Network: Часть 1. База.

Published: 15.06.2026 12:28 | Updated: 15.06.2026 15:16

Сокеты и сетевой стек: инженерное понимание того, как приложения общаются по сети.

Почему понимание сетей начинается не с TCP/IP

Когда инженеры начинают изучать сети, они обычно сталкиваются с абстракциями высокого уровня: HTTP-запросами, REST API, WebSocket-соединениями или клиентами баз данных. Всё это создает иллюзию, что работа по сети сводится к отправке запросов и получению ответов.
Однако в реальности между вашим приложением и сетью существует целый слой механизмов операционной системы. Именно здесь находятся сокеты — фундаментальный интерфейс, через который программы получают доступ к сетевому стеку.
Понимание сокетов важно не только для системных программистов. Оно помогает объяснить:

- почему возникают таймауты;
- откуда берутся блокировки в приложениях;
- как работают асинхронные серверы;
- почему высоконагруженные системы проектируются определенным образом;
- что на самом деле происходит при вызове `send()` или `recv()`.

Чтобы мыслить как инженер, важно понимать не только **что происходит**, но и **где именно это происходит**.

Сокет — это не соединение

Одна из самых распространенных ошибок заключается в том, что сокет воспринимают как сетевое соединение.
На самом деле это не совсем так.
Сокет — это программный интерфейс, предоставляемый операционной системой для взаимодействия приложения с сетевым стеком.
Другими словами, сокет представляет собой точку входа приложения в сетевую подсистему операционной системы.
Важно разделять два понятия:

Понятие	**Что означает**
Socket	Интерфейс доступа к сетевому стеку
TCP Connection	Логическое соединение между двумя узлами

TCP-соединение существует внутри TCP-стека операционной системы и представляет собой набор состояний и параметров протокола.
Сокет же является способом управления этим соединением.
Можно сказать так:

> Сокет — это дверная ручка, а TCP-соединение — помещение за этой дверью.

Сокет в Unix: сеть как обычный ввод-вывод

Одной из самых элегантных идей Unix является концепция:

> «Всё является файлом».

Это не означает, что сокет хранится на диске как файл. Речь идет о единой модели ввода-вывода.
В Unix-подобных системах сокет представлен **файловым дескриптором** (File Descriptor, FD).
Файловый дескриптор — это целочисленный идентификатор открытого ресурса, которым управляет ядро операционной системы.
Например:

```
import socket


```

```
s = socket.socket()
print(s.fileno())
```

Вывод:

```
3
```

Число 3 — это файловый дескриптор.
Это означает, что многие операции ввода-вывода используют единый интерфейс:

```
read()
write()
close()
```

или их сетевые аналоги:

```
recv()
send()
close()
```

Главный вывод здесь заключается в следующем:

> Для Unix сеть является еще одним видом ввода-вывода.

Именно поэтому механизмы работы с файлами и сетью во многом похожи.

Что происходит при создании сокета

Создание сокета выглядит очень просто:

```
socket()
```

Но за этим вызовом скрывается значительный объем работы операционной системы.
При вызове `socket()` происходит следующее:

1. Выполняется системный вызов.

2. Процесс передает управление ядру.
3. Ядро создает внутренний объект сокета.
4. Выделяются необходимые структуры данных.
5. Инициализируются буферы отправки и приема.
6. Создается запись в таблице файловых дескрипторов процесса.
7. Приложению возвращается файловый дескриптор.

Схематически процесс выглядит так:

```
Приложение
      ↓
socket()
      ↓
Системный вызов
      ↓
Ядро ОС
      ↓
Создание socket object
      ↓
Выделение буферов
      ↓
Возврат File Descriptor
```

После этого приложение получает возможность взаимодействовать с сетевым стеком через созданный дескриптор.
Важно понимать:

> Сам сокет существует внутри ядра. Приложение получает лишь ссылку на этот объект.

User Space и Kernel Space

Современные операционные системы разделяют пространство выполнения на две области.

User Space

Здесь выполняются пользовательские приложения:

- браузеры;

- базы данных;
- Python-приложения;
- веб-серверы.

Программы в пользовательском пространстве имеют ограниченные привилегии и не могут напрямую обращаться к оборудованию.

