

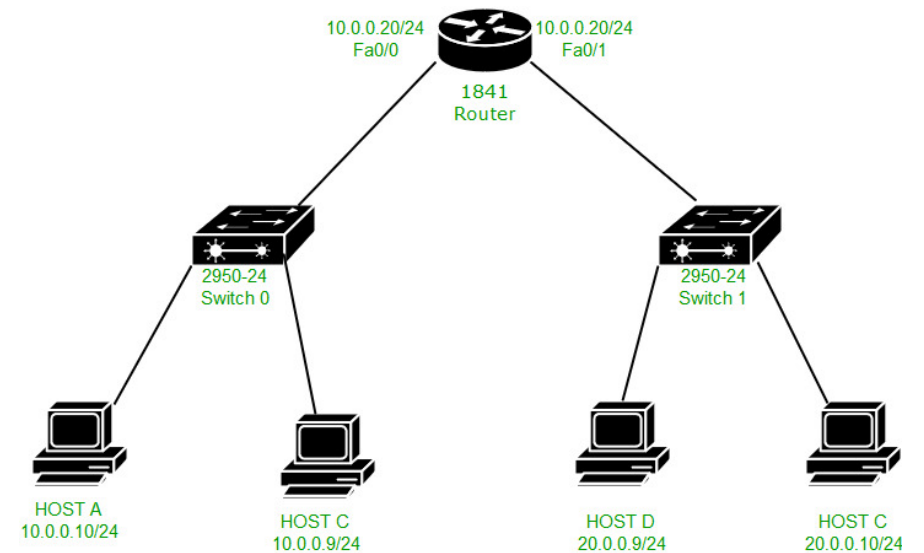
# Системное программирование

Лекция 8

Компьютерные сети

# Сеть

- **Компьютерная сеть** – множество компьютеров, связанных между собой каналами передачи информации.
- **Узел сети** – устройство, участвующее в процессе передачи данных по сети.
- **Конечная точка** – узел сети либо процесс на узле сети, являющийся исходным отправителем или итоговым получателем данных.



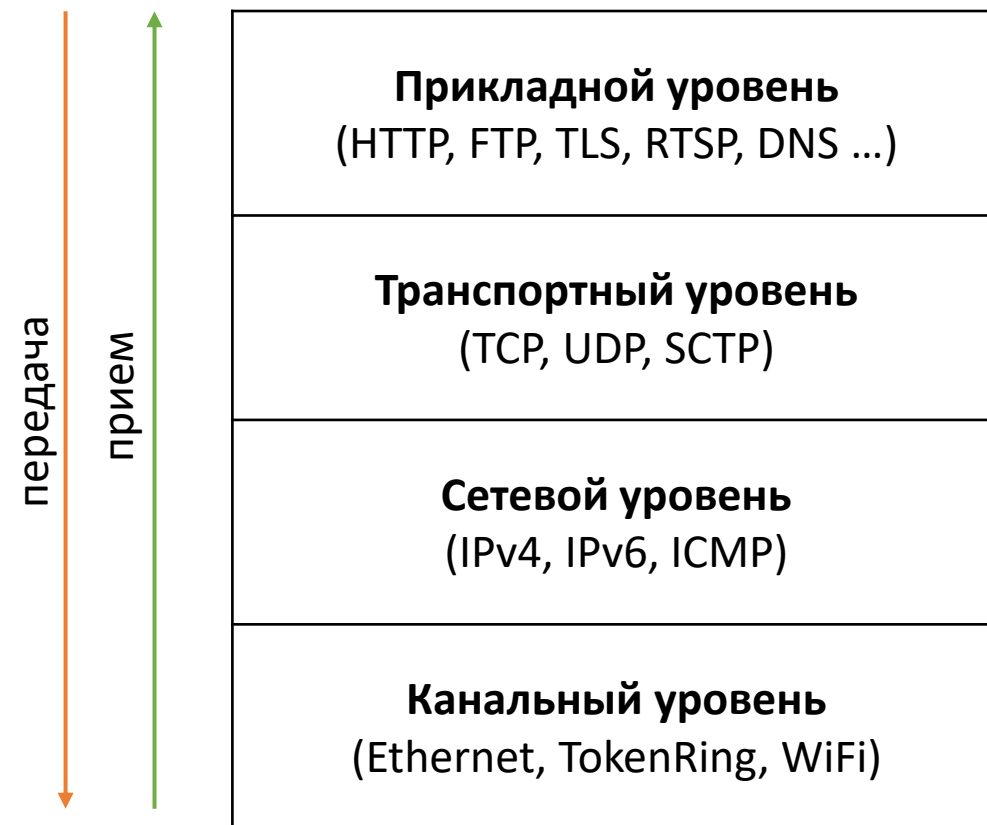
# Модель TCP/IP

Процесс передачи данных разделяется на несколько уровней, на каждом из которых решается соответствующая задача.

