Содержание

1 Введение в компьютерные сети

Сила современных вычислений заключается в возможности объединения нескольких машин в распределенную систему. Сети позволяют масштабировать нагрузку, повышать надежность и обеспечивать коммуникацию между узлами. Большинство базовых протоколов, используемых сегодня, были разработаны в 1980-х годах.

Основная задача физического уровня — передача информации между двумя непосредственно соединенными машинами. Технически это реализуется путем изменения физических параметров среды передачи, например, напряжения в проводнике.

2 Физический уровень: передача сигнала

2.1 Цифровые и аналоговые сигналы

Различают два основных способа передачи данных:

- **Цифровой сигнал:** имеет фиксированное количество валидных значений (например, 0 и 1). Любые промежуточные значения интерпретируются как ближайшее валидное, что обеспечивает высокую устойчивость к шуму.
- **Аналоговый сигнал:** имеет бесконечное множество значений. Потенциально переносит больше информации, но крайне чувствителен к помехам, так как любой шум приводит к неправильному декодированию.

В современных вычислительных сетях используются цифровые сигналы. Для передачи битов фиксируются уровни напряжения (например, 1 V — логический 0, 3 V — логическая 1).

2.2 Проблема синхронизации и тактирования

Передача последовательности битов требует договоренности о временной сетке. Если отправитель меняет напряжение раз в секунду, получатель должен замерять его с тем же интервалом. Однако часы на разных устройствах имеют дрейф, что делает синхронизацию нетривиальной задачей. При передаче длинной последовательности одинаковых битов (например, 100 000 единиц) константное напряжение в проводе не позволяет получателю точно определить количество переданных битов без идеального тактового генератора.

Определение: Манчестерское кодирование

Метод физического кодирования, при котором каждый такт (тик) часов разбивается на две половины. Значение бита определяется направлением перехода напряжения в

середине такта.

рис. 1 иллюстрирует логику манчестерского кодирования (согласно IEEE 802.3):

- **Логический 0:** переход от низкого уровня к высокому (rising edge).
- **Логическая 1:** переход от высокого уровня к низкому (falling edge).

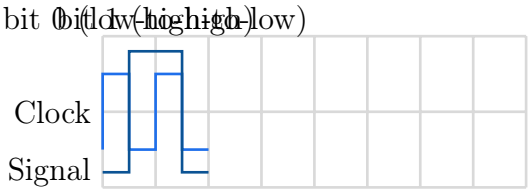

Рис. 1 – Схематичное представление манчестерского кодирования

Такой подход гарантирует изменение напряжения хотя бы раз за такт, что позволяет получателю самосинхронизироваться и восстанавливать частоту передатчика.

3 Уровень передачи данных: Ethernet

Над физическим уровнем строится уровень передачи данных (Data Link Layer), оперирующий структурированными блоками информации — **кадрами** (frames).

3.1 Структура кадра Ethernet II

Типичный кадр имеет размер от 64 до 1518 байт и включает следующие поля:

1. **Destination MAC Address (6 байт):** адрес получателя.
2. **Source MAC Address (6 байт):** адрес отправителя.
3. **EtherType (2 байта):** указывает тип протокола сетевого уровня (например, IPv4 или IPv6).
4. **Payload (46–1500 байт):** полезная нагрузка.
5. **CRC Checksum (4 байта):** контрольная сумма для проверки целостности данных.

3.2 MAC-адресация

Определение: MAC-адрес

Уникальный 48-битный идентификатор сетевого интерфейса, зашиваемый производителем на этапе производства.

MAC-адрес структурирован следующим образом:

- Первые 3 байта (*Organizationally Unique Identifier*, OUI) выдаются централизованной организацией конкретному производителю (например, Intel).
- Последние 3 байта назначаются производителем серийно.

Коммутаторы (свитчи) в локальной сети строят таблицу соответствия MAC-адресов портам. При получении кадра свитч смотрит в поле Destination MAC и пересылает данные только в нужный кабель, что обеспечивает базовую маршрутизацию на втором уровне.