Kernel Space

В пространстве ядра находятся:

- драйверы устройств;
- планировщик процессов;
- файловые системы;
- сетевой стек;
- управление памятью.

Ядро обладает полным доступом к аппаратным ресурсам компьютера.

Системные вызовы: мост между приложением и ОС

Когда приложение вызывает:

```
sock.send(data)
```

происходит не обычный вызов функции.
Выполняется системный вызов (system call).
Процесс выглядит следующим образом:

```
Python приложение
        ↓
send()
        ↓
Переход User Mode → Kernel Mode
        ↓
Обработка сетевым стеком ОС
        ↓
Возврат управления приложению
```

Этот переход требует:

- переключения режима процессора;
- сохранения контекста выполнения;
- проверки прав доступа;
- передачи управления ядру.

Поэтому системные вызовы являются относительно дорогими операциями. Именно по этой причине высокопроизводительные приложения стремятся:

- уменьшать количество системных вызовов;
- отправлять данные пакетами;
- использовать механизмы асинхронного ввода-вывода.

TCP-соединение как машина состояний

TCP представляет собой протокол с сохранением состояния (stateful protocol). Это означает, что каждое соединение обладает собственным набором состояний. Основные состояния TCP:

```
CLOSED
LISTEN
SYN_SENT
SYN_RECEIVED
ESTABLISHED
FIN_WAIT_1
FIN_WAIT_2
TIME_WAIT
CLOSE_WAIT
LAST_ACK
```

Переход между состояниями определяется событиями внутри протокола. Например:

```
Клиент                      Сервер
   |                           |
   | ------ SYN ------------>  |
   |                            |
   | <---- SYN + ACK -------    |
   |                            |
```

```
   | ------ ACK ----------->   |
   |                           |
 ESTABLISHED              ESTABLISHED
```

После завершения трехэтапного рукопожатия соединение считается установленным.
Важно понимать:

> TCP-соединение — это не просто канал передачи данных, а конечный автомат со строго определенными правилами перехода между состояниями.

Жизненный цикл TCP-соединения

Работу TCP можно условно разделить на три этапа.

1. Установка соединения

Используется механизм **Three-Way Handshake**.

```
Client → SYN → Server
Server → SYN-ACK → Client
Client → ACK → Server
```

На этом этапе стороны согласуют:

- начальные номера последовательностей;
- возможность установления связи;
- параметры соединения.

2. Передача данных

После установления соединения TCP обеспечивает:

- надежную доставку;
- контроль порядка пакетов;
- повторную передачу потерянных сегментов;
- подтверждение доставки (ACK);
- управление потоком передачи.

3. Завершение соединения

Закрытие TCP-соединения также требует обмена служебными пакетами.
Обычно используется схема:

```
FIN
ACK
FIN
ACK
```

Это гарантирует корректное завершение передачи данных и освобождение ресурсов.

Почему это важно

Даже если вы работаете исключительно с высокоуровневыми библиотеками, понимание этих механизмов помогает объяснить многие реальные проблемы:

- зависание сетевых операций;
- большое количество соединений в состоянии `TIME_WAIT`;
- таймауты;
- проблемы масштабирования;
- неожиданное поведение асинхронных приложений.

Высокоуровневые инструменты скрывают детали реализации, но не отменяют их существования.
Чем лучше инженер понимает фундаментальные принципы работы сети, тем эффективнее он способен диагностировать и решать проблемы в реальных системах.

Буферы сокетов: почему `send()` не означает отправку данных

Многие инженеры интуитивно воспринимают вызов:

```
sock.send(data)
```

как действие:

> «данные немедленно отправились по сети».

На самом деле это не так.
Между приложением и физической сетью существует несколько промежуточных слоёв, и один из важнейших элементов этой цепочки — **буферы сокета**.

Буферизация как фундамент сетевого взаимодействия

У каждого TCP-сокета существуют два основных буфера:

- **Send Buffer** — буфер отправки;
- **Receive Buffer** — буфер приёма.

