#лекции Сетевое программирование в C# - это возможность создания приложений, которые могут обмениваться данными между различными устройствами через сеть. Сетевые приложения могут взаимодействовать с другими приложениями на удаленных компьютерах, обмениваться данными и ресурсами, и выполнять различные задачи.

Основы сетевого программирования в C# включают в себя следующее:

1. Создание соединения: Для установления соединения между клиентом и сервером в C# используются классы TcpListener и TcpClient. TcpListener прослушивает определенный порт и ожидает входящих подключений, а TcpClient - устанавливает соединение с сервером.

2. Обмен данными: Для передачи данных между клиентом и сервером в C# используется класс NetworkStream. NetworkStream предоставляет потоковый интерфейс для отправки и приема данных между устройствами.

3. Протоколы: Для обмена данными между устройствами в C# используются различные протоколы, такие как TCP, UDP, HTTP и другие. Каждый протокол имеет свои особенности и используется для определенных задач.

4. Обработка ошибок: При сетевом программировании в C# необходимо учитывать возможные ошибки, такие как потеря данных, проблемы с соединением и другие. Для обработки ошибок используются исключения, которые позволяют обработать возможные ошибки и продолжить работу программы.

5. Многопоточность: Сетевые приложения в C# могут быть многопоточными, что позволяет обрабатывать множество запросов одновременно. Для создания многопоточных приложений в C# используются классы Thread и ThreadPool.

6. Асинхронное программирование: В C# также можно создавать асинхронные приложения, которые позволяют продолжать работу программы, даже если некоторые операции еще не завершены. Для создания асинхронных приложений используются ключевые слова async и await.

==Основные понятия==

Основными понятиями, связанными с сетевым программированием, являются:

IP-адрес – это уникальный адрес, идентифицирующий устройство в интернете или локальной сети. IP означает «Интернет-протокол» – набор правил, регулирующих формат данных, отправляемых через интернет или локальную сеть.

По сути, IP-адрес – это идентификатор, позволяющий передавать информацию между устройствами в сети: он содержит информацию о местоположении устройства и обеспечивает его доступность для связи. IP-адреса позволяют различать компьютеры, маршрутизаторы и веб-сайты в интернете и являются важным компонентом работы интернета.

Хост — любое устройство, предоставляющее сервисы формата «клиент-сервер» в режиме сервера по каким-либо интерфейсам и уникально определённое на этих интерфейсах.

1. Клиент-серверная архитектура: Клиент-серверная архитектура - это модель сетевого взаимодействия, где одно устройство (сервер) предоставляет услуги или ресурсы, а другое устройство (клиент) запрашивает эти услуги или ресурсы. Сервер и клиент могут обмениваться данными и сообщениями через сеть.

Клиент - это программа, которая запускается на устройстве пользователя и предоставляет пользовательский интерфейс для взаимодействия с приложением. Клиент обычно инициирует запросы к серверу, получает ответы и обрабатывает их для отображения пользователю.

Сервер - это программа, которая запускается на удаленном компьютере и предоставляет данные и функции, доступные клиентам через сетевые соединения. Сервер обрабатывает запросы от клиентов, обычно выполняет операции над данными, использует базы данных и возвращает результаты клиенту.

Соединение между клиентом и сервером осуществляется через сеть. Клиент и сервер могут быть реализованы на разных устройствах, например, клиент может быть запущен на персональном компьютере, а сервер - на удаленном сервере, который управляется компанией или облачным провайдером.

В клиент-серверной архитектуре могут использоваться различные типы соединений, например, TCP/IP, HTTP, FTP и другие. Клиент и сервер могут использовать различные языки программирования и технологии, но они должны использовать совместимый протокол для обмена данными.

==Недостатки клиент-серверной архитектуры==

Преимущества клиент-серверной архитектуры включают:

- Разделение логики приложения между клиентом и сервером, что может улучшить производительность и масштабируемость приложения.
- Централизованное хранение данных на сервере, что может уменьшить риск потери данных и облегчить обновление и управление данными.
- Распределенная обработка запросов, что может снизить нагрузку на клиентские устройства и ускорить ответы на запросы.
- Улучшенная безопасность, так как сервер может управлять доступом к данным и ресурсам.

Недостатки клиент-серверной архитектуры включают:

- Необходимость надежной и быстрой сети для обмена данными между клиентом и сервером.
- Необходимость разработки и поддержки двух разных программных приложений - клиента и сервера.
- Увеличенная сложность в реализации и тестировании приложения.