4 Сетевой уровень: IP (Internet Protocol)

Для связи узлов в глобальной сети MAC-адресации недостаточно, так как она не масштабируется. Используется протокол IP, вводящий иерархическую систему адресации.

4.1 IPv4: адресация и подсети

IPv4-адрес — это 32-битное число, обычно записываемое в виде четырех октетов (от 0 до 255). Всего существует около $4 \cdot 10^9$ адресов, что значительно меньше количества современных онлайн-устройств.

Примечание

Для решения проблемы дефицита адресов используется **NAT** (*Network Address Translation*). Устройства во внутренней сети имеют частные IP, а роутер транслирует их в один публичный адрес, запоминая порты отправителей для обратной доставки ответов.

Подсети определяются префиксом (*CIDR notation*). Например, маска /23 означает, что первые 23 бита фиксированы (адрес сети), а оставшиеся 9 бит ($2^9 = 512$) доступны для хостов. Два адреса зарезервированы: адрес сети (все нули в суффиксе) и *broadcast* (все единицы).

4.2 Поле TTL (Time to Live)

В IP-пакете присутствует поле **TTL**. Оно ограничивает количество пересылок (хопов). Каждый роутер уменьшает TTL на 1. Если TTL становится равным 0, пакет уничтожается, а отправителю посылается уведомление. Это предотвращает бесконечное зацикливание пакетов при ошибках маршрутизации.

5 Транспортный уровень: TCP и UDP

Протокол IP доставляет пакеты между хостами, но не гарантирует надежность. Эти задачи решают протоколы транспортного уровня. Для идентификации конкретной программы на хосте вводятся **порты** (16-битные числа).

5.1 UDP (User Datagram Protocol)

UDP предоставляет минимальную абстракцию. Он позволяет отправлять одиночные сообщения (датаграммы) без установки соединения.

- **Плюсы:** минимальные задержки, отсутствие накладных расходов на подтверждение.
- **Минусы:** нет гарантий доставки, порядка пакетов или отсутствия дубликатов.

5.2 TCP (Transmission Control Protocol)

TCP предоставляет абстракцию надежного двустороннего канала (стрима).

- **Надежность:** каждый пакет нумеруется (*Sequence Number*). Получатель подтверждает получение (*ACK*). Если подтверждение не пришло, данные отправляются повторно.
- **Порядок:** TCP собирает байты в исходном порядке, даже если IP-пакеты пришли вразнобой.
- **Управление потоком:** протокол динамически меняет скорость отправки, чтобы не перегрузить сеть.

```
struct tcp_header {
```

```
    uint16_t source_port;
    uint16_t dest_port;
    uint32_t seq_number;
    uint32_t ack_number;
    uint16_t flags; // SYN, ACK, FIN, etc.
    uint16_t window_size;
    uint16_t checksum;
    uint16_t urgent_ptr;
};
```

Листинг 1 – Структура заголовка TCP (фрагмент)

Итоги раздела

- **Физический уровень:** цифровые сигналы и манчестерское кодирование решают проблему синхронизации.
- **Ethernet (L2):** использует MAC-адреса для доставки кадров внутри локального сегмента.
- **IP (L3):** обеспечивает глобальную маршрутизацию. TTL предотвращает циклы, а NAT экономит адреса IPv4.
- **Транспорт (L4):** UDP ориентирован на скорость, TCP — на надежность и соблюдение порядка байтов.
- **Endianness:** в сетевых протоколах принят порядок **Big-Endian**, независимо от архитектуры хоста.

6 Абстракция сокета и системный вызов socket()

В операционных системах семейства Unix сетевое взаимодействие реализовано через расширение концепции «всё есть файл». Для работы с сетью используется абстракция **сокета** — программного интерфейса, который инкапсулирует детали сетевых протоколов и предоставляет файловый дескриптор для передачи данных.

