Содержание

1.Правила и принципы проектирования переносимых программ

Переносимость программного обеспечения — это способность программы функционировать на различных аппаратных и программных платформах с минимальными изменениями в исходном коде. Переносимость особенно важна в условиях разнообразия операционных систем (Windows, Linux, macOS), процессорных архитектур (x86, ARM, RISC-V) и устройств (ПК, смартфоны, встроенные системы). Для достижения переносимости необходимо следовать следующим принципам и правилам:

• Использование стандартных интерфейсов и библиотек.

Программисты должны отдавать предпочтение стандартизированным библиотекам и интерфейсам, которые поддерживаются большинством платформ. Например, стандарт POSIX (Portable Operating System Interface) предоставляет унифицированный набор функций для работы с файлами, процессами и потоками, совместимый с большинством UNIX-подобных систем и частично с Windows. Использование стандарта ANSI C или C++ также способствует переносимости. Проприетарные API, такие как WinAPI для Windows или Cocoa для macOS, следует избегать или изолировать с помощью абстракций.

• Избежание зависимости от специфических особенностей платформы.

Код не должен зависеть от уникальных характеристик оборудования или операционной системы. Например: Размер типов данных. Размер типа int может варьироваться (32 или 64 бита) в зависимости от архитектуры. Для переносимости следует использовать типы с фиксированным размером, такие как int32_t или uint64_t из заголовочного файла <cstdint> в C++. Системные пути. Пути к файлам различаются между системами (например, C:\folder\file в Windows против /folder/file в Linux). Для работы с путями рекомендуется использовать кроссплатформенные библиотеки, такие как std::filesystem в C++17 или модули os.path в Python. Байтовая последовательность (endianness). Разные процессоры используют big-endian или little-endian для хранения данных. Программист должен учитывать это при работе с сетевыми протоколами или бинарными файлами, используя функции преобразования, такие как htonl и ntohl в C.

• Выбор кроссплатформенных языков программирования.

Языки программирования с высоким уровнем абстракции упрощают переносимость. Java использует виртуальную машину JVM, которая изолирует код от особенностей платформы. Программа на Java будет работать на любой системе с установленной JVM. Python предоставляет интерпретатор, который абстрагирует детали операционной системы. Однако программист должен учитывать различия в поведении библиотек (например, threading в Windows и Linux). C/C++ требуют более тщательной работы с переносимостью, но использование кроссплатформенных фреймворков (Qt, Boost) и компиляторов (GCC, Clang) значительно упрощает задачу.

• Учет различий между системами.

Архитектура процессоров. Код, оптимизированный для x86, может некорректно работать на ARM из-за различий в наборах инструкций. Для переносимости следует избегать использования низкоуровневых инструкций (например, ассемблерных вставок). Кодировки символов. Разные системы могут использовать разные кодировки (UTF-8, UTF-16, ASCII). Для работы с текстом рекомендуется использовать библиотеки, такие как ICU (International Components for Unicode). Системные вызовы. Например, создание потоков или управление памятью различается между Windows и Linux. Для абстрагирования от таких различий используются библиотеки, такие как POSIX Threads (pthreads) или Boost.Thread.

• Тестирование на разных платформах.

Переносимость невозможно гарантировать без тестирования. Разработчик должен: Использовать виртуальные машины или контейнеры для тестирования на различных ОС. Применять системы непрерывной интеграции (CI), для автоматического запуска тестов на разных платформах. Проверять поведение программы в условиях ограниченных.

• Модульность и абстракция.

Код должен быть организован так, чтобы платформозависимые части были изолированы. Например, использование шаблона "Фасад" или выделение платформозависимого кода в отдельные модули позволяет легко заменять реализацию для каждой платформы.

• Документация и условные компиляции.

Для языков вроде C/C++ можно использовать директивы условной компиляции (#ifdef, #endif) для включения платформозависимого кода. Однако злоупотребление условной компиляцией усложняет поддержку кода, поэтому лучше минимизировать её использование. Применение этих принципов позволяет создавать программы, которые легко адаптируются к новым платформам, снижая затраты на разработку и поддержку.

2.Сетевое программное обеспечение. Архитектура сети. Инкапсуляция данных

Сетевое программное обеспечение — это комплекс программных компонентов, обеспечивающих взаимодействие устройств в компьютерных сетях. Оно включает в себя протоколы, драйверы, серверное и клиентское ПО, а также утилиты для диагностики и управления сетью. Основная цель сетевого ПО — обеспечить надежную, безопасную и эффективную передачу данных.

Архитектура сети

Архитектура сети определяет, как организованы устройства, протоколы и методы передачи данных в сети. Она включает следующие элементы:

- Топология сети.

Топология описывает физическую или логическую структуру сети:

o Звезда: все устройства подключены к центральному узлу (например, коммутатору). Проста в управлении, но уязвима при отказе центрального узла.

o Шина: все устройства подключены к общей линии передачи данных. Проста в реализации, но чувствительна к сбоям.

o Кольцо: устройства соединены последовательно, образуя замкнутый контур. Надежна, но добавление новых устройств может быть сложным.

o Ячеистая (mesh): каждое устройство связано с несколькими другими, что повышает отказоустойчивость, но увеличивает сложность. Топология влияет на производительность, масштабируемость и стоимость сети.

- Протоколы.

Протоколы — это наборы правил, определяющих формат, порядок и обработку данных. Примеры:

o TCP/IP: основа Интернета, включает протоколы IP (маршрутизация) и TCP (надежная доставка).

o HTTP/HTTPS: протоколы для передачи веб-страниц.

o FTP: протокол для передачи файлов. Протоколы стандартизированы международными организациями, такими как IETF (Internet Engineering Task Force).

- Методы передачи данных.

Сети используют различные методы передачи:

o Коммутация пакетов: данные разбиваются на пакеты, которые передаются независимо и собираются на стороне получателя (используется в Интернете).

o Коммутация каналов: создается выделенный канал между отправителем и получателем (используется в традиционной телефонии).

o Коммутация сообщений: данные передаются целиком, что редко используется в современных сетях.

- Модель OSI и TCP/IP.

Архитектура сети часто описывается через модели, такие как:

o Модель OSI (Open Systems Interconnection) делит сетевые функции на семь уровней:

1. Физический (передача битов через кабели, Wi-Fi).
2. Канальный (формирование кадров, контроль ошибок, например, Ethernet).
3. Сетевой (маршрутизация, например, IP).
4. Транспортный (надежность передачи, например, TCP, UDP).
5. Сеансовый (управление сессиями).
6. Представления (форматирование и шифрование данных).
7. Прикладной (интерфейс для приложений, например, HTTP).

o Модель TCP/IP более практична и включает четыре уровня: канальный, сетевой, транспортный и прикладной. Она широко используется в Интернете.

Инкапсуляция данных

Инкапсуляция данных — это процесс упаковки данных для передачи через сеть, при котором к данным добавляются служебные заголовки (и иногда трейлеры) на каждом уровне модели OSI или TCP/IP. Этот процесс можно представить как "упаковку письма в конверт" для доставки.

- Процесс инкапсуляции:

1. Прикладной уровень: приложение формирует данные (например, текст HTTP-запроса).
2. Транспортный уровень: данные разбиваются на сегменты (например, TCP-сегменты). К ним добавляется заголовок с номерами портов, порядковыми номерами и контрольной суммой.
3. Сетевой уровень: сегмент упаковывается в IP-пакет, к которому добавляется заголовок с IP-

адресами отправителя и получателя.

4. Канальный уровень: пакет превращается в кадр, к которому добавляются заголовок (MAC-адреса) и трейлер (для проверки целостности).

- Пример инкапсуляции в TCP/IP: Если пользователь отправляет веб-страницу через HTTPS:
  - Данные (HTML-код) шифруются на уровне SSL/TLS.
  - Шифрованные данные передаются в TCP, где формируется сегмент с заголовком (порт 443).
  - Сегмент инкапсулируется в IP-пакет с адресами отправителя и получателя.
  - Пакет превращается в кадр Ethernet с MAC-адресами сетевых интерфейсов.
- Декапсуляция: на стороне получателя процесс идет в обратном порядке — заголовки снимаются, и данные передаются приложению.

Инкапсуляция обеспечивает модульность и независимость уровней, позволяя каждому уровню обрабатывать свои задачи (маршрутизацию, контроль ошибок, форматирование) без вмешательства в работу других.

3. Основные проблемы, возникающие при разработке стека протоколов

Разработка стека протоколов — сложный процесс, связанный с необходимостью обеспечения совместимости, производительности, безопасности и масштабируемости. Основные проблемы включают:

- Совместимость.

Новые протоколы должны быть совместимы с существующими стандартами. Например:

- o Переход с IPv4 на IPv6 требует поддержки обоих протоколов, так как IPv4 все еще широко используется.
- o Протоколы верхнего уровня (например, HTTP/3) должны работать с нижними уровнями (UDP вместо TCP). Решение: разработка переходных механизмов (например, туннелирование IPv6 в IPv4) и тщательное тестирование на совместимость.

- Производительность.

Протоколы должны минимизировать задержки и обеспечивать высокую пропускную способность. Проблемы:

- o Высокая вычислительная нагрузка при обработке заголовков или шифровании.
- o Ограниченная пропускная способность в беспроводных сетях. Решение: оптимизация алгоритмов (например, сжатие заголовков в HTTP/2), использование аппаратного ускорения (например, для AES-шифрования) и адаптивные механизмы управления перегрузкой.

- Безопасность.

Сети уязвимы для атак, таких как перехват данных, подмена (spoofing) или атаки типа "отказ в обслуживании" (DDoS). Проблемы:

- o Внедрение шифрования (например, TLS) увеличивает накладные расходы.
- o Уязвимости в протоколах (например, устаревшие версии SSL). Решение: использование современных криптографических алгоритмов, многофакторной аутентификации и механизмов обнаружения атак.

- Масштабируемость.

Протоколы должны поддерживать рост числа пользователей, устройств и объема данных. Проблемы:

- o Ограничения адресации (например, нехватка адресов в IPv4).
- o Перегрузка сетевых узлов при увеличении трафика. Решение: использование протоколов с большим адресным пространством (IPv6), распределенных архитектур (CDN) и алгоритмов балансировки нагрузки.

- Надежность.

Протоколы должны гарантировать доставку данных даже в условиях сбоев. Проблемы:

- o Потеря пакетов из-за перегрузки или нестабильного соединения.
- o Дублирование или неправильный порядок доставки пакетов. Решение: внедрение механизмов подтверждения доставки (ACK в TCP), повторной передачи и управления порядком пакетов.

- Сложность реализации.

Разработка стека протоколов требует учета множества факторов (например, кроссплатформенности, поддержки устаревших систем), что усложняет код. Решение: использование модульного подхода, где каждый уровень протокола реализуется независимо.

- Стандартизация.

Протоколы должны быть согласованы между производителями и разработчиками, что требует длительных процессов стандартизации в организациях, таких как IETF или IEEE. Задержки в стандартизации могут замедлить внедрение новых технологий.

Решение этих проблем требует баланса между функциональностью, эффективностью и простотой, а также постоянного тестирования и обновления протоколов.

4. Службы и протоколы. Типы служб

Службы в компьютерных сетях — это функциональные возможности, предоставляемые сетью для приложений или пользователей. Службы реализуются через протоколы и классифицируются по способу взаимодействия между узлами.

- Службы без установления соединения (connectionless).

В таких службах данные передаются без предварительного установления канала связи. Пример: протокол UDP (User Datagram Protocol).

- o Характеристики:
- Быстрая передача, так как нет накладных расходов на установление и разрыв соединения.
- Нет гарантии доставки, контроля порядка или обработки ошибок.
- Подходит для приложений, где важна скорость, а потеря отдельных пакетов допустима (например, потоковое видео, DNS-запросы).
- o Пример: в видеоконференциях UDP используется для передачи видео, так как задержка важнее потери отдельных кадров.

- Службы с установлением соединения (connection-oriented).

Перед передачей данных создается виртуальное соединение, которое поддерживается до завершения обмена. Пример: протокол TCP (Transmission Control Protocol).

- o Характеристики:
- Гарантирует доставку данных, их порядок и отсутствие дублирования.
- Требует больше ресурсов из-за необходимости установления соединения (трехэтапное рукопожатие в TCP) и подтверждений.
- Подходит для приложений, где надежность критична (например, передача файлов, веб-страницы).
- o Пример: при загрузке файла через FTP TCP обеспечивает доставку всех данных без потерь.

- Службы с подтверждением.

Отправитель получает уведомление о том, что данные успешно доставлены. Это характерно для TCP, где получатель отправляет подтверждение (ACK) для каждого сегмента.

- o Характеристики:
- Повышает надежность, но увеличивает задержки из-за дополнительных сообщений.
- Используется в приложениях, где важна целостность данных (например, электронная почта).
- o Пример: протокол SMTP для отправки email использует подтверждения для гарантии доставки.

- Службы без подтверждения.

Подтверждение доставки не предусмотрено, что снижает накладные расходы, но увеличивает риск потери данных. Пример: UDP.

- o Характеристики:
- Минимальная нагрузка на сеть.
- Подходит для приложений реального времени, таких как VoIP или онлайн-игры.
- o Пример: в играх UDP передает данные о положении игроков, где важна скорость, а потеря пакета не критична.

- Дополнительные типы служб:
- o Службы с гарантированной пропускной способностью: обеспечивают фиксированную скорость передачи (используются в сетях QoS для видео или голоса).
- o Службы с приоритетами: пакеты с высоким приоритетом передаются раньше (например, в VoIP).

Каждая служба выбирается в зависимости от требований приложения: надежность против скорости, простота против функциональности.

5. Службы и протоколы. Примитивы служб

Примитивы служб — это базовые операции, которые сетевая служба предоставляет приложениям для взаимодействия с сетью. Они представляют собой интерфейс между приложением и протоколами, обеспечивая стандартизированный способ выполнения сетевых операций. Примитивы могут варьироваться в зависимости от типа службы (с установлением соединения или без).

Основные примеры примитивов:

- Connect.

Устанавливает соединение между двумя узлами сети. Используется в службах с установлением соединения, таких как TCP.

- o Параметры: адрес получателя (IP и порт), параметры соединения (например, тайм-аут).
- o Пример: приложение вызывает connect() для соединения с веб-сервером по HTTP.
- o Реализация: в TCP выполняется трехэтапное рукопожатие (SYN, SYN-ACK, ACK).

- Send.

Отправляет данные от одного узла к другому.

- o Параметры: данные (буфер), адрес получателя, флаги (например, приоритет).
- o Пример: отправка HTTP-запроса на сервер через send().
- o Реализация: данные разбиваются на сегменты (TCP) или датаграммы (UDP) и передаются через нижние уровни.

- Receive.

Принимает данные от отправителя.

- o Параметры: буфер для хранения данных, максимальный размер данных.
- o Пример: веб-браузер вызывает receive() для получения HTML-страницы.
- o Реализация: протокол проверяет порядок и целостность данных (в TCP) или просто передает их приложению (в UDP).

- Disconnect.

Завершает соединение между узлами.

- o Параметры: идентификатор соединения, тип завершения (мягкое или жесткое).
- o Пример: закрытие соединения с сервером после завершения передачи файла.
- o Реализация: в TCP выполняется четырехэтапное завершение (FIN, ACK, FIN, ACK).

Типы примитивов:

- Синхронные примитивы.

Программа ожидает завершения операции перед продолжением выполнения. Например, вызов send() блокирует выполнение, пока данные не будут переданы.

- o Преимущества: простота реализации, предсказуемое поведение.
- o Недостатки: снижение производительности в многозадачных приложениях.

- Асинхронные примитивы.

Операция выполняется в фоновом режиме, а приложение продолжает работу. Программа получает уведомление (например, через callback или событие) по завершении операции.

- o Преимущества: высокая производительность, особенно в приложениях реального времени.
- o Недостатки: сложность реализации, необходимость управления состоянием.
- o Пример: асинхронный send() в Node.js с использованием промисов.

- Дополнительные примитивы:

- o Listen: ожидание входящих соединений (используется серверами, например, в HTTP).
- o Accept: принятие входящего соединения от клиента.
- o Bind: привязка сокета к определенному адресу и порту.

Реализация примитивов

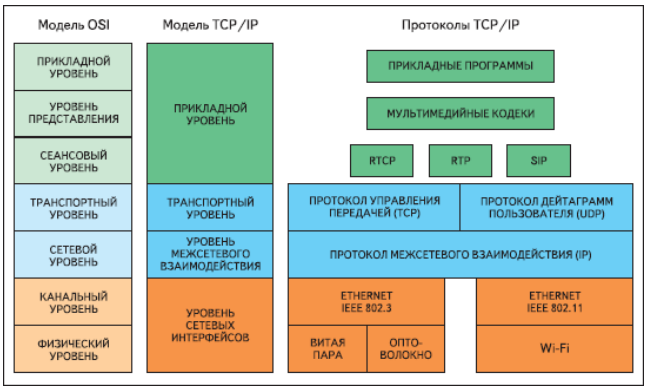
Примитивы реализуются через API операционной системы (например, сокеты в UNIX или Winsock в Windows). Программист взаимодействует с примитивами через высокоуровневые библиотеки (например, socket в Python или java.net в Java), которые скрывают детали низкоуровневой работы с протоколами.

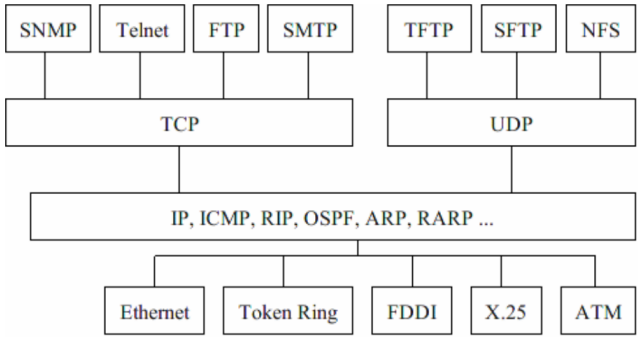
Пример использования примитивов:

Клиент-серверное приложение на TCP:

1. Сервер вызывает bind() и listen() для ожидания соединений.
2. Клиент вызывает connect() для установления соединения.
3. Клиент и сервер обмениваются данными через send() и receive().
4. После завершения обмена вызывается disconnect().