Мы постоянно пользуемся клиент-серверное взаимодействие, это все наши сайты, браузеры - всё это является результатом этого взаимодействия.

По своей сути сервер - это обычный компьютер, на котором запущено специальное приложение, которое обрабатывает входящие данные определённым образом. То есть это приложение отвечающее за логику, там происходит работа с данными большого объёма.

Сервер работает по принципу ожидания. То есть запущено приложение которое ждёт входного сигнала, когда поступает какой-либо входящий сигнал, он этот сигнал читает и обрабатывает. Если синал корректный - выполняет какие-то ддействия и даёт ответ, если запрос не корректный, посылает сообщение об ошибке, либо вообще игнориует клиентское приложение.

Можно сказать так - сервер это мозг, а пользовательское приложение обращается к мозгу за какой-то информацией и получает ответ.

Самый наиболее распостранённый пример клиентского приложения это браузер. Браузер посылает запрос на сервер сайта типа вот открой эту страничку, браузер это обрабатывает и отправяет ему соответсвующий html код, который уже отображется в клиентском приложении.

Серверное приложение всегда находится в режиме ожидания, в режиме прослушивания, как только приходит сигнал, серверное приложение реагирует и начинает обрабатывать этот запрос (посылать не посылать данные).

Это тоже самое если бы вы подошли к человеку,человек сидел на скамейке, никого не трогал, находился в расслабленном состоянии, а вы подошли к нему и спросили что-то. То же самое происходит с сервером. Человек что-то обдумывает и даёт вам ответ.

1. Протоколы: Протоколы - это наборы правил и форматов данных, которые используются для обмена информацией между устройствами в сети. Протоколы могут определять формат сообщений, порядок их передачи, правила обработки ошибок и другие аспекты сетевого взаимодействия. Некоторые из наиболее распространенных протоколов включают TCP, UDP, HTTP, FTP и другие.

Два самых распостранённый сетевых протокола это TCP и UDP (протоколы обмена).

TCP устанавливает чёткое соединение между клиентом и сервером (гарантия того что всё будет доставлено, будет посылать запрос до тех пор пока не будет подтверждения о том что всё получено, пока конечно привышено количество запросов). Соответственно если вы отправили нное кол-во раз и не удалось, он выдаёт сообщение об ошибке (слишком долгое ожидание). Минус TCP - работает медленно, создаёт большую нагрузку и уходит много времени на установку соединения

Типа TCP - это когда вы подошли к человеку и лично что-то сказали, убедились что он вас точно видит и слышит.

A UDP протокол - это когда вы взяли микрофон и что-то сказали толпе людей, кто-то отвернулся, кто-то не услышал и вас это не волнует.

3. IP-адреса и порты: IP-адреса и порты - это основные идентификаторы устройств в сети. IP-адрес используется для идентификации устройства в сети, а порт - для идентификации конкретного приложения на устройстве. Приложение может быть доступно на одном или нескольких портах.

==4. Сокеты: == Сокеты - это программные интерфейсы для взаимодействия приложений через сеть. Сокеты позволяют приложениям устанавливать соединения, передавать данные и закрывать соединения. Сокеты могут использоваться как на стороне клиента, так и на стороне сервера.

5. Многопоточность: Многопоточность - это возможность приложения обрабатывать несколько задач одновременно. В сетевом программировании многопоточность может использоваться для обработки множества запросов одновременно и улучшения производительности приложения.

6. Асинхронное программирование: Асинхронное программирование - это подход, при котором приложение может продолжать работу, не ожидая завершения операции ввода-вывода или других блокирующих операций. В сетевом программировании асинхронное программирование позволяет приложению эффективно управлять множеством соединений и обрабатывать большие объемы данных.

7. Базы данных: Базы данных - это наборы данных, организованные в определенном формате и доступные для использования приложениями. В сетевом программировании базы данных используются для хранения и обмена данными между приложениями. Различные типы баз данных могут использоваться для различных целей, например, для хранения пользовательских данных, конфигурационных файлов и других типов информации.

==Алогритм работы сервера:==

Алгоритм работы сервера в зависимости от его назначения может иметь различные особенности. Однако, общий алгоритм работы может быть описан следующим образом:

1. Запуск сервера. При запуске сервера создаются необходимые системные ресурсы, например, сокеты для приема и передачи данных, а также память для хранения информации о клиентах и их запросах.

2. Ожидание подключений. Сервер ожидает поступления запросов на подключение от клиентов. Когда клиент подключается, сервер создает новый поток выполнения или процесс для обработки его запроса.