Создание сокета осуществляется системным вызовом `socket()`:

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
    perror("socket failed");
    exit(1);
}
```

Листинг 2 – Создание дескриптора сокета

Основные параметры вызова:

- **Address Family (AF):** определяет протокол адресации. `AF_INET` для IPv4 или `AF_INET6` для IPv6.
- **Type:** определяет семантику передачи. `SOCK_STREAM` соответствует потоковому протоколу TCP, `SOCK_DGRAM` — протоколу датаграмм UDP.
- **Protocol:** обычно устанавливается в 0, что позволяет системе выбрать протокол по умолчанию для заданной пары (Family, Type).

7 Реализация TCP-клиента

Процесс подключения клиента к серверу состоит из подготовки адреса и вызова `connect()`.

7.1 Адресация и порядок байт

Сетевые структуры данных требуют строгого соблюдения *Network Byte Order* (Big-endian). Для преобразования локальных данных (Host Byte Order) используются функции `htons()` (*host to network short*) и `htonl()`.

Определение: Структура sockaddr_in

Специфичная для IPv4 структура, содержащая семейство адресов (`sin_family`), порт (`sin_port`) и бинарное представление IP-адреса (`sin_addr`).

Для преобразования строкового представления IP (например, "127.0.0.1") в бинарное используется функция `inet_pton()` (*presentation to network*).

7.2 Установка соединения

Системный вызов `connect()` инициирует процедуру «трехстороннего рукопожатия» (TCP 3-way handshake).

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(80); // Network order!
inet_pton(AF_INET, "93.184.216.34", &addr.sin_addr);

if (connect(fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
    perror("connect failed");
}
```

Листинг 3 – Подключение клиента к серверу

8 Реализация TCP-сервера

В отличие от клиента, сервер должен зарезервировать порт в системе и ожидать входящих подключений.

8.1 Жизненный цикл серверного сокета

1. **bind():** связывает сокет с конкретным IP-адресом и портом. Использование адреса `INADDR_ANY` (0.0.0.0) позволяет принимать пакеты на все сетевые интерфейсы машины.
2. **listen():** переводит сокет в пассивный режим ожидания. Параметр *backlog* определяет максимальный размер очереди необработанных соединений в ядре.
3. **accept():** извлекает первое соединение из очереди. Этот вызов **блокирует** поток исполнения до появления клиента и возвращает **новый** файловый дескриптор, выделенный специально для обмена данными с этим клиентом. Слушающий дескриптор при этом остается свободным для приема новых вызовов.

Примечание

Важно: `accept()` возвращает адрес подключившегося клиента. Для хранения адреса произвольного протокола (IPv4 или IPv6) рекомендуется использовать структуру `struct sockaddr_storage`, которая имеет достаточный размер для любого семейства.

9 Тонкости эксплуатации и обработки ошибок

9.1 Проблема TIME_WAIT и опция SO_REUSEADDR

После закрытия TCP-соединения со стороны сервера, порт переходит в состояние `TIME_WAIT` (обычно на 1-4 минуты). Это защитный механизм, предотвращающий попадание «запоздавших» пакетов старого соединения в новое. Однако это мешает немедленному перезапуску сервера на том же порту.

Для обхода этого ограничения используется опция сокета:

```
int opt = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

9.2 Сигнал SIGPIPE и MSG_NOSIGNAL

Если одна сторона закрыла соединение, а вторая пытается выполнить запись в сокет, ядро ОС по умолчанию посылает процессу сигнал `SIGPIPE`. Реакция по умолчанию на этот сигнал — завершение процесса. В серверном ПО это недопустимо.

Для предотвращения этого эффекта запись следует выполнять через `send()` с флагом `MSG_NOSIGNAL`:

```
ssize_t sent = send(client_fd, buf, len, MSG_NOSIGNAL);
if (sent < 0 && errno == EPIPE) {
    // Connection closed, handle gracefully
}
```

10 Визуализация взаимодействия

На рис. 2 представлен полный цикл взаимодействия через Sockets API.

Итоги раздела

- Сокет — это файл, но с дополнительной логикой адресации и состояния соединения.
- `bind()` и `listen()` переводят сокет в режим приема подключений.
- `accept()` порождает новый FD для каждой сессии.
- Использование `htons()` обязательно для порта в структурах `sockaddr`.
- Флаг `MSG_NOSIGNAL` критичен для стабильности сервера при разрывах соединений.

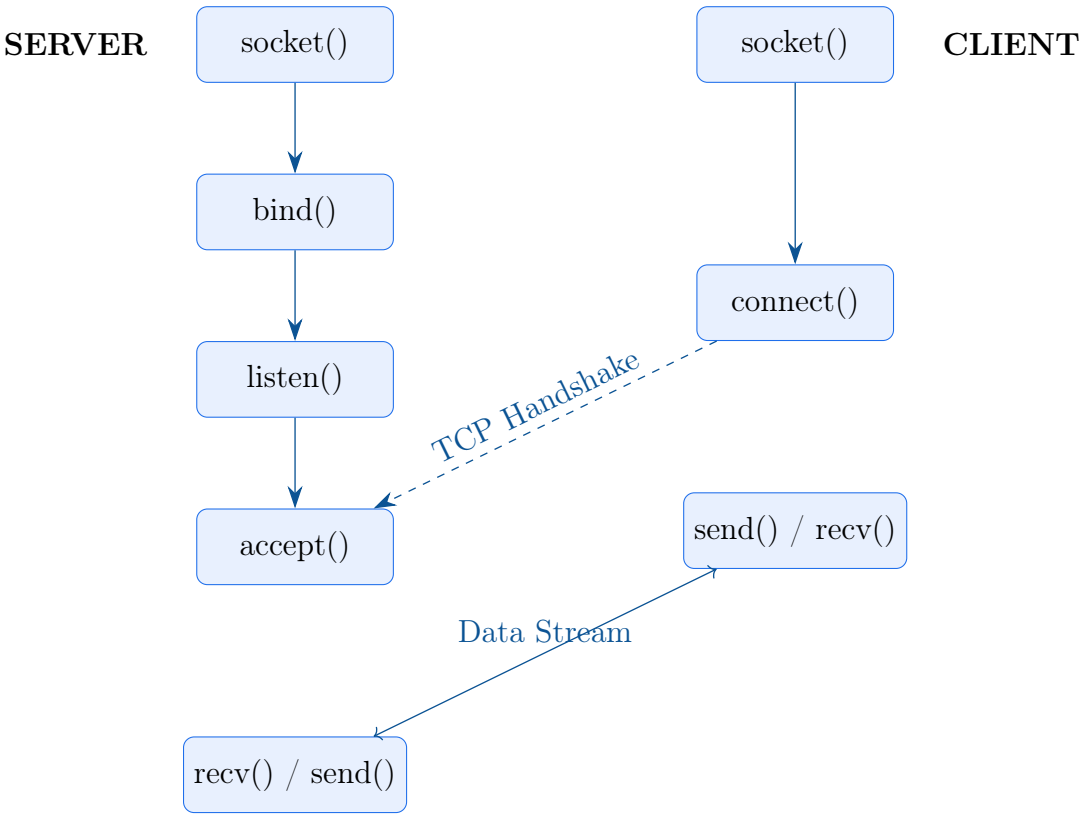

Рис. 2 – Жизненный цикл сетевого взаимодействия

1113 Сетевой стек: от физического уровня до транспортногоВысокопроизводительный ввод-вывод: мультиплексирование и epoll 21.12.202523.12.2025

11 Проблема масштабируемости: Thread-per-connection

Традиционный подход к написанию серверов заключается в выделении отдельного потока исполнения (*thread*) на каждое клиентское соединение. Несмотря на простоту реализации, данная модель сталкивается с жесткими архитектурными ограничениями при росте нагрузки:

- **Расход памяти:** каждому потоку требуется собственный стек. По умолчанию в Linux это может быть до 8 MB, хотя фактически аллоцируется меньше, при тысячах соединений затраты оперативной памяти становятся критическими.
- **Context Switching:** планировщик ОС вынужден постоянно переключать контекст между тысячами потоков. Затраты на сохранение/восстановление регистров процессора и сброс кэшей (L1/L2) начинают превышать время полезной работы сервера.
- **Блокировки:** потоки большую часть времени проводят в состоянии ожидания (*blocked*) на системных вызовах `read()` или `write()`, что неэффективно использует ресурсы CPU.

12 Неблокирующий ввод-вывод и ошибка EAGAIN

Для решения проблемы простоя потоков используется перевод файловых дескрипторов в неблокирующий режим (*non-blocking mode*).

Определение: Неблокирующий сокет

Режим работы сокета, при котором системные вызовы `read` и `write` возвращают управление немедленно. Если данных для чтения нет или буфер записи полон, вызов возвращает ошибку `EAGAIN` или `EWOULDBLOCK`.

Перевод дескриптора в этот режим осуществляется через `fcntl()`:

