

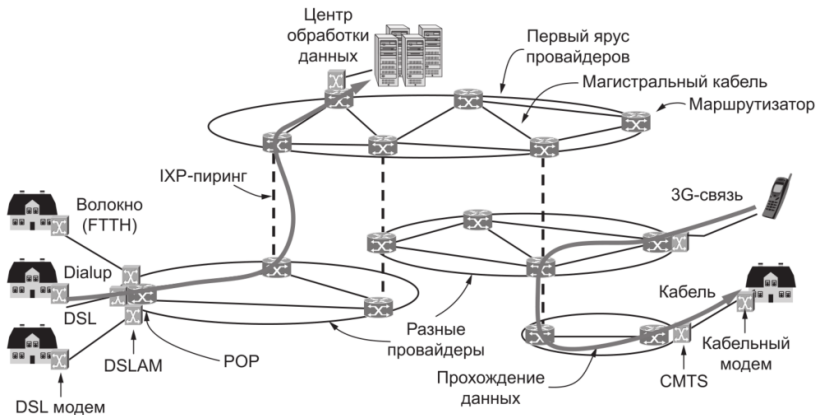
# Работа с сетью

## Низкий уровень

Юрий Литвинов  
yurii.litvinov@gmail.com

29.09.2020г

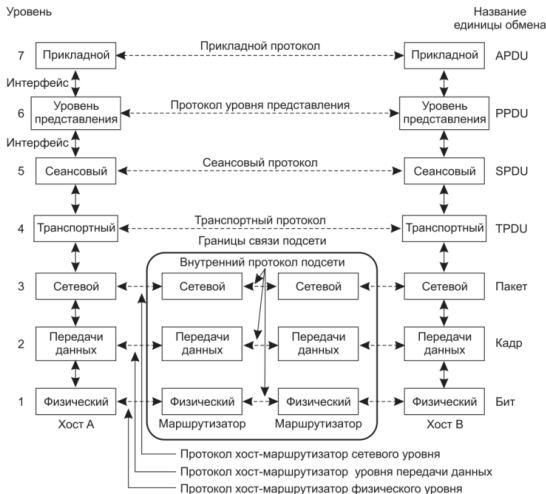
# Архитектура глобальной сети



© Э. Таненбаум

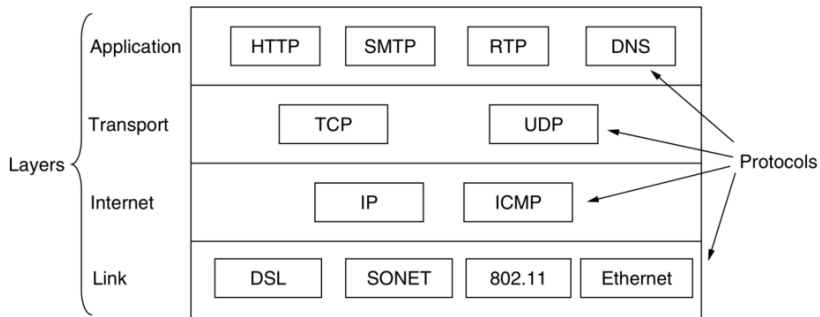
# Уровневая архитектура

## Модель OSI



© Э. Таненбаум

# Модель TCP/IP



© Э. Таненбаум

# Физический уровень

- ▶ Физические параметры канала (электрические, электромагнитные, ...)
- ▶ Ethernet (витая пара), USB, xDSL, Bluetooth, IEEE 802.11 (WiFi), оптические сети, спутниковая связь, мобильные сети (GSM, EDGE, LTE) и т.д.
  - ▶ RFC 1149 “IP over Avian Carriers” (<https://tools.ietf.org/html/rfc1149>)
- ▶ Отвечает только за передачу сигнала в рамках среды распространения между двумя точками
- ▶ Вопросы кодирования битов уровнями сигнала, синхронизации, помехоустойчивости, мультиплексирования
- ▶ Передаёт биты или блоки битов