3. Обработка запроса. Сервер получает запрос от клиента и анализирует его, определяет, какой тип запроса это и как ему следует быть обработан. Это может включать в себя проверку аутентификации клиента, чтение или запись данных в базу данных, выполнение каких-либо действий и т.д.

4. Отправка ответа. После обработки запроса сервер формирует ответ и отправляет его клиенту через сетевое соединение. Это может включать в себя передачу данных,

сообщений об ошибках или других сообщений, которые могут быть полезны для клиента.

5. Закрытие соединения. После того, как ответ был отправлен клиенту, сервер закрывает соединение и освобождает свои ресурсы, чтобы быть готовым к обработке новых запросов от клиентов.

6. Повторение процесса. Шаги 2-5 повторяются для каждого нового подключения, пока сервер не будет остановлен или пока не будет достигнут предел количества одновременных подключений.

В C#

==Socket TCP server== - это сервер, который использует протокол TCP/IP для обмена данными с клиентами, подключенными к нему через сокеты. Протокол TCP/IP - это стандартный протокол для сетевого взаимодействия, который обеспечивает надежную передачу данных в сети.

В сетевом программировании сервер создает сокет и прослушивает определенный порт на своем хосте, ожидая подключения клиентов. Когда клиент подключается к серверу, сервер создает новый сокет для обмена данными с этим клиентом. Обмен данными между клиентом и сервером происходит через созданный сокет.

==Алгоритм работы==

Один из основных компонентов сервера TCP - это слушающий сокет (listening socket), который прослушивает определенный порт на хосте сервера и ожидает новых подключений. Когда новое подключение устанавливается, сервер создает новый сокет для обмена данными с клиентом, который был только что подключен.

Для создания TCP-сервера на C# используется класс System.Net.Sockets.TcpListener.

`TcpListener` - это класс в .NET Framework, который предоставляет простой интерфейс для прослушивания входящих соединений TCP/IP и принятия их. Он используется для создания серверных приложений, которые прослушивают сетевые порты и принимают входящие соединения от клиентов.

Некоторые из наиболее часто используемых методов и свойств `TcpListener` следующие:

Методы:

- `Start()` - запускает прослушивание входящих соединений.
- `Stop()` - останавливает прослушивание входящих соединений.
- `AcceptTcpClient()` - принимает входящее соединение и возвращает объект `TcpClient`, который можно использовать для взаимодействия с клиентом.
- `BeginAcceptTcpClient(AsyncCallback callback, object state)` - начинает асинхронный процесс принятия входящих соединений. При завершении операции, в результате которой получен объект `TcpClient`, вызывается метод обратного вызова `callback`.

Свойства:

- `LocalEndpoint` - возвращает локальную конечную точку прослушивания.
- `Pending()` - возвращает значение, указывающее, есть ли ожидающие соединения клиента.
- `Server` - возвращает объект `Socket`, который используется для прослушивания входящих соединений.

Класс `NetworkStream` предоставляет интерфейс для чтения и записи данных через сетевой сокет. Некоторые из наиболее часто используемых методов и свойств `NetworkStream` следующие:

Методы:

- `Read(byte[] buffer, int offset, int size)` - читает данные из сетевого потока и сохраняет их в буфере. Параметры `buffer` - массив байтов для чтения, `offset` - смещение в массиве для начала чтения и `size` - количество байтов для чтения.
- `Write(byte[] buffer, int offset, int size)` - записывает данные в сетевой поток из буфера. Параметры `buffer` - массив байтов для записи, `offset` - смещение в массиве для начала записи и `size` - количество байтов для записи.

Свойства:

- `CanRead` - возвращает значение, указывающее, поддерживает ли сетевой поток чтение.
- `CanWrite` - возвращает значение, указывающее, поддерживает ли сетевой поток запись.
- `DataAvailable` - возвращает значение, указывающее, есть ли доступные данные для чтения из сетевого потока.
- `ReadTimeout` - устанавливает или возвращает время ожидания в миллисекундах для операции чтения из сетевого потока.
- `WriteTimeout` - устанавливает или возвращает время ожидания в миллисекундах

TCPClient - это класс в пространстве имен System.Net.Sockets, который представляет собой клиентское приложение для TCP-соединения. TCPClient обеспечивает установку соединения с сервером и передачу данных между клиентом и сервером.