6. Эталонная модель TCP/IP.





1**. Уровень сетевого интерфейса** (Link Layer / Network Interface Layer)

Этот уровень объединяет функции физического и канального уровней модели OSI. Он отвечает за передачу данных через физическую сеть и взаимодействие с оборудованием. Формирование кадров (фреймов) данных для передачи через физическую среду. Управление доступом к среде. Обнаружение и исправление ошибок на уровне. Преобразование данных в сигналы и обратно. Определение физических адресов (MAC-адресов) для идентификации устройств в локальной сети.

- Ethernet (IEEE 802.3) — для проводных сетей. - Wi-Fi (IEEE 802.11) — для беспроводных сетей.
- PPP (Point-to-Point Protocol) — для прямых соединений.
- Технологии вроде Bluetooth или оптоволокна.

2**. Сетевой уровень** (Internet Layer)

Этот уровень отвечает за маршрутизацию и доставку пакетов между устройствами в разных сетях, даже если они находятся на разных концах света. Логическая адресация: использует IP-адреса (например, 192.168.1.1) для идентификации устройств в глобальной сети. Маршрутизация: определяет путь, по которому данные (пакеты) будут передаваться от отправителя к получателю через промежуточные узлы (маршрутизаторы).

- IP (Internet Protocol): IPv4 или IPv6. IP передаёт пакеты, но не гарантирует их доставку, порядок или отсутствие дубликатов.
- ICMP (Internet Control Message Protocol): используется для диагностики (например, команда `ping`).

3. **Транспортный уровень** (Transport Layer)

Этот уровень обеспечивает передачу данных между хостами, добавляя надёжность, управление потоком и контроль ошибок. Установление соединения (для TCP): гарантирует, что данные дойдут до получателя. Сегментация и сборка: делит данные на сегменты (или датаграммы) и собирает их на стороне получателя. Контроль порядка. Управление потоком. Обнаружение и исправление ошибок.

- TCP (Transmission Control Protocol): Надёжный, с установлением соединения. Гарантирует доставку, порядок и отсутствие ошибок.
- UDP (User Datagram Protocol): Ненадёжный, без установления соединения. Быстрее, чем TCP, но не гарантирует доставку. Используется для потокового видео, игр, DNS-запросов.
- Другие протоколы, такие как SCTP (Stream Control Transmission Protocol), иногда применяются для специфических задач.

4. **Прикладной уровень** (Application Layer)

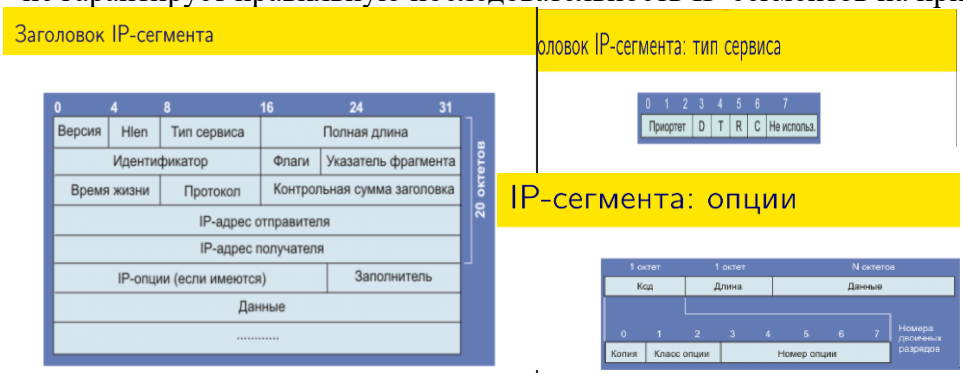
Этот уровень объединяет функции сеансового, представления и прикладного уровней модели OSI. Он отвечает за взаимодействие приложений с сетью и предоставление данных пользователю. Предоставление интерфейса для приложений: позволяет программам (браузерам, почтовым клиентам) отправлять и получать данные через сеть. Управление сеансами: поддерживает связь между приложениями (например, синхронизация в видеозвонках). Поддержка специфических функций: например, передача файлов, отправка писем, доступ к веб-страницам.

- HTTP/HTTPS: Для доступа к веб-страницам.
- FTP: Для передачи файлов.
- SMTP, IMAP, POP3: Для работы с электронной почтой.
- DNS: Для преобразования доменных имён (например, google.com) в IP-адреса.
- RTP (Real-time Transport Protocol): Для потоковой передачи мультимедиа (видео, аудио).
- SIP (Session Initiation Protocol): Для установления VoIP-звонков (например, в Skype или Zoom).

7. Протокол IPv4. Адресация IPv4.

Протокол IPv4

- реализует обмен информации пакетами (IP-сегментами) (максимальный размер – 65535 байт);
- является протоколом взаимодействия без установления логического соединения;
- для адресации узлов сети используется адрес длиной 4 байта; обеспечивает в случае необходимости фрагментацию IP-сегментов;
- IP-сегменты имеют конечное время жизни в сети;
- не гарантирует надежность доставки IP-сегментов адресату;
- не имеет средств управления интенсивностью передачи IP-сегментов посылающей стороной (flow control);
- не гарантирует правильную последовательность IP-сегментов на принимающей стороне.



Тип сервиса:

Приоритеты: 0 - обычный уровень 1 - приоритетный 2 – немедленный 3 - срочный 4 – экстренный 5 - ceitic/ecp 6 - межсетевое управление 7 - сетевое управление

Формат поля TOS определен в документе RFC-1349. D=1 требует минимальной задержки, T=1 - высокую пропускную способность, R=1 - высокую надежность, C=1 - низкую стоимость.

Опции:

Конец списка опций – используется, если опции не укладываются в поле заголовка (смотри также поле "заполнитель")

Никаких операций - используется для выравнивания октетов в списке опций

Ограничения, связанные с секретностью (для военных приложений)

Свободная маршрутизация. Используется для того, чтобы направить дейтограмму по заданному маршруту

Запись маршрута – используется для трассировки

Идентификатор потока – устарело.

Жесткая маршрутизация – используется, чтобы направить дейтограмму по заданному маршруту Временная метка Интернет

Адресация IPv4.

IP address Classes

Class	# Network Bits	# Hosts Bits	Decimal Address Range	Subnet mask
Class A	8 bits	24 bits	1-126	255.0.0.0
Class B	16 bits	16 bits	128-191	255.255.0.0
Class C	24 bits	8 bits	192-223	255.255.255.0
Class D	Reserved for Multicasting		224-239	N/A
Class E	Reserved for R. & D		240-255	N/A

Специальные IP адреса

CIDR address block	Description	Reference
0.0.0.0/8	Current network (only valid as source address)	RFC 1700
10.0.0.0/8	Private network	RFC 1918
127.0.0.0/8	Loopback	RFC 5735
169.254.0.0/16	Link-Local	RFC 3927
172.16.0.0/12	Private network	RFC 1918
192.0.0.0/24	Reserved (IANA)	RFC 5735
192.0.2.0/24	TEST-NET-1, Documentation and example code	RFC 5735
192.88.99.0/24	IPv6 to IPv4 relay	RFC 3068
192.168.0.0/16	Private network	RFC 1918
198.18.0.0/15	Network benchmark tests	RFC 2544
198.51.100.0/24	TEST-NET-2, Documentation and examples	RFC 5737
203.0.113.0/24	TEST-NET-3, Documentation and examples	RFC 5737
224.0.0.0/4	Multicasts (former Class D network)	RFC 3171
240.0.0.0/4	Reserved (former Class E network)	RFC 1700
255.255.255.255	Broadcast	RFC 919

IP-адрес представляет собой четырехбайтовое число, старшие (крайние левые) биты которого определяют класс IP-адреса.

Класс A: 0.xxx.xxx.xxx — 127.xxx.xxx.xxx (/8, 8 бит для сети, 24 для хостов).

Класс B: 128.0.xxx.xxx — 191.255.xxx.xxx (/16, 16 бит для сети, 16 для хостов).

Класс C: 192.0.0.xxx — 223.255.255.xxx (/24, 24 бита для сети, 8 для хостов).

Класс D: 224.0.0.0 — 239.255.255.255 (для мультикаста).

Класс E: 240.0.0.0 — 255.255.255.255 (зарезервировано).

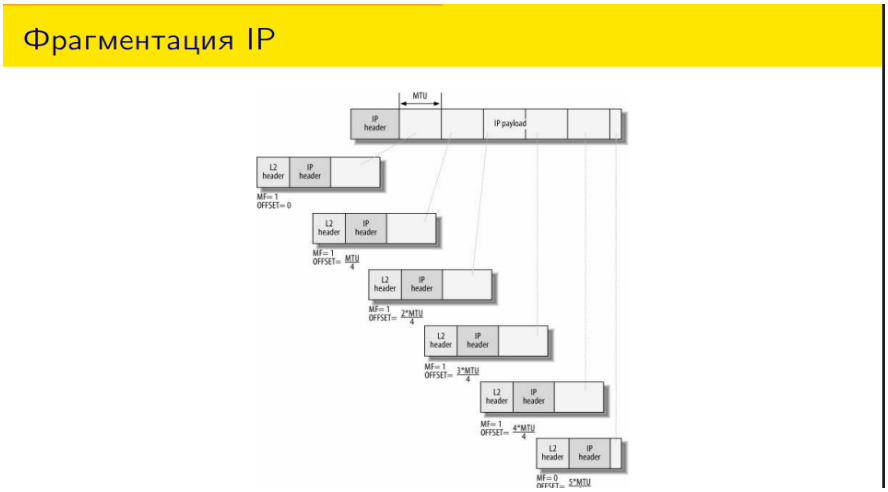
8. Протокол IPv4. Фрагментация IPv4-сегментов.

Протокол IPv4 в вопросе 7.

Фрагментация IPv4-сегментов.

Для того, чтобы существовала возможность передачи IP-сегментов через сети различного типа, межсетевой протокол обеспечивает адаптацию их размера к требованиям каждой сети. Это дает возможность, например, IP-сегментам, порожденным в сети на базе Ethernet (максимальный размер кадра – 1526 байт), беспрепятственно перемещаться до адресата по сети на базе X.25 (максимальный размер кадра - 128 байт). Изменение размера IP-сегмента в процессе перемещения по сети может быть связано и с соображениями эффективности передачи.

Каждый IP-фрагмент представляет собой полноценный IP-сегмент со своим собственным IP-заголовком. Однако заголовки всех IP-фрагментов содержат одинаковый идентификатор, совпадающий с идентификатором исходного IP-сегмента. Это позволяет распознавать все IP-фрагменты, относящиеся к одному исходному IP-сегменту.



IP-модуль на принимающем IP-фрагменты узле в ситуации, когда он должен транслировать IP-сегмент далее по сети, имеет три варианта действий с фрагментами:

- переслать IP-фрагменты далее неизменными;
- разбить (если в этом есть необходимость) полученные IP-фрагменты на более короткие IP-фрагменты;
- восстановить исходный IP-сегмент из фрагментов.

В работе с IP-фрагментами на принимающей стороне используется специальный таймер, который с приходом первого фрагмента IP-сегмента устанавливается в исходное состояние (для UNIX-реализаций это, обычно, 30 сек) и начинает обратный счет.

9. Характеристика, назначение и особенности работы протокола ICMP.

Протокол передачи команд и сообщений об ошибках (ICMP – internet control message protocol, RFC-792) ICMP позволяет маршрутизатору либо конечному узлу сообщить узлу-отправителю об ошибках, с которыми машрутизатор столкнулся при передаче какого-либо IP-пакета от данного конечного узла.

Управляющие сообщения ICMP не могут направляться промежуточному маршрутизатору, который участвовал в передаче пакета, с которым возникли проблемы, так как для такой посылки нет адресной информации - пакет несет в себе только адрес источника и адрес назначения, не фиксируя адреса промежуточных маршрутизаторов

ICMP-протокол сообщает об ошибках в IP-дейтограммах, но не дает информации об ошибках в самих ICMP-сообщениях

ICMP-протокол сообщает об ошибках в IP-дейтограммах, но не дает информации об ошибках в самих ICMP-сообщениях. ICMP использует IP, а IP-протокол должен использовать ICMP.

ICMP-протокол сообщает об ошибках в IP-дейтограммах, но не дает информации об ошибках в самих ICMP-сообщениях. ICMP использует IP, а IP-протокол должен использовать ICMP. В случае ICMP-фрагментации сообщение об ошибке будет выдано только один раз на дейтограмму, даже если ошибки были в нескольких фрагментах.

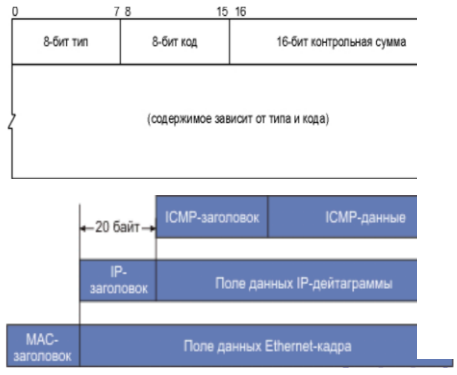
Задачи ICMP:

- передача отклика на пакет или эхо на отклик;
- контроль времени жизни дейтограмм в системе;
- реализует переадресацию пакета;
- выдает сообщения о недостижимости адресата или о некорректности параметров;
- формирует и пересылает временные метки;
- выдает запросы и отклики для адресных масок и другой информации.

ICMP-сообщения об ошибках не выдаются:

- на ICMP-сообщение об ошибке;
- при мультикастинг или широковещательной адресации;
- для фрагмента дейтаграммы (кроме первого);
- для дейтаграмм, чей адрес отправителя является нулевым, широковещательным или мультикастинговым.

Заголовок ICMP



Там еще много подозрительных типов!

Тип	Описание
4	Сдерживание источника (отключение источника при переполнении очереди)
9	Объявление маршрутизатора
10	Запрос маршрутизатора
13	Запрос метки времени
14	Ответ с меткой времени
15	Информационный запрос
16	Информационный ответ
17	Запрос адресной маски (RFC-950)
18	Отклик на запрос адресной маски (RFC-950)
30	Трассировка маршрута (RFC-1393)
31	Ошибка преобразования датаграммы (RFC-1475)
32	Перенаправление для мобильного узла
33	IPv6 Where-Are-You (где вы находитесь)
34	IPv6 I-Am-Here (я здесь)
35	Запрос перенаправления для мобильного узла
36	Отклик на запрос перенаправления для мобильного узла
37	Запрос доменного имени (Domain Name Request)
38	Ответ на запрос доменного имени (Domain Name Reply)
39	SKIP

Тип 12 – Проблема с параметрами дейтаграммы

Код	Описание
0	Ошибка в ip-заголовке
1	Отсутствует необходимая опция

Тип 0 – Эхо-ответ (ping-отклик)

Код	Описание
0	Эхо-ответ (ping-отклик)

Тип 8 – Эхо-запрос

Код	Описание
0	Эхо запрос (ping-запрос)

Тип 11 – Превышение временного интервала (для дейтаграммы время жизни истекло) (ttl=0)

Код	Описание
0	при передаче
1	при сборке (случай фрагментации)

Тип 5 - Переадресовать (изменить маршрут)

Код	Описание
0	Переадресовать дейтаграмму в сеть (устарело)
1	Переадресовать дейтаграмму на ЭВМ
2	Переадресовать дейтаграмму для типа сервиса (tos) и сети
3	Переадресовать дейтаграмму для типа сервиса и ЭВМ

Тип 3 – Адресат недоступен

Код	Описание
0	Сеть недостижима
1	ЭВМ не достижима
2	Протокол не доступен
3	Порт не доступен
4	Необходима фрагментация сообщения
5	Исходный маршрут вышел из строя
6	Сеть места назначения не известна
7	ЭВМ места назначения не известна
8	Исходная ЭВМ изолирована
9	Связь с сетью места назначения административно запрещена
10	Связь с ЭВМ места назначения административно запрещена
11	Сеть не доступна для данного вида сервиса
12	ЭВМ не доступна для данного вида сервиса
13	Связь административно запрещена с помощью фильтра.
14	Нарушение старшинства ЭВМ
15	Дискриминация по старшинству

10. Характеристика протокола UDP. Понятие «порт протокола».

Понятие «порт протокола».

Порт протокола транспортного уровня идентифицируемый номером системный ресурс, выделяемый приложению, выполняемому на некотором сетевом хосте, для связи с приложениями, выполняемыми на других сетевых хостах (в том числе с другими приложениями на этом же хосте).

Порты необходимы по двум причинам:

1. используются как идентификатор для разделения траспортных сессий между одинаковыми оконечными точками
2. используются для идентификации протокола прикладного уровня и ассоциированного с ним сервиса

Для каждого из протоколов TCP и UDP стандарт определяет возможность одновременного выделения на хосте до 65536 уникальных портов, идентифицирующихся номерами от 0 до 65535.

Для каждого из протоколов TCP и UDP стандарт определяет возможность одновременного выделения на хосте до 65536 уникальных портов, идентифицирующихся номерами от 0 до 65535. Порты TCP не пересекаются с портами UDP. То есть, порт 1234 протокола TCP не будет мешать обмену по UDP через порт 1234!!!

Системные (0-1023)

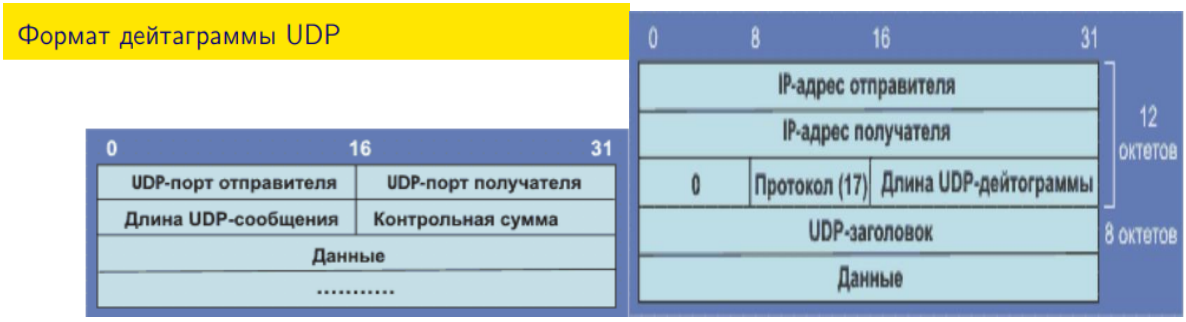
Пользовательские (1024-49151)

Динамические и/или частные (49152-65535)

Для большинства ОС системный диапазон доступен только с привилегиями администратора!