# Канальный уровень

- ▶ Общение напрямую соединённых устройств сети
- ▶ PPP (Point to Point Protocol)
- ▶ Понятия MAC и LLC (Logical Link Control)
  - ▶ MAC-адрес: D8-FB-5E-E5-55-67
- ▶ Вопросы коррекции ошибок физического уровня (коды Хэмминга, Рида-Соломона, свёрточные коды и прочая алгебра с теорией чисел), повтора передачи пропавших данных, управления скоростью передачи
- ▶ Передаёт фреймы (или кадры)

# Сетевой уровень

- ▶ Сеть из нескольких устройств
- ▶ Вопросы поиска оптимального маршрута внутри сети (роутинга), передачи по принципиально разным сетям (например, один пакет по оптоволокну, второй — через спутник)
- ▶ IP (Internet Protocol)
- ▶ Понятие IP-адреса (IPv4, IPv6)
- ▶ Передаёт пакеты

# Транспортный уровень

- ▶ Соединение двух устройств через сеть
- ▶ Вопросы надёжности доставки, разделения-сборки сообщения, правильного порядка сообщений, подтверждения и повторной отправки
- ▶ Протоколы TCP (Transmission Control Protocol), UDP (User Datagram Protocol)
  - ▶ TCP — протокол, гарантирующий доставку данных в правильном порядке, без потерь и порчи, если это вообще возможно
    - ▶ Передача файлов, текстовых данных (включая веб-страницы), веб-сервисы
  - ▶ UDP — протокол, позволяющий отправлять “датаграммы” без гарантий их доставки или доставки в правильном порядке, но в разы быстрее TCP
    - ▶ Стриминг фильмов, музыки, компьютерные игры



# Сеансовый уровень

- ▶ Установление, поддержание и закрытие соединения
- ▶ Протокол TCP

# Уровень представления

- ▶ Кодировка и представление передаваемых данных
  - ▶ Шифрование
  - ▶ Сериализация/десериализация

# Прикладной уровень

- ▶ Общение конкретных приложений
- ▶ Протоколы HTTP, FTP, SMTP и т.д.
- ▶ Протоколы поверх HTTP: REST, SOAP и т.д.

# IP-адреса

- ▶ IPv4: 192.168.0.1 (4 байта)
  - ▶ Уникален в рамках подсети (не глобально уникальный)
  - ▶ 192.168.x.x, 172.16-31.x.x, 10.x.x.x — адреса, зарезервированные для локальных подсетей
  - ▶ 127.0.0.1 (точнее, 127.x.x.x) — loopback (локальный адрес самого компа), часто используется для отладки
  - ▶ Маска подсети — битовая маска, определяющая кусок IP-адреса
- ▶ IPv6: fe80::488f:1f6:9030:46c7%10 (16 байт)

# Формат пакета IPv4



© Э. Таненбаум

# Порты и сокеты

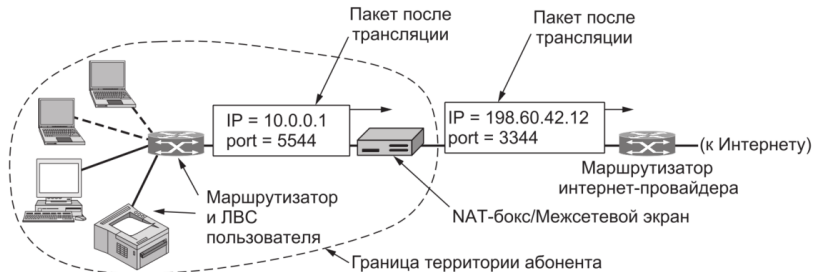
- ▶ Порт — число от 1 до 65535, привязан к сетевому интерфейсу
- ▶ Ресурс, управляемый ОС
- ▶ Типичные порты
  - ▶ 22 — SSH
  - ▶ 25 — SMTP
  - ▶ 80 — HTTP
  - ▶ 443 — HTTPS
  - ▶ 666 — Doom
  - ▶ Первые 1024 порта зарезервированы
- ▶ Ненужные порты обычно закрыты на уровне ОС (фаерволл), чтобы было труднее взломать компьютер — поэтому ваше первое сетевое приложение, скорее всего, не заработает
- ▶ Сокет — программный интерфейс к порту
- ▶ Сетевой стек — важная часть операционной системы, сокеты — способ для прикладного программиста с ним работать

