить пиковые нагрузки, сдвинув транзакции со времени максимальных нагрузок на время с низкой загрузкой. В такой схеме заказы отправляются в очередь сообщений, а принимающая сторона читает их тогда, когда это удобно системе базы данных. Таким образом, нагрузка на систему сглаживается по времени, так что сервер, обрабатывающий транзакции, может быть дешевле и не требовать модернизации:

Функциональные возможности Message Queuing

Технология Message Queuing является службой, которая поставляется как часть операционной системы Windows. Ниже перечислены ее основные функциональные возможности:

- Сообщения могут пересылаться в автономной среде. То есть приложению-отправителю и приложению-получателю вовсе не обязательно выполняться в одно и то же время.
- В экспресс-режиме сообщения могут пересылаться очень быстро. В экспресс-режиме сообщения просто сохраняются в памяти.
- Для механизма восстановления сообщения могут отправляться с гарантированной доставкой. Такие сообщения сохраняются в файлах и доставляются даже в случае перезагрузки сервера.
- Очереди сообщений могут защищаться с применением списков контроля доступа и указания в них, каким пользователям разрешено отправлять или получать сообщения из очереди. Кроме того, сообщения могут шифроваться для исключения вероятности их

прочтения с помощью сетевых анализаторов пакетов, а также снабжаться приоритетами, чтобы те из них, которые имеют более высокий приоритет, обрабатывались быстрее.

- В Message Queuing 3.0 поддерживается возможность отправки многоадресных (multicast) сообщений.
- В Message Queuing 4.0 поддерживается возможность распознавания вредоносных сообщений. Для таких сообщений может быть определена специальная очередь.

Например, в случае, если после прочтения сообщения из обычной очереди, далее оно должно вставляться в базу данных, но по какой-то причине этого не происходит, это сообщение может быть отправлено в очередь вредоносных сообщений. Впоследствии этой очередью вредоносных сообщений должен кто-нибудь заняться и выяснить, по какой причине адрес сообщения не удалось преобразовать.

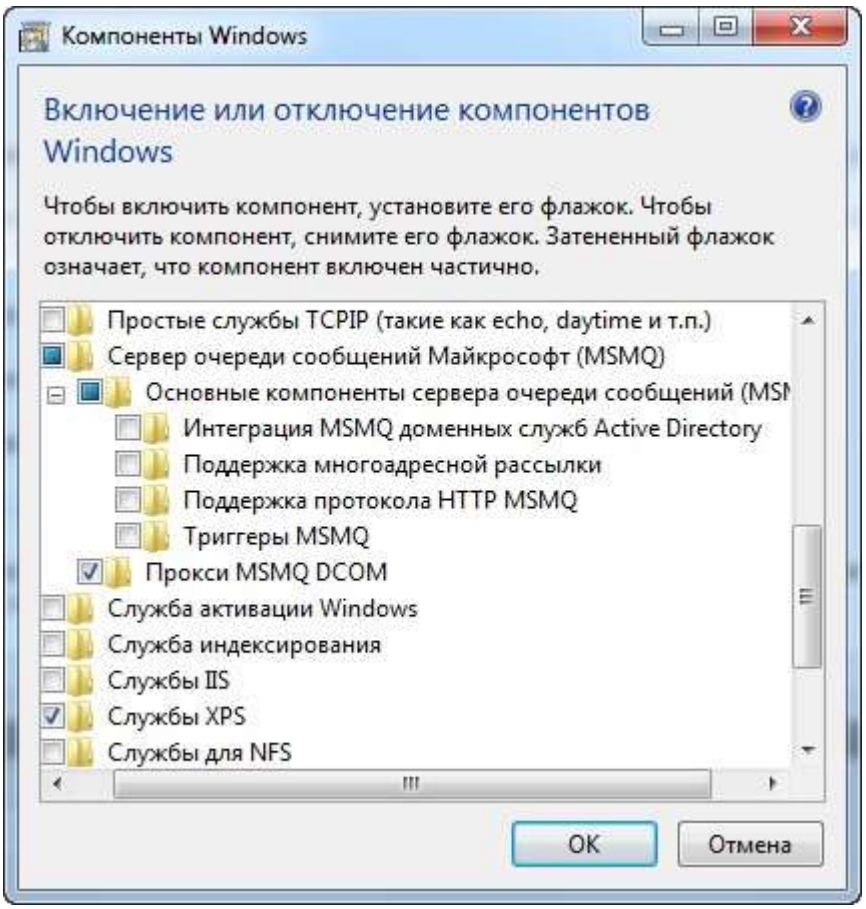
- В Message Queuing 5.0 поддерживаются более безопасные алгоритмы аутентификации, и может обрабатываться большее количество очередей. (В Message Queuing 4.0 при обработке нескольких тысяч очередей начинали возникать проблемы с производительностью.)