Схематично это выглядит так:

```
Приложение
     ↓
Send Buffer
     ↓
TCP/IP Stack
     ↓
Сеть
     ↓
TCP/IP Stack
     ↓
Receive Buffer
     ↓
Приложение
```

Буфер отправки (Send Buffer)

Когда приложение вызывает:

```
sock.send(data)
```

обычно происходит следующее:

1. Данные копируются из пользовательского пространства в буфер ядра.

2. Функция возвращает управление приложению.
3. TCP самостоятельно определяет:
   -- когда отправить сегменты;
   -- какого размера они будут;
   -- требуется ли повторная передача.

Это означает:

> Успешный вызов `send()` означает только то, что данные были приняты ядром ОС.

Но это **не гарантирует**, что:

- данные уже покинули компьютер;
- удалённая сторона их получила;
- удалённая сторона их обработала.

Буфер приёма (Receive Buffer)

Когда сетевой пакет приходит на машину:

1. Сетевая карта получает данные.
2. Ядро обрабатывает пакет.
3. Данные помещаются в Receive Buffer.
4. Приложение вызывает `recv()` и считывает их.

Важно понимать:

> TCP может получить данные раньше, чем приложение будет готово их обработать.

Буферизация позволяет временно компенсировать разницу скоростей между сетью и приложением.

Почему буферы необходимы

Без буферов возникли бы серьёзные проблемы.
Например:

```
Сеть → очень быстро
Приложение → временно занято
```

Если бы промежуточного хранилища не существовало, пакеты пришлось бы немедленно отбрасывать.
Буферы позволяют:

- сглаживать кратковременные всплески нагрузки;
- компенсировать разницу скоростей работы компонентов;
- уменьшать количество системных вызовов;
- повышать общую эффективность передачи данных.

Flow Control: защита от перегрузки получателя

Представим ситуацию:

```
Отправитель → 1 Гбит/с
Получатель → обрабатывает 100 Мбит/с
```

Очевидно, что рано или поздно буфер получателя переполнится.
TCP решает эту проблему с помощью механизма **Flow Control**.

Как работает управление потоком

Получатель сообщает отправителю:

> «Сейчас я могу принять ещё N байт данных».

Эта информация передаётся через параметр **Receive Window (rwnd)**.
Если буфер начинает заполняться:

```
Receive Window ↓
```

отправитель автоматически снижает скорость передачи.
Если буфер полностью заполнен:

```
Window = 0
```

отправитель временно прекращает отправку.
Таким образом TCP предотвращает потерю данных из-за перегрузки принимающей стороны.

Backpressure: обратное давление в системах

Flow Control является частным случаем более общей концепции — **Backpressure**.
Идея проста:

> Медленные компоненты системы должны иметь возможность замедлять быстрые.

Это встречается не только в сетях:

- TCP-соединения;
- очереди сообщений;
- реактивные системы;
- стриминговые платформы;
- асинхронные пайплайны обработки данных.

Без механизма обратного давления системы быстро становятся нестабильными.

Блокирующие сокеты

По умолчанию большинство сокетов работают в **blocking mode**.
Это означает:

```
data = sock.recv(1024)
```

будет ждать до тех пор, пока:

- не появятся данные;
- соединение не будет закрыто;
- не возникнет ошибка.

Что происходит внутри ОС

Если данных нет:

```
recv()
    ↓
Ядро проверяет Receive Buffer
    ↓
Буфер пуст
    ↓
```

```
Поток переводится в состояние ожидания
    ↓
CPU переключается на выполнение других задач
```

Когда данные появляются:

```
Пакет поступил
    ↓
Поток пробуждается
    ↓
recv() возвращает данные
```

Проблема блокирующего подхода

Представим сервер:

```
1 поток → 1 соединение
```

Если одновременно подключатся:

```
10 000 клиентов
```

потребуется:

```
10 000 потоков
```

Это приводит к серьёзным накладным расходам:

- увеличение потребления памяти;
- затраты на переключение контекста;
- снижение производительности.

Такой подход плохо масштабируется.

Неблокирующие сокеты

В неблокирующем режиме:

```
sock.setblocking(False)
```

вызов:

```
sock.recv(1024)
```

никогда не будет ждать.
Возможны два сценария:

Данные доступны

```
recv()
    ↓
Данные возвращаются сразу
```

Данных нет

Возвращается ошибка:

```
BlockingIOError
```

или соответствующий код ошибки операционной системы.

