



## Урок 6

# Сетевые функции ядра

Сетевые модели OSI/ISO. Реализация сетевых функций в ядре. Сокеты Беркли. Взаимодействие сервера и операционной системы на примере TCP/IP сервера.

### Введение

### Сетевые модели

#### OSI/ISO

#### TCP/IP

#### OSI и TCP

#### Практическая реализация сетевых стандартов на данный момент

##### Прикладные протоколы (пользовательские и системные)

##### Уровень транспортной безопасности

##### Транспортный уровень – гарантированная и негарантированная доставка

##### Сетевой уровень

##### Уровень сетевых интерфейсов

### Реализация сетевых служб в ядре операционной системы

#### Сокеты Беркли

##### Типы транспортных протоколов

##### Функции сокетов Беркли

###### Инициализация сокета – socket()

###### gethostbyname() – получение IP-адреса

###### bind() и connect() – привязывание адресов и порта и подключение

###### close() – закрытие сокета

[listen\(\) и accept\(\) – слушать сокет и принимать соединения](#)

[Функции чтения и записи](#)

[Идентификация процессов](#)

[Как работает сервер/клиент](#)

[Как работает TCP-сервер](#)

[Как работает TCP-клиент](#)

[Как работает UDP сервер](#)

[Как работает UDP-клиент](#)

[Пример реализации TCP-сервера](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

# Введение

Сетевые операционные системы появились не сразу. Появление поддержки сети было знаковым событием в эволюции ОС. Многопользовательский режим с поддержкой сети – это отдельно выделяемый уровень выполнения в ОС Linux по сравнению с режимами без сетевой поддержки.

Сетевые возможности – важный компонент современных систем: в эпоху SAAS и облачных вычислений сомневаться в этом не приходится. Реализация сетевых возможностей имеет много уровней: оборудование, ядра операционной системы и прикладные программы нуждаются во взаимодействии.

Рассмотрим сетевые модели, которые используются в настоящее время.

## Сетевые модели

### OSI/ISO

OSI/ISO – это классическая семиуровневая модель, которая обычно рассматривается в учебной литературе и пользуется успехом у отечественного государственного заказчика. Эта модель – академическая попытка создания универсальной модели взаимодействия систем.

Название OSI/ISO расшифровывается как Open Systems Interconnection model (OSI model). Модель OSI является стандартом ISO/IEC 7498-1.

В русскоязычных документах и стандартах модель может фигурировать под именем ЭМВОС, что расшифровывается как «эталонная модель взаимодействия открытых систем». Применительно к ней существует отечественный стандарт ГОСТ Р ИСО/МЭК 7498-1-99).

Модель OSI/ISO разработана в 1978 году и состоит из 7 уровней, которые частично соответствуют применяемым на практике технологиям.

Уровни модели:

- 7=Прикладной;
- 6=Представления;
- 5=Соединения;
- 4=Транспортный;
- 3=Сетевой;
- 2=Канальный;
- 1=Физический.

Предполагалось, что реализация данной модели позволит осуществлять унифицированное взаимодействие между машинами разных архитектур и операционных систем, с различными кодировками, разрядностью и т.д.

Рассмотрим, каким образом должно осуществляться взаимодействие между уровнями.

Прикладной уровень предоставляет прикладному программному обеспечению доступ к сетевым услугам и передаёт запрос на нижний уровень.

Уровень представления осуществляет перекодировку сообщений таким образом, чтобы работа осуществлялась между машинами с разными кодировками/разрядностью. Уровень представления запрашивает сеансовый уровень.

Сеансовый уровень устанавливает сеанс между отправителем и получателем с заданным уровнем надёжности.

Транспортный уровень осуществляет доставку сообщений от отправителя до получателя. На более низком уровне доставка реализуется через сетевой уровень.

Сетевой уровень осуществляет глобальную адресацию и маршрутизацию. Таким образом, если на сеансовом уровне соединение выглядит как установленный сеанс связи, например, с двусторонней связью, а на транспортном – как непосредственное соединение отправителя и получателя, то на сетевом уровне появляются многочисленные промежуточные станции – шлюзы и маршрутизаторы. Тем не менее, даже сетевой уровень является довольно высокоуровневым и алгоритмическим. Он определяет, какое сообщение на какой адрес следует отправлять, но не решает, каким образом сообщение будет доставлено до следующей в маршруте машины: это уже задача канального уровня.