```
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

Примечание

Простой цикл опроса (*busy-waiting*) неблокирующих сокетов приведет к 100 % загрузке CPU бесполезной работой. Необходим механизм, позволяющий ядру ОС уведомлять процесс о готовности конкретных дескрипторов.

13 Эволюция мультиплексирования: от select до epoll

Мультиплексирование позволяет одному потоку наблюдать за состоянием множества дескрипторов одновременно.

1. **select():** исторически первый интерфейс. Ограничен 1024 дескрипторами и требует передачи битовых масок из User mode в Kernel mode при каждом вызове. Сложность — $O(N)$.
2. **poll():** снимает ограничение на количество дескрипторов, но по-прежнему требует линейного сканирования массива событий в ядре и в приложении. Сложность — $O(N)$.

3. **epoll():** современный механизм Linux. Состояние наблюдаемых дескрипторов хранится внутри ядра. Приложение получает только список дескрипторов, на которых **реально** произошло событие. Сложность — $O(1)$.

14 Системный интерфейс epoll

Работа с `epoll` строится вокруг специального дескриптора «интересов», создаваемого вызовом `epoll_create1()`.

14.1 Управление интересами: epoll_ctl()

Этот вызов позволяет добавлять (`EPOLL_CTL_ADD`), удалять или изменять условия наблюдения за сокетами.

- `EPOLLIN`: данные доступны для чтения.
- `EPOLLOUT`: в буфере есть место для записи.
- `EPOLLET`: Edge-Triggered режим (уведомление только при изменении состояния).

14.2 Ожидание событий: epoll_wait()

Процесс блокируется на этом вызове, пока не произойдет хотя бы одно событие или не истечет таймаут.

```
struct epoll_event events[MAX_EVENTS];
int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);

for (int n = 0; n < nfds; ++n) {
    if (events[n].data.fd == listen_fd) {
        // accept() new connection and add to epoll
    } else {
        // handle_client(events[n].data.fd);
    }
}
```

Листинг 4 – Пример обработки событий в Event Loop

15 Архитектура Event Loop

Использование `epoll` порождает паттерн «Петля событий» (*Event Loop*), лежащий в основе архитектур Node.js, Nginx и высокопроизводительных Python-фреймворков.

16 Инверсия управления и машины состояний

Переход к асинхронному вводу-выводу требует изменения логики приложения. Вместо линейного выполнения кода («прочитать данные, затем обработать, затем отправить»), программист должен разбивать логику на цепочки колбэков или использовать **корутины** (*coroutines*).

Ядро ОС уведомляет нас, что сокет готов к записи, но мы можем иметь лишь часть данных. Нам необходимо сохранять состояние каждого соединения (*state machine*) в пользовательском пространстве, так как стек вызовов больше не является хранилищем состояния сессии.