Наиболее полной моделью, описывающей взаимодействие по сети (сетевой моделью) является **модель OSI**.

На практике удобнее оперировать более простой **сетевой моделью TCP/IP**.

В рамках модели TCP/IP выделяется 4 уровня: прикладной, транспортный, сетевой и канальный.



# Канальный уровень

На **канальном уровне** определяется, как данные передаются между соседними узлами сети.

Например, протокол Ethernet описывает процесс передачи данных по проводу (витой паре или коаксиальному проводу), а протокол WiFi – беспроводную передачу данных.

В современных сетях каждая точка подключения имеет уникальный **MAC-адрес**.

<b>Прикладной уровень</b> (HTTP, FTP, TLS, RTSP, DNS ...)
<b>Транспортный уровень</b> (TCP, UDP, SCTP)
<b>Сетевой уровень</b> (IPv4, IPv6, ICMP)
<b>Канальный уровень</b> (Ethernet, TokenRing, WiFi)

# Сетевой уровень

На **сетевом уровне** происходит определение маршрута между отправителем и получателем и передача данных по этому маршруту.

На сетевом уровне данные, полученные от транспортного уровня, разбиваются на **пакеты**.

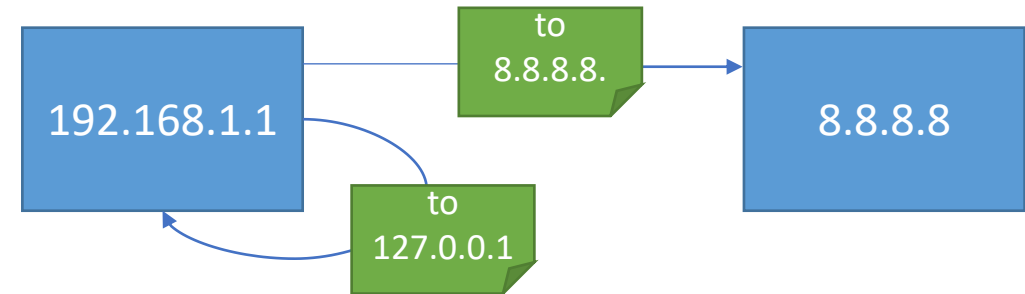
На сетевом уровне общение происходит между непосредственно узлами сети (хостами). Каждый хост имеет уникальный в пределах сети **IP-адрес**.

<b>Прикладной уровень</b> (HTTP, FTP, TLS, RTSP, DNS ...)
<b>Транспортный уровень</b> (TCP, UDP, SCTP)
<b>Сетевой уровень</b> (IPv4, IPv6, ICMP)
<b>Канальный уровень</b> (Ethernet, TokenRing, WiFi)

# IP-адрес

В сети IPv4 IP-адресом является 4-байтовое значение. Адрес записывается в форме 4 чисел (например, **192.128.100.201**).

Специальным адресом\* является loopback-адрес **127.0.0.1** ( **localhost** ). Указывая localhost в качестве получателя, хост может посылать сообщения самому себе.



\*на самом деле, вся подсеть 127.0.0.0/8 соответствует loopback

# Транспортный уровень

На **транспортном уровне** определяется, как происходит передача данных между приложением-отправителем и приложением-получателем.

Во время отправки сообщение прикладного уровня при необходимости разбивается на блоки ограниченной длины – сегменты или датаграммы.

На транспортном уровне каждой конечной точке на узле сети соответствует **порт** – целое число из диапазона [1, 65535].

На данном уровне работают несколько протоколов, в т.ч. TCP и UDP. **Пространства портов разных протоколов не пересекаются.**

<b>Прикладной уровень</b> (HTTP, FTP, TLS, RTSP, DNS ...)
<b>Транспортный уровень</b> (TCP, UDP, SCTP)
<b>Сетевой уровень</b> (IPv4, IPv6, ICMP)
<b>Канальный уровень</b> (Ethernet, TokenRing, WiFi)

# Диапазоны портов

Порты делятся на 3 категории:

**1-1023** – общеизвестные/системные порты (Well-Known Ports) – порты, которые могут быть зарезервированы под конкретные широко используемые приложения или системные службы.