Канальный уровень в общем случае не связывает отправителя и получателя. Если все вышестоящие уровни связывали отправителя и получателя непосредственно, задача канального уровня – связать ближайших отправителей и получателей среди маршрутизаторов, включая отправителя и получателя. Таблицы маршрутизации есть на всех машинах, даже формально не являющихся маршрутизаторами. Таким образом, канальный уровень зависит от среды передачи и решает задачи идентификации абонентов, работающих в одной среде.

Физический уровень задаёт способ кодирования данных с использованием заданной физической среды (витая пара, радиоканал, оптическая связь и т.д.)

Разработанный проект оказался слишком громоздким, и в практической реализации успел распространиться стандарт TCP/IP. стек TCP/IP содержал всего четыре уровня. Именно TCP/IP реализуют все современные сетевые операционные системы.

## TCP/IP

Для обозначения TCP/IP иногда также применяется термин DOD Model (USA Department of Defence). стек протоколов родился из проекта NCP (Network Control Program), реализующей сетевой стек в TCP/IP.

В модели TCP/IP выделяют четыре уровня:

- 4=Уровень приложений;
- 3=Транспортный уровень;
- 2=Сетевой уровень;
- 1=Уровень сетевых интерфейсов.

Модель TCP/IP не различает физический и канальный уровни (которые, впрочем и на практике описываются одним документом – например, IEEE 802.3, IEEE 802.11), в то время как модель OSI/ISO обладает уровнями представления и соединения, которые едва ли могут быть однозначно сопоставлены с действующими протоколами. Четырёхуровневую модель иногда неудобно рассматривать в контексте современных механизмов шифрования, потому что протоколы HTTP и HTTPS в TCP/IP оказываются на одном уровне – при этом, например, в модели OSI протокол HTTPS представляется двумя протоколами: собственно HTTP на прикладном уровне и TLS – на представительском/сеансовом.

Рассмотрим уровни TCP/IP.

Уровень приложений в TCP/IP реализуется соответствующими прикладными программами. Для реализации прикладного протокола требуется открыть соответствующий сокет транспортного уровня – как правило, TCP или UDP. Большинство применяемых на практике протоколов (HTTP, SMTP, POP3, IMAP, DNS, DHCP, NTP, SNMP, XMPP и т.д.) относятся к прикладному уровню. Прикладные протоколы относительно просты в реализации, и разработчик программного обеспечения свободен в создании новых.

На транспортном уровне в TCP/IP решаются задачи идентификации приложения (фактически транспортный уровень реализует механизм межпроцессного взаимодействия) и гарантированной/негарантированной доставки. Как правило, используются протокол TCP для гарантированной доставки (создаёт сеанс связи) или UDP для негарантированной (в этом случае речь идёт о дейтаграммах, названных так по аналогии с телеграммами). Существуют и другие протоколы транспортного уровня – как с гарантированной доставкой, так и без нее.

Сетевой уровень в модели TCP/IP классически представлен протоколом IP и связанными с ним протоколами – такими, как ICMP, служащий для индикации проблем сетевого уровня и IGMP, предназначенный для управления подписками на multicast-рассылки (IP-TV). Тесно связаны с протоколом IP и протоколы, обеспечивающие его работу, в том числе прикладного уровня (DHCP, DNS), и протоколы маршрутизации (технически могут быть реализованы протоколы как сетевого, так и прикладного уровня, при этом протокол TCP или UDP задействуется для транспорта). Для взаимодействия IP и протоколов канального уровня, использующих MAC-адреса, берёт протокол ARP.

В связи с исчерпанием пространства IP-адресов (IPv4) постепенно внедряется IPv6, который можно воспринимать как альтернативный сетевой протокол со своими обслуживающими протоколами (ICMPv6, а в общем случае с отсутствием необходимости в DHCP также и ARP).

Если прикладные протоколы определяются практически решаемыми задачами, то выбор между UDP и TCP обычно определён потребностью в гарантированной или негарантированной доставке. Возможны варианты использования и того и другого: например, DNS – для ответов на запрос хоста используется UDP, а для обмена файлами зон и DNSsec – TCP.

В перспективе возможно использование протоколов IPv6 совместно с транспортными протоколами SCTP и DCCP (широкому распространению которых, помимо прочего, препятствует популярная система NAT, работающая на уровне ядра – как правило, это сетевой фильтр).