Возникает новая проблема

Если постоянно выполнять:

```
while True:
    sock.recv(...)
```

получится так называемый **busy waiting**.
Процессор будет бесконечно проверять наличие данных.
Это приведёт к бесполезной загрузке CPU.
Требуется другой подход.

Multiplexing: обслуживание множества соединений

Основная идея заключается в следующем:

> Один поток должен уметь ожидать события сразу на множестве сокетов.

Именно для этого существуют механизмы мультиплексирования.

Select: первое поколение

Механизм:

```
select()
```

позволяет передать ядру список сокетов:

```
"Сообщи мне, какие из них готовы к работе"
```

Ядро возвращает список:

- готовых к чтению;
- готовых к записи;
- завершившихся ошибкой.

Ограничения select

Основные недостатки:

- ограничение на количество дескрипторов;
- необходимость каждый раз передавать весь список сокетов;
- линейная сложность проверки.

Для небольших систем этого достаточно.
Для highload-приложений — нет.

Poll: улучшенная версия

`poll()` устраняет некоторые ограничения `select()`.

Преимущества:

- отсутствие жёсткого ограничения на количество сокетов;
- более удобный интерфейс.

Однако остаётся проблема:

```
O(n)
```

Операционная система всё ещё вынуждена проверять все дескрипторы.

Epoll: основа современных высоконагруженных серверов

Linux предлагает механизм:

```
epoll
```

Основная идея:

> Не спрашивать постоянно, какие сокеты готовы.

Вместо этого:

> ОС сама сообщает об изменениях состояния.

Как работает epoll

Регистрация интересующих событий

```
epoll_ctl()
```

Приложение сообщает:

```
"Меня интересуют эти сокеты"
```

Ожидание событий

```
epoll_wait()
```

Поток засыпает.

Возникновение события

Например:

```
Поступили данные
```

Ядро помещает событие во внутреннюю очередь.

Пробуждение приложения

`epoll_wait()` возвращает:

```
Только активные сокеты
```

Почему epoll масштабируется лучше

Если имеется:

```
100 000 соединений
```

и активны только:

```
200
```

то:

select/poll

Проверяют:

```
100 000 дескрипторов
```

epoll

Возвращает:

```
200 активных дескрипторов
```

Это существенно снижает нагрузку на систему.

Event Loop: фундамент асинхронного программирования

Большинство современных серверов используют событийную модель.
Принцип работы:

```
while True:
    события = epoll_wait()

    for событие in события:
        обработать()
```

Именно на этой идее построены:

- asyncio;
- uvloop;
- Node.js;
- Nginx;
- FastAPI;
- Tornado.

Почему асинхронность работает

Распространённое заблуждение:

> Асинхронность делает код быстрее.

На самом деле:

> Асинхронность уменьшает время простоя.

Большая часть сетевых приложений тратит время на ожидание:

- данных от клиента;
- ответа базы данных;
- ответа внешнего API.

Event Loop позволяет эффективно использовать это время ожидания.

Ментальная модель современных серверов

Традиционный подход:

```
Клиент
 ↓
Поток
 ↓
Блокировка
```

Современный подход:

```
Клиенты
   ↓
epoll
   ↓
Event Loop
   ↓
Небольшое количество потоков
```

Что происходит после вызова `send()`: путь данных внутри системы

С точки зрения инженера отправка данных выглядит предельно просто:

```
sock.send(data)
```

Однако за этим вызовом скрывается целая цепочка взаимодействий между приложением, операционной системой, памятью и сетевым оборудованием. Чтобы понимать производительность сетевых приложений, необходимо видеть весь путь данных.
Упрощённо он выглядит следующим образом:

```
Приложение
    ↓
User Space
    ↓
Системный вызов
    ↓
Kernel Space
    ↓
TCP/IP Stack
    ↓
Сетевая карта (NIC)
    ↓
Физическая сеть
```

Системные вызовы и переход в ядро

Когда приложение вызывает:

```
send()
recv()
connect()
accept()
```

происходит не обычный вызов функции.
Выполняется **системный вызов** (system call).
Это означает:

```
User Mode
    ↓
Kernel Mode
    ↓
Обработка запроса ядром
```

```
    ↓
Возврат в User Mode
```

Почему системные вызовы дороги

Во время перехода в режим ядра процессору необходимо:

- сохранить текущее состояние выполнения;
- переключить уровень привилегий;
- передать управление ядру;
- выполнить необходимую обработку;
- восстановить пользовательский контекст.