**1024-49151** – зарегистрированные порты – порты, которые могут быть зарезервированы под конкретные приложения.

**49152-65535** – динамические/эфемерные порты - порты, которые гарантированно не могут быть зарезервированы под какое-либо конкретное приложение.

За резервирование портов отвечает организация IANA.

В UNIX-подобных ОС получить общеизвестный порт может только привилегированный процесс.



# TCP

**Протокол TCP** (Transmission Control Protocol) – протокол надежной передачи *потока данных*.

До начала передачи данных происходит установление **соединения**.

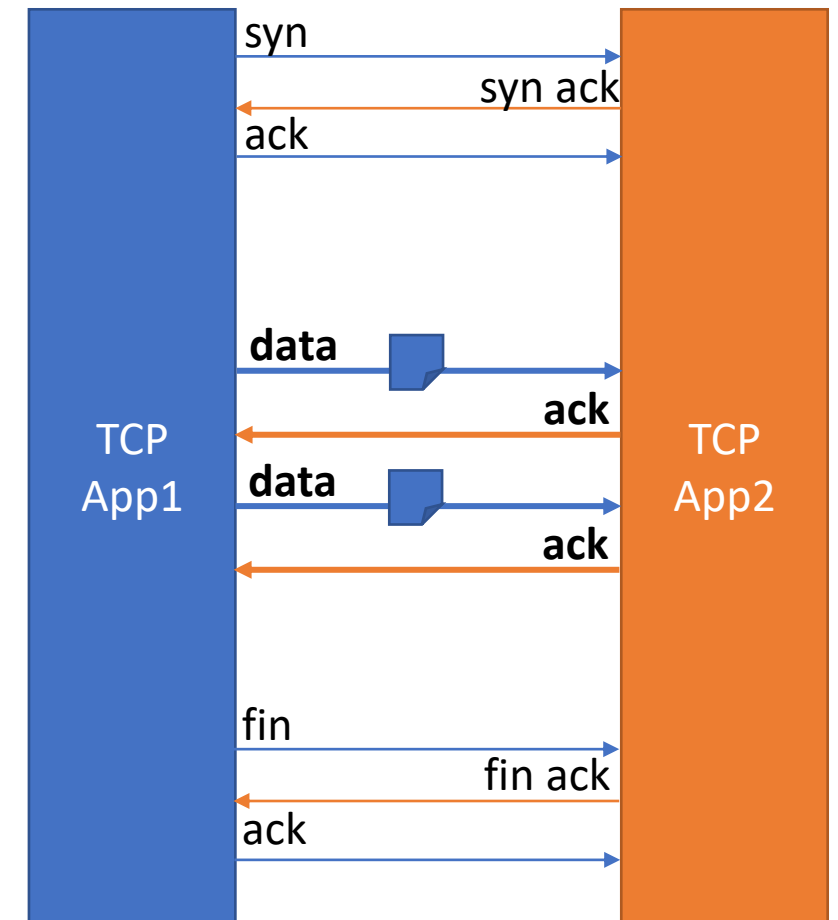
Единицей передачи в TCP является **сегмент**.

Отправляемые данные разделяются на сегменты при отправке и собираются из сегментов при получении.

При передаче данных по протоколу TCP получатель отправляет подтверждение о получении каждого сегмента. Если отправитель не получает подтверждения, то он отправляет сегмент заново.

Если происходит обрыв соединения (по желанию одной из сторон или из-за неполадок), участник соединения получает извещение - *невозможна вечная блокировка*

*TCP не сохраняет границы записанных данных => 2 сообщения по 8 байт могут быть приняты, как 1 сообщение из 16 байт.*



# UDP

**Протокол UDP** (User Datagram Protocol)– протокол ненадежной передачи *сообщений*.

Единицей передачи в UDP является **датаграмма**.

В протоколе UDP соединения не устанавливаются – датаграммы просто отправляются на указанный адрес.

В случае получения датаграммы получатель не отправляет никаких подтверждений => невозможно узнать о потере датаграммы в ходе передачи.