Характеристика протокола UDP

Протокол UDP (RFC-768) является одним из основных протоколов, расположенных непосредственно над IP. Он предоставляет прикладным процессам транспортные услуги, немногим отличающиеся от услуг протокола IP. Протокол UDP обеспечивает доставку дейтограмм, но не требует подтверждения их получения. Протокол UDP не требует соединения с удаленным модулем UDP ("бессвязный"протокол)



Длина включает в себя заголовок.

Контрольная сумма рассчитывается с использованием псевдозаголовка

Контрольная сумма с использованием псевдозаголовка необходима как защита от дейтаграмм, ошибочно направленных по другому адресу. Контрольная сумма UDP не является обязательной. Если она не подсчитывается, ее значение равно 0 (настоящая нулевая контрольная сумма кодируется всеми единицами).

11. Характеристика протокола TCP. Понятие «порт протокола».

Понятие «порт протокола» в 10 вопросе.

TCP – Transmission Control Protocol. Это надежный протокол с установлением соединений, позволяющий без ошибок доставлять байтовый поток с одной машины на любую другую машину объединенной сети. Он разбивает входной поток байтов на отдельные сообщения и передает их межсетевому уровню. В пункте назначения этот протокол собирает из полученных сообщений выходной поток. Кроме того, TCP осуществляет управление потоком.

Механизмы надежности TCP:

Контрольная сумма – позволяет определить повреждение данных и/или заголовка TCP-сегмента.

Определение дублирующихся сегментов – "дубли"выкидываются.

Повторная передача – используются подтверждения о доставке данных.

Отсутствие подтверждений вместе с механизмом таймеров приводит к повторной передаче данных.

Корректная последовательность – TCP собирает сегменты в правильном порядке.

Таймеры – позволяют управлять повторной передачей сегментов

TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Описание флагов в 12 вопросе.

12. Характеристика протокола TCP. Назначение флагов в TCP-сегменте.

Характеристика протокола TCP в вопросе 11.

Назначение флагов в TCP-сегменте.

1. Urgent Pointer (URG) — флаг срочных данных

- Назначение: указывает, что в сегменте содержатся срочные данные, которые нужно обработать вне очереди.

- Как работает: если флаг URG установлен (1), в заголовке TCP указывается значение Urgent Pointer (указатель на срочные данные). Это поле показывает, где заканчиваются срочные данные в потоке. Принимающая сторона должна немедленно обработать эти данные, не дожидаясь остального потока.

2. Acknowledgement (ACK) — флаг подтверждения

- Назначение: подтверждает успешное получение данных и указывает, какие данные ожидаются дальше.

- Как работает: если флаг ACK установлен (1), поле **Acknowledgement Number** в заголовке TCP указывает следующий ожидаемый байт данных. Это часть механизма надёжной доставки TCP: отправитель знает, что данные до этого номера байта успешно доставлены.

3. Push Function (PSH) — флаг немедленной передачи

- Назначение: - Указывает, что данные нужно немедленно передать приложению на принимающей стороне, не дожидаясь заполнения буфера.

- Как работает: если флаг PSH установлен (1), TCP передаёт данные приложению сразу после получения, минуя буферизацию. Это ускоряет обработку данных, когда задержка критична.

4. Reset the Connection (RST) — флаг сброса соединения

- Назначение: принудительно сбрасывает (закрывает) соединение в случае ошибок или некорректных действий.

- Как работает: если флаг RST установлен (1), соединение немедленно разрывается, и все ресурсы, связанные с ним, освобождаются.

5. Synchronize (SYN) — флаг синхронизации

- Назначение: Инициирует установление TCP-соединения между двумя устройствами.

- Как работает: Если флаг SYN установлен (1), сегмент используется для начала трёхэтапного рукопожатия (Three-Way Handshake):

1. Клиент отправляет SYN с начальным номером последовательности (Sequence Number).
2. Сервер отвечает сегментом SYN-ACK, подтверждая SYN и отправляя свой номер последовательности.
3. Клиент отправляет ACK, завершая установление соединения.

6. No More Data from Sender (FIN) — флаг завершения передачи

- Назначение: указывает, что отправитель завершил передачу данных и хочет закрыть соединение.

- Как работает: если флаг FIN установлен (1), это сигнал, что новых данных от отправителя не будет.

7. NS (Nonce Sum) — флаг защиты от подделки

- Назначение: Используется для защиты от подделки TCP-сегментов в рамках механизма ECN (Explicit Congestion Notification) и для проверки целостности данных.

8. CWR (Congestion Window Reduced) — флаг уменьшения окна перегрузки

- Назначение: Уведомляет получателя, что отправитель уменьшил своё окно перегрузки в ответ на сигнал о перегрузке сети.

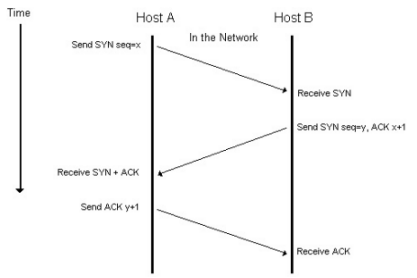
9. ECE (ECN-Echo) — флаг уведомления о перегрузке

- Назначение: Уведомляет отправителя о том, что сеть испытывает перегрузку, чтобы он мог принять меры (например, уменьшить скорость передачи).

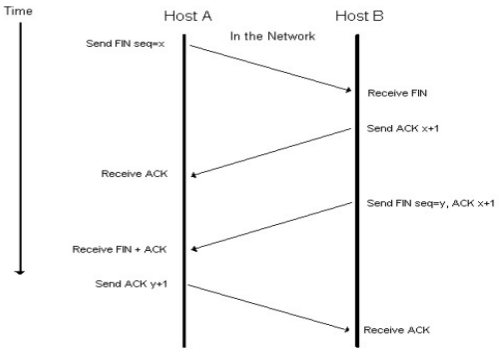
13. Характеристика протокола TCP. Установление и разрыв соединения TCP (на уровне протокола).

Характеристика протокола TCP в вопросе 11.

Установка соединения TCP



Разрыв соединения TCP



Установление соединения в TCP происходит через трёхэтапное рукопожатие:

1. Клиент отправляет SYN (Synchronize): Клиент (например, ваш браузер) инициирует соединение, отправляя сегмент с установленным флагом SYN.

В сегменте указывается начальный Sequence Number (номер последовательности, ISN), например, Seq = x. Этот номер используется для отслеживания порядка байтов.

2. Сервер отвечает SYN-ACK (Synchronize-Acknowledge): Сервер получает SYN, подтверждает его, отправляя сегмент с флагами SYN и ACK.

Флаг ACK подтверждает получение SYN клиента: поле Acknowledgement Number устанавливается в Ack = x + 1, что означает, что сервер ожидает следующий байт после x.

Сервер также отправляет свой начальный номер последовательности, например, Seq = y.

3. Клиент подтверждает с ACK (Acknowledge): Клиент получает SYN-ACK и отправляет сегмент с флагом ACK, подтверждая получение серверного SYN.

Поле Ack = y + 1 указывает, что клиент готов принимать данные, начиная с байта y + 1.

Клиент устанавливает свой номер последовательности Seq = x + 1.

Разрыв соединения в TCP происходит следующим образом.

1. Первая сторона отправляет FIN (Finish):

Одна из сторон (например, клиент) решает закрыть соединение и отправляет сегмент с флагом FIN.

Клиент указывает свой текущий номер последовательности, например, Seq = x.

2. Вторая сторона подтверждает с ACK (Acknowledge):

Сервер получает FIN и отправляет сегмент с флагом ACK, подтверждая, что он принял FIN.

Поле Ack = x + 1 указывает, что сервер получил все данные до x.

Сервер может продолжать отправлять данные, если у него есть что передать.

3. Вторая сторона отправляет свой FIN:

Когда сервер завершает отправку данных, он отправляет свой сегмент с флагом FIN, например, Seq = y.

4. Первая сторона подтверждает с ACK:

Клиент получает FIN от сервера и отправляет сегмент с флагом ACK, подтверждая его: Ack = y + 1.

14. Характеристика протокола TCP. Управление потоком. Скользящее окно. Механизм медленного старта.

Характеристика протокола TCP в вопросе 11.

Управление потоком с помощью скользящего окна в TCP

Основные понятия

- **Скользящее окно (Window):** Это диапазон байтов, которые отправитель может передать без получения подтверждения (ACK) от получателя. Размер окна определяется получателем и указывается в поле **Window Size** в заголовке TCP.

Этапы управления потоком

(a) Начальное состояние

- **Описание:** На этом этапе отправитель может передать байты 1–4, так как окно скольжения охватывает эти байты (выделены сплошной рамкой).

- **Описание:** Получатель успешно принял байты 1–3 и отправил подтверждение (ACK) с номером `Ack = 4`, что означает, что он ожидает байт 4 и далее.

(в) Дальнейшее продвижение окна

- **Описание:** Получатель подтвердил байт 4 (отправил `Ack = 5`), и окно снова сдвигается.

Принципы работы скользящего окна

1. **Размер окна:** Получатель сообщает отправителю, сколько данных он готов принять (Window Size), исходя из своих ресурсов (например, размера буфера). Если буфер получателя заполнен, он может уменьшить размер окна, вплоть до 0 (остановка передачи).

2. **Подтверждение:** Получатель отправляет ACK, указывая, какие данные успешно приняты. Окно сдвигается вправо, включая новые байты.

3. **Контроль перегрузки:** Если сеть перегружена, размер окна может быть уменьшен (например, с помощью механизмов ECN, как CWR/ECE), чтобы избежать потерь пакетов.

4. **Повторная передача:** Если данные теряются (например, байт 4 не был получен), получатель продолжает отправлять ACK с последним успешно принятым байтом (например, `Ack = 4`), и отправитель повторно передаёт потерянные данные.

Механизм медленного старта.

Congestion Control: Slow Start – с каждой успешной передачей размер буфера удваивается до половины точки насыщения– далее по 1 сегменту. Вплоть до размера окна. Congestion Avoidance – противовес медленному старту. Уменьшает размер окна в 2 раза, но не менее 2 сегментов. В случае таймаута – уменьшается до 1 сегмента и запускается механизм медленного старта.

15. Характеристика протокола TCP. Алгоритм Нэгла (Nagle) и синдром узкого окна.

Характеристика протокола TCP в вопросе 11.

Синдром узкого окна

Синдром узкого окна Такого рода проблема возникает в том случае, когда данные поступают отправителю крупными блоками, а интерактивное приложение адресата считывает информацию побайтно.

Такого рода проблема возникает в том случае, когда данные поступают отправителю крупными блоками, а интерактивное приложение адресата считывает информацию побайтно. Кларк предложил не посылать уведомление о ненулевом значении ширины окна при считывании одного байта, а лишь после освобождения достаточно большого пространства в буфере. Например, когда адресат готов принять MSS байтов или когда буфер наполовину пуст.

Алгоритм Нэгла.

Алгоритм Нэгла, предложенный Джоном Нэглом в 1984 году, направлен на повышение эффективности передачи данных в TCP за счёт уменьшения количества маленьких сегментов, отправляемых по сети. Он решает проблему, связанную с тем, что отправка небольших порций данных, например одного байта, сопровождается значительными накладными расходами из-за заголовков TCP и IP, которые могут быть больше самих данных. Это особенно актуально для сетей с высокой задержкой, где такие передачи создают лишний трафик и снижают пропускную способность.

Суть работы алгоритма заключается в том, что TCP задерживает отправку данных, если их объём меньше максимального размера сегмента, известного как MSS. Вместо немедленной отправки данные накапливаются в буфере. Отправка происходит, только когда буфер заполняется до MSS, либо когда приходит подтверждение от получателя о доставке предыдущего сегмента, либо истекает таймер ожидания, обычно около 200 миллисекунд. При этом алгоритм следит за тем, чтобы новые данные отправлялись, только если все предыдущие сегменты подтверждены, или если накопленный объём данных позволяет сформировать полный сегмент.

Представим ситуацию: пользователь вводит текст в приложении, таком как Telnet, где каждый символ — это один байт. Без алгоритма Нэгла каждый символ отправлялся бы отдельным сегментом с большим заголовком, что создавало бы избыточный трафик. С алгоритмом Нэгла TCP собирает эти символы в буфер, пока не наберётся достаточно данных или не придёт подтверждение, после чего отправляет их одним сегментом, экономя ресурсы сети.

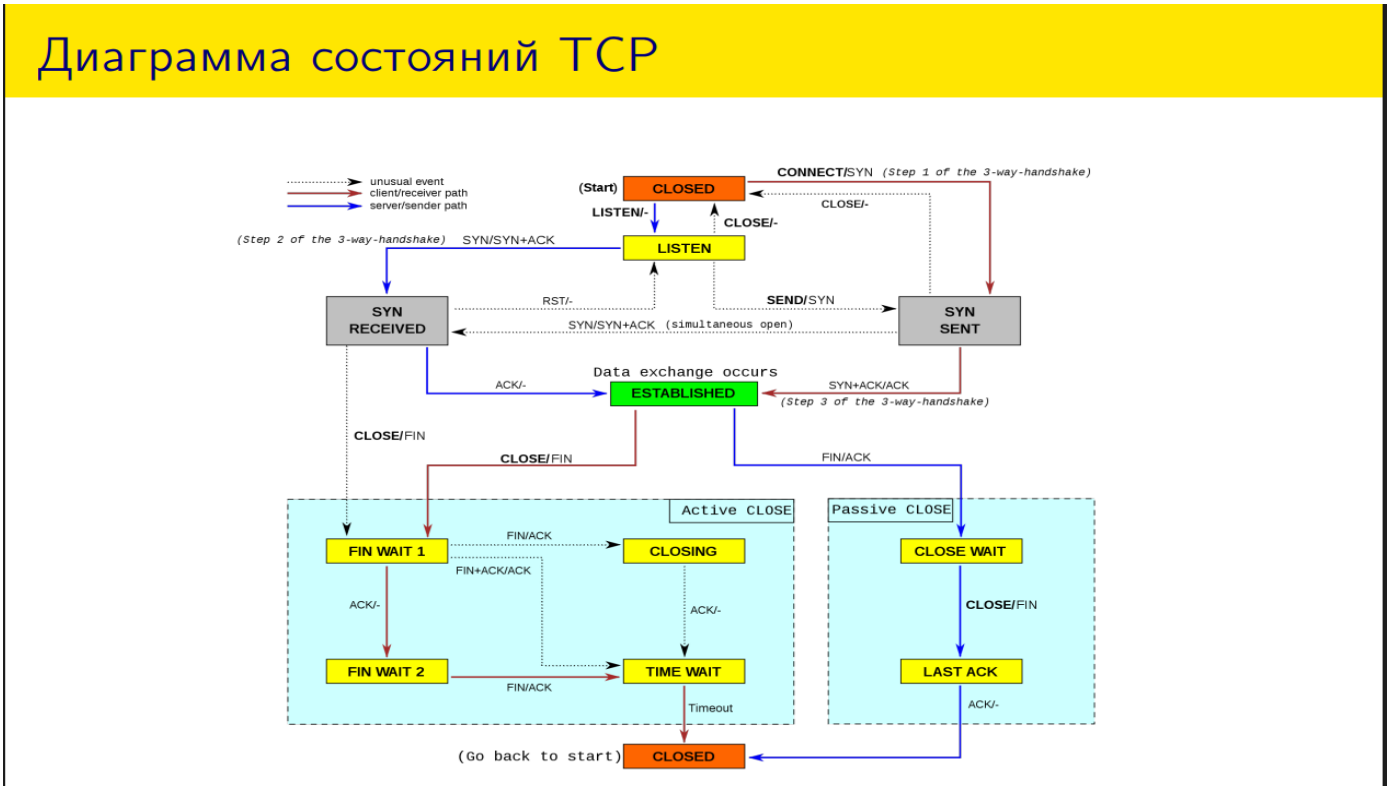
Алгоритм эффективно снижает нагрузку на сеть, особенно в условиях высокой задержки, где маленькие сегменты сильно замедляют передачу. Он позволяет отправлять данные большими порциями, что улучшает пропускную способность. Однако у алгоритма есть и обратная сторона: он может вызывать задержки, так как данные ждут в буфере. Это особенно заметно в интерактивных приложениях, таких как онлайн-игры или чаты, где важна мгновенная передача. Проблема усугубляется, если получатель использует механизм задержки подтверждения, когда он ждёт больше данных, чтобы отправить подтверждение, а отправитель, в свою очередь, ждёт это подтверждение, чтобы отправить новые данные. Такой "застой" увеличивает задержку.

Для приложений, где низкая задержка критична, алгоритм Нэгла часто отключают с помощью опции TCP_NODELAY. Это заставляет TCP отправлять данные сразу, даже если они маленькие, но ценой увеличения трафика из-за большего количества сегментов. Такой подход оправдан, например, в голосовых звонках или играх, где важна скорость, а не экономия пропускной способности.

Алгоритм Нэгла представляет собой компромисс между эффективностью передачи и скоростью отклика. Он полезен для задач, связанных с передачей больших объёмов данных, таких как загрузка файлов, но может быть неэффективен для приложений, требующих немедленной отправки. Несмотря на развитие сетей, он остаётся важной частью TCP, хотя его влияние в современных условиях стало менее значительным благодаря другим оптимизациям.

16. Характеристика протокола TCP. Диаграмма состояний TCP.

Характеристика протокола TCP в вопросе 11.



Начинается всё с состояния CLOSED, где соединение ещё не установлено. Сервер переходит в состояние LISTEN после выполнения команды LISTEN, ожидая входящих запросов на соединение. Клиент, инициирующий соединение, отправляет сегмент с флагом SYN (первая часть трёхэтапного рукопожатия), переходя в состояние SYN SENT. Сервер, получив SYN, отвечает сегментом SYN+ACK (вторая часть рукопожатия) и переходит в состояние SYN RECEIVED. Клиент, получив SYN+ACK, отправляет ACK (третья часть рукопожатия) и переходит в состояние ESTABLISHED, где начинается обмен данными. Сервер, получив ACK, также переходит в ESTABLISHED.