Методы:

- `Connect` : устанавливает соединение с сервером, указанным в конструкторе. Если соединение не установлено, метод генерирует исключение. Пример использования:
- `GetStream` : возвращает объект NetworkStream для текущего соединения, который можно использовать для чтения и записи данных. Пример использования:
- `Close` : закрывает соединение с сервером и освобождает все ресурсы, связанные с текущим экземпляром TCPClient. Пример использования:
- `Connected` : указывает, установлено ли соединение с сервером. Если соединение установлено, свойство возвращает true, в противном случае - false. Пример использования:

Свойства:

- `ReceiveTimeout` : указывает время ожидания приема данных в миллисекундах. Если за это время не получены данные, метод Read() генерирует исключение. По умолчанию это свойство установлено в 0, что означает бесконечное ожидание. Пример использования:
- `SendTimeout` : указывает время ожидания отправки данных в миллисекундах. Если за это время данные не отправлены, метод Write() генерирует исключение. По умолчанию это свойство установлено в 0, что означает бесконечное ожидание. Пример использования:
- `Dispose` : освобождает все ресурсы, связанные с текущим экземпляром TCPClient. Этот метод может вызываться явно или с помощью конструкции using. Пример использования:

Из этого следует что

TCPListenet позволяет создать серверное приложение, которое ожидает входящих соединений от клиентских приложений. Как только клиентское приложение устанавливает соединение с сервером, **TCPListener** создаёт новый экземляр класса **TCPClient**, который представляет собой соединение между клиентом и сервером. Далее с помощью** NetworkStream**, котороый может быть получен из экземпляра** TcpClient**, сервер может отправлять и принимать данные от клиента.

Пример кода на C#, демонстрирующий создание TCP-сервера:

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

class TCPServer {
    static void Main() {
        // Создаем TcpListener, который будет прослушивать входящие соединения на порту 8080
        TcpListener server = new TcpListener(IPAddress.Parse("127.0.0.1"), 8080);
        // Запускаем TcpListener
        server.Start();
        Console.WriteLine("Server started");

        while (true) {
            // Принимаем входящее соединение от клиента и создаем новый TcpClient для обмена
            TcpClient client = server.AcceptTcpClient();
            Console.WriteLine("Client connected");

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

            // Создаем буфер для чтения данных от клиента
            byte[] buffer = new byte[1024];

            // Читаем данные от клиента и записываем их в буфер
            int bytesRead = stream.Read(buffer, 0, buffer.Length);

            // Преобразуем байты в строку
            string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);
```

```
            Console.WriteLine("Received: {0}", data);

            // Создаем ответное сообщение
            byte[] response = Encoding.ASCII.GetBytes("Hello, client!");

            // Отправляем ответное сообщение клиенту через сетевой поток
            stream.Write(response, 0, response.Length);
            Console.WriteLine("Sent: {0}", Encoding.ASCII.GetString(response));

            // Закрываем TcpClient и NetworkStream
            client.Close();
            Console.WriteLine("Client disconnected");
        }
    }
}
```

Код клиентского приложения может выглядеть так:

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

class TCPClientExample
{
    static void Main()
    {
        string serverIp = "127.0.0.1"; // IP-адрес сервера
        int port = 8080; // порт сервера
        try
        {
            // Создание объекта TCPClient и подключение к серверу
            TcpClient client = new TcpClient(serverIp, port);

            // Получение потока для отправки сообщений на сервер
            NetworkStream stream = client.GetStream();

            // Отправка сообщения на сервер
            string message = "Hello, server!";
            byte[] data = Encoding.UTF8.GetBytes(message);
            stream.Write(data, 0, data.Length);

            // Получение ответа от сервера
            data = new byte[256];
            StringBuilder response = new StringBuilder();
            int bytes = stream.Read(data, 0, data.Length);
            response.Append(Encoding.UTF8.GetString(data, 0, bytes));
            Console.WriteLine("Server response: {0}", response);

            // Закрытие соединения
            stream.Close();
            client.Close();
```

```
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception: {0}", e);
        }
    }
}
```

IP-адрес 127.0.0.1 является специальным адресом, который называется "localhost" или "loopback". Этот адрес обычно используется для тестирования сетевых приложений на одном компьютере без реальной сети.

Когда вы используете IP-адрес 127.0.0.1 в качестве адреса сервера, вы подключаетесь к тому же компьютеру, на котором запущен клиент. Это удобно для отладки и тестирования приложений, поскольку вы можете запустить сервер и клиент на одном и том же компьютере без необходимости наличия физической сети.

Класс WebClient - это класс в пространстве имен System.Net, который предоставляет простой и удобный способ выполнения HTTP-запросов и получения HTTP-ответов от веб-серверов.