При получении датаграммы производится контроль целостности данных\*, датаграммы с ошибками отбрасываются без уведомлений.

\*опционален в IPv4, но все современные устройства по умолчанию генерируют и проверяют контрольные суммы датаграмм



# Сравнение

## TCP

- моделирует поток данных -> не разделяет данные (2 сообщения по 8 байт могут быть приняты как 1 фрагмент данных из 16 байт);
- гарантирует доставку данных и сохраняет порядок записи данных;
- требует установления соединения перед передачей данных.

## UDP

- явно разделяет записанные данные (2 сообщения по 8 байт будут приняты, как 2 отдельных сообщения);
- не гарантирует ни доставку, ни порядок получения датаграмм;
- не требует установления соединения.

# Прикладной уровень

На **прикладном уровне** приложение определяет непосредственно протокол обмена данными (типы сообщений, допустимые операции и пр).

К примеру, веб-браузер и веб-сервер общаются по протоколу HTTP - это протокол, ориентированный на передачу текстовых данных, с допустимым набором операций GET/POST/ (UPDATE/DELETE/...).

Установление защищенного соединения (аутентификация сторон + шифрование) происходит на этом уровне.

<b>Прикладной уровень</b> (HTTP, FTP, TLS, RTSP, DNS ...)
<b>Транспортный уровень</b> (TCP, UDP, SCTP)
<b>Сетевой уровень</b> (IPv4, IPv6, ICMP)
<b>Канальный уровень</b> (Ethernet, TokenRing, WiFi)

# Сетевой порядок байтов

Разные платформы могут использовать разный порядок байтов для представления чисел.

На x86 числа хранятся в обратном порядке (Little Endian). На AVR числа хранятся в прямом порядке (Big Endian, Network Endian).

Для унификации все числовые данные, используемые в процессе передачи по сети, должны быть представлены **в прямом порядке**.

**IP-адреса и номера портов должны быть в прямом порядке!**

```
int i = 0xAABBCCDD;
```

Little Endian



Big Endian



# Преобразование порядка байтов

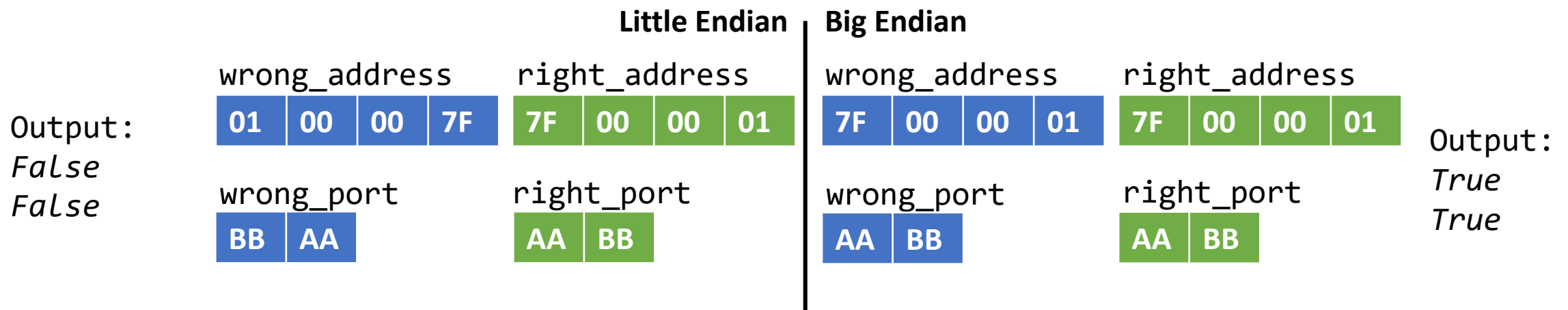
Для преобразования порядка байтов используются функции

```
uint32_t htonl(uint32_t hostlong); /*Host to network long*/  
uint16_t htons(uint16_t hostshort); /*Host to network short*/  
uint32_t ntohl(uint32_t netlong); /*Network to host long*/  
uint16_t ntohs(uint16_t netshort); /*Network to host short*/
```

На Big-Endian архитектурах эти функции ничего не делают. На Little-Endian архитектурах функции меняют порядок байтов переданного числа.