Из-за того, что Message Queuing является частью операционной системы, установить версию Message Queuing 5.0 в системе Windows XP или Windows Server 2003 не получится. Эта версия входит в состав ОС Windows Server 2008 R2 и Windows 7.

Продукты Message Queuing

Версия Message Queuing 5.0 поставляется в составе Windows 7 и Windows Server 2008 R2. В Windows 2000 входила версия Message Queuing 2.0, в которой не было поддержки ни протокола HTTP, ни многоадресных сообщений. Версия Message Queuing 3.0 поставлялась в составе Windows XP и Windows Server 2003, а версия Message Queuing 4.0 — в составе Windows Vista и Windows Server 2003.

При использовании ссылки Turn Windows Features on or off (Включение или отключение компонентов Windows), которая предлагается в Windows 7 в окне Configuring Programs and Features (Программы и компоненты), можно обнаружить отдельный раздел с опциями, касающимися Message Queuing:



В этом разделе доступны для выбора перечисленные ниже компоненты:

Microsoft Message Queue (MSMQ) Server Core

Основные компоненты сервера очереди сообщений (MSMQ), которые необходимы для получения базовой функциональности Message Queuing.

Active Directory Domain Services Integration

Интеграция MSMQ доменных служб Active Directory. Это средство позволяет записывать имена очередей сообщений в Active Directory. С помощью этой опции можно находить очереди в Active Directory и защищать их на основе пользователей и групп пользователей Windows.

MSMQ HTTP Support

Поддержка протокола HTTP MSMQ. Поддержка MSMQ HTTP позволяет отправлять и принимать сообщения, используя протокол HTTP.

Triggers

С помощью триггеров создаются экземпляры приложений при поступлении нового сообщения.

Multicast Support

Поддержка многоадресной рассылки. Позволяет отправлять сообщения группам серверов.

MSMQ DCOM Proxy

С помощью DCOM-прокси система может подключаться к удаленному серверу, используя API-интерфейс DCOM.

После установки Message Queuing в системе должна быть обязательно запущена служба Message Queuing (показана на рисунке). Эта служба читает

и записывает сообщения, а также взаимодействует с другими серверами Message Queuing для осуществления маршрутизации сообщений по сети:
В Message Queuing сообщения записываются и читаются из специальной очереди сообщений. Сами сообщения и очереди сообщений имеют несколько атрибутов, которые требуют пояснений.

Сообщения

Сообщение посылается в очередь сообщений. Сообщение включает тело, содержащее пересылаемые данные, и метку — заголовок сообщения. В тело сообщения может быть помещена любая информация. В .NET имеется набор форматировщиков, преобразующих данные, которые помещаются в тело. Помимо метки и тела сообщение включает дополнительную информацию об отправителе, конфигурацию таймаута, идентификатор транзакции или приоритет.
Очереди сообщений содержат сообщения нескольких типов:

- *Нормальное сообщение* отправляется любым приложением.
- *Подтверждающее сообщение (acknowledgment message)* уведомляет о состоянии нормального сообщения. Подтверждающие сообщения отправляются в административные очереди, чтобы уведомить об успехе или сбое при отправке нормальных сообщений.
- *Ответные сообщения* отправляются принимающим приложением, когда исходный отправитель требует некоторого специального ответа.
- *Отчетные сообщения* генерируются системой Message Queuing. К этой категории относятся тестовые сообщения и сообщения отслеживания маршрутизации.