Отдельно стоит отметить стек протоколов IPX/SPX. Идентификация приложений в нём выполнялась в сетевом протоколе IPX, а SPX реализовал транспортные возможности. В настоящее время данный протокол не применяется, но идеология IPX в значительной степени повлияла на IPv6.

Основные протоколы:

- IP – Internet Protocol;
- ARP – Address Resolution Protocol;
- TCP – Transmission Control Protocol;
- UDP – User Datagram Protocol;
- ICMP – Internet Control Message Protocol;
- IGMP – Internet Group Management Protocol;
- DNS – Domain Name System;
- DHCP – Dynamic Host Configuration Protocol;
- NTP – Network Time Protocol;
- SNMP – Simple Network Management Protocol.

Большинство упомянутых протоколов стека TCP/IP описаны в соответствующих документах RFC.

Также упомянутые протоколы:

- IPX – Internetwork Packet Exchange;
- SPX – Sequenced Packet Exchange;
- SCTP – Stream Control Transmission Protocol;
- DCCP – Datagram Congestion Control Protocol.

Часто используемые стандарты канального уровня:

- RFC – 1661: PPP – Point to Point Protocol;
- IEEE 802.3 – Ethernet;
- IEEE 802.1Q – VLAN;
- IEEE 802.11 – WiFi;
- ITU G.992.1 – ADSL;
- ITU G.991.1 – HDSL.

PPP – в чистом виде протокол канального уровня, поэтому он используется в туннелях; остальные спецификации определяют и физический уровень.

## OSI и TCP

Существуют разные варианты сопоставления действующих протоколов с уровнями модели, но они относятся к спекулятивным. На практике вместо модели OSI/ISO используется четырехуровневая модель TCP/IP. У этих моделей есть общие черты: например, идея с инкапсуляцией. На самом верхнем уровне некий объем данных снабжается заголовком и отправляется на нижестоящий уровень. Нижестоящий уровень упаковывает несколько фрагментов данных в один или, наоборот, разрезает чересчур объемную последовательность на несколько, снабжает своими заголовками и передает на следующий уровень. Так информация доходит до физической среды передачи данных, а на другой стороне провода или передающей среды происходит сборка в обратном направлении. Подобная иерархия напоминает матрешку.

В простейшем случае кадр канального уровня с заголовком включает в себя IP-пакет с IP-заголовком, в котором содержится либо TCP-сегмент (со своим заголовком), или UDP-дейтаграмму (со своим заголовком) – они уже вмещают данные или их часть, непосредственно полученные от программного обеспечения.

Притом, что прикладной уровень модели TCP примерно схож с уровнем OSI/ISO, он также включает в себя аспекты прикладного и сеансового уровня (последний в разной степени может реализовываться самим прикладным протоколом, а также TCP в целом предоставляет сеансовое подключение). Условно к представительскому и сеансовому уровням модели OSI относят процессы, связанные с шифрованием, кодировкой, установкой сеанса связи – при этом задачи кодировки в них могут и не решаться (они решаются в составе прикладного протокола или в составе передаваемых файлов, как, например, в HTTP/HTML). Нечто похожее на уровень представления наблюдается при использовании инструкций htons/ntohs и htonl/ntohl. К уровню представления могут относиться также стандарты кодировок, алгоритмы сжатия, аудио- и видео-кодеки и т.д. К сеансовому уровню можно отнести механизм удаленных вызовов процедур и реализации API (несмотря на то, что они могут использовать в том числе и протокол HTTP в качестве транспортного уровня). Притом если вопрос соотнесения практических протоколов с представительским и сеансовым уровнем спекулятивно можно решить, то на практике строгого перехода с уровня выше на уровень ниже и обратно не будет.

Транспортный и сетевой уровень обычно соотносятся однозначно, с сетевым тоже не возникает трудностей. Канальный и физический уровни модели OSI в TCP/IP объединяются в уровень сетевых интерфейсов.

# Практическая реализация сетевых стандартов на данный момент

## Прикладные протоколы (пользовательские и системные)

К прикладным относятся большинство пользовательских протоколов (HTTP, SMTP, XMPP), а также большая часть протоколов, необходимых для нормальной работы системы в сети (DHCP, DNS, NTP). Прикладные протоколы реализуются за счёт программ, работающих в пользовательском пространстве. Для доступа к стеку TCP/IP они запрашивают сокет через интерфейсы ядра операционной системы.