# Преобразование порядка байтов

```
uint32_t wrong_address = 0x7f000001; /* 127.0.0.1 */
uint32_t right_address = htonl(0x7f000001); //OK
puts(wrong_address == right_address ? "True" : "False");
uint16_t wrong_port = 43707;
uint16_t right_port = htons(43707); //OK
puts(wrong_port == right_port ? "True" : "False");
```



# Размеры данных, выравнивание и сеть

Архитектуры машин в сети, а значит размеры фундаментальных типов и выравнивание могут отличаться.

Наиболее надежным вариантом решения обеих проблем является сериализация структур в поток байтов по определенному протоколу, однако в простых случаях такое решение может быть избыточным.

Проблема разницы размеров решается\* использованием типов данных явного размера (`short` -> `int16_t`, `int` -> `int32_t` и т.д.).

Проблема выравнивания решается\*\* через директивы компилятора `#pragma pack(push, 1)` и `#pragma pack(pop)`. Данные директивы поддерживаются MSVC, GCC и clang.

\*за исключением машин с не-8-битным байтом

\*\*может вызвать проблемы на машинах с со строгими требованиями на выравнивание

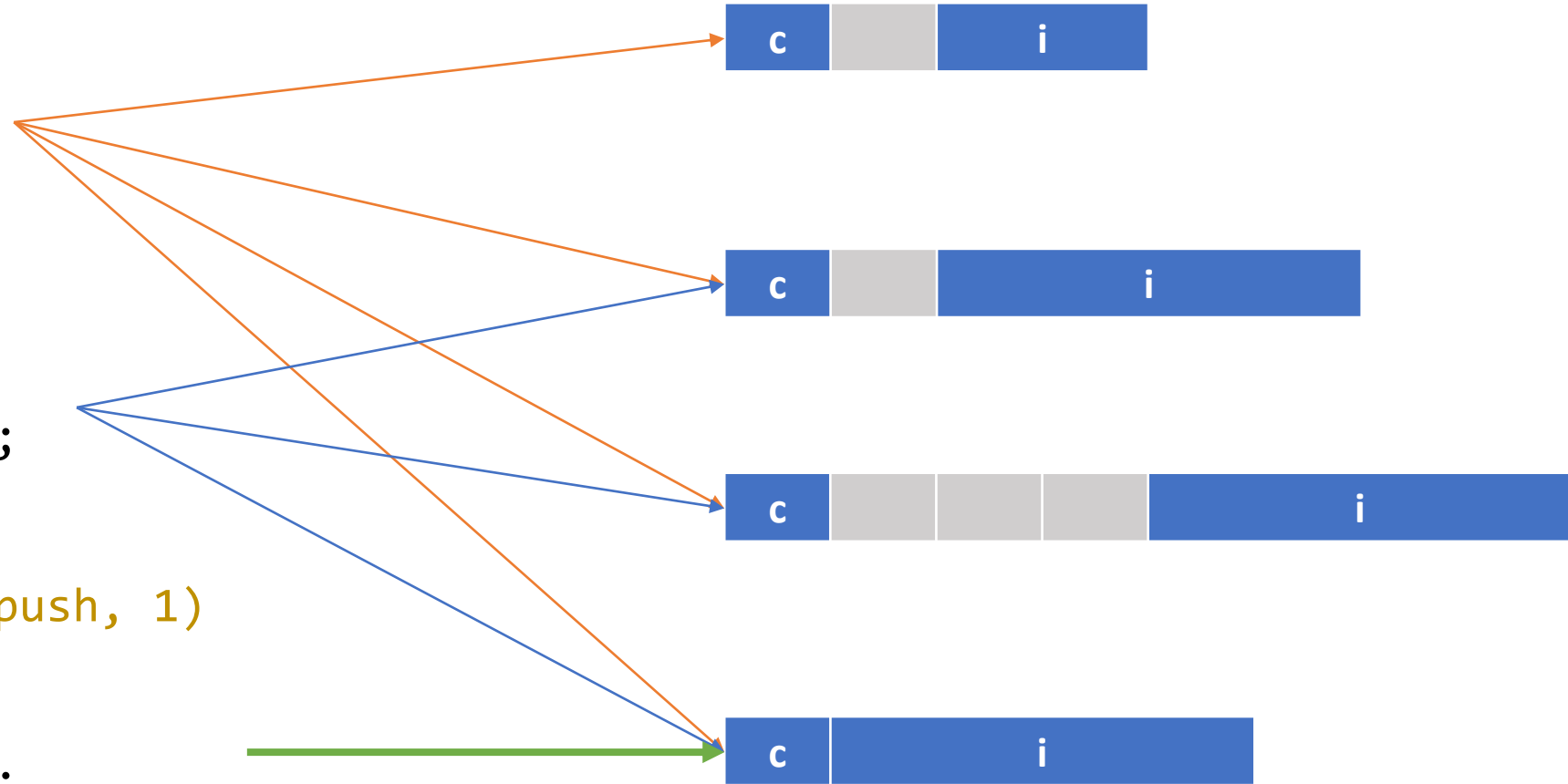


# Размеры данных, выравнивание и сеть

```
struct S{  
    char c;  
    int i;  
};
```

```
struct S2{  
    int8_t c;  
    int32_t i;  
};
```

```