Сообщение может обладать приоритетом, определяющим порядок, в котором сообщения будут читаться из очереди. Сообщения сортируются в очереди в соответствии с приоритетами, поэтому следующим из очереди всегда читается то сообщение, которое имеет наивысший приоритет.
Сообщения имеют два режима доставки: экспресс (express) и восстановимый (recoverable). Экспресс-сообщения доставляются очень быстро, потому что в качестве хранилища очереди используется оперативная память. Восстановимые сообщения сохраняются в файлах на каждом шаге маршрута — до тех пор, пока они не будут доставлены. Таким образом, доставка сообщений гарантируется, даже если компьютер будет перегружен или произойдет сбой сети.
Транзакционные сообщения — это специальная версия восстановимых сообщений. Благодаря транзакционным сообщениям гарантируется, что сообщения будут доставлены только однажды, и в том же порядке, в каком были отправлены. С транзакционными сообщениями приоритеты не используются.

Очередь сообщений

Очередь сообщений представляет собой своего рода "накопительный бункер" для сообщений. Сообщения, сохраняемые на диске, размещаются в каталоге <windows>\system32\msmq\storage.

Общедоступные или частные очереди обычно применяются для отправки сообщений, но существуют также и другие типы очередей.

Типы очередей сообщений MSMQ

Тип очереди	**Описание**
Общедоступная (public) очередь	Публикуется в Active Directory. Информация о таких очередях реплицируется в доменах Active Directory. Для получения информации о таких очередях можно воспользоваться средствами просмотра и поиска. К общедоступной очереди можно обращаться, не зная имени компьютера, на котором она расположена. Такую очередь можно переместить с одной системы на другую и клиент этого не заметит. В среде рабочей группы (Workgroup) невозможно создавать общедоступные очереди, потому что им нужна служба Active Directory.
Частные (private) очереди	Не публикуются в Active Directory. Эти очереди доступны, только когда известны их полные путевые имена. Частные очереди могут использоваться в среде Workgroup.
Журнальные (journal) очереди	Служат для хранения копий сообщений после того, как они были получены или отправлены. Включение протоколирования для общедоступной или частной очереди автоматически создает журнальную очередь. Журнальные очереди бывают двух типов: исходное протоколирование и целевое протоколирование. Исходное протоколирование включается свойствами сообщения - журнальные сообщения сохраняются на исходной системе. Целевое протоколирование включается свойствами очереди - эти сообщения сохраняются в журнальной очереди целевой системы.
Очереди мертвых писем (dead-letter)	Хранят сообщения, если они не появляются на целевой системе по истечении определенного периода времени. В противоположность синхронному программированию, где ошибки обнаруживаются немедленно, при использовании Message Queuing ошибки должны обрабатываться иначе. Очередь мертвых писем можно проверять для обнаружения не доставленных сообщений.
Административные очереди	Содержат подтверждения об отправленных сообщениях. Отправитель может указать административную очередь, из которой он будет получать уведомления об успешной отправке

	сообщений.
Очередь ответов	Применяется, когда требуется нечто большее, чем простое подтверждение о факте отправки в качестве ответа со стороны получателя. Принимающее приложение может посылать ответные сообщения обратно исходному отправителю.
Очередь отчетов	Используется для тестовых сообщений. Очереди отчетов могут быть созданы изменением типа (или категории) общедоступной или частной очереди на предопределенный идентификатор {55EE8F33-CCE9-11CF-B108-0020AFD61CE9}. Очереди отчетов удобны в качестве инструмента тестировании для отслеживания сообщений на их маршруте.
Системные очереди	Являются частными и используются самой системой Message Queuing. Эти очереди предназначены для административных сообщений, хранения уведомлений и обеспечения правильного порядка доставки транзакционных сообщений.