Даже если операция выполняется быстро, стоимость переключения режимов остаётся.
Поэтому высокопроизводительные системы стремятся:

- уменьшать количество системных вызовов;
- выполнять операции пакетно;
- избегать лишних переключений между режимами работы процессора.

Копирование данных: скрытая стоимость сетевых операций

Многие считают, что основная стоимость сетевого взаимодействия связана исключительно с передачей данных по сети.
На практике значительную часть времени могут занимать операции копирования памяти.
Рассмотрим типичный сценарий отправки данных.

Этап 1. Данные находятся в пользовательском пространстве

Например:

```
response = b"Hello World"
sock.send(response)
```

Переменная `response` располагается в памяти процесса.

```
User Space RAM
```

Этап 2. Копирование в буфер ядра

При вызове `send()` происходит копирование:

```
User Space RAM
        ↓
Kernel Send Buffer
```

После этого функция может вернуть управление приложению.

Этап 3. Работа TCP-стека

Ядро выполняет:

- сегментацию данных;
- формирование TCP-заголовков;
- формирование IP-пакетов;
- управление окнами передачи;
- контроль повторных отправок.

Этап 4. Передача сетевой карте

Далее данные передаются драйверу сетевого устройства.

```
TCP/IP Stack
      ↓
NIC Driver
```

Этап 5. Отправка в сеть

Сетевая карта отправляет данные через физический канал связи.

```
NIC
 ↓
Ethernet/Wi-Fi
```

```
 ↓
Сеть
```

Почему копирование становится проблемой

Для небольших объёмов данных это практически незаметно.
Однако в высоконагруженных системах ситуация меняется.
Например:

- веб-сервер передаёт большие файлы;
- CDN обслуживает миллионы запросов;
- прокси пересылает гигабайты данных.

В таких случаях оказывается, что:

> Копирование памяти может стоить дороже самой передачи данных.

Именно поэтому инженеры стремятся уменьшить количество копирований.

Zero-Copy: уменьшение количества копирований

Zero-Copy — это группа техник, позволяющих минимизировать перемещение данных между различными областями памяти.
Важно понимать:

> Zero-Copy не всегда означает полное отсутствие копирования.

Чаще речь идёт о сокращении количества промежуточных копий.

Обычная схема передачи файла

```
Диск
 ↓
Kernel Buffer
 ↓
User Space
 ↓
Kernel Buffer
 ↓
NIC
```

Происходит несколько копирований.

Использование `sendfile()`

Системный вызов:

```
sendfile()
```

позволяет передавать данные напрямую:

```
Диск
 ↓
Kernel Buffer
 ↓
NIC
```

Приложение не участвует в перемещении данных.

Почему это важно

Преимущества:

- меньше загрузка CPU;
- меньше операций копирования;
- ниже задержки;
- выше пропускная способность системы.

Именно поэтому современные веб-серверы активно используют Zero-Copy техники.

RAM: настоящий центр сетевого взаимодействия

Когда говорят о сетях, обычно представляют:

- кабели;
- маршрутизаторы;
- пакеты;
- протоколы.

Однако большинство операций происходит вовсе не в сети.
Они происходят в **оперативной памяти (RAM)**.

Что такое RAM

RAM (Random Access Memory) — оперативная память компьютера, в которой располагаются:

- код программ;
- данные приложений;
- структуры ядра;
- сетевые буферы;
- очереди пакетов.

Проще говоря:

> RAM является рабочей областью всей системы.

CPU работает именно с RAM

Процессор практически никогда не взаимодействует напрямую:

- с дисками;
- с сетевыми устройствами.