#pragma pack(push, 1)  
struct S3{  
    int8_t c;  
    int32_t i;  
};  
#pragma pack(pop)
```



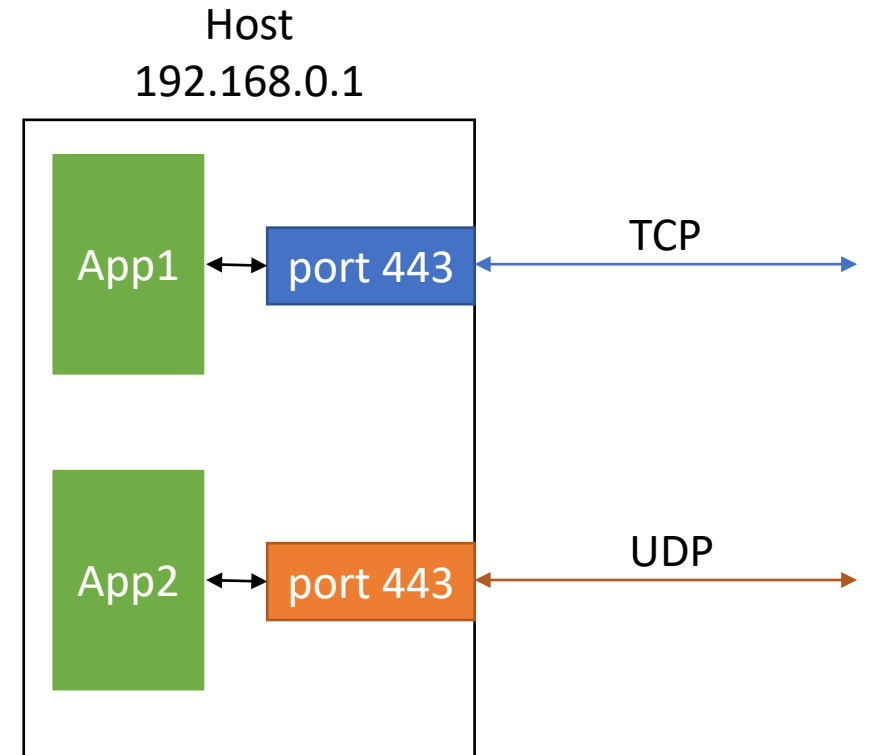
# Конечная точка

Каждое приложение, работающее с сетью, является **конечной точкой** сети.

Конечная точка определяется 3-мя параметрами:

- IP-адрес (255.255.255.1);
- Протокол транспортного уровня (TCP/UDP/SCTP/...);
- Порт (0-65535).

Соединение в целом описывается 5 параметрами (2 адреса, 2 порта и протокол).



# Сокеты

**Сокет** – абстракция конечной точки.

В UNIX с сокетами можно работать, как с обычными файлами. В частности, дескриптор сокета является обычным файловым дескриптором.

Для сокетов определены операции приема и передачи данных (вызовы `send/recv`), однако поддерживаются и обычные файловые операции чтения/записи (вызовы `read/write`).

Закрывается сокет вызовом `close()`.

# Сокеты

Сокеты открываются вызовом `socket()`.

```
int sockfd = socket(int domain, int type, int protocol);
```

Аргументы:

- `domain` – домен сокета (`AF_INET` для IPv4, `AF_UNIX` для UNIX и т.д.);
- `type` – тип сокета (см. след слайд);
- `protocol` – протокол соединения (если 0, выбирается автоматически).

Вызов возвращает дескриптор сокета или -1 в случае ошибки.

Дескриптор закрывается вызовом `close()`.