# DNS, NAT

- ▶ DNS — сопоставление непонятным IP-адресам читаемых доменных имён
  - ▶ Более-менее глобальный сервис
  - ▶ DNS-запрос по доменному имени (google.com) возвращает IP-адрес (64.233.164.113), только после этого возможен “настоящий” запрос
  - ▶ Есть локальные DNS-сервера, есть общеизвестные (например, 8.8.8.8, Google Public DNS)
  - ▶ localhost — всегда (более-менее) раскрывается в 127.0.0.1
- ▶ NAT — Network Address Translation, механизм, позволяющий компьютерам с локальными IP получать ответы из Интернет (только если они инициировали запросы)
- ▶ Ports forwarding — механизм, позволяющий компьютерам за NAT принимать входящие запросы
- ▶ Прокси — программа, которая пересылает запросы (и может делать с ними что-нибудь)

# Как работает NAT

Или ещё одна причина, почему ваше первое сетевое приложение не заработает



© Э. Таненбаум



# Полезные консольные команды

- ▶ `ping` — проверка соединения с указанным IP или доменным именем, показывает время отклика узла
  - ▶ Удалённый компьютер имеет право не отвечать
- ▶ `tracert` (`traceroute`) — показывает все узлы, через которые шёл запрос со временами их отклика (если они хотят откликнуться)
  - ▶ Хороший способ диагностировать проблемы с интернетом
- ▶ `ipconfig` под Windows, `ifconfig` под Linux — узнать всё про локальный сетевой интерфейс (IP-адреса, MAC-адреса, используемые DNS и т.д.)
  - ▶ Наиболее полезен `ipconfig /all`

## Полезные консольные команды (2)

- ▶ netcat, nc — позволяет опросить указанный порт или наоборот, прикинуться сервером, работающим по данному порту, очень полезна при отладке сетевых приложений
  - ▶ Под Windows не входит в стандартную поставку, надо ставить отдельно
- ▶ telnet — открывает TCP-соединение с заданным хостом на заданный порт
  - ▶ Пример:  
telnet smtp.gmail.com 25  
220 smtp.gmail.com ESMTP m71-v6sm2246896lje.84 - gsmtpt  
HELP  
214 2.0.0 <https://www.google.com/search?btnI&q=RFC+5321> m71-v6sm2246896lje.84  
- gsmtpt
  - ▶ Выйти — Ctrl + ']', quit
  - ▶ Под Windows не входит в стандартную поставку, надо ставить отдельно

# Работа с сетью в .NET

- ▶ Пространство имён System.Net
- ▶ Классы TcpListener, TcpClient, UdpClient — управляют стеком протоколов, предоставляют сокеты или потоки байтов
- ▶ Класс Socket — абстракция сетевого соединения (сокета)
- ▶ Чаще всего в реальной жизни обработка запросов на сервере асинхронна — каждый клиент обслуживается своей задачей в пуле потоков
- ▶ Dns — класс, отвечающий за работу с DNS-службой
- ▶ IPEndPoint — абстракция адреса (IP-адрес + порт)

# Минимальный пример, сервер

```
static void Main(string[] args)
{
    const int port = 8888;
    var listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
    Console.WriteLine($"Listening on port {port}...");
    using (var socket = listener.AcceptSocket())
    {
        var stream = new NetworkStream(socket);
        var streamReader = new StreamReader(stream);
        var data = streamReader.ReadToEnd();
        Console.WriteLine($"Received: {data}");
    }
    listener.Stop();
}
```

# Минимальный пример, клиент

```
static void Main(string[] args)
{
    const int port = 8888;
    using (var client = new TcpClient("localhost", port))
    {
        Console.WriteLine($"Sending to port {port}...");
        var stream = client.GetStream();
        var writer = new StreamWriter(stream);
        writer.Write("Hello, world!");
        writer.Flush();
    }
}
```