Его работа выглядит следующим образом:

```
RAM
 ↓
CPU
 ↓
RAM
```

Это означает:

> Любая обработка сетевых данных неизбежно проходит через оперативную память.

Иерархия памяти

Скорость различных уровней памяти сильно отличается.

```
Регистры CPU
     ↓
L1 Cache
     ↓
L2 Cache
     ↓
L3 Cache
     ↓
RAM
     ↓
SSD/HDD
```

Чем выше уровень:

- тем быстрее доступ;
- тем меньше объём.

Почему кэш настолько важен

Современные процессоры значительно быстрее оперативной памяти.
Без кэшей CPU большую часть времени простаивал бы в ожидании данных.
Поэтому используются:

- **L1 Cache** — самый быстрый;
- **L2 Cache**;
- **L3 Cache** — общий для нескольких ядер.

Принцип локальности

Работа кэшей основана на двух идеях.

Temporal Locality

Если данные использовались недавно:

> велика вероятность, что они понадобятся снова.

Spatial Locality

Если используются данные по определённому адресу:

> вероятно, скоро понадобятся соседние области памяти.

Виртуальная память: иллюзия непрерывного пространства

Процессы не работают напрямую с физической памятью.
Каждый процесс получает собственное виртуальное адресное пространство.
Например:

```
Процесс A → 0x1000
Процесс B → 0x1000
```

Оба процесса видят одинаковые адреса.
Но физически они могут находиться в совершенно разных областях RAM.

MMU: перевод адресов

За преобразование адресов отвечает специальный аппаратный компонент:

```
MMU
(Memory Management Unit)
```

Он выполняет отображение:

```
Виртуальный адрес
       ↓
Физический адрес
```

Страничная организация памяти

Память разделяется на страницы.
Типичный размер страницы:

```
4 КБ
```

Существуют:

- виртуальные страницы;
- физические страницы.

Операционная система хранит таблицы соответствия между ними.

Page Fault: когда страницы нет в памяти

Если процесс обращается к отсутствующей странице:

```
Page Fault
```

ядро должно:

1. Найти данные на диске.
2. Загрузить страницу в RAM.
3. Обновить таблицы страниц.
4. Возобновить выполнение программы.

Это чрезвычайно дорогая операция.

Как сеть использует RAM

Рассмотрим получение пакета.

```
NIC
 ↓
DMA
 ↓
RAM
 ↓
Kernel Buffer
 ↓
Socket Buffer
 ↓
User Space
 ↓
Приложение
```

Вся обработка происходит через оперативную память.
Отсюда важный вывод:

> Сеть — это механизм перемещения данных между RAM разных компьютеров.

DMA: доступ к памяти без участия CPU

DMA означает:

```
Direct Memory Access
```

Этот механизм позволяет устройствам работать с памятью напрямую.

Без DMA

```
NIC
 ↓
CPU
 ↓
RAM
```

Процессор участвует в каждом копировании.

С DMA

```
NIC
 ↓
RAM
```

Сетевая карта самостоятельно записывает данные в память.
CPU подключается только на этапе обработки.

Почему DMA настолько важен

Преимущества:

- разгружает процессор;
- увеличивает пропускную способность;

- уменьшает задержки;
- позволяет эффективно работать с большими объёмами данных.

Современные сетевые системы были бы невозможны без DMA.

Прерывания и Polling

Когда приходят новые данные, необходимо уведомить процессор.
Существуют два подхода.

Interrupt-Based

Сетевая карта генерирует прерывание.

```
NIC
 ↓
Interrupt
 ↓
CPU
```

Процессор немедленно начинает обработку.

Polling

Процессор самостоятельно проверяет наличие данных.

```
CPU
 ↓
NIC ?
 ↓
CPU
 ↓
NIC ?
```

Современный подход

Сегодня обычно используется гибридная модель.
При небольшой нагрузке:

```
Interrupts
```

При высокой:

```
Polling
```

Это позволяет добиться лучшего баланса между задержками и производительностью.

Инженерное понимание производительности сети

Важно осознать что производительность сети определяется не только скоростью канала связи.
Ограничивающими факторами могут быть:

- количество системных вызовов;
- операции копирования памяти;
- промахи кэша процессора;
- page faults;
- эффективность работы DMA;
- реализация сетевого стека ОС.