В состоянии ESTABLISHED обе стороны обмениваются данными. Соединение может быть закрыто двумя способами: активным (Active CLOSE) или пассивным (Passive CLOSE) закрытием. При активном закрытии одна сторона (например, клиент) отправляет сегмент с флагом FIN, переходя в состояние FIN WAIT 1. Вторая сторона (сервер) отвечает сегментом ACK и переходит в состояние CLOSE WAIT, а клиент, получив ACK, переходит в FIN WAIT 2. Сервер, завершив передачу данных, отправляет свой FIN, переходя в состояние LAST ACK, а клиент, получив FIN, отправляет ACK и переходит в состояние TIME WAIT. В TIME WAIT клиент ожидает некоторое время (обычно 2 MSL — около 30–60 секунд), чтобы убедиться, что все задержавшиеся сегменты обработаны, после чего переходит обратно в CLOSED. Сервер, получив последний ACK, сразу переходит в CLOSED.

Если закрытие инициируется другой стороной (пассивное закрытие), то процесс начинается с получения FIN: сторона переходит в CLOSE WAIT, отправляет свой FIN и переходит в LAST ACK, ожидая финального ACK для завершения. Также из состояния ESTABLISHED возможно одновременное закрытие (Simultaneous CLOSE), когда обе стороны отправляют FIN и переходят в состояние CLOSING, обмениваясь ACK, после чего переходят в TIME WAIT и затем в CLOSED.

Диаграмма также учитывает нештатные ситуации: сегмент с флагом RST (Reset) может быть отправлен из любого состояния, переводя соединение сразу в CLOSED, например, при ошибках или сбоях. Переходы между состояниями сопровождаются событиями, такими как команды CLOSE (закрытие соединения) или тайм-ауты (например, в TIME WAIT).

17. Понятие интерфейса сокетов. Функциональные средства интерфейса.

Интерфейс сокетов – это программный интерфейс, предоставляющий приложениям механизм для сетевого взаимодействия через стандартизированные функции. Сокеты абстрагируют низкоуровневые детали сетевых протоколов (TCP, UDP и др.), позволяя программам обмениваться данными по сети.
Функциональные средства:

- Распределение локальных ресурсов – выделение памяти и системных ресурсов под сокет.
- Задание локальной и удаленной оконечных точек связи – указание IP-адресов, портов и типа протокола.
- Инициирование соединения (клиент) – функция connect() для установки соединения с сервером.
- Передача дейтаграммы (клиент) – отправка данных без установки соединения (UDP).
- Ожидание входящего запроса на установление соединения (сервер) – функции bind(), listen(), accept().
- Передача или прием данных – send(), recv() (TCP) или sendto(), recvfrom() (UDP).
- Определение момента поступления данных – использование механизмов select(), poll(), epoll() или kqueue() для асинхронного контроля.
- Выработка срочных данных – обработка внеполосных (OOB) данных (TCP).
- Обработка входящих срочных данных – чтение OOB-данных через флаг MSG_OOB.
- Корректное завершение соединения – функции shutdown() и close().
- Обрабоотка запроса на завершение соединения от удаленного участника соединения – детектирование разрыва связи (recv() возвращает 0).
- Аварийное прекращение связи – принудительный разрыв соединения (close() без shutdown()).
- Устранение последствий аварийных ситуаций или аварийного прекращения связи – освобождение ресурсов после нештатного завершения.
- Освобождение локальных ресурсов после завершения связи – закрытие сокета (close()).

18. Понятие сокета. Типы и структуры сокетов. Семейства адресов.

Сокет – это программный интерфейс для сетевого взаимодействия, представляющий собой конечную точку соединения. В ОС сокет идентифицируется целым числом (дескриптором), который ОС хранит в таблице дескрипторов (наряду с файловыми дескрипторами).
Адрес сокета – это структура данных, включающая:

- Идентификатор адресного семейства (например, AF_INET).
- Адресную информацию (IP-адрес, порт и др.), зависящую от семейства адресов.

Общая структура адреса:

```
struct sockaddr{
u_char      sa_len;
u_short     sa_family;
char        sa_data[14];
}
```

Для AF_INET:

```
struct sockaddr_in{
u_char              sin_len;
u_short             sin_family;
u_short             sin_port;
struct in_addr      sin_addr;
char                sin_zero[8]; // Выравнивание (обычно нули)
}
```

Сокет может использоваться для различных видов связи, поэтому приложение должно явно указывать:

- Тип сокета (потоковый, датаграммный и т. д.).
- Номер порта (для идентификации сервиса).
- Локальный и удаленный адреса (IP-адреса компьютеров).

Типы сокетов:

- SOCK_STREAM – поддерживают надежные, упорядоченные, полнодуплексные потоки октетов в режиме с установлением соединения.
- SOCK_SEQPACKET – аналог с дополнительным сохранением границ между записями (как в датаграммах).
- SOCK_DGRAM – передача данных в виде датаграмм в режиме без установления соединения.
- SOCK_RAW – аналог с дополнительной возможностью доступа к протокольным заголовкам и другой информации нижнего уровня. "Неструктурированные" или "сырые" сокеты.

Адресное семейство соответствует конкретной среде взаимодействия:

- AF_UNIX (AF_LOCAL) – поддерживает межпроцессное взаимодействие в пределах одной системы.
- AF_INET – поддерживает взаимодействие по протоколам IPv4.
- AF_INET6 – взаимодействие по протоколам IPv6.

19. Сокеты Berkeley. Разрешение имен хостов, протоколов, сетевых сервисов.

Сетевые приложения работают с тремя основными типами идентификаторов:
- Имена хостов (example.com)
- Номера протоколов (TCP = 6)
- Сетевые сервисы (HTTP = 80)

Преобразование между этими форматами осуществляется через:
- Системные базы данных (/etc/hosts, /etc/protocols, /etc/services)
- Сетевые службы (DNS для имен хостов)
- Специальные API-функции

Данные передаются по сети в виде последовательности октетов. Сетевой порядок октетов – при котором первый (с наименьшим адресом) октет содержит старшие (наиболее значимые).
Последовательности октетов неудобны в обработке на хостах, где предпочтительнее аппаратно поддерживаемые типы, в особенности целочисленные. Значения этих типов хранятся с другим порядком байт – хостовым.
Для преобразования значений типов uint16_t и uint32_t из хвостового порядка байт в сетевой служат функции htons и htonl, обратную операцию осуществляют функции ntohs и ntohl.

Получение имен хостов:

```
struct hostent {
char    *h_name;        //имя хоста
char    **h_aliases;    //массив псевдонимов хоста
int     h_addrtype;     //тип адреса (AF_INET или AF_INET6)
int     h_length;       //длина адреса в байтах
char    **h_addr_list; }    //массив указателей на сетевые адреса хоста
```

Функции для работы:

- sethostent – устанавливает соединение с базой, остающееся открытым после вызова gethostent, если значение аргумента stayopen отлично от 0.
- gethostent – последовательно читает элементы базы, возвращая результат в структуре типа hostent.
- endhostent – закрывает соединение с базой.
- gethostbyname – возвращает результат в структуре типа hostent для заданного имени хоста.
- gethostbyaddr – возвращает результат в структуре типа hostent для заданного в структуре addr.

Для произвольного доступа используются функции getnameinfo и getaddrinfo.
Локальная база данных сетевых протоколов располагается по адресу /etc/protocols.
Структура protoent содержит по крайней мере:
- char *p_name – официальное имя протокола.
- char **p_aliases – массив указателей на альтернативные имена протокола, завершаемый пустым указателем.
- int p_proto – номер протокола.

Доступ к базе данных сетевых протоколов:
- void setprotoent (int stayopen);
- struct protoent *getprotoent (void);
- struct protoent *getprotobyname (const char *name);
- struct protoent *getprotobynumber (int proto);
- void endprotoent (void);

Локальная база данных сетевых сервисов располагается по адресу /etc/services.
Структура servent содержит по крайней мере:
- char *s_name – официальное имя сервиса.
- char **s_aliases – массив указателей на альтернативные имена сервиса, завершаемый пустым указателем.
- int s_port – номер порта, соответствующий сервису (в сетевом порядке байт).
- char *s_proto – имя протокола для взаимодействия с сервисом.

Доступ к базе данных сетевых сервисов:
- void setservent (int stayopen);
- struct servent *getservent (void);
- struct servent *getservbyname (const char *name, const char *proto);
- struct servent *getservbyport (int port, const char *proto);
- void endservent (void);

20. Сокеты Berkeley. Основные системные вызовы.

• **socket** – создать оконечную точку коммуникации.
int socket(int domain, int type, int protocol);
Возвращает: файловый дескриптор или -1 при ошибке
domain задает набор протоколов для коммуникации (PF_UNIX, PF_INET, PF_INET6 и т.д.)
type задает семантику коммуникации (SOCK_STREAM, SOCK_SEQPACKET, SOCK_DGRAM, SOCK_RAW)
protocol зависит от домена и указывается, если существует несколько протоколов для конкретного типа сокета.
• **sockerpair** – создать неименованную пару подключенных сокетов (аналог неименованных каналов чекрез сокеты)
int socketpair(int domain, int type, int protocol, int sv[2]);
Возвращает дескрипторы сокетов.
• **bind** – привязать имя к сокету
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
Привязывает к сокету sockfd локальный адрес my_addr длиной addrlen.
• **listen** – слушать соединения на сокете
int listen(int s, int backlog);
Успех = 0, ошибка = -1.
Применим только к SOCK_STREAM или SOCK_SEQPACKET. Backlog – максимальная длина очереди жидающих соединений.
• **accept** – прием соединений
int accept (int sd, struct sockaddr *restrict address, socklen_t *restrict address_len);
Успех – дескриптор, ошибка -1.
• **connect** – инициирует соединение на сокете
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
Успешное соединение – 0, ошибка -1.
Если sockfd ссылается на сокет типа SOCK_DGRAM, то serv_addr – адрес по умолчанию, куда посылаются датаграммы, и единственный адрес, откуда они принимаются. Если сокет имеет тип SOCK_STREAM или SOCK_SEQPACKET, то connect попытается установить соединение с другим сокетом (serv_addr – адрес длиной addrlen).
Сокеты с протоколами, основанными на соединении, обычно могут устанавливать соединение только 1 раз; с протоколами без соединения могут использовать connect многократно, чтобы изменить адрес назначения.
• **send**, **sendto**, **sendmsg** – отправить сообщение в сокет
int send(int s, const void *msg, size_t len, int flags);
int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
send можно использовать, только если сокет в соединенном состоянии. sendto и sendmsg – в любое время.
Возвращают количество отправленных символов или -1 в случае ошибки.
• **recv**, **recvfrom**, **recvmsg** – получить сообщение из сокета
int recv(int s, void *buf, size_t len, int flags);
int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
Флаги:
○ MSG_OOB – запрашиваются только экстренные данные (зависит от протокола).
○ MSG_DONTROUTE (send) – не использовать маршрутизацию при отправке пакета.
○ MSG_DONTWAIT (send) – включает неблокирующий режим.
○ MSG_TRUNC (recv) – возвращает реальную длину пакета (пакетные протоколы).
○ MSG_PEEK (recv) – не удалять прочитанные данные.
○ MSG_WAITALL (recv) – для сокетов типа SOCK_STREAM значит, что вызывающий процесс блокируется до получения всего запрошенного объема данных.
• **close** – закрыть файловый дескриптор.
int close(int fd);
0 – успех, -1 – ошибка.
• **shutdown** – закрыть файловый дескриптор.
int shutdown (int sd, int how);
0 – успех, -1 – ошибка.
how – что именно завершается: SHUT_RD прекращает прием, SHUT_WR – отправку, SHUT_RDWR – и то, и другие.
• **select** и **pselect** ждут изменения статуса нескольких файловых дескрипторов.
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
n на единицу больше самого большого номера дескриптора из всех наборов.
timeout – верхняя граница времени, которое пройдет перед возвратом из select.

21. Сокеты Berkeley. Опции сокетов. Уровни опций.

Опции сокетов позволяют настраивать их поведение на разных уровнях сетевого стека. Они управляют:

- Буферизацией данных (размеры буферов, таймауты).
- Режимами работы (широковещание, отладка, повторное использование адресов).
- Параметрами протоколов (TTL, MTU, алгоритм Нэгла).

Опции влияют на функционирование сокетов. Их значения можно опросить или изменить с помощью функций getsockopt и setsockopt.
int getsockopt (int sd, int level, int option_name, void *restrict option_value, socklen_t *restrict option_len);
int setsockopt (int sd, int level, int option_name, const void *option_value, socklen_t option_len);
level – указывает к какому уровню относится опция.

- SOL_SOCKET – опции сокета;
- SOL_TCP – опции для протокола TCP;
- SOL_IP – опции для протокола IP.

SOL_SOCKET

Опция	Описание
SO_ERROR	Статус ошибок
SO_TYPE	Тип сокета
SO_PROTOCOL	Протокол
SO_DEBUG	Отладочная информация о работе сокета
SO_ACCEPTCONN	Является ли сокет слушающим
SO_SNDBUF	Размер буфера отправки
SO_RCVBUF	Размер буфера приема
SO_RCVLOWATM	Мин. число байт, обрабатываемых при выводе
SO_SNDLOWAT	Мин. число байт, обрабатываемых при вводе
SO_RCVTIMEO	Время ожидания поступления данных при вводе
SO_SNDTIMEO	Время ожидания отправки данных при выводе
SO_TIMESTAMP	Включить передачу отметок времени
SO_BROADCAST	Перевод в широковещательный режим
SO_OOBINLINE	Данные out-of-band помещаются (= 1)
SO_REUSEADDR	Использование занятого адреса (для bind)
SO_LINGER	Блокировать ли процесс при закрытии дескриптора до передачи буферезированных данных (и на какой срок)

SOL_TCP

Опция	Описание
TCP_NODELAY	Отключает алгоритм Нэгла
TCP_MAXSEG	Устанавливает макс. размер сегметна (MSS)
TCP_CORK	Буферизация до полного заполнения буфера

SOL_IP

Опция	Описание
IP_HDRINCL	Ручное формирование IP-заголовка для RAW-сокетов
IP_OPTIONS	Установка IP-опций или возвращение установленных
IP_TTL	Задает Time To Live
IP_TOS	Установка поля Type of Service (приоритет трафика)
IP_PMTU_DISCOVER	Включает/отключает Path MTU Discovery (поиск максимального MTU на пути)
IP_MTU	Получение текущего MTU (Maximum Transmission Unit) для сокета

22. Модель сетевого взаимодействия клиент-сервер.

Термин "клиент-сервер"означает такую архитектуру программного комплекса, в которой его функциональные части взаимодействуют по схеме "запрос-ответ".

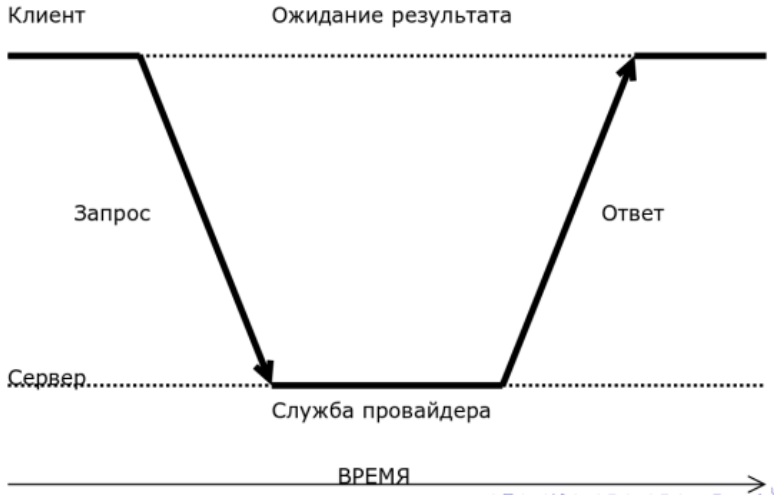
С сетевой точки зрения

Если рассмотреть две взаимодействующие части этого комплекса, то одна из них (клиент) выполняет активную функцию, т. е. инициирует запросы, а другая (сервер) пассивно на них отвечает. По мере развития системы роли могут меняться, например, некоторый программный блок будет одновременно выполнять функции сервера по отношению к одному блоку и клиента по отношению к другому.

С функциональной точки зрения

Процессы, реализующие некоторую службу, например ФС или БД называются ***серверами***.

Процессы, запрашивающие службы у серверов службы путем пересылки запроса и последующего ожидания ответа от сервера, называются ***клиентами***.



Необходимость применения принципа организации взаимодействия типа клиент-сервер связана с решением проблемы согласования условий соединения. Модель взаимодействия типа клиент-сервер предусматривает согласование условий соединения наиболее простым способом: она требует, чтобы для взаимодействия любой пары приложений, один из участников приступал к работе заранее и ждал (возможно в течение неопределенного времени) до тех пор, пока к нему не обратится второй участник соединения.

Архитектурные особенности модели

Модель клиент-сервер характеризуется следующими особенностями:

• *Централизация*: Сервер управляет ресурсами и данными, а клиенты обращаются к нему для их получения.

• *Масштабируемость*: Один сервер может обслуживать множество клиентов одновременно. Однако с увеличением числа клиентов может потребоваться балансировка нагрузки.

• *Упрощённая безопасность*: Централизованный контроль позволяет легче реализовать аутентификацию, разграничение прав и ведение журналов событий.

Преимущества модели клиент-сервер:

- Централизованное администрирование и обновление
- Эффективная реализация политики безопасности
- Четкое разделение обязанностей между клиентом и сервером

Недостатки:

- Зависимость от сервера (точка отказа)
- Необходимость в высокопроизводительном сервере при большой нагрузке
- Сложности с масштабированием при резком росте числа клиентов

23. Трехуровневая клиент-серверная модель, варианты распределения на физически двухзвенной архитектуре. Вертикальное, горизонтальное и peer-to-peer распределение.