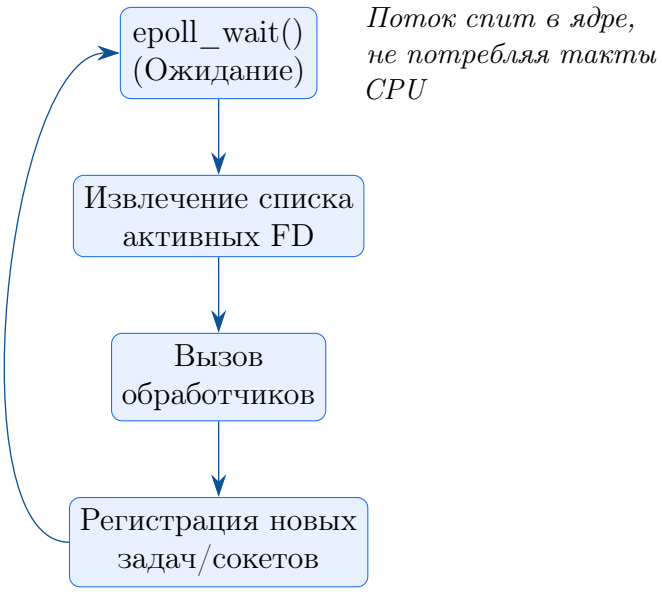

Рис. 3 – Схематичное представление Event Loop

Итоги раздела

- Модель `epoll` позволяет эффективно обрабатывать десятки тысяч соединений в одном потоке.
- Ключевое преимущество `epoll` — ядро ОС берет на себя мониторинг дескрипторов, возвращая приложению готовый список событий.
- Обязательным условием работы мультиплексирования является использование неблокирующего режима (`O_NONBLOCK`).
- Современные протоколы (HTTP/3) и высоконагруженные системы почти полностью отказались от блокирующего ввода-вывода в пользу событийных моделей.

17 Служба доменных имен (DNS)

Протоколы сетевого (IP) и транспортного (TCP/UDP) уровней оперируют исключительно числовыми идентификаторами — IP-адресами и портами. Для обеспечения удобства человеческого взаимодействия используется иерархическая распределенная система **DNS** (*Domain Name System*).

Определение: DNS

Сетевой протокол прикладного уровня, предназначенный для трансляции человекочитаемых доменных имен (например, `google.com`) в машиночитаемые IP-адреса.

Процесс разрешения имени (*resolution*) обычно происходит прозрачно для программиста через библиотечные вызовы ОС. Основным современным интерфейсом для этого является функция `getaddrinfo()`. Она заменяет устаревшую `gethostbyname()`, так как поддерживает IPv6 и позволяет фильтровать результаты по типу сокета.

```
struct addrinfo hints, *res;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; // IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM;
```

```
5
6  if (getaddrinfo("google.com", "80", &hints, &res) == 0) {
7      struct addrinfo *p = res;
8      while(p) {
9          p = p->ai_next;
10     }
11     freeaddrinfo(res);
12 }
```

Листинг 5 – Разрешение доменного имени через getaddrinfo

18 Маршрутизация на прикладном уровне (L7 Proxy)

Современные веб-технологии позволяют содержать сотни различных сервисов на одном физическом хосте с одним IP-адресом. Поскольку протоколы L3 и L4 ничего не знают о доменах, демультиплексирование трафика происходит на **прикладном уровне** (L7).

В протоколе HTTP клиент обязан передавать заголовок `Host`. Прокси-сервер (например, Nginx или HAProxy) принимает TCP-соединение на 80-м или 443-м порту, анализирует текстовый заголовок запроса и перенаправляет трафик соответствующему внутреннему сервису.

Примечание

Именно благодаря L7-маршрутизации возможен виртуальный хостинг. Без этого механизма каждому доменному имени требовался бы уникальный публичный IPv4-адрес.