Именно поэтому оптимизация высоконагруженных систем требует понимания не только протоколов, но и архитектуры операционных систем.

Почему сети кажутся сложными

Многие инженеры воспринимают сети как набор отдельных тем:

- HTTP;
- TCP;
- сокеты;
- DNS;
- маршрутизация;
- асинхронность.

Из-за этого формируется ощущение, что сети состоят из множества несвязанных технологий.
На самом деле всё значительно проще.
Большинство концепций в сетях являются разными уровнями описания одной и той же задачи:

> Как надёжно передать данные между двумя процессами, работающими на разных компьютерах.

Всё остальное — детали реализации этой идеи.

Главная мысль: данные и их передача — разные вещи

Это одна из важнейших концепций, которую должен понимать любой инженер.
Необходимо различать:

- **что передаётся**;
- **как это передаётся**.

Что передаётся

Это данные прикладного уровня:

- HTML-страницы;
- JSON-документы;
- изображения;
- файлы;
- команды API;
- сообщения чата.

Для приложения это осмысленные объекты.
Например:

```
{
  "name": "Markus",
  "role": "developer"
}
```

Или:

```
with open("report.pdf", "rb") as f:
    data = f.read()
```

Для инженера это:

- файл;
- объект;
- документ;
- структура данных.

Как передаётся

Для сети всё выглядит иначе.
Сеть не знает о существовании:

- файлов;
- JSON;
- HTTP;
- изображений.

Она работает только с последовательностями байтов.
Например:

```
01001000
01100101
01101100
01101100
01101111
```

Для сетевого стека существует только поток данных.
Именно поэтому важно понимать:

> В сети не существует файлов. Существуют только байты.

Интернет — это сеть сетей

Слово Internet происходит от:

```
Inter-network
```

или:

> «сеть сетей».

Это означает, что Интернет не является единой централизованной системой.
Он состоит из множества независимых сетей.
Например:

```
Домашний провайдер
        ↓
Магистральный провайдер
        ↓
Дата-центр
        ↓
Облачная инфраструктура
        ↓
Сервер приложения
```

Каждая из этих сетей:

- принадлежит разным организациям;
- управляется независимо;
- использует собственное оборудование.

Интернет существует потому, что эти сети согласились использовать общие протоколы взаимодействия.

Сеть по своей природе ненадёжна

Это одна из фундаментальных идей компьютерных сетей.
Интуитивно кажется:

> Если данные отправлены, они должны дойти.

В действительности всё наоборот.
Сеть изначально предполагается ненадёжной средой.

Что может происходить в реальности

Пакеты могут:

Теряться

Причины:

- перегрузка маршрутизаторов;
- ошибки передачи;
- переполнение буферов.

Приходить в неправильном порядке

Например:

```
Пакет 1
Пакет 2
Пакет 3
```

могут прийти как:

```
Пакет 2
Пакет 1
Пакет 3
```

Дублироваться

Из-за повторных передач:

```
Пакет 1
Пакет 1
Пакет 2
```

Задерживаться

Некоторые пакеты могут прибыть значительно позже остальных.

Надёжность — не свойство сети

Это очень важная идея.
Надёжность обеспечивается не сетью.
Она создаётся протоколами верхних уровней.
Например:

- TCP;
- TLS;
- прикладными механизмами повторных запросов.

Иными словами:

> Сеть представляет собой хаос, а протоколы пытаются навести в нём порядок.

Зачем нужны уровни абстракции

Современные сети чрезвычайно сложны.
Если бы каждому приложению приходилось самостоятельно:

- управлять сигналами;
- определять маршруты;
- реализовывать повторную передачу;
- обрабатывать ошибки,

разработка программ была бы практически невозможной.
Поэтому используются уровни абстракции.

Идея уровневой архитектуры

Каждый уровень решает строго определённый набор задач.
При этом он скрывает детали реализации нижележащих уровней.
Схематично это выглядит следующим образом:

```
Приложение
     ↓
Сокеты
     ↓
TCP
     ↓
IP
```

```
     ↓
Ethernet
     ↓
Физическая среда
```

Каждый слой предоставляет сервис вышестоящему.