## Уровень транспортной безопасности

Если используется надёжное соединение, применяется SSL (Socket Secure Layer), фактически TLS (Transport Layer Security). В таком случае прикладной протокол инкапсулируется в TLS-сессию, которая уже использует открытый сокет. TLS, который можно отнести к сеансовому уровню модели OSI/ISO, используется в пользовательском пространстве.

## Транспортный уровень – гарантированная и негарантированная доставка

К транспортному уровню относятся STREAM-протоколы, использующие STREAM-сокеты TCP и SCTP. Они принадлежат к транспортному уровню, обладают определенными чертами сеансового уровня. Данные STREAM-протоколов называются сегментами: например, TCP-сегмент. Один сегмент несёт фрагмент установленной сессии.

Кроме того, к транспортному уровню относятся DATAGRAM-протоколы, использующие DATAGRAM-сокеты: UDP и DCCP. Данные в этих протоколах называются дейтаграммами. Такие протоколы не осуществляют контроля доставки и порядка получения сообщений.

Транспортные протоколы позволяют идентифицировать приложения на хосте. Для этого используется двухбайтное пространство STREAM-портов (обычно называемых TCP-портами, даже если используется SCTP) и не связанное с ним пространство DATAGRAM-портов (обычно называемых UDP-портами, даже если используется DCCP). Именно поэтому в стеке протоколов TCP/IP транспортный уровень осуществляет механизм межпроцессного взаимодействия. Программы пользовательского окружения лишь запрашивают доступ к сокету (аналогично доступу к файлам), в то время как механизм сборки пакетов и контроля сессий реализуется на уровне ядра.

Возможна и самостоятельная сборка с применением механизма RAW-сокеты, но для этого – например, в Linux – для запуска приложения требуются права ROOT.

Все приложения, которые не используют идентификацию портов, могут быть обработаны либо на уровне ядра (ICMP, IGMP и т.д.), либо также через сырые (RAW) сокеты. Дополнительный вариант – реализовать собственный механизм подтверждений и инкапсулировать его в UDP. Такой протокол будет обладать чертами сеансового: примером может быть протокол QUIC от Google.

## Сетевой уровень

Сетевой уровень осуществляет связь между хостами. Это становится возможным благодаря глобальной логической адресации (например, IP-адресам), где каждый адрес, как правило, содержит адрес сети и хоста.

На этом уровне решаются задачи маршрутизации пакетов. Таблицы маршрутизации присутствуют на каждой машине, в том числе и на устройствах, не являющихся маршрутизаторами. Связанные протоколы (ICMP, IGMP) являются частью механизма IP и обрабатываются на уровне ядра.

## Уровень сетевых интерфейсов

Уровень сетевых интерфейсов (канальный уровень) решает задачи физической адресации и идентификации сетевых интерфейсов (MAC-адреса, MAC- Media Access Control, контроль доступа к среде). MAC-адрес играет роль позывного в радиосети и позволяет отличить трафик, направленный данному получателю, от трафика, направленного к соседям.

Если прикладной протокол, как правило, однозначно определяет, через TCP или UDP он будет работать, то для работы IPv6 уже необходима инфраструктура IP-адресов и доменных имён (в том числе и записи AAAA DNS-серверов), и какой будет протокол канального уровня – как правило, не играет роли. Если переподключиться с WiFi на Ethernet, работа в сети продолжится аналогично. При этом TCP-сессии будут разорваны, а SCTP – нет.

Уровень сетевых интерфейсов реализуется соответствующими драйверами сетевых устройств. Например, для поддержки VLAN в Linux используется модуль ядра 8031Q.

# Реализация сетевых служб в ядре операционной системы

На уровне ядра решаются следующие задачи:

- работа с IP-пакетами (и сетевой фильтр, в том числе NAT);
- маршрутизация;
- связь сетевого и канального уровня (ARP);
- предоставление доступа к сокетам.

Прикладные протоколы реализуют прикладные программы самостоятельно, используя предоставленный ядром операционной системы механизм сокетов.