Модель клиент-сервер всегда была предметом множества дебатов и споров. Один из главных вопросов состоял в том, как точно разделить клиента и сервер. Четкого разделения здесь не может быть.

Трехуровневая модель

- уровень пользовательского интерфейса
- уровень обработки
- уровень данных

1. Уровень пользовательского интерфейса обычно реализуется на клиентах. Этот уровень содержит программы, посредством которых пользователь может взаимодействовать с приложением. Простейший вариант пользовательского интерфейса не содержит ничего, кроме символьного (или графического) дисплея.

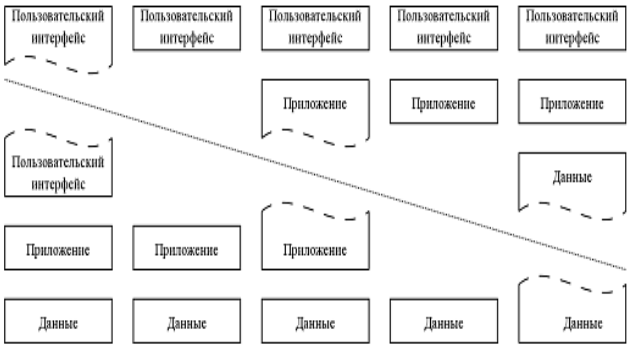
2. На этом уровне трудно выделить какие-то закономерности, обычно здесь реализуется логика работы программы.

3. На этом уровне находятся программы, которые поставляют данные обрабатывающим их приложениям. В простейшем варианте уровень данных реализуется файловой системой, однако часто может использоваться и полнофункциональная база данных. В модели клиент-сервер этот уровень обычно находится на стороне сервера. Специфическим требованием этого уровня является требование сохранности – это означает, что когда приложение не работает, данные должны сохраняться в определенном месте в расчете на дальнейшее использование.

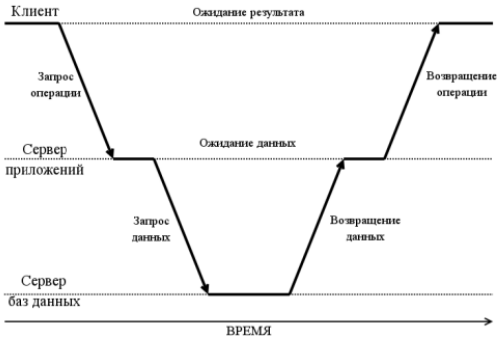
Логическая архитектура (трёхуровневая) может быть реализована на **различном количестве физических узлов**:

- ***Двухзвенная архитектура***: пользовательский интерфейс и логика обработки находятся на клиенте, а сервер отвечает за данные.
- ***Трёхзвенная архитектура***: каждый из уровней размещается на отдельной машине — клиент (UI), сервер приложений (логика), сервер БД (данные).

Физически двухзвенная архитектура



Физически трехзвенная архитектура



Вертикальное распределение

Во множестве приложений обработка данных организована как многозвенная архитектура приложений клиент-сервер. Особенностью такого типа является то, что он о достигается размещением логически разных компонентов на различных физических машинах.

Горизонтальное распределение

При таком типе распределения клиент или сервер может содержать физически разделенные части логически однородного модуля, причем работа с каждой из частей может протекать независимо. Это делается для выравнивания нагрузки. горизонтальное распределение используется для масштабирования нагрузки: например, если сервер базы данных перегружен, его можно заменить кластером БД с репликами. Также бизнес-логику можно распределить на несколько серверов (например, микросервисы).

peer-to-peer

Для некоторых несложных приложений выделенного сервера может не быть вообще. Такую организацию обычно называют одноранговым распределением. Применяется в ситуациях, где:

низкие требования к безопасности, нет нужды в централизованном хранении данных, высока важность децентрализации (например, торрент-сети, мессенджеры).

24. Архитектура клиента. Определение местонахождения сервера.

Процессы, запрашивающие службы у серверов службы путем пересылки запроса и последующего ожидания ответа от сервера, называются ***клиентами***.

Приложения действующие в качестве клиентов концептуально проще приложений выполняющих функции серверов. основной части клиентского программного обеспечения не нужно взаимодействовать с несколькими серверами основная часть клиентского программного обеспечения применяется в виде обычных прикладных программ – не требуются привилегии большая часть клиентского обеспечения не заботится о правилах защиты, поскольку в этом вопросе оно полагается на ОС.

Клиенты могут быть разных типов:

- ***Тонкие клиенты*** — содержат минимальную логику, полагаясь на сервер (например, веб-браузеры).
- ***Толстые клиенты*** — значительная часть бизнес-логики выполняется на клиентской машине (например, бухгалтерские программы).
- ***Мобильные клиенты*** — оптимизированы под мобильные устройства, сочетают локальную обработку с сетевым доступом к серверу.

Взаимодействие между клиентом и сервером чаще всего реализуется через стандартные сетевые протоколы, такие как TCP/IP. Обмен данными может происходить синхронно (клиент ждёт ответа от сервера) или асинхронно (в фоновом режиме).

Определение местонахождения сервера – первейшая и одна из самых выжных задач клиентского программного обеспечения!!!

- доменное имя или IP-адрес сервера могут быть заданы в виде константы во время трансляции клиентской программы
- клиентская программа может требовать у пользователя указывать имя сервера при ее вызове
- информация о местонахождении сервера предоставляется из постоянного хранилища данных
- для поиска сервера используется отдельный протокол

Еще один аспект поиска сервера следует из того, какой тип сервиса предоставляется сервером. От этого зависит нужно ли использовать какой-либо определенный сервер или же можно использовать первый ответивший. В качестве альтернативы в ответе сервера может также содержаться список других серверов.

25. Архитектура клиента. Алгоритмы работы клиентов. Функция Connect для UDP.

Приложения действующие в качестве клиентов концептуально проще приложений выполняющих функции серверов. основной части клиентского программного обеспечения не нужно взаимодействовать с несколькими серверами основная часть клиентского программного обеспечения применяется в виде обычных прикладных программ – не требуются привилегии большая часть клиентского обеспечения не заботится о правилах защиты, поскольку в этом вопросе оно полагается на ОС

Задача построения клиента с использованием протокола TCP является самой простой из всех задач сетевого программирования.

Алгоритм клиента TCP:

1 Найти IP-адрес и номер порта протокола сервера, с которым необходимо установить связь.

2 Распределить сокет

3 Указать, что для соединения нужен произвольный, неиспользуемый порт протокола на локальном компьютере и позволить ПО TCP выбрать такой порт

4 Подключить сокет к серверу

5 Выполнить обмен данными с сервером по протоколу прикладного уровня

6 Закрыть соединение

Алгоритм клиента UDP**:**

1. Найти IP-адрес и порт сервера.
2. Создать сокет (socket()).
3. *Не требуется устанавливать соединение*, но можно использовать connect() для указания адреса сервера (об этом ниже).
4. Отправить данные с помощью sendto() или send() (если был вызван connect()).
5. Получить ответ через recvfrom() или recv().
6. Закрыть сокет (close()).

Хотя функция connect() традиционно ассоциируется с TCP, она также может использоваться с сокетами UDP. В контексте UDP она не устанавливает соединение, как это происходит в TCP, а выполняет следующие задачи:

- привязывает сокет к конкретному адресу и порту сервера;
- позволяет использовать send() и recv() вместо sendto() и recvfrom(), упрощая код;
- ограничивает приём данных — сокет будет получать только пакеты от указанного адреса;
- снижает накладные расходы на повторное указание адреса сервера в каждой отправке.

connect() в UDP **не устанавливает TCP-соединение**, но «привязывает» сокет к конкретному адресу;

26. Классификация серверов. Алгоритм последовательного сервера с установлением логического соединения.

Процессы, реализующие некоторую службу, например ФС или БД называются ***серверами***.

По внутренней архитектуре

- Последовательный
- Параллельный

По типу используемого сервиса

- С установлением соединения
- Без установления соединения

По состоянию

- С сохранением состояния (stateful)
- Без сохранения состояния (stateless)

Многопоточное решение применяется, если затраты на создание и переключение между потоками невелики и при этом требуется совместное использование или обмен данными между соединениями.

Многопроцессная модель применяется для достижения максимального распараллеливания. Если используются процессы, то появляется возможность использовать для обработки внешние программы.

При использовании *асинхронного ввода/вывода* обработка запросов ведется только в одном потоке, поэтому сервер имеет практически такую же производительность, что и последовательный, даже на компьютере с несколькими процессорами. Удобно применять, если сервер должен иметь доступ к данным разных соединений или на обработку каждого запроса не требуется много времен

Серверы с установлением логического соединения

Преимущества:

- простота программирования

Недостатки:

- для каждого логического соединения требуется создавать отдельный сокет
- требуется трехэтапное квитирование при установке и разрыве соединения, что невыгодно для использования передачи небольших объемов данных в небольшой сети
- простаивающее соединение, по которому не проходят пакеты, напрасно используют ресурсы

Серверы без установления логического соединения

Преимущества:

- нет накладных расходов на установление и разрыв соединения
- можно реализовывать широковещательные или групповые рассылки
- не требуются ресурсы на поддержание соединения

Недостатки:

- самостоятельная реализация механизмов управления передачей: квитирование, тайм-ауты, оптимизация трафика, контроль надежности

stateful & stateless серверы

Информация о состоянии – это обновляемая сервером информация о ходе взаимодействия с клиентом. Информация о состоянии применяется для эффективной оптимизации сервера. Если на сервере сохраняются какие-либо данные о запросах клиента, то:

- можно значительно сократить объем передаваемой информации и ускорить работу сервера
- информация о состоянии также может сохраняться для использования даже в случае перезагрузки сервера.

Отрицательная сторона: информация о состоянии, хранящаяся на сервере, может стать ошибочной, если сообщения были потеряны, продублированы или доставлены не в исходном порядке,

либо если клиент аварийно перезапустился. Соответственно и ответ сервера, основанный на ошибочной информации может быть ошибочным.

Покажем обобщенный алгоритм работы последовательного сервера ***с установлением логического соединения***:

1. Создать сокет и установить связь с локальным адресом.
2. Перевести сокет в пассивный режим, подготавливая его для использования сервером.
3. Принять из сокета следующий запрос на установление соединения и получить новый сокет для соединения.
4. Считывать в цикле запросы от клиента, формировать ответы и отправлять клиенту.
5. После завершения обмена данными с конкретным клиентом закрыть соединение и возвратиться к этапу 3 для приема нового запроса на установление соединения.

На рисунке 2 показана схема организации работы последовательного сервера с установлением логического соединения.

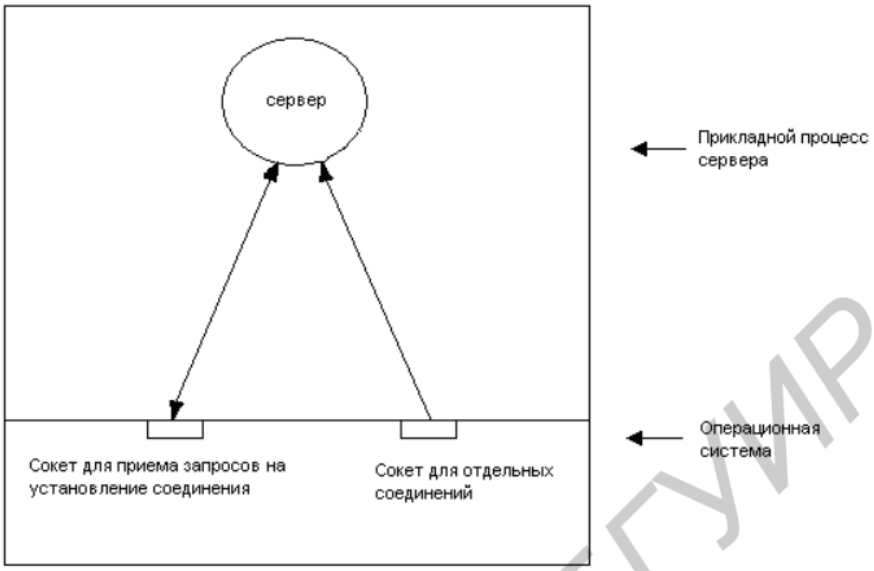


Рисунок 2 – Схема организации работы последовательного сервера с установления логического соединения

Используется в службах, требующих незначительного времени для обработки каждого запроса, однако требующих надежного протокола доставки сообщений. За счет больших издержек на установку и завершение соединения среднее время отклика часто значительно выше, чем у предыдущего сервера.

Наиболее распространенный тип сервера, поскольку сочетает надежный протокол (подходит и для глобальных сетей) с возможностью одновременной работы с несколькими клиентами.

```
main( void)
{
...
char * proto = "tcp";
int sd, rsd, rlen, readed;
...
while(1) {
rlen = sizeof( remote);
rsd = accept( sd, (struct sockaddr *)&remote, &rlen);
 if( (readed = recv(rsd, buf, 512,0)) != -1) {
 gettimeofday( &timev, NULL);
 t_now = ctime( &(timev.tv_sec));
 send(rsd, t_now, strlen(t_now), 0);
 }
 close( rsd);
 }
 return 0;
}
```

27. Классификация серверов. Алгоритм последовательного сервера без установления логического соединения.

Сервер — это программа или устройство, предоставляющее определённые услуги (сервисы) другим программам или устройствам, называемым клиентами, через сеть.
По сути каждый сервер функционирует по следующему алгоритму:
В нем создается сокет и выполняется привязка сокета к порту. Затем сервер входит в бесконечный цикл, в котором он принимает очередной запрос, поступающий от клиента, обрабатывает этот запрос, формирует ответ и отправляет его клиенту.
Время обработки запроса сервером – это общее количество времени, которое требуется серверу для обработки одного отдельно взятого запроса
Время отклика – задержка между временем отправки клиентом запроса и временем получения ответа от сервера.
Время отклика всегда больше времени обработки запроса, однако при применении очереди запросов, подлежащих обработке, отклик может занимать значительно больше времени нежели время обработки запроса.

Классификация серверов:
По внутренней архитектуре
- Последовательный
- Параллельный (несколько запросов за раз)
  - С порождением потока/процесса по запросу
  - С предварительным порождением потоков/процессов (prethreading/preforking)
  - Мультисервисные (суперсервер)
- Асинхронные

По типу используемого сервиса
- С установлением соединения
- Без установления соединения

По состоянию
- С сохранением состояния (stateful)
- Без сохранения состояния (stateless)

Основные:
- Последовательный сервер без установления логического соединения
- Последовательный сервер с установлением логического соединения
- Параллельный сервер без установления логического соединения
- Параллельный сервер с установлением логического соединения

Серверы с установлением логического соединения
Преимущества:
- простота программирования

Недостатки:
- для каждого логического соединения требуется создавать отдельный сокет
- требуется трехэтапное квитирование при установке и разрыве соединения, что невыгодно для использования передачи небольших объемов данных в небольшой сети
- простаивающее соединение, по которому не проходят пакеты, напрасно используют ресурсы

Серверы без установления логического соединения

Преимущества:

- нет накладных расходов на установление и разрыв соединения
- можно реализовывать широковещательные или групповые рассылки
- не требуются ресурсы на поддержание соединения

Недостатки:

- самостоятельная реализация механизмов управления передачей: квитирование, тайм-ауты, оптимизация трафика, контроль надежности

stateful & stateless серверы

Информация о состоянии – это обновляемая сервером информация о ходе взаимодействия с клиентом.

Информация о состоянии применяется для эффективной оптимизации сервера.

Если на сервере сохраняются какие-либо данные о запросах клиента, то:

- можно значительно сократить объем передаваемой информации и ускорить работу сервера
- информация о состоянии также может сохраняться для использования даже в случае перезагрузки сервера.

Отрицательная сторона: информация о состоянии, хранящаяся на сервере, может стать ошибочной, если сообщения были потеряны, продублированы или доставлены не в исходном порядке, либо если клиент аварийно перезапустился. Соответственно и ответ сервера, основанный на ошибочной информации может быть ошибочным.

Алгоритм последовательного сервера без установления логического соединения Используется в службах, требующих незначительного времени для обработки каждого запроса.

Опишем обобщенный алгоритм работы последовательного сервера без установления логического соединения:

1. Создать сокет сервера (связав его с доступным портом и локальным адресом).
2. Считывать в цикле запросы от клиента, формировать ответы и отправлять клиенту в соответствии с прикладным протоколом.

На рисунке показана схема организации работы последовательного сервера без установления логического соединения. Требуется только один поток выполнения, который обеспечивает взаимодействие сервера со многими клиентами с использованием одного сокета.

28. Классификация серверов. Алгоритм параллельного сервера без установления логического соединения.

Классификация серверов в вопросе 27

Алгоритм параллельного сервера без установления логического соединения.

Редко применяемый тип сервера. Во многих случаях затраты на создание потоков или процессов не оправдывают повышения эффективности, достигнутого за счет параллелизма

Ведущий поток:

1 Создать сокет и привязать его к общепринятому адресу службы

2 В цикле считывать запросы с помощью recvfrom и создавать новые ведомые потоки (процессы) для формирования ответа

Ведомый поток:

1 Работа потока начинается с получения конкретного запроса от ведущего потока, а также доступа к сокету

2 Сформировать ответ согласно прикладному протоколу и отправить его клиенту с использованием функции sendto

3 Завершить работу потока

29. Классификация серверов. Алгоритм параллельного сервера с установлением логического соединения.

Классификация серверов в вопросе 27

Алгоритм параллельного сервера с установлением логического соединения.

Ведущий поток:

1 Создать сокет и привязать его к общепринятому адресу
службы

2 Перевести сокет в пассивный режим

3 Вызвать в цикле функцию accept для получения очередного запроса от клиента и создать новый ведомый поток или процесс для формирования ответа

Ведомый поток:

1 Работа начинается с получения доступа к соединению, полученному от ведущего потока

2 Выполнять в цикле работу с клиентом через соединение

3 Закрыть соединение и завершить работу.

30. Классификация серверов. Алгоритм сервера с асинхронным вводом-выводом.

Классификация серверов в вопросе 27

Алгоритм сервера с асинхронным вводом-выводом.