Почему это важно инженеру

Представим обычный HTTP-запрос.

```
requests.get("https://example.com")
```

Инженер думает:

> «Отправить GET-запрос».

Однако в действительности происходит множество процессов.

Уровень приложения

Формируется HTTP-запрос.

```
GET / HTTP/1.1
Host: example.com
```

Транспортный уровень

TCP обеспечивает:

- установку соединения;
- контроль доставки;
- повторную передачу;
- управление потоком.

Сетевой уровень

IP определяет:

- каким маршрутом передавать данные;

- куда доставить пакеты.

Канальный уровень

Ethernet отвечает за:

- передачу внутри локальной сети;
- взаимодействие с сетевым оборудованием.

Физический уровень

Происходит передача сигналов:

- электрических;
- оптических;
- радиочастотных.

Абстракции позволяют скрывать сложность

Именно благодаря слоям инженер может использовать:

```
requests.get(...)
```

не задумываясь о:

- TCP-флагах;
- размере окон;
- работе DMA;
- устройстве Ethernet-кадров.

Однако при возникновении проблем именно понимание нижележащих уровней позволяет эффективно диагностировать систему.

Финальная инженерная модель сети

Теперь можно собрать всё воедино.
Когда приложение отправляет данные, происходит следующее:

```
Приложение
     ↓
Socket API
```

```
      ↓
System Call
      ↓
Kernel Space
      ↓
TCP/IP Stack
      ↓
Send Buffer
      ↓
NIC Driver
      ↓
DMA
      ↓
Сетевая карта
      ↓
Физическая сеть
      ↓
Маршрутизаторы
      ↓
Удалённая машина
      ↓
NIC
      ↓
DMA
      ↓
Kernel Buffer
      ↓
TCP/IP Stack
      ↓
Receive Buffer
      ↓
recv()
      ↓
Приложение
```

Это полная цепочка движения данных.

Что должен понимать инженер о сетях

После изучения выше написанного материала надо понимать следующие идеи.

1. Сеть — это передача байтов

Не файлов.
Не HTTP.
Не JSON.
А именно байтов.

2. Сокет — это интерфейс ОС

Сокет не является соединением.
Он предоставляет приложению доступ к сетевому стеку.

3. TCP является машиной состояний

Каждое соединение обладает:

- состояниями;
- буферами;
- механизмами управления передачей.

4. Буферизация критически важна

Операции `send()` и `recv()` работают через буферы.
Они не взаимодействуют с сетью напрямую.

5. Асинхронность связана с ожиданием

Большинство сетевых приложений ограничены не вычислениями, а ожиданием событий.

6. Производительность зависит не только от сети

Большое влияние оказывают:

- системные вызовы;
- копирование памяти;
- работа кэшей процессора;
- DMA;

- реализация сетевого стека ОС.

7. Абстракции скрывают сложность

Высокоуровневые библиотеки делают разработку удобной.
Но понимание того, что происходит под ними, отличает инженера от пользователя технологий.

Заключение

Понимание сетей начинается не с изучения протоколов и не с написания серверов.
Оно начинается с понимания того, что происходит между приложением и операционной системой.

Мы рассмотрели:

- сокеты как интерфейс доступа к сети;
- системные вызовы и взаимодействие с ядром;
- TCP и его модель состояний;
- буферы и управление потоком;
- блокирующий и неблокирующий ввод-вывод;
- `select`, `poll` и `epoll`;
- роль оперативной памяти в сетевых взаимодействиях;
- механизмы Zero-Copy и DMA;
- принципы уровневой архитектуры сетей.

Все эти темы объединяет одна идея:

> Сеть представляет собой механизм перемещения данных между процессами, работающими на разных компьютерах, а операционная система выступает посредником, обеспечивающим эту коммуникацию.

Понимание этой модели создаёт фундамент, на котором строится дальнейший путь изучения:

- модели OSI и TCP/IP;
- маршрутизации;
- DNS;
- HTTP;
- TLS;

- высоконагруженных распределённых систем.

Именно с этого момента изучение сетей перестаёт быть набором разрозненных технологий и превращается в целостную инженерную дисциплину.