Сложнее выглядит ситуация с канальным и физическим уровнем. Фактически такой доступ тоже может реализовать прикладная программа, например, напрямую взаимодействуя с портами (/dev/ttyS0 и т.д. для Linux), но для имплементации этого взаимодействия в стек TCP/IP необходимо решение на уровне ядра. Родные решения для работы с Ethernet и т.д., уже включены в ядро, либо включаются как модули ядра. Внешние задачи – такие, как работа с нестандартным оборудованием или туннелирование – могут решаться как программное обеспечение в пользовательском пространстве, но, как правило, для этого требуется создать соответствующий файл устройства и либо использовать существующий модуль ядра (например, для TUN и TAP), либо реализовать свой.

Возможны случаи, когда взаимодействие происходит несколько раз между окружением ядра и пользовательским окружением. Например, туннель OpenVPN потребует модуль ядра tun для доступа к интерфейсу tun. Именно через этот интерфейс приложения будут получать доступ к стеку TCP/IP. Информация от устройства tun будет направляться в пользовательское пространство OpenVPN, а затем снова в ядро, но уже в сетевой интерфейс, обслуживающий Ethernet или WiFi.



# Сокеты Беркли

Технология сокетов Беркли является фундаментальной для стека протоколов TCP/IP. Она построена по образцу файлов (исходя из принципа unix-way «всё есть файл»).

## Типы транспортных протоколов

Различаются протоколы с установкой соединения и протоколы без установки соединения. Под первым в большинстве случаев подразумевается протокол TCP, под вторым – UDP.

Протоколы с установкой соединения используются в случаях, когда требуется надёжное соединение, когда прикладная программа не решает вопросы сборки пакетов и проверки последовательности их приёма, а также когда важна последовательность приёма данных и идентичность полученных данных отправленным. Так, текстовый документ или двоичный файл будет отправляться в большинстве случаев через протокол с установкой соединения. Бывают и исключения: например, если контроль за ошибками и коррекция решаются вышестоящим протоколом (например, протоколом прикладного уровня QUIC от Google).

Протоколы без установки соединения используются, когда требуется доставка небольших порций данных и когда потеря или порядок переданных сообщений не критичен, а скорость передачи важнее гарантированной доставки всех сообщений. В частности, UDP используется для онлайн-трансляций, VoIP: если несколько сообщений выпадут, это приведёт к искажению голоса, если же мы будем ожидать гарантированной доставки, может возникнуть задержка, которая практически неизбежна при использовании для этих целей TCP. В то же время на качество передачи будет влиять и канал связи: например, низкая скорость соединения может потребовать применения соответствующих кодеков, которые уменьшают размер передаваемой информации благодаря снижению качества голоса, т.е. выдавая роботизированный, как из радики, голос. Подробнее на эту тему можно почитать по ключевому слову «вокодеры».

Кроме того, UDP может использоваться в реализации онлайн-игр.

## Функции сокетов Беркли

### Инициализация сокета – `socket()`

Функция инициализации возвращает дескриптор сокета (аналогия с файловым дескриптором, произошедшая из unix-way). При этом указывается стек протоколов, который будет использоваться (IPv4, IPv6, Unix-сокеты, использующие для межпрограммного взаимодействия файлы), тип используемого протокола (STREAM, DGRAM, RAW) и протокол.

Тип используемого протокола:

- STREAM – для передачи с установкой соединения (используется в TCP и SCTP);
- DGRAM – для передачи дейтаграмм, то есть без установки соединения (используется в UDP и DCCP);
- RAW – сырой сокет, т.е. непосредственно работа с IP-пакетами.

Протокол:

- TCP;
- UDP;
- SCTP;
- DCCP;

- RAW (для сырых сокетов).

### **gethostbyname() – получение IP-адреса**

Если IP-адрес неизвестен, доменное имя необходимо в него преобразовать. Это будет сделано прозрачно для пользователя: фактически будет выполнено обращение DNS-клиента к DNS-серверу, а при определённых ситуациях и чтение файла /etc/hosts.

### **bind() и connect() – привязывание адресов и порта и подключение**

bind() используется сервером, чтобы связать с сокетом прослушиваемые IP-адрес и порт.

connect() служит, чтобы подключиться к серверу. При этом в аргументах функции указывается в составе структуры информация, с какого IP-адреса, на какой IP-адрес и какой порт будет происходить подключение.

Для перевода сокета в прослушивающее состояние (актуально для сервера) используется listen().

### **close() – закрытие сокета**

Для закрытия сокета используется close().