# Типы сокетов

```
int sockfd = socket(int domain, int type, int protocol);
```

Тип сокета должен быть одним из 3 констант:

- `SOCK_STREAM` – сокет потоковой передачи данных (TCP).
- `SOCK_DGRAM` – сокет ненадежной передачи сообщений (UDP).
- `SOCK_RAW` – «сырой» сокет (прямой доступ к сетевому уровню).

Формально определена константа `SOCK_SEQPACKET` (сокет надежной передачи сообщений), но для IP-сокетов этот тип не реализован.

# Присвоение адреса

Для выполнения приема/передачи данных сокет должен получить адрес. Для присвоения адреса используется вызов `bind()`:

```
int bind(int sockfd, const sockaddr* addr, socklen_t addrlen);
```

Аргументы:

- `sockfd` – дескриптор сокета;
- `addr` – адрес, к которому привязывается сокет;
- `addrlen` – длина адреса.

Структура `sockaddr` не используется напрямую, вместо нее, в зависимости от типа сокета используются `sockaddr_in` (для сокетов `AF_INET`) или `sockaddr_un` (для сокетов `AF_UNIX`).

# Структура sockaddr\_in

```
struct sockaddr_in {  
    sa_family_t    sin_family; /* =AF_INET */  
    in_port_t      sin_port;   /* порт в прямом порядке */  
    struct in_addr sin_addr;    /* адрес (см. ниже) */  
};
```

```
struct in_addr {  
    in_addr_t      s_addr; /* адрес в прямом порядке */  
};
```

# Присвоение адреса

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0); /*TCP сокет*/

sockaddr_in addr{};
addr.sin_port = htons(40001);
addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_family = AF_INET; /*конечная точка IPv4 127.0.0.1:40001*/

bind(sock_fd, (sockaddr*)&addr, sizeof(addr));
/*...*/
```



# Установление соединения (SOCK\_STREAM)

В случае сокетов типа SOCK\_STREAM, до начала передачи данных необходимо установить **соединение**.

При установлении соединения выделяются **2 стороны**:

1. Сторона, ожидающая соединение (выполняет вызовы `bind()` + `listen()` + `accept()`).
2. Сторона, инициирующая соединение (выполняет вызов `connect()`).

# Ожидание соединения (SOCK\_STREAM)

Перевод сокета в режим приема подключений осуществляется вызовом listen:

```
int listen(int sockfd, int backlog);
```

Аргументы:

sockfd – дескриптор сокета (сокеты должны быть привязаны к адресу);

backlog – длина очереди приема соединений.

Вызов возвращает 0 в случае успеха и -1 при ошибке.

В аргументе backlog передается максимальное количество подключений, ожидающих приема. Например, при backlog==3 первые 3 запроса на соединение будут ожидать приема, а 4-й будет сразу отвергнут.

**Замечание:** вызов не принимает никаких соединений, он только переключает режим работы сокета!

# Установка соединения (SOCK\_STREAM)

Прием запроса на соединение осуществляется вызовом accept:

```
int accept(int sockfd, sockadd* addr, socklen_t* addrlen);
```

Аргументы:

- sockfd – дескриптор сокета (сокет должен быть в режиме приема подключений);
- addr – буфер для адреса инициатора соединения [опционален];
- addrlen – указатель на размер буфера.

По адресу addrlen до вызова должен быть записан размер буфера, после успешного вызова сюда же будет записан размер записанного адреса.

Вызов возвращает дескриптор **нового сокета**, который ассоциирован с принятым подключением или -1 при ошибке.

Вызов **блокирует** поток до получения нового соединения.

# Установка соединения (SOCK\_STREAM)

Соединение инициируется вызовом connect:

```
int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
```

Аргументы:

- sockfd – дескриптор сокета;
- addr – целевой адрес;
- addrlen – размер адреса.

Вызов возвращает -1 при ошибке.

**Замечание:** если сокет не был явно привязан к конечной точке вызовом bind(), он будет привязан к случайному динамическому порту автоматически.

# Установка соединения (SOCK\_STREAM)

Сторона сервера