19 Эталонная модель OSI

Для структурирования сетевых технологий используется семиуровневая модель взаимодействия открытых систем (*Open Systems Interconnection*).

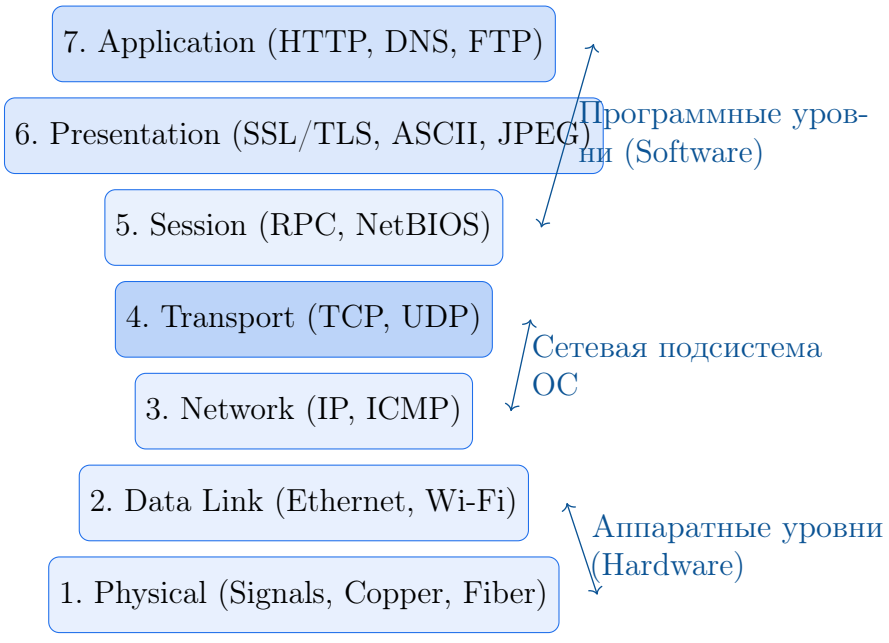

Рис. 4 – Семиуровневая модель OSI

Важно понимать, что в современном стеке протоколов TCP/IP уровни 5 (Session) и 6 (Presentation) обычно не выделяются в отдельные сущности и реализуются либо внутри прикладного протокола, либо библиотеками типа OpenSSL.

20 Эволюция протоколов: HTTP/3 и QUIC

Традиционно надежная передача данных (HTTP/1.1 и HTTP/2) строилась поверх TCP. Однако TCP обладает рядом фундаментальных недостатков: задержки при установке TLS-соединения и проблема «блокировки начала очереди» (*Head-of-line blocking*).

Примечание

В HTTP/3 произошел архитектурный сдвиг: вместо TCP используется протокол **QUIC**, работающий поверх **UDP**. Надежность и шифрование реализуются не на уровне ядра ОС, а на прикладном уровне, что позволяет быстрее устанавливать соединение и эффективнее обрабатывать потери пакетов.

21 Заключение курса

Изучение сетевого стека замыкает цикл понимания архитектуры вычислительной системы. Мы прошли путь от электрических импульсов и манчестерского кодирования до сложных абстракций асинхронного ввода-вывода и распределенных прикладных протоколов. Современная ОС — это в первую очередь коммуникационная среда, обеспечивающая взаимодействие процессов через строго определенные уровни абстракции.

Итоги раздела

- DNS — критическая инфраструктура, связывающая человеческую логику имен с машинной логикой IP-адресов.
- Модель OSI — фундаментальный справочный каркас, позволяющий инженерам говорить на одном языке.
- Маршрутизация может происходить на разных уровнях: L2 (MAC), L3 (IP), L4 (Port) и L7 (HTTP Host).
- Будущее сетей движется в сторону выноса логики управления потоком из ядра ОС в User-space (QUIC, DPDK).