Существует также механизм для разрешения доменных имён gethostbyname(). При его задействовании происходит обращение к файлу /etc/hosts, к локальной DNS-службе, которая уже использует поиск по распределённой базе соответствий IP-адреса для указанного имени. Хотя DNS – это протокол прикладного уровня, в данном контексте он совершенно особенный, так как используется независимо от того, создаём мы сокет с установкой соединения, без установки соединения или даже сырой сокет.

Для закрытия сокета служит close(). Если процесс выполнил fork(), каждый порождённый процесс также должен выполнять close().

### **listen() и accept() – слушать сокет и принимать соединения**

Если реализуется сервер, для приёма входящих соединений необходимо перевести сокет в состояние слушающего. Для этого используется listen().

Кроме того, для TCP-соединения дополнительно необходимо принимать каждое входящее соединение с помощью accept() и в дальнейшем обслуживать созданную сессию.

### **Функции чтения и записи**

Для чтения и записи используются различные механизмы: первоначально (read() и write()), как для файлов, затем recv() и send().

### **Идентификация процессов**

В стеке протоколов TCP/IP для идентификации приложения используется номер – порт, позволяющий идентифицировать как приложение на хосте-отправителе (которому нужно вернуть ответ), так и приложение на хосте-получателе (которому следует передать сообщение). Два байта позволяют идентифицировать до 65545 портов. При этом пространство портов тоже распределено специфическим образом.

С 1 по 1023 порты, как правило, зарезервированы для стандартных служб. Например:

- 20 – передача данных по протоколу FTP;
- 21 – передача команд по протоколу FTP;
- 22 – SSH;

- 23 – telnet;
- 70 – gopher – предшественник HTTP;
- 80, 8080 – HTTP;
- 443 – HTTPS.

В исходящих дейтаграммах UDP-соединения в качестве исходящего порта может быть также указан 0.

В Linux прослушивание портов до 1023 номера требует прав root.

Назначение портов определяется той же Администрацией адресного пространства Интернет (IANA – Internet Assigned Numbers Authority), которая назначает IP-адреса. При этом возможно использование программным обеспечением и незарегистрированных портов или даже портов, зарегистрированных для других целей.

Порты с номерами выше 49152 называются динамическими и используются клиентским программным обеспечением.

Считается, что для TCP и UDP-протоколов используется независимая нумерация портов: два порта с одним и тем же номером, TCP и UDP, могут быть открыты независимо, не мешая друг другу. Впрочем, это не совсем верно, так как, кроме TCP и UDP, существуют и другие протоколы. Корректнее говорить, что имеется диапазон портов от 0 до 65545 для STREAM-протоколов (как правило, TCP, но также и SCTP, а в теории и для других возможных протоколов с установкой соединения, при условии, что ядро операционной системы будет их поддерживать) и от 0 до 65545 для DGRAM-протоколов (как правило, UDP, но также и DCCP, а в перспективе и для других возможных протоколов без установки соединения, при условии, что ядро операционной системы также будет их поддерживать).

Итак, если протокол не является ни TCP/SCTP, ни UDP/DCCP, для корректной работы с ним необходимо, чтобы протокол поддерживался на уровне ядра операционной системы, – в противном случае следует использовать механизм сырых сокетов.

Вариант 1: протокол поддерживается ядром операционной системы, т.е. на уровне ядра может быть извлечён номер порта получателя, и, соответственно, сегмент или дейтаграмма будет направлена в соответствующий порт STREAM или DGRAM. В частности, так и обстоит дело с SCTP и DCCP, которые можно воспринимать как перспективные альтернативы протоколам TCP и UDP.

Вариант 2: необходима реализация через сырые (RAW) сокеты. Как понятно из варианта 1, поскольку невозможно определить порт (структура пакета известна только программному обеспечению конкретного пользователя) или такой механизм не используется вообще, единственный оставшийся вариант направить данные слушающему сырому сокету – это направить пришедший пакет всем слушающим сырым сокетами. В таком случае задача определения порта ложится на программное обеспечение пользователя. Если открыто несколько сырых сокетов, пакет получит каждый из них.

## Как работает сервер/клиент

### Как работает TCP-сервер

Сервер инициализирует сокет (IPv4, STREAM, TCP). Затем сервер привязывает к сокету порт и IP-адрес, чтобы определить, по какому IP-адресу и на каком порту предстоит слушать. Чтобы сокет перешёл в состояние прослушиваемого (которое, кстати, можно заметить в выводе netstat) используется функция listen(). Далее запускается бесконечный цикл, в котором для сокета с установкой соединения используется accept(), чтобы принять входящее соединение.