1 Создать сокет и привязать его к общепринятому адресу службы. Добавить сокет к списку сокетов, через которые может осуществляться ввод/вывод

2 Использовать функцию select для получения информации о готовности существующих сокетов к вводу/выводу

3 Если готов первоначальный сокет, то использовать функцию accept для получения очередного запроса на установление соединения и добавить новый сокет к списку сокетов, через которые может осуществляться ввод/вывод

4 Если готов сокет, отличный от первоначального, использовать функцию recv для получения очередного запроса, сформировать ответ и передать ответ клиенту с использованием функции send

5 Перейти к пункту 2

31. Классификация серверов. Серверы с предварительным порождением потоков.

Классификация серверов в вопросе 27

Серверы с предварительным порождением потоков.

Сервер с предварительным порождением потоков (*prethreading*) — это тип параллельного сервера, который заранее создаёт фиксированное количество потоков (пул), каждый из которых готов принимать и обрабатывать клиентские запросы. Потоки существуют постоянно и не создаются заново при каждом новом соединении, что снижает затраты на переключение контекста и ускоряет обработку.

Алгоритм работы сервера с предварительным порождением потоков:

1 При запуске сервера создаётся пул потоков.

2 Каждый поток входит в цикл, в котором ожидает подключения клиента.

3 Один из потоков принимает соединение (вызов accept) и обрабатывает запрос клиента.

4 После завершения работы поток не завершается, а возвращается в состояние ожидания нового запроса.

5 Главный поток управляет пулом и может отслеживать его загрузку или динамически регулировать количество потоков.

Преимущество этого подхода заключается в том, что он позволяет избежать затрат на создание нового потока при каждом клиентском подключении. Это повышает производительность при большом количестве подключений, особенно в высоконагруженных системах.

preforking

- сервер с предварительным созданием дочерних процессов с параллельным вызовом accept
- сервер с предварительным созданием дочерних процессов с блокировкой для защиты accept
- сервер с предварительным созданием дочерних процессов с использованием взаимного исключения для защиты accept
- сервер с предварительным созданием дочерних процессов с последующей передачей дескриптора сокета дочерним процессам

prethreading

- сервер с предварительным созданием потоков с использованием взаимного исключения для защиты accept
- сервер с предварительным созданием потоков, главный поток вызывает accept

32. Классификация серверов. Серверы с предварительным порождением процессов.

Классификация серверов в вопросе 27

Серверы с предварительным порождением процессов.

Сервер с предварительным порождением процессов (preforking) — это тип параллельного сервера, в котором заранее создаётся фиксированное количество дочерних процессов, каждый из которых готов принимать и обрабатывать клиентские запросы. В отличие от серверов с динамическим порождением, процессы создаются один раз при запуске сервера, что снижает накладные расходы, связанные с их созданием при каждом новом подключении.

Алгоритм работы сервера с предварительным порождением процессов:

1 При запуске сервер создаёт пул дочерних процессов.

2 Каждый дочерний процесс входит в цикл ожидания подключений от клиентов.

3 Один из процессов принимает входящее соединение (вызов accept) и обрабатывает клиентский запрос.

4 После завершения работы с клиентом процесс не завершается, а возвращается в состояние ожидания следующего соединения.

5 Главный (родительский) процесс может отслеживать состояние дочерних процессов и при необходимости перезапускать упавшие или регулировать их количество.

Преимущества серверов с предварительным порождением процессов включают отсутствие затрат на создание процесса при каждом подключении, а также повышенную устойчивость, так как сбой одного дочернего процесса не влияет на остальные. Однако серверы с предварительным порождением процессов требуют больше ресурсов на переключение контекста по сравнению с потоками, менее эффективно используют память и процессор.

preforking

- сервер с предварительным созданием дочерних процессов с параллельным вызовом accept
- сервер с предварительным созданием дочерних процессов с блокировкой для защиты accept
- сервер с предварительным созданием дочерних процессов с использованием взаимного исключения для защиты accept
- сервер с предварительным созданием дочерних процессов с последующей передачей дескриптора сокета дочерним процессам

prethreading

- сервер с предварительным созданием потоков с использованием взаимного исключения для защиты accept
- сервер с предварительным созданием потоков, главный поток вызывает accept

33. Классификация серверов. Мультисервисные серверы (на примере xinetd).

Классификация серверов в вопросе 27

Мультисервисные серверы (на примере xinetd).

Мультисервисный сервер (или суперсервер) — это сервер, который не реализует сам функциональность конечного сервиса, а лишь слушает сетевые порты и при получении запроса запускает соответствующий сервис (дочернюю программу)

Используется не отдельный сервер, а "враппер" для сервисов.

- уменьшение надежности
- ограничения по количеству открытых сокетов
- потребляет меньше ресурсов

Принцип работы xinetd:

1. При запуске xinetd открывает сокеты и привязывает их к нужным портам, указанным в конфигурационных файлах.
2. В цикле мониторинга (через select, poll и т.п.) xinetd ожидает активности на сокетах.
3. При поступлении запроса (соединение TCP или дейтаграмма UDP) он:

- определяет, какой службе соответствует данный порт,
- порождает процесс (через fork() и exec()) с нужной службой,
- перенаправляет стандартные потоки службы (stdin/stdout/stderr) в сетевой сокет,
- и возвращается к ожиданию следующих подключений.

Преимущества xinetd:

- Централизованное управление сервисами.
- Экономия ресурсов — демоны запускаются по запросу.

Недостатки:

- Задержка при первом подключении.
- Единая точка отказа.

```
1 /* Если работаем с помощью xinetd, то
2 перенаправляем stdout и stderr в сокет) */
3 if( xinetd)
4 {
5 if ( dup2( STDIN_FILENO, STDOUT_FILENO) == −1) return 1;
6 if ( dup2( STDIN_FILENO, STDERR_FILENO) == −1) return 1; }
```

Мультисервисный сервер без установления соединения

1 Сервер открывает набор сокетов UDP и привязывает к портам служб.
2 Используется таблица соответствия сокетов службам
3 с помощью select сервер переводится в состояние ожидания дейтаграммы
4 для обработки каждой отдельной дейтаграммы вызывается соответствующая функция

Мультисервисный сервер с установлением соединения

1 Сервер открывает набор сокетов TCP и привязывает к портам служб.
2 Используется таблица соответствия сокетов службам
3 с помощью select сокеты переводится в состояние ожидания нового соединения
4 Если готов один из первоначальных сокетов, то создаем новый сокет.
5 Для обработки каждого соединения вызывается соответствующая функция

Модульный мультисервисный сервер с установлением соединения

1 Сервер открывает набор сокетов TCP и привязывает их к портам служб.
2 Используется таблица соответствия сокетов службам
3 с помощью select сокеты переводится в состояние ожидания нового соединения
4 После поступления запроса вызываем fork для создания ведомого процесса
5 ведомый процесс закрывает все ненужные сокеты
6 ведомый процесс производит замещение процесса с помощью вызова из семейства exec

34. Классификация серверов. Методы улучшения функционирования серверов.

Классификация серверов в вопросе 27

Методы улучшения функционирования серверов.

1 Функционирование в фоновом режиме: с помощью fork запускаем копию программы, а родительский процесс убивается. При этом должны закрываться все унаследованные дескрипторы файлов.

2 Сервер должен отключать управляющий терминал, чтобы не получать от него сигналы:
fd=open("/dev/tty", O_RDWR); ioctl(fd, TIOCNOTTY, 0); close(fd);

3 Сервер должен переходить в безопасный каталог используя chdir Необходимо изменить маску по умолчанию для создания файлов используя umask(027)

4 Необходимо открыть стандартные дескрипторы для корректной работы библиотечных процедур:
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
fd=open("/dev/null", O_RDWR); //ввод dup(fd); //вывод
dup(fd); //ошибки

5 Сервер должен предотвращать запуск нескольких копий (лок-файлы или др. системные функции)

6 Желательно игнорировать сигналы, не относящиеся к работе сервера

7 Использование пула потоков или процессов

Создание пула позволяет обрабатывать множество запросов параллельно без накладных расходов на создание новых потоков/процессов при каждом подключении.

8 Асинхронная обработка ввода-вывода

Применение неблокирующих сокетов и механизмов select, poll, epoll, kqueue снижает простои и повышает масштабируемость сервера.

9 Кеширование данных

Хранение часто запрашиваемых данных в оперативной памяти позволяет значительно сократить время отклика.

10 Балансировка нагрузки

Распределение запросов между несколькими экземплярами сервера с помощью балансировщиков нагрузки (например, Nginx, HAProxy) помогает избежать перегрузки отдельных узлов.

11 Ограничение ресурсов и защита от перегрузки

Установка лимитов на количество одновременных подключений, времени выполнения запросов и защита от DoS-атак предотвращают сбои при высокой нагрузке.

12 Масштабируемая архитектура

Разделение на микросервисы, кластеризацию и репликацию позволяет серверу адаптироваться к росту нагрузки.

35. Текстовое и бинарное представление данных в протоколах.

При проектировании прикладных протоколов один из ключевых аспектов — выбор формата представления данных, передаваемых между программами. Наиболее широко используются два типа представления: текстовое и бинарное.

Бинарный формат — это способ кодирования данных в виде последовательности байтов, максимально приближенной к машинному представлению. Все поля имеют жёстко определённую структуру, соответствующую внутренним форматам хранения данных в памяти компьютера.

Когда выгодно использовать бинарное представление?

- Необходимость передачи больших блоков данных и разработчик формата действительно позаботился о достижении максимальной плотности полезной информации в потоке. Пример: мультимедиа форматы.
- Существует жесткое ограничение времени и/или инструкций, необходимых для интерпретации данных. Пример: протоколы сетевого уровня.

Преимущества:

- Компактность: данные занимают минимально возможный объём.
- Высокая производительность: не требуется парсинг строк — данные сразу обрабатываются в нужной форме.
- Подходит для автоматической обработки: удобно для программного взаимодействия без участия человека.

Недостатки:

- Нечитаемость: бинарные данные сложно анализировать вручную, без специальных утилит.
- Сложность отладки и тестирования.
- Проблемы с переносимостью: разные системы могут по-разному интерпретировать одни и те же байты.

Текстовое представление — это способ кодирования данных в виде строк символов, предназначенных для восприятия человеком. Наиболее часто используется кодировка UTF-8, данные передаются в формате текстовых структур (например, JSON, XML, YAML).

Дуг Макилрой (изобретатель pipes):

- Будьте готовы к тому, что вывод каждой программы станет вводом другой, еще неизвестной программы.
- Не загромождайте вывод посторонней информацией.
- Избегайте строгих табличных или двоичных форматов ввода.
- Не настаивайте на интерактивном вводе

Преимущества:

- Читаемость: легко воспринимаются и анализируются человеком.
- Простота отладки и тестирования: можно использовать простые инструменты.
- Универсальность: текстовые потоки легко обрабатываются в различных языках программирования и операционных системах.
- Гибкость: формат может легко расширяться или изменяться.

Недостатки:

- Избыточность: текстовые данные часто занимают больше места (например, число 1234 занимает 2 байта в бинарном виде, но 4 символа в тексте).
- Меньшая производительность: требуется парсинг строк, что увеличивает нагрузку.
- Риск неоднозначности при неправильной интерпретации структуры.

Если необходима производительность, то можно внедрить сжатие текстового потока на уровне выше или ниже. Часто такая конструкция является более производительной, чем бинарное представление.

Упрощается интерпретация и анализ взаимодействия приложений, а также написание тестовых программ. Серверные процессы часто запускаются с помощью суперсерверов подобных inetd/xinetd, так что сервер получает команды на стандартный ввод и отправляет ответ на стандартный вывод.

Можно взаимодействовать с сервером или клиентом с помощью программ telnet или netcat.

36. Проектирование протоколов прикладного уровня. Механизмы протокола.

Проектирование прикладного протокола — это процесс создания формата и правил обмена данными между программами через сеть с учетом целей, архитектуры и требований системы. Основная цель — обеспечить надёжную, эффективную и понятную коммуникацию между приложениями, с учётом архитектурных особенностей системы, требований к надёжности, скорости, безопасности и масштабируемости.

Процесс проектирования обычно включает следующие **этапы**:

1. Анализ требований (какую задачу должен решать протокол)
2. Выбор модели взаимодействия (ориентация на соединения)
3. Определение формата сообщений и логики обмена
4. Механизмы надёжности и управления сессиями
5. Реализация и тестирование

Определение свойств проектируемого протокола

1 Является ли протокол ориентированным на соединения?

2 Нужно ли использовать запросы и ответы для обмена сообщениями?

3 Нужна ли возможность обмена асинхронными сообщениями?

Ориентация на соединения

Прикладной протокол ориентирован на соединения (т.е. работает поверх TCP или SCTP) Без установления соединения – к ним относятся прикладные протоколы, для которых нет нужды в задержках на установление/разрыв соединений, а также поддержке надежной передачи данных. При необходимости надежность реализуется средствами прикладного протокола.

Обмен сообщениями

Под сообщением подразумевается простая структура данных, которыми обмениваются слабо-связные системы. В качестве противовеса можно привести сильно-связные системы, например RPC. Проблема в том, что слабо- и сильно-связные системы – граничные части целого спектра.

Асинхронный обмен

Синхронная peer-to-peer модель

Модель в стиле «запрос-ответ», где каждая из сторон может являться клиентом и/или сервером. Большинство протоколов являются синхронными: почта, WWW и т. п.

Часть протоколов не может быть построена по синхронным принципам — например: сетевые файловые системы, системы именования, мультикаст-сообщения и т.п.

Механизмы протокола

Как и какие задачи выполняет протокол?

- Структурирование – заголовки, концевики.
- Кодирование – представление.
- Отчетность – каким образом описываются ошибки.
- Асинхронность – как производится обмен независимыми сообщениями.
- Аутентификация – каким образом стороны идентифицируют и проверяют друг друга. Конфиденциальность – защита от перехвата или измения данных.

Структурирование

Octet-stuffing

Например SMTP – команды заканчиваются переводом каретки (CR-LF)

Octet-counting
Сообщение разделяется на заголовок и тело; в заголовке указывается размер данных в теле сообщения.

Connection-blasting
Например информационное соединение FTP.

Кодирование
Зависит от задачи! Под кодированием подразумевается формат представления команд и данных. Например для SMTP он определяется в RFC 822. В Интернете MIME является «де-факто» стандартом представления данных.

Отчетность
«Теория ответных кодов» состоит из 3-х цифр:
1 Успех-неуспех, либо постоянное или временное состояние;
2 ответственная часть системы;
3 идентификация конкретной ситуации.
Код может дополняться текстовым сообщением для человека.
«Теория ответных кодов» имеет 2 нерешенных недостатка:

- коды используются как для ответа на операцию, так и для индикации изменения состояния прикладного протокола;
- код не указывает, кому адресуется ошибка – пользователю, администратору или программисту.

Асинхронность
Протокол является асинхронным, если он поддерживает независимые обмены по одному соединению.
Наиболее широко распространенный подход – pipelining, позволяет делать множество запросов к серверу, но требует, чтобы запросы были обслужены последовательно.

Аутентификация
Механизмы специфичны для каждого протокола. Например, FTP использует один механизм, HTTP – другой, а SMTP (классический) не использует его вовсе. Рекомендуется к использованию SASL (RFC 222 – Simple Authentication and Security Layer) – фреймворк для аутентификации сторон, использующий в том числе OTP (one-time-passwords)

Конфиденциальность
Как правило, подразумевается использование криптографических средств для шифрования данных. Наиболее часто используемый механизм – TLS/SSL – использует шифрование соединения, а не отдельных объектов.
Можно использовать различные подходы для организации:

- прослушивать на одном порту незащищенные соединения, а на другом – защищенные (SSL);
- прослушивать и защищенные, и незащищенные соединения на одном и том же порту, что требует поддержки со стороны протокола прикладного уровня (TLS).

37. Проектирование протоколов прикладного уровня. Свойства протокола.

Проектирование прикладного протокола написано в вопросе 36

Свойства протокола

При проектировании протокола необходимо учитывать некоторые свойства:

1 Масштабируемость
2 Эффективность
3 Простоту
4 Расширяемость
5 Устойчивость

Масштабируемость

Хорошо спроектированный протокол должен быть масштабируемым.
Не все протоколы поддерживают асинхронность. Частая практика – использовать несколько соединений для утилизации канала, однако при этом необходимо учитывать:

- свойства транспортного протокола;
- сервер трактует каждое новое соединение по протоколу прикладного уровня, как независимое (проблема аутентификации и доступа к ресурсам сессии).

Эффективность

Хорошо спроектированный протокол должен быть эффективным. Например, использование octet-staffing'а упрощает реализацию протокола, в то время, как octet-counting потребляет немного меньше ресурсов.

Простота

Хорошо спроектированный протокол должен быть простым. Хорошее правило для определения насколько протокол прост:

- в хорошо спроектированном протоколе усилия для изменения пропорциональны сложности изменений;
- в плохо спроектированном протоколе необходимо приложить много усилий для любых изменений.

Расширяемость

Хорошо спроектированный протокол должен быть расширяемым. Невозможно предсказать заранее, с какими проблемами столкнется протокол. Таким образом желательно предусмотреть способы расширения функциональности.

Устойчивость

Хорошо спроектированный протокол должен быть устойчивым. Внезапно! Но для протоколов принцип устойчивости Постела «будь либерален к тому, что принимаешь и консервативен к тому, что отсылаешь» может привести к обратному эффекту. Проблема в количестве и качестве различных реализаций одного и того же протокола

38. Характеристика протокола HTTP. Функционирование протокола HTTP

Протокол **HTTP (HyperText Transfer Protocol)** — это прикладной протокол уровня модели OSI, используемый для передачи гипертекста в интернете, в первую очередь между клиентами (обычно браузерами) и серверами.

1 HTTP функционирует по модели «запрос-ответ»: клиент инициирует запрос, а сервер обрабатывает его и возвращает ответ.

2 Каждый HTTP-запрос обрабатывается независимо; сервер не сохраняет информацию о предыдущих запросах. Для реализации сессий используются механизмы, такие как cookies и session tokens.

3 HTTP использует текстовый формат для передачи данных. Заголовки HTTP позволяют легко расширять функциональность протокола без изменения его ядра.