```
int listen_fd = socket(AF_INET, SOCK_STREAM, 0); // создаем сокет
sockaddr_in addr{}, other_addr;
socklen_t other_len= sizeof(other_addr);
addr.sin_port = htons(40001);
addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_family = AF_INET;

bind(listen_fd, (sockaddr*)&addr, sizeof(addr)); // привязываем адрес к сокету
listen(listen_fd, 1); // переключаем сокет в режим приема соединений
while(true){
    // принимаем соединение
    int sock_fd = accept(listen_fd, (sockaddr*)&other_addr, &other_len);
    /* работаем с соединением */
    close(sock_fd); //закрываем сокет соединения
}
```

# Установка соединения (SOCK\_STREAM)

Сторона клиента

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0); // создаем сокет

sockaddr_in addr{};
addr.sin_port = htons(40001);
addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_family = AF_INET;
connect(sock_fd, (sockaddr*)&addr, sizeof(sockaddr)); // подключаемся по адресу
/* работаем с соединением */
```

# Передача данных (SOCK\_STREAM)

Передача данных производится вызовами send/recv

```
ssize_t send(int sockfd, const void* buf, size_t len, int flags);
```

```
ssize_t recv(int sockfd, void* buf, size_t len, int flags);
```

Вызовы возвращают число полученных/отправленных байт.

Аргументы, за исключением flags, аналогичны аргументам read()/write().

В flags могут быть переданы следующие константы: **MSG\_WAITALL** (ждать получения заданного количества байт), **MSG\_PEEK** (прочитать данные без удаления из буфера ожидания), **MSG\_DONTWAIT** – не ждать завершения операции.

# Разрыв соединения (SOCK\_STREAM)

Соединение разрывается автоматически при закрытии сокета вызовом `close()`. Запретить отправку новых данных любой стороне можно вызовом `shutdown()`:

```
int shutdown(int sockfd, int how);
```

В аргументе `sockfd` передается дескриптор сокета.

В аргументе `how` передается одна из констант:

- `SHUT_RD` – запретить прием новых данных;
- `SHUT_WR` – запретить отправку новых данных;
- `SHUT_RDWR` – запретить все.

Вызов возвращает -1 при ошибке.



# Передача данных (SOCK\_DGRAM)

В случае сетевых сокетов типа **SOCK\_DGRAM** (протокол UDP), процесс передачи определяется спецификой данного типа сокетов.

- Каждый вызов `send()/sendto()` отправляет отдельное сообщение (а не часть единого потока данных).
- Если сообщение не может передано одной датаграммой, то возможно получение ошибки **EMSGSIZE**.
- Максимальный размер датаграммы – 64КБ.
- Вызов `recv()/recvfrom()` всегда читает только 1 сообщение независимо от запрошенного размера.
- ...

# Передача данных (SOCK\_DGRAM)

- Сокеты **SOCK\_DGRAM** не требуют установления соединения между 2 узлами – прием и передача данных возможны сразу после `bind()`.
- Сокет принимает все датаграммы, предназначенные конечной точке, если не был вызван `connect()`.
- Вызов `connect()` не устанавливает соединение, он всего лишь определяет адрес отправки по умолчанию и разрешает прием датаграмм только с заданного адреса (т.е. фильтрует датаграммы).

# Передача данных (SOCK\_DGRAM)

Если необходимо реализовать общение одновременно с несколькими конечными точками, то вместо `send()/recv()` нужно использовать `sendto()/recvfrom()`.

Вызов `sendto()` эквивалентен `send()`, но требует указания адреса получателя при каждой отправке.

Вызов `recvfrom()` возвращает адрес отправителя датаграммы вместе с самой датаграммой.

```
ssize_t sendto(int sockfd, const void* buf, size_t len, int flags,  
               const sockaddr* dest_addr, socklen_t addrlen);
```

```
ssize_t recvfrom(int sockfd, void* buf, size_t len, int flags,  
                 sockaddr* src_addr, socklen_t* addrlen);
```

# «Сырые» сокеты

«Сырые» сокеты позволяют получать доступ напрямую к сетевому (`AF_INET`, `SOCK_RAW`) или канальному уровню (`AF_PACKET`, `SOCK_RAW/SOCK_DGRAM`), и вручную формировать непосредственно IP-пакеты/Ethernet-кадры или сообщения служебных протоколов (ICMP, ARP и т.д.).

«Сырой» сокет может быть получен только при наличии соответствующих привилегий.

- ответственность за корректную реализацию возлагается на программиста;
- + можно реализовывать собственные протоколы более высоких уровней целиком в пространстве пользователя;
- + можно *делать разное*.