Административные инструменты для работы с Message Queuing

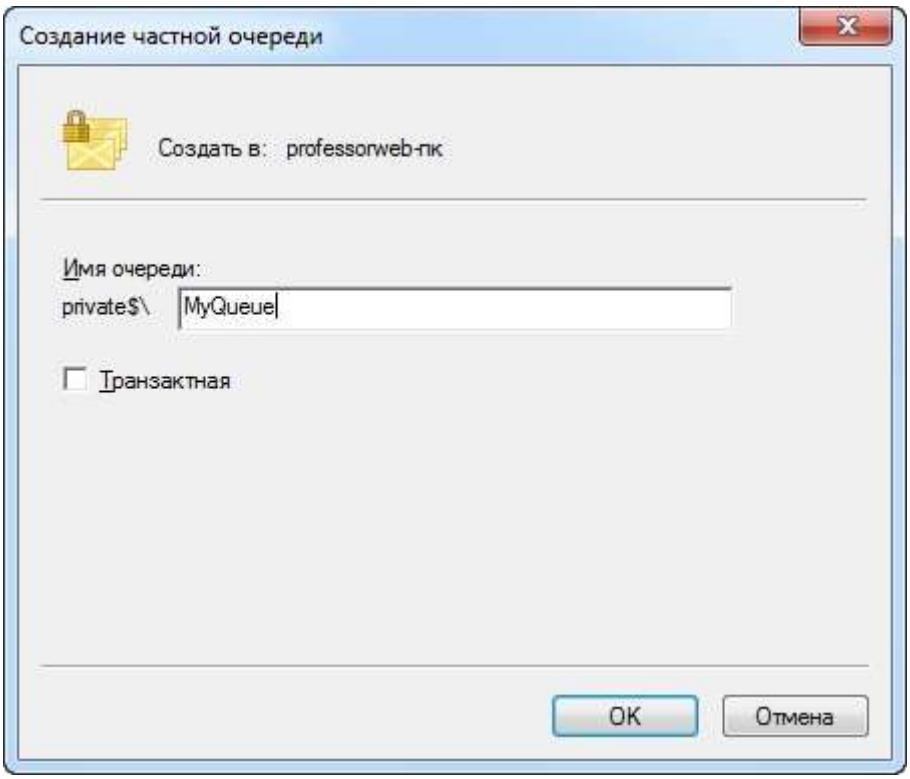
Прежде чем переходить к рассмотрению программного взаимодействия с Message Queuing, следует ознакомиться с административными инструментами, которые поставляются в составе операционной системы Windows для создания и управления очередями и сообщениями.

Описанные здесь инструменты применяются для работы не только с Message Queuing. Возможности, касающиеся Message Queuing, становятся доступными в этих инструментах только после установки Message Queuing в системе.