4 HTTP используется также в качестве базового протокола для коммуникации пользовательских агентов с прокси-серверами и другими системами Интернет, в том числе и использующие протоколы SMTP, NNTP, FTP, Gopher, XMPP и многих других.

5. В качестве протокола транспортного уровня HTTP использует TCP.

Инициация запроса

Когда пользователь вводит URL в браузере, происходит преобразование доменного имени в IP-адрес через DNS. Затем браузер устанавливает TCP-соединение с сервером

Формирование HTTP-запроса

Браузер формирует HTTP-запрос, который включает:

1.Стартовую строку с методом (GET, POST…), URI и версией протокола.

2. Заголовки, содержащие дополнительную информацию (Host, User-Agent, Accept…).

3. Тело запроса, присутствующее при отправке данных.

Обработка запроса сервером

Сервер получает запрос, анализирует его и формирует ответ, который включает:

1. Стартовую строку ответа с кодом состояния (например, 200 OK, 404 Not Found).

2.Заголовки ответа, такие как Content-Type, Content-Length, Set-Cookie.

3.Тело ответа, содержащее запрашиваемые данные, например, HTML-документ или изображение.

Получение и отображение ответа клиентом

Браузер получает ответ, интерпретирует его и отображает содержимое пользователю. При необходимости он может инициировать дополнительные запросы для загрузки связанных ресурсов (CSS, JavaScript, изображения).

Методы HTTP: Протокол поддерживает различные методы для выполнения операций:

GET — получение ресурса.

POST — отправка данных на сервер.

PUT — обновление ресурса.

DELETE — удаление ресурса.

HEAD, OPTIONS, PATCH и другие для специфических задач.

Коды состояния: Сервер возвращает коды состояния для информирования клиента о результате обработки запроса:
2xx — успешная обработка.
3xx — перенаправление.
4xx — ошибка клиента.
5xx — ошибка сервера.
Поддержка кэширования: HTTP позволяет использовать механизмы кэширования для снижения нагрузки на сервер и ускорения загрузки страниц.
Шифрование данных: Для обеспечения безопасности используется HTTPS — расширение HTTP с поддержкой SSL/TLS, обеспечивающее шифрование передаваемых данных.

P.S. полезно знать!

URI — это общий термин, обозначающий строку символов, идентифицирующую ресурс в интернете. URI может указывать на ресурс по его местоположению, имени или обоим параметрам одновременно.

URL (пользователь)— это подмножество URI, которое указывает на местоположение ресурса и способ его получения. URL содержит информацию о протоколе доступа, домене, порте и пути к ресурсу.

```
<схема(протокол)>://<домен>:<порт>/<путь>?<параметры>#<фрагмент>
```

URN (сервер)— это подмножество URI, которое идентифицирует ресурс по его имени в определённом пространстве имён, не указывая его местоположение. URN обеспечивает постоянную и уникальную идентификацию ресурса.

```
urn:<пространство_имён>:<уникальный_идентификатор>
```

39. Характеристика протокола HTTP. Запросы HTTP

Характеристика написана в вопросе 38

Запросы позволяют клиенту запрашивать ресурсы, отправлять данные и управлять состоянием на сервере.

Структура HTTP-запроса

1.Стартовая строка

Содержит метод запроса, путь к ресурсу (URI) и версию протокола.

```
GET /index.html HTTP/1.1
```

2.Заголовки (Headers)

Представляют собой пары "ключ: значение", передающие дополнительную информацию о запросе. Заголовки могут быть:

- **-Общие (General Headers):** применяются как к запросам, так и к ответам.
- **-Заголовки запроса (Request Headers):** относятся только к запросам.
- **-Заголовки сущности (Entity Headers):** описывают тело сообщения, если оно присутствует.

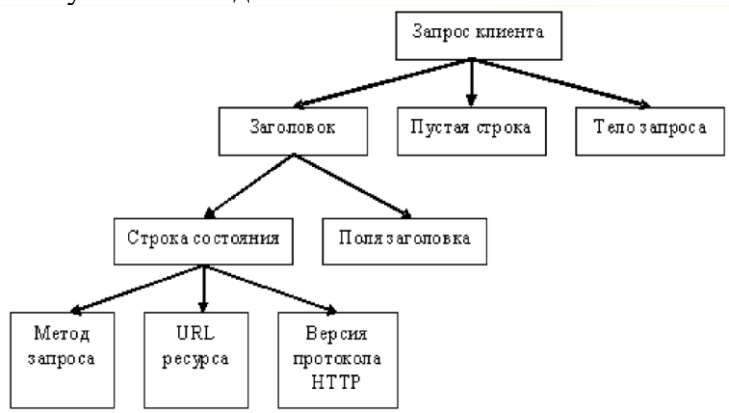
```
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

3.Пустая строка

Отделяет заголовки от тела запроса.

4.Тело запроса (Body)

Содержит данные, отправляемые на сервер (например, данные формы). Присутствует не во всех запросах, чаще всего используется с методами POST и PUT.



Основные методы HTTP-запросов

Методы определяют действие, которое клиент хочет выполнить

GET: Запрашивает представление ресурса. Не должен изменять состояние сервера.
POST: Отправляет данные на сервер, часто используется для создания новых ресурсов.
PUT: Заменяет текущий ресурс данными запроса.
DELETE: Удаляет указанный ресурс.
HEAD: Аналогичен GET, но без тела ответа; используется для получения метаданных.
OPTIONS: Возвращает методы, поддерживаемые сервером для указанного ресурса.
PATCH: Вносит частичные изменения в ресурс.
TRACE: Возвращает полученный запрос; используется для диагностики.
CONNECT: Устанавливает туннель к серверу, часто используется для HTTPS.
Также можно делать кастомные методы.

```
POST /submit-form HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
name=Aboba&email=Aboba@example.com
```

40. Характеристика протокола HTTP. HTTP-ответ

Характеристика написана в вопросе 38

Ответ содержит информацию о результате обработки запроса и, при необходимости, возвращает запрошенные данные.

HTTP-ответ состоит из следующих компонентов:

1.Статусная строка (Status Line)
Версию протокола, код состояния, фраза причины

```
HTTP/1.1 200 OK
```

2.Заголовки (Headers)
Заголовки могут включать:

Content-Type: тип возвращаемого содержимого (например, text/html)
Content-Length: длина тела ответа в байтах
Server: информация о сервере
Set-Cookie: установка cookie в браузере клиента
Cache-Control: инструкции по кэшированию

3.Пустая строка
Отделяет заголовки от тела ответа.
4.Тело ответа (Body)
Содержит непосредственно данные, возвращаемые сервером, такие как HTML-документ, изображение, JSON-объект и т.д. Тело может отсутствовать, например, в ответах на запросы с методом HEAD.
Коды состояния HTTP делятся на пять категорий:
2xx — успешная обработка.
3xx — перенаправление.
4xx — ошибка клиента.
5xx — ошибка сервера.

```
HTTP/1.1 200 OK
Date: Mon, 01 May 2025 12:00:00 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 137

<!DOCTYPE html>
<html>
<head>
  <title>Пример страницы</title>
</head>
<body>
  <h1>Добро пожаловать!</h1>
</body>
</html>
```

41. Характеристика протокола HTTP. Безопасность, аутентификация, cookies.

Характеристика написана в вопросе 38

Безопасность HTTP

Протокол HTTP изначально не обеспечивает шифрование данных, что делает его уязвимым к атакам типа «человек посередине» (MITM). Для защиты данных используется HTTPS — HTTP поверх TLS/SSL, обеспечивающий:

Конфиденциальность: шифрование передаваемых данных.
Целостность: предотвращение изменения данных в процессе передачи.
Аутентификацию: подтверждение подлинности сервера.
Использование HTTPS особенно важно при передаче чувствительных данных, таких как учетные данные и персональная информация.

Аутентификация в HTTP

Аутентификация подтверждает личность пользователя. Основные методы:

Базовая аутентификация: передача имени пользователя и пароля в заголовке запроса, закодированных в Base64. Требует использования HTTPS для безопасности.
Аутентификация с использованием токенов: после входа пользователь получает токен (например, JWT), который отправляется с каждым запросом в заголовке Authorization.
Аутентификация с использованием cookies : после успешного входа сервер устанавливает cookie с идентификатором сессии, который автоматически отправляется с последующими запросами.
Каждый метод имеет свои особенности и применяется в зависимости от требований безопасности и архитектуры приложения.

Cookies

Cookies — это небольшие фрагменты данных, хранящиеся в браузере пользователя, которые используются для хранения информации о состоянии сеанса, предпочтениях и аутентификации.
После входа в систему сервер может установить cookie с идентификатором сессии. Браузер автоматически отправляет этот cookie с каждым последующим запросом, позволяя серверу идентифицировать пользователя без повторной аутентификации.
Для повышения безопасности cookies рекомендуется использовать следующие атрибуты:

Secure: гарантирует, что cookie будет отправляться только по защищенному соединению (HTTPS).
HttpOnly: предотвращает доступ к cookie через JavaScript, снижая риск атак XSS.
SameSite: ограничивает отправку cookie только с запросами с того же сайта, что помогает предотвратить атаки CSRF.

42.Проектирование протоколов прикладного уровня на базе HTTP.

Проектирование прикладных протоколов на базе HTTP — это создание надстроек над HTTP, обеспечивающих взаимодействие между клиентами и серверами в различных приложениях. HTTP, как протокол прикладного уровня, предоставляет основу для обмена сообщениями, на которой можно строить более сложные протоколы, такие как REST.

Причины использования

1. Знакомство и зрелость технологии

HTTP — это один из самых известных и документированных протоколов. Его легко понять, внедрить и протестировать. Разработчики, системные администраторы и архитекторы хорошо знакомы с его принципами.

2.Совместимость с браузерами и инструментами

HTTP — нативный протокол веб-браузеров, что делает его идеальным для клиент-серверных приложений, ориентированных на пользователей.

3.Повторное использование существующих библиотек

Большое количество клиентских и серверных библиотек (на всех популярных языках программирования) сокращают время разработки и упрощают поддержку.

4.Скорость разработки (прототипирования)

Использование HTTP даёт возможность быстро развернуть сервер через CGI, FastCGI, RESTful-обвязки и другие фреймворки.

5.Встроенные механизмы безопасности

Существуют готовые схемы аутентификации: Basic, Digest, TLS/SSL. Это удобно, особенно для простых сценариев.

6.Сетевые преимущества

HTTP-пакеты часто проходят через прокси, NAT и фаерволы без дополнительной настройки. Это позволяет приложениям работать «из коробки» в сложных сетевых условиях.

Проблемы выбора при использовании HTTP

1. **Сложность протокола**

Хотя HTTP изначально прост, со временем он оброс множеством расширений:

-постоянные соединения (keep-alive),
-кэширование,
-сжатие,
-согласование содержимого (content negotiation),
-поддержка частичных запросов (byte ranges).

Эти функции хороши для веба, но часто не нужны прикладным протоколам, создавая ненужную сложность и накладные расходы.

2. **Избыточность**

-Протокол работает поверх TCP, и каждое соединение требует установки сессии.
-Текстовый формат заголовков увеличивает размер передаваемых данных.
-Высокая задержка при частом обмене короткими сообщениями (в отличие от, например, WebSocket или UDP).

3**. Безопасность**

-Механизмы HTTP-аутентификации (Basic/Digest) имеют ограничения, включая хранение общих секретов.
-TLS/SSL подвержены ряду практических проблем:
  -Неудобство управления сертификатами (особенно самоподписанными).
  -Устаревшие или уязвимые алгоритмы шифрования по умолчанию.

4. **Совместимость с прокси, кэшами и NAT**

-Это преимущество может обернуться проблемой: нестандартные методы, URI-схемы и коды состояния могут быть проигнорированы, отфильтрованы или модифицированы промежуточными узлами.
-Необходимо понимать, как поведение прокси влияет на логику приложения.

43. Особенности программирования приложений с использованием протокола ICMP (на примере реализации утилит ping или traceroute).

ICMP (Internet Control Message Protocol) — это протокол сетевого уровня, входящий в стек TCP/IP, который используется для передачи управляющих сообщений и уведомлений об ошибках в IP-сетях. Он играет ключевую роль в диагностике сетевых соединений, и такие утилиты, как **ping** и **traceroute**, являются классическими примерами его применения.

1. Основы протокола ICMP

- ICMP работает на сетевом уровне и инкапсулируется непосредственно в IP-пакеты, не используя порты, как это делают транспортные протоколы (TCP или UDP). Его основное назначение — передача сообщений, таких как:
- **Эхо-запрос (Echo Request)** и **Эхо-ответ (Echo Reply)** — используются в ping для проверки доступности узла.
- **Сообщение о превышении времени жизни (Time Exceeded)** — применяется в traceroute для определения маршрута.
- **Сообщение о недостижимости узла (Destination Unreachable)** — сигнализирует о проблемах доставки.
- Эти сообщения имеют определённую структуру, включающую заголовок (тип, код, контрольная сумма) и, при необходимости, данные. Программисту важно понимать, как формировать и интерпретировать такие сообщения.
- Использовать select для управления таймаутами.

2. Особенности программирования с ICMP

Создание приложений, использующих ICMP, связано с рядом технических нюансов.

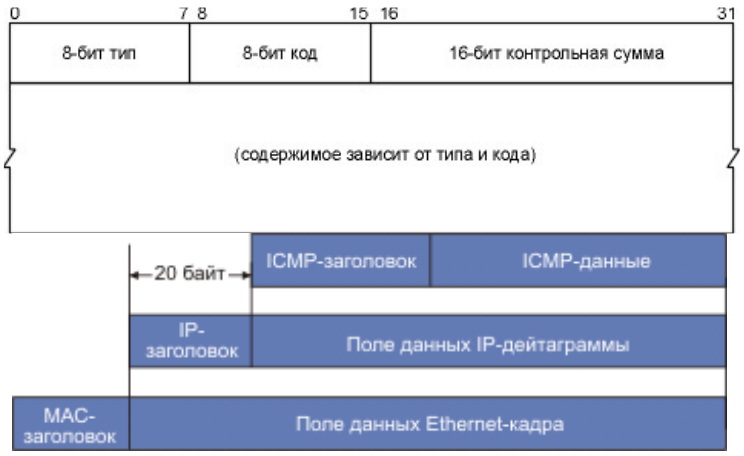
2.1. Использование raw-сокетов

Для работы с ICMP применяются **raw-сокеты** (сырые сокеты), которые позволяют напрямую взаимодействовать с IP-пакетами, включая их заголовки. Это отличает их от обычных сокетов TCP или UDP, где заголовки формируются автоматически.

- В языке C в Unix-подобных системах raw-сокет создаётся так:
```
int sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

2.2. Формирование ICMP-пакетов

При использовании raw-сокетов разработчик вручную формирует ICMP-пакеты. Структура пакета ICMP (+ ICMP упакованный) выглядит следующим образом:



Тип и код определяют назначение сообщения (например, 8/0 для эхо-запроса, 0/0 для эхо-ответа).
Идентификатор и **номер последовательности** помогают отличать отправленные запросы от полученных ответов.
Контрольная сумма вычисляется по всему сообщению (заголовок + данные) для проверки целостности.

3. Реализация утилиты ping

Ping проверяет доступность узла, отправляя ICMP эхо-запросы (тип 8) и ожидая эхо-ответы (тип 0). Рассмотрим процесс создания такой утилиты.

3.1. Алгоритм реализации

1. **Создание сокета**: `int sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);`
2. **Формирование запроса**:
   - Тип: 8 (Echo Request).
   - Идентификатор: уникальное значение (например, PID процесса).
   - Номер последовательности: инкрементируется для каждого запроса.
   - Временная метка в данных для расчёта RTT.
3. **Отправка пакета**:
   - Используется `sendto()` с указанием IP-адреса назначения.
4. **Приём ответа**:
   - `recvfrom()` получает ICMP-пакет.
   - Проверяется, что тип = 0 (Echo Reply) и идентификатор совпадает.
   - RTT вычисляется как разница между временем получения и отправки.
5. **Периодичность**:
   - Запросы отправляются с интервалом (например, 1 секунда) через таймер.

4. Реализация утилиты traceroute

Traceroute определяет маршрут до узла, используя механизм TTL (Time To Live) и ICMP-сообщения о превышении времени жизни (тип 11).

4.1. Алгоритм реализации

1. **Создание сокетов**:
   - Для отправки: UDP-сокет (традиционная реализация) или ICMP-сокет.
   - Для приёма: raw-сокет для ICMP.
2. **Отправка пакетов**:
   - Начинаем с TTL = 1, увеличивая его на каждом шаге.
   - Для UDP отправляем датаграммы на случайный порт (>33434).
   - Для ICMP отправляем эхо-запросы.
3. **Приём ответов**:
   - Промежуточные маршрутизаторы возвращают **ICMP Time Exceeded** (тип 11).
   - Целевой узел отвечает **ICMP Echo Reply** (для ICMP) или **ICMP Port Unreachable** (для UDP).
4. **Управление TTL**:
   ```
   int ttl = 1;
   setsockopt(sock, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
   ```
5. **Таймауты**:
   - Используется select() для ожидания ответа в течение заданного времени.

5. Общие сложности и нюансы

- **Права доступа:** Необходимы привилегии root для raw-сокетов.
- **Фильтрация:** Пакеты фильтруются по идентификатору, чтобы исключить чужие ответы.
- **Безопасность:** Неправильная реализация может привести к уязвимостям (например, ICMP-флуд).
- **Фрагментация:** Нужно учитывать возможность фрагментированных пакетов.

44. Атаки ICMP-flooding и Smurf: особенности реализации.

ICMP-flooding и **Smurf** — это два типа атак типа "отказ в обслуживании" (DoS), использующих протокол ICMP (Internet Control Message Protocol) для перегрузки целевой системы. Обе атаки направлены на исчерпание ресурсов жертвы, но различаются по методам реализации и масштабам воздействия.