# Канал работает в обе стороны

## Сервер

```
static void Main(string[] args)
{
    const int port = 8888;
    var listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
    Console.WriteLine($"Listening on port {port}...");
    using (var socket = listener.AcceptSocket())
    {
        var stream = new NetworkStream(socket);
        var reader = new StreamReader(stream);
        var data = reader.ReadLine();
        Console.WriteLine($"Received: {data}");

        Console.WriteLine($"Sending \"Hi!\"");
        var writer = new StreamWriter(stream);
        writer.Write("Hi!");
        writer.Flush();
    }
    listener.Stop();
}
```

# Канал работает в обе стороны

## Клиент

```
static void Main(string[] args)
{
    const int port = 8888;
    using (var client = new TcpClient("localhost", port))
    {
        Console.WriteLine($"Sending \"Hello!\" to port {port}...");
        var stream = client.GetStream();
        var writer = new StreamWriter(stream);
        writer.WriteLine("Hello!");
        writer.Flush();

        var reader = new StreamReader(stream);
        var data = reader.ReadToEnd();
        Console.WriteLine($"Received: {data}");
    }
}
```

# Немного асинхронности

## Сервер

```
static async Task Main(string[] args)
{
    const int port = 8888;
    var listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
    Console.WriteLine($"Listening on port {port}...");
    using (var socket = await listener.AcceptSocketAsync())
    {
        var stream = new NetworkStream(socket);
        var reader = new StreamReader(stream);
        var data = await reader.ReadLineAsync();
        Console.WriteLine($"Received: {data}");

        Console.WriteLine($"Sending \"Hi!\"");
        var writer = new StreamWriter(stream);
        writer.AutoFlush = true;
        await writer.WriteAsync("Hi!");
    }
    listener.Stop();
}
```



# Или, более типично

## Сервер

```
static async Task Main(string[] args) {
    const int port = 8888;
    var listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
    Console.WriteLine($"Listening on port {port}...");
    while (true) {
        var socket = await listener.AcceptSocketAsync();
        Task.Run(async () => {
            var stream = new NetworkStream(socket);
            var reader = new StreamReader(stream);
            var data = await reader.ReadLineAsync();
            Console.WriteLine($"Received: {data}");

            Console.WriteLine($"Sending \"Hi!\"");
            var writer = new StreamWriter(stream);
            await writer.WriteAsync("Hi!");
            await writer.FlushAsync();

            socket.Close();
        });
    }
}
```

# Теперь можно писать и читать одновременно

Полнодуплексное соединение, на примере сервера

```
private static async Task Main(string[] args) {  
    ...  
    while (true) {  
        var client = await listener.AcceptTcpClientAsync();  
        Writer(client.GetStream());  
        Reader(client.GetStream());  
    }  
}
```

```
private static void Writer(NetworkStream stream) {  
    Task.Run(async () => {  
        ...  
    });  
}
```

```
private static void Reader(NetworkStream stream) {  
    Task.Run(async () => {  
        ...  
    });  
}
```

# Например

```
private static void Writer(NetworkStream stream)
{
    Task.Run(async () =>
    {
        var writer = new StreamWriter(stream) { AutoFlush = true };
        while (true)
        {
            Console.WriteLine(">");
            var data = Console.ReadLine();
            await writer.WriteLineAsync(data + "\n");
        }
    });
}
```

# UdpClient

## Сервер

```
static async Task Main(string[] args)
{
    const int port = 8888;
    var udpClient = new UdpClient(port);
    Console.WriteLine($"Listening on port {port}...");
    var received = await udpClient.ReceiveAsync();
    var data = Encoding.UTF8.GetString(received.Buffer);
    Console.WriteLine($"Received: {data}");
}
```

# UdpClient

## Клиент

```
static async Task Main(string[] args)
{
    const int port = 8888;
    var udpClient = new UdpClient();

    Console.WriteLine($"Sending \"Hello!\" to port {port}...");
    var data = Encoding.UTF8.GetBytes("Hello!");
    await udpClient.SendAsync(data, data.Length, "localhost", port);
}
```

Не следует посылать UDP-датаграммы более 508 байт размером