Создание очередей сообщений

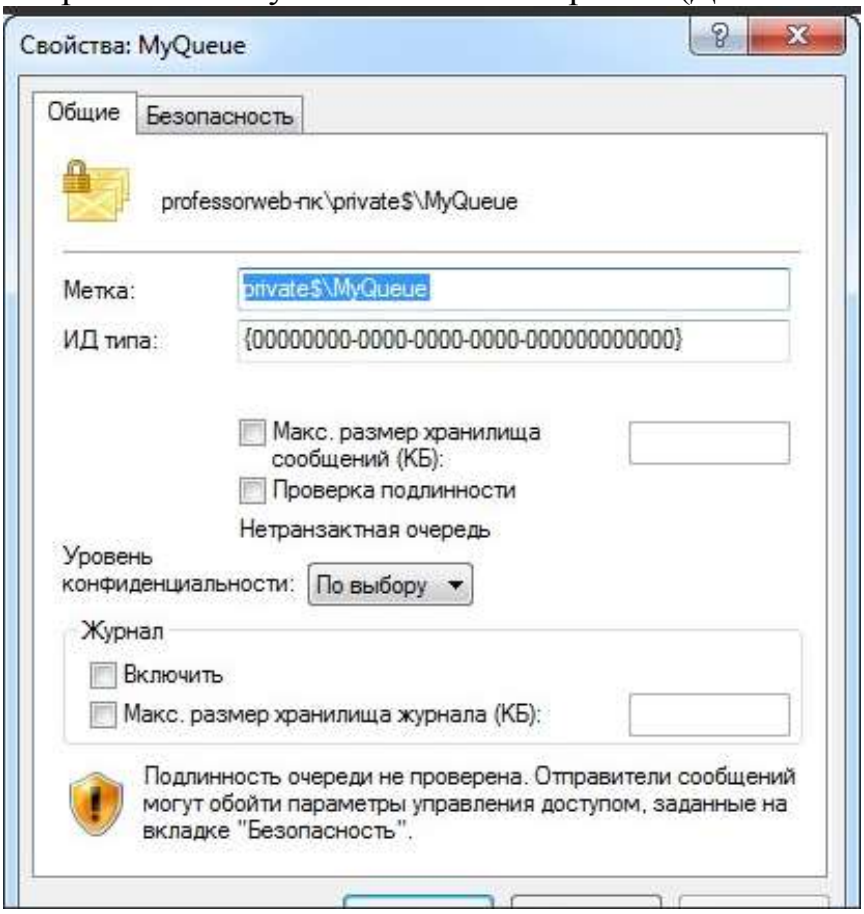
Очереди сообщений могут создаваться с помощью оснастки Computer Management (Управление компьютером) консоли управления MMC. В системе Windows 7 оснастку Computer Management можно запустить, выбрав в меню Start (Пуск) пункт Control Panel --> Administrative Tools --> Computer Management (Панель управления --> Администрирование --> Управление компьютером). В панели древовидного представления Message Queuing находится ниже элемента Services and Applications (Службы и приложения).

Выбрав Private Queues (Частные очереди) или Public Queues (Общедоступные очереди), можно создать новую очередь из меню Action (Действие), как показано на рисунке. С общедоступными очередями можно работать, только если Message Queuing сконфигурирована в режиме Active Directory:



Свойства Message Queuing

После создания очереди в оснастке Computer Management можно модифицировать ее свойства, выделив очередь в древовидной панели и выбрав в меню пункт Action --> Properties (Действие --> Свойства):



Здесь для конфигурирования доступно несколько опций:

- Label (Метка) — имя очереди, которое может использоваться для ее поиска.
- Type ID (Идентификатор типа), который по умолчанию устанавливается в {00000000-0000-0000-0000-000000000000}, для отображения множественных очередей на единственную категорию типа.

Отчетные очереди, как было сказано ранее, используют специфический идентификатор типа. Type ID представляет собой универсальный уникальный идентификатор (UUID) или глобально уникальный идентификатор (GUID).

Специальные идентификаторы типа могут быть созданы утилитами *uuidgen.exe* или *guidgen.exe*. Утилита uuidgen.exe — это инструмент командной строки, служащий для создания уникальных идентификаторов, а guidgen.exe — графическая ее версия для создания UUID.

- Во избежание переполнения диска, максимальный размер всех сообщений (Limit message storage to (KB)) в очереди может ограничиваться.
- Отмеченный флажок Authenticated (Проверка подлинности) позволяет записывать и читать сообщения в очереди только аутентифицированным пользователям.
- Опция Privacy Level (Уровень конфидециальности) позволяет шифровать содержимое сообщения. Возможные значения: None (Нет), Optional (Необязательно) или Body (Тело). None означает, что шифрованные сообщения не принимаются, Body — принимаются только шифрованные сообщения, а значение по умолчанию Optional — принимаются те и другие.
- Целевое протоколирование может быть сконфигурировано опцией Journal (Журнал). С помощью этой опции обеспечивается сохранение в журнале копий принятых сообщений. Для журнальных сообщений очереди может быть указан максимальный размер занятого дискового пространства. По достижении максимального размера целевой журнал очищается.
- При опции конфигурации Multicast (Групповой очереди) можно определить групповой IP-адрес для очереди. Один и тот же групповой IP-адрес может применяться разными узлами в сети, так что сообщение, отправленное по одному адресу, принимается множеством очередей.