Приняв соединение, сервер может создать `fork()`, чтобы не блокировать прослушивающую очередь. `Fork` не относится к механизму сокетов Беркли, но по понятным причинам необходим для использования. Для каждого вызова `fork()` необходимо закрывать прослушиваемый сокет: `close()`;

Получив сообщение, сервер использует процедуры чтения и записи. При этом он задействует разные варианты: возможны исторические процессы, родственные операциям с файлами `read` и `write`, но обычно это `recv` и `send` или даже вызовы, определенные для конкретной реализации протокола.

После завершения работы сокет закрывается: `close()`.

Обратите внимание, что при использовании механизма сокетов Беркли механизмы TCP для нас остаются скрыты.

### **Как работает TCP-клиент**

Клиент так же, как и сервер, инициализирует сокет (IPv4, STREAM, TCP). Далее, если речь идёт о доменном имени (например, наш клиент будет скачивать файл, и адрес сервера указан в виде доменного имени), необходимо выяснить его IP-адрес, используя вызов `gethostbyname()`.

`gethostbyname ()` входит в состав механизма сокета Беркли.

Затем мы (в отличие от сервера, использующего `bind()`) вызываем `connect()`, в параметрах указывая и наш IP-адрес, и IP-адрес, к которому мы будем подключаться. Исходящий порт выбирается из диапазона динамических портов автоматически.

Далее отправляем серверу сообщение, получаем ответ, закрываем соединение.

### **Как работает UDP сервер**

Сервер инициализирует сокет (IPv4, DGRAM, TCP).

Привязываем (`bind`) к сокету порт и IP-адрес.

Чтобы сокет перешел в состояние прослушиваемого, используем `listen()`.

При этом запускается бесконечный цикл, но, в отличие от TCP, мы не ждём принятия соединения, а сразу ожидаем принятия сообщения. Получив сообщение, отправляем ответ.

После завершения работы сокет закрывается: `close()`.

### **Как работает UDP-клиент**

Клиент инициализирует сокет (IPv4, DGRAM, TCP).

При необходимости разрешаем доменное имя (`gethostbyname ()`).

Затем мы (в отличие от сервера, использующего `bind()`) вызываем `connect()`, в параметрах указывая наш IP-адрес и IP-адрес, к которому будем подключаться. Исходящий порт выбирается из диапазона динамических портов автоматически.

Далее отправляем серверу сообщение, получаем ответ, закрываем соединение.

При использовании `send()` перед отправкой нужно выполнить `connect()`. В отличие от TCP, соединения в этом случае не происходит, а только запоминаются IP-адрес отправителя, IP-адрес и порт получателя, куда будет отправлено сообщение.

## Пример реализации TCP-сервера

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    int sock, listener;
    struct sockaddr_in addr;
    char buf[2048];
    int bytes_read;
    listener = socket(AF_INET, SOCK_STREAM, 0);
    if(listener < 0)
    {
        perror("socket");
        exit(1);
    }
    addr.sin_family = AF_INET;
    addr.sin_port = htons(8888);
    addr.sin_addr.s_addr=htonl(INADDR_ANY);
    if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("bind");
        exit(2);
    }
    printf("Начинаем слушать \n");
    listen(listener, 1);
    while(1)
    {
        sock = accept(listener, NULL, NULL);
        if(sock < 0)
        {
            perror("accept");
            exit(3);
        }
        if (!fork())
        {
            close(listener);
            send(sock, "Echo server. Hello\n",19,0);
            while(1)
            {
                printf("Ожидаем сообщение...\n");
                bytes_read = recv(sock, buf, 2048, 0);
                if(bytes_read <= 0) break;
                printf("Получено %d bytes\tСообщение: %s\n", bytes_read, buf);
                printf("Отправляю принятое сообщение клиенту\n");
                send(sock, buf, bytes_read, 0);
            }

            close(sock);
            exit(0);
        }
    }
    return 0;
}
```

Скомпилируем:

```
# gcc tinecho.c -o tinecho
```

Запускаем:

```
.tinyecho
```

Проверяем:

```
netstat -ntl
```

```
oga@uho:~$ netstat -ntl
Активные соединения с интернетом (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp      0      0 127.0.0.1:587      0.0.0.0:*        LISTEN
tcp      0      0 127.0.0.1:53       0.0.0.0:*        LISTEN
tcp      0      0 0.0.0.0:21        0.0.0.0:*        LISTEN
tcp      0      0 0.0.0.0:22        0.0.0.0:*        LISTEN
tcp      0      0 127.0.0.1:631     0.0.0.0:*        LISTEN
tcp      0      0 0.0.0.0:8888      0.0.0.0:*        LISTEN
tcp      0      0 127.0.0.1:25      0.0.0.0:*        LISTEN
tcp      0      0 127.0.0.1:953     0.0.0.0:*        LISTEN
tcp6     0      0 :::443            :::*             LISTEN
tcp6     0      0 :::80             :::*             LISTEN
tcp6     0      0 :::53             :::*             LISTEN
tcp6     0      0 :::22             :::*             LISTEN
tcp6     0      0 :::1:631          :::*             LISTEN
tcp6     0      0 :::1:953          :::*             LISTEN
oga@uho:~$ _
```

Проверяем в списке процессов:

```
ps ax|grep tinyecho
```

```
oga@uho:~$ ps ax|grep tinyecho
 4362 tty2      S+   0:00 ./tinyecho
 4432 tty4      S+   0:00 grep --color=auto tinyecho
oga@uho:~$
```

Подключаемся через telnet:

```
telnet 127.0.0.1 8888
```

```
oga@uho:~$ telnet 127.0.0.1 8888
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Echo server. Hello
Hello
Hello
Test
Test
-
```

Проверяем:

```
netstat -ntl
```

```
oga@uho:~$ netstat -nt
Активные соединения с интернетом (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp      0      0 127.0.0.1:8888      127.0.0.1:59006    ESTABLISHED
tcp      0      0 127.0.0.1:59006     127.0.0.1:8888     ESTABLISHED
tcp      0      0 192.168.131.200:22  192.168.131.1:59776 ESTABLISHED
oga@uho:~$
```

Проверяем в списке процессов:

```
ps ax|grep tinyecho
```

```
oga@uho:~$ ps ax|grep tinyecho
4362 tty2      S+   0:00 ./tinyecho
4499 tty2      S+   0:00 ./tinyecho
4523 tty4      S+   0:00 grep --color=auto tinyecho
oga@uho:~$
```

Один процесс обслуживает принятие соединений, другой – установленное соединение.

Консоль выдачи сервера:

```
oga@uho:~/SRV_TCP$ gcc tinyecho.c -o tinyecho
oga@uho:~/SRV_TCP$ ./tinyecho
Начинаем слушать
Ожидаем сообщение...
Получено 7 bytes      Сообщение: Hello

Отправляю принятое сообщение клиенту
Ожидаем сообщение...
Получено 6 bytes      Сообщение: Test

Отправляю принятое сообщение клиенту
Ожидаем сообщение...
```

# Практическое задание

1. В разобранный пример подключиться через telnet к серверу еще несколько раз (2-3 соединения). Что вы видите через netstat, ps ax?
2. Написать или найти готовый код TCP-клиента, разобрать совместную работу.
3. Написать или найти готовый код UDP-сервера и клиента.
4. Переделать код TCP-сервера или найти готовый без fork. Попробовать подключиться, посмотреть, что происходит.
5. \* Переделать код TCP-сервера с использования процессов на использование потоков. Сравнить.
6. \* Найти и разобрать код SCTP-сервера и клиента. Подключиться, проанализировать сходства и различия с TCP.
7. \* Найти и разобрать реализацию модуля ядра Linux для работы сетевого устройства (можно учебное, без практической ценности).

*Примечание. Задания со звездочкой предназначены для тех, кому недостаточно заданий 1-4 и требуются более сложные задачи.*

## Дополнительные материалы

1. Таненбаум Э. Компьютерные сети
2. Стивенс У.Р. UNIX: Разработка сетевых приложений
3. <http://www.ibm.com/developerworks/ru/library/au-tcpsystemcalls/index.html>
4. <http://rus-linux.net/MyLDP/algol/analiz-variantov-realizacii-TCP-IP-servera.html>

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [https://ru.wikipedia.org/wiki/Сокеты\\_Беркли](https://ru.wikipedia.org/wiki/Сокеты_Беркли)
2. <http://www.unixnetworkprogramming.com/src.html>