Класс WebClient поддерживает множество функций, включая следующие:

1. Загрузка данных из удаленного ресурса в виде массива байтов, строки или файла на локальном диске.
2. Загрузка данных асинхронно с использованием асинхронных методов.
3. Отправка данных на сервер методом POST или PUT.
4. Использование прокси-сервера для доступа к удаленному ресурсу.
5. Установка параметров запроса, таких как заголовки, тип содержимого и кодировка.
6. Поддержка авторизации на удаленном сервере.
7. Использование куки для сохранения сеанса между запросами.

Пример использования WebClient для загрузки данных с удаленного сервера в виде строки:

```
using System.Net;

class Program
{
    static void Main(string[] args)
    {
        WebClient client = new WebClient();
        string result = client.DownloadString("https://www.example.com");
        Console.WriteLine(result);
    }
}
```

Этот пример загружает данные с URL-адреса https://www.example.com в виде строки и выводит их в консоль.

Класс WebClient предоставляет множество методов и свойств для работы с сетевыми запросами и ответами. Ниже приведены наиболее распространенные методы и свойства:

Методы:

- DownloadData(string address): Загружает содержимое указанного ресурса в виде массива байтов.
- DownloadFile(string address, string fileName): Загружает содержимое указанного ресурса и сохраняет его в указанный файл на локальном диске.
- DownloadString(string address): Загружает содержимое указанного ресурса в виде строки.
- UploadData(string address, byte[] data): Отправляет указанные данные на указанный ресурс методом POST.
- UploadFile(string address, string fileName): Отправляет содержимое указанного файла на указанный ресурс методом POST.
- UploadString(string address, string data): Отправляет указанные данные на указанный ресурс методом POST.

Свойства:

- BaseAddress: Получает или задает базовый адрес URI для всех запросов в WebClient.
- Headers: Получает коллекцию заголовков, которые будут отправлены с каждым запросом.
- Credentials: Получает или задает объект, содержащий информацию об авторизации, который будет использоваться для каждого запроса.
- Proxy: Получает или задает объект, содержащий информацию о прокси-сервере, который будет использоваться для каждого запроса.
- UseDefaultCredentials: Получает или задает значение, указывающее, следует ли использовать учетные данные пользователя по умолчанию для каждого запроса.

Загрузка и сериализация данных с сайта

Пример кода для загрузки и десериализации данных с сайта jsonplaceholder:

```
using System.Net;
using Newtonsoft.Json;

class Program
{
    static void Main(string[] args)
    {
        using (WebClient client = new WebClient())
        {
            string jsonData = client.DownloadString("https://jsonplaceholder.typicode.com/pos
            List<Post> posts = JsonConvert.DeserializeObject<List<Post>>(jsonData);

            foreach (Post post in posts)
```

```
            {
                Console.WriteLine("Title: {0}", post.Title);
                Console.WriteLine("Body: {0}", post.Body);
            }
        }
    }
}

public class Post
{
    public int UserId { get; set; }
    public int Id { get; set; }
    public string Title { get; set; }
    public string Body { get; set; }
}
```

Этот пример загружает данные с ресурса https://jsonplaceholder.typicode.com/posts в формате JSON с помощью метода DownloadString() и десериализует их в объекты класса Post с помощью метода DeserializeObject() из библиотеки Newtonsoft.Json.

Затем мы можем обработать каждый пост и вывести его заголовок и тело на консоль. В этом примере мы создали класс Post для хранения данных каждого поста. Это можно настроить в соответствии с требованиями конкретного приложения.

Для загрузки файла по сети с использованием метода `client.DownloadFile` необходимо указать адрес ресурса и путь к файлу на локальном диске, на который будет сохранен загруженный файл. Метод `DownloadFile` выполнит загрузку файла и автоматически сохранит его на локальный диск.

Пример кода:

```
using System.Net;

class Program
{
    static void Main(string[] args)
    {
        using (WebClient client = new WebClient())
        {
            string remoteUri = "http://example.com/file.zip";
            string fileName = @"C:\Downloads\file.zip";
            client.DownloadFile(remoteUri, fileName);
        }
    }
}
```

В этом примере мы создаем новый экземпляр класса WebClient и указываем адрес удаленного файла в переменной `remoteUri` и путь к файлу, в который будет сохранен загруженный файл, в переменной `fileName`. Метод `DownloadFile` загружает файл и сохраняет его в указанном файле на локальном диске.

Важно убедиться, что у приложения есть права на запись в указанную директорию на локальном диске. Если это не так, то загрузка файла завершится неудачей.