1. ICMP-flooding – это прямая атака, при которой злоумышленник отправляет большое количество ICMP-пакетов (обычно Echo Request, как в утилите ping) на целевую систему. Цель — перегрузить сетевые или вычислительные ресурсы жертвы, что может привести к замедлению или полному отказу в обслуживании. **Особенности реализации:**

- **Генерация большого количества ICMP-пакетов**: Злоумышленник использует специальные инструменты или скрипты для отправки множества ICMP Echo Request пакетов на целевой хост. Это может быть реализовано с помощью утилит, таких как hping3, или через написание собственных программ на языках вроде C или Python.
- **Подмена IP-адреса (IP spoofing)**: Чтобы скрыть свою личность и затруднить отслеживание, злоумышленник может подделывать IP-адрес отправителя. Это делается путём формирования пакетов с фальшивым IP-адресом источника, что особенно эффективно при использовании raw-сокетов.
- **Использование ботнетов**: Для усиления атаки злоумышленники могут задействовать ботнеты — сети зараженных компьютеров, которые одновременно отправляют ICMP-пакеты на жертву. Это позволяет распределить нагрузку и увеличить масштаб атаки.

2. Smurf – это более изощрённая атака, которая также использует ICMP-пакеты, но с механизмом **усиления** за счёт широковещательных запросов. В этой атаке злоумышленник отправляет ICMP Echo Request на широковещательный адрес сети, подделывая IP-адрес отправителя как IP-адрес жертвы. В результате все хосты в сети отправляют ответы (ICMP Echo Reply) на жертву, что приводит к её перегрузке. **Особенности реализации:**

- **Использование широковещательных адресов**: Злоумышленник отправляет ICMP Echo Request на широковещательный адрес сети (например, 192.168.1.255 для сети 192.168.1.0/24). При этом в пакете подделывается IP-адрес отправителя, указывая IP жертвы.
- **Усиление атаки**: Все активные хосты в сети, получившие широковещательный запрос, отправляют ICMP Echo Reply на поддельный IP-адрес (жертву). Таким образом, один отправленный пакет может вызвать множество ответов, что усиливает нагрузку на жертву пропорционально количеству хостов в сети.
- **Масштабирование**: Эффективность атаки Smurf зависит от размера сети: чем больше хостов в сети, тем сильнее усиление. В крупных сетях это может привести к значительной перегрузке жертвы.

3. Основные различия между атаками

ICMP-flooding:
Прямая атака, наводняющая жертву ICMP-пакетами.
Требует значительных ресурсов со стороны атакующего или использования ботнетов для масштабирования.

- **Smurf**:
  - Использует механизм усиления за счёт широковещательных запросов.
  - Один отправленный пакет может вызвать множество ответов, что делает атаку более эффективной при меньших ресурсах атакующего.

4. Меры защиты

- **Фильтрация ICMP-трафика**: Настройка межсетевых экранов (firewall) для ограничения или блокировки входящих ICMP Echo Request пакетов, особенно от недоверенных источников.
- **Отключение ответов на широковещательные запросы**: Настройка хостов и маршрутизаторов для игнорирования ICMP Echo Request, отправленных на широковещательные адреса. Это предотвращает участие в атаках Smurf.
- **Ограничение скорости ICMP-трафика**: Настройка систем для ограничения количества обрабатываемых ICMP-пакетов в единицу времени, что помогает предотвратить перегрузку.
- **Использование антиспуфинг фильтров**: Блокировка пакетов с поддельными IP-адресами на уровне маршрутизаторов, что затрудняет проведение атак с подменой IP.

45. IP-spoofing. Особенности создания пакетов протокола TCP (на примере атак SYN-flooding и Land).

IP-spoofing — это техника подделки IP-адреса источника в заголовке IP-пакета. Она позволяет злоумышленнику скрыть свой реальный адрес, выдать себя за доверенный хост или перенаправить ответы на другой IP. Используется в атаках типа "отказ в обслуживании" (DoS), таких как SYN-flooding и Land.

Реализация IP-spoofing

- Используются **raw-сокеты** для формирования пакетов вручную:

```
int sock = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

- Активируется опция IP_HDRINCL для управления заголовком IP:

```
setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
```

Структура IP-заголовка

- **Версия**: 4 (IPv4).
- **Длина заголовка (IHL)**: 5 (20 байт).
- **Общая длина**: заголовок + данные.
- **Идентификатор (id)**: уникальный для пакета.
- **Флаги и смещение**: 0 (без фрагментации).
- **TTL**: 255.
- **Протокол**: 6 (TCP).
- **Контрольная сумма**: вычисляется по заголовку.
- **IP-адрес источника**: поддельный.
- **IP-адрес назначения**: адрес жертвы.

SYN-flooding: атака на основе TCP

SYN-flooding — атака, при которой отправляется множество TCP SYN-пакетов на сервер, но рукопожатие не завершается. Это перегружает сервер, исчерпывая его ресурсы.

Особенности создания пакетов для SYN-flooding

- **TCP-заголовок**:
  - Флаг **SYN** установлен (syn = 1).
  - Случайный номер последовательности (seq).
  - Порт источника: случайный.
  - Порт назначения: целевой порт жертвы.
- **IP-spoofing**: поддельный IP-адрес источника, чтобы сервер отправлял SYN-ACK на несуществующий адрес.
- **Пример кода**:
- struct tcphdr *tcp_hdr = (struct tcphdr *)(sendbuf + sizeof(struct iphdr));
- tcp_hdr->source = htons(sport); // случайный порт
- tcp_hdr->dest = htons(dport); // порт жертвы
- tcp_hdr->seq = htonl(rand()); // случайный seq
- tcp_hdr->doff = 5; // длина заголовка
- tcp_hdr->syn = 1; // флаг SYN
  tcp_hdr->window = htons(128); // размер окна

Механизм атаки

Сервер резервирует ресурсы для каждого SYN-запроса, но из-за поддельного IP не получает ACK, что приводит к переполнению очереди соединений.

Land: атака с зацикливанием

Land — атака, при которой TCP SYN-пакет имеет одинаковые IP-адреса и порты источника и назначения. Это заставляет систему установить соединение с самой собой, вызывая зацикливание.

Особенности создания пакетов для Land

- **TCP-заголовок**:
  - Флаг **SYN** установлен.
  - Порты источника и назначения идентичны.
- **IP-заголовок**:
  - IP-адрес источника и назначения совпадают (например, 192.168.0.1).
- **Результат**: система отправляет SYN-ACK сама себе, что может привести к отказу в обслуживании.

Пример структуры пакета

- Источник: 192.168.0.1:80
- Назначение: 192.168.0.1:80
- Флаг SYN: 1

Общие особенности создания TCP-пакетов

- **Псевдозаголовок**: включает IP-адреса, протокол и длину сегмента для вычисления контрольной суммы TCP.
- **Поля TCP**:
  - **Порты**: случайные (SYN-flooding) или одинаковые (Land).
  - **Номер последовательности**: уникален для каждого пакета.
  - **Флаги**: SYN для обеих атак.
- **Контрольная сумма**: вычисляется по псевдозаголовку, заголовку TCP и данным.

46. Протокол IPv6. Характеристики. Преимущества и отличия от IPv4.

IPv6 (Internet Protocol version 6) — это сетевой протокол, разработанный для замены IPv4 (Internet Protocol version 4). Он был стандартизирован в RFC-2460 и RFC-4291 и призван устранить ключевые ограничения IPv4, такие как нехватка адресного пространства, а также предложить новые возможности для современных сетей. IPv6 — это развитие сетевых технологий, устраняющее ограничения IPv4 и предоставляющее новые возможности для масштабируемости, безопасности и эффективности.

Характеристики IPv6

1. Расширенное адресное пространство

IPv6 использует **128-битные адреса**, что обеспечивает до 3.4×10^{38} уникальных адресов, в отличие от 32-битных адресов IPv4 (4.3×10^{9} адресов). Поддерживает больше уровней иерархии адресации и упрощает авто-конфигурацию.

2. Упрощенный формат заголовка

Заголовок IPv6 имеет фиксированную длину **40 байт**, что снижает издержки на обработку пакетов. Некоторые поля IPv4 удалены или стали опциональными.

3. Поля заголовка IPv6

- **Version**: Указывает версию протокола (6 для IPv6).
- **Traffic Class (8 бит)**: Определяет класс трафика или приоритет; значение по умолчанию — 0, может изменяться промежуточными узлами.
- **Flow Label (20 бит)**: Позволяет помечать пакеты одного потока для специальной обработки.
- **Payload Length (16 бит)**: Указывает длину данных после заголовка (до 65 535 байт; для больших данных используется Jumbo Payload).
- **Next Header (8 бит)**: Указывает тип следующего заголовка.
- **Hop Limit (8 бит)**: Ограничивает количество прыжков пакета; при достижении 0 пакет уничтожается.
- **Source Address (128 бит)** и **Destination Address (128 бит)**: Адреса отправителя и получателя.

4. Типы адресов

- **Unicast**: Адрес одного интерфейса.
- **Anycast**: Адрес группы интерфейсов, пакет доставляется ближайшему из них.
- **Multicast**: Адрес группы интерфейсов, пакет доставляется всем участникам группы.

5. Представление адресов

Адреса записываются как восемь групп по четыре шестнадцатеричные цифры, разделенные двоеточиями (например, fedc:ba98:7654:3210:fedc:ba98:7654:3210). Лишние нули можно сокращать с помощью: (например, 1080:0:0:0:8:800:200c:417a → 1080::8:800:200c:417a).

6. Авто-конфигурация Поддерживает **SLAAC (Stateless Address Autoconfiguration)**, что упрощает настройку адресов без DHCP.

7. Безопасность Встроенная поддержка **IPsec** для шифрования и аутентификации.

Преимущества IPv6

1. **Решение проблемы нехватки адресов**
   Огромное адресное пространство устраняет необходимость в NAT и поддерживает рост числа устройств (например, IoT).
2. **Упрощенная маршрутизация**
   Фиксированный заголовок и отсутствие фрагментации на маршрутизаторах ускоряют обработку пакетов.
3. **Улучшенная поддержка расширений**
   Гибкое кодирование опций упрощает внедрение новых функций.
4. **Эффективный мультикаст**
   Замена широковещания мультикастом снижает нагрузку на сеть.
5. **Повышенная безопасность**
   IPsec встроен в протокол, обеспечивая защиту данных.

Отличия от IPv4

Характеристика	IPv4	IPv6
Длина адреса	32 бита	128 бит
Число адресов	~4.3×10^{9}	~3.4×10^{38}
Формат заголовка	Переменная длина	Фиксированная (40 байт)
Фрагментация	Выполняется маршрутизаторами	Только отправителем
Широковещание	Есть	Нет (заменено мультикастом)
NAT	Широко используется	Не требуется
Авто-конфигурация	DHCP или вручную	SLAAC
Поля заголовка	Больше полей (включая checksum)	Упрощены, добавлен Flow Label

47. Протокол IPv6. Типы адресов. Запись адреса.

В протоколе IPv6 существуют три основных типа адресов:

1. **Unicast**
   Адрес unicast идентифицирует один интерфейс. Пакет, отправленный на такой адрес, доставляется конкретному интерфейсу.
2. **Anycast**
   Адрес anycast идентифицирует группу интерфейсов, обычно расположенных на разных узлах. Пакет, отправленный на anycast-адрес, доставляется одному из интерфейсов в группе, как правило, ближайшему по метрике маршрутизации.
3. **Multicast**
   Адрес multicast идентифицирует группу интерфейсов, обычно на разных узлах. Пакет, отправленный на multicast-адрес, доставляется всем интерфейсам в группе.

Примечание: Широковещательные адреса (broadcast), которые использовались в IPv4, в IPv6 отсутствуют. Их функции заменены multicast-адресами.

Запись адреса в IPv6

Адреса IPv6 представляют собой 128-битные числа и записываются в виде восьми групп по четыре шестнадцатеричные цифры, разделённые двоеточиями. Пример полного адреса:
fedc:ba98:7654:3210:fedc:ba98:7654:3210

Для удобства записи применяются следующие правила:

- **Сокращение ведущих нулей**: В каждой группе можно опускать ведущие нули.
  Пример: 1080:0000:0000:0000:0008:0800:200c:417a → 1080:0:0:0:8:800:200c:417a.
- **Сокращение последовательных нулей с помощью** ::: Если в адресе есть последовательные группы, состоящие только из нулей, их можно заменить двойным двоеточием (::), но это сокращение можно использовать только один раз.
  Пример: 1080:0:0:0:8:800:200c:417a → 1080::8:800:200c:417a.

Таким образом, адреса IPv6 записываются в шестнадцатеричном формате с возможностью сокращения для упрощения восприятия и ввода.

48. Многоадресная передача. Понятие и механизмы. Адресация и область действия IPv4.

Многоадресная передача — это способ доставки данных сразу **нескольким получателям**, без необходимости дублировать каждый пакет для каждого получателя отдельно (как в unicast) и без отправки всем (как в broadcast). Это **оптимальная модель**, особенно при передаче потокового мультимедиа, данных в играх, обновлений и других ситуациях, когда один источник вещает группе клиентов.

Механизм multicast позволяет сетевому интерфейсу принимать только те кадры, которые адресованы **конкретной группе**, к которой он подписан.

1. На канальном уровне (Ethernet, Wi-Fi)

- **Ethernet** использует **специальные MAC-адреса**, начинающиеся с `01:00:5e`, чтобы обозначить пакеты, предназначенные группе.
- В **Wi-Fi (802.11)** возможны задержки из-за **режима энергосбережения (power-save mode)** — кадры буферизуются до пробуждения устройств.

2. На сетевом уровне (IPv4 / IPv6)

Смотреть дальше

3. На транспортном уровне

Multicast использует **UDP**, так как TCP требует установления соединения "один-к-одному". Часто используется модель "один-ко-многим", где **один отправитель** рассылает данные **множеству подписчиков**.

4. Сеансы многоадресной передачи

Сеанс определяется **адресом multicast-группы + портом**. Это позволяет, например, отделить аудио и видео потоки:

```
226.1.1.1:5000 — аудио
226.1.1.2:5000 — видео
```

Многоадресные IPv4-адреса (Multicast addresses)

IPv4-многоадресные адреса находятся в **классе D** — это адреса от **224.0.0.0 до 239.255.255.255**

Структура:

- Первые 4 бита всегда 1110, что идентифицирует адрес как multicast.
- Последние 28 бит — **идентификатор группы**.

Категории и специальные группы

Адрес	Назначение
224.0.0.0/24	**Локальные адреса** (link-local)
224.0.0.1	Все хосты (все устройства в сегменте)
224.0.0.2	Все маршрутизаторы в сети
224.0.0.9	Протокол маршрутизации RIP v2
224.0.1.1	NTP
239.0.0.0/8	**Административно ограниченные адреса**

Адреса из 224.0.0.0/24 **никогда не маршрутизируются** за пределы локальной сети. Используются для служебных целей (протоколы, маршрутизация, соседство и т. д.).

Ограничение области действия (Scope) с помощью TTL

Многоадресные IPv4-пакеты **не содержат специального поля "scope"**. Вместо этого используется значение **TTL** — оно управляет тем, **насколько далеко пакет может уйти**.

TTL	Область действия
0	Только внутри узла (loopback)
1	Только в пределах локальной сети
<32	В пределах сайта (site-local)
<64	В пределах региона (regional)
<128	В пределах континента (continental)
≤255	Глобальная область (global)

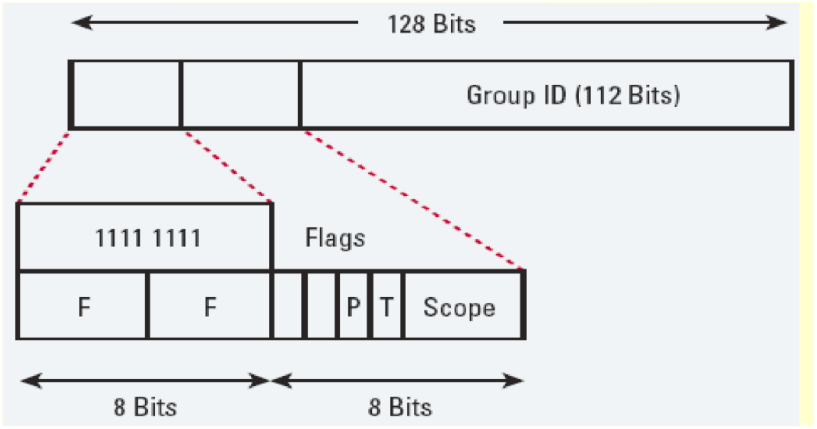
Диапазон 239.0.0.0/8 зарезервирован для **ограничения области действия по административному принципу**, а не только TTL.

49. Многоадресная передача. Понятие и механизмы. Адресация и область действия IPv6.

Часть вопроса см в вопросе 48

Все multicast-адреса в IPv6 начинаются с префикса **FF00::/8**

Общая структура IPv6 multicast-адреса



- FF — идентификатор multicast-адреса
- **Флаги (Flags)** — определяют свойства адреса (например, постоянный или временный)
- **Scope** — **область действия** (определяет, где допустима передача)
- **Group ID** — идентификатор группы

Области действия (Scope) в IPv6

Scope определяет **границу**, в пределах которой допустима доставка multicast-пакета:

Scope (десятичное)	Область действия	Применение
0x1	Node-local (локально на узле)	Внутреннее взаимодействие на хосте
0x2	Link-local (в пределах сети)	Протоколы маршрутизации, NDP
0x4	Admin-local	Административно ограниченная сеть
0x5	Site-local	В пределах сайта (организации)
0x8	Organization-local	Внутри организации
0xE	Global	В глобальной сети (интернет)

Примеры специальных IPv6 multicast-адресов

Адрес	Назначение	Scope
FF02::1	Все узлы (all-nodes)	Link-local (0x2)
FF02::2	Все маршрутизаторы	Link-local (0x2)
FF01::1	Все узлы на **одном узле**	Node-local (0x1)
FF05::1:3	Все DHCP-серверы сайта	Site-local (0x5)
FF0E::101	Пример глобального multicast	Global (0xE)

Флаги (Flags)

Флаг	Назначение
T	0 — статический адрес (well-known), 1 — временный
P	1 — адрес создан на основе префикса
R	Зарезервировано
M	Зарезервировано

Присоединение к группам (Join) в IPv6

Для работы с группами используются системные вызовы и структуры:

- IPV6_JOIN_GROUP / IPV6_LEAVE_GROUP
- Структура: struct ipv6_mreq
  Содержит:
  - imr_multiaddr — адрес группы
  - imr_interface — индекс интерфейса