# Сетевое программирование от Биджа

## *Использование Интернет Сокетов*

**Брайан “Beej Jorgensen” Холл**
**beej@beej.us**

**Версия 3.0.15**
**Июль 3, 2012**

**Copyright © 2012 Brian “Beej Jorgensen” Hall**

**Перевод: Андрей Косенко aikos55@gmail.com**

# Beej's Guide to Network Programming

## *Using Internet Sockets*

**Brian "Beej Jorgensen" Hall**
**beej@beej.us**

**Version 3.0.15**
**July 3, 2012**

**Copyright © 2012 Brian "Beej Jorgensen" Hall**

Спасибо всем, кто помогал мне в написании этого документа в прошлом и будущем. Спасибо Ashley за уговоры превратить дизайн обложки в лучший образчик программистского художества, на которое я способен. Спасибо всем, кто делает Свободные программы и пакеты, которые я использовал при создании этого Руководства: GNU, Linux, Slackware, vim, Python, Inkscape, Apache FOP, Firefox, Red Hat и многие другие. И наконец, большое спасибо буквально тысячам из вас, кто присылал свои предложения по улучшению руководства и слова ободрения.

Я посвящаю это руководство моим величайшим героям и вдохновителям в мире компьютеров: Дональду Кнуту (Donald Knuth), Брюсу Шнайеру (Bruce Schneier), Ричарду Стивенсу (W. Richard Stevens), Стиву Возняку (The Woz), моим Читателям и всему Free and Open Source Software Community.

Англоязычный вариант этой книги написан автором на XML в редакторе vim на Slackware Linux с инструментами GNU. “Картина” обложки и диаграммы сделаны на Inkscape. XML преобразован в HTML и XSL-FO пользовательскими скриптами Python. XSL-FO вывод затем был преобразован Apache FOP в PDF документы с использованием шрифтов Liberation. Все инструменты это 100% Free and Open Source Software.[1]

За исключением взаимно достигнутого участниками данной работы согласия, автор предлагает всё как есть и не делает никаких заявлений и не даёт никаких гарантий относительно данной работы, выражений, предположений, предписаний или чего-либо иного, включая, без ограничений, гарантии прав собственности, возможности сбыта, применимость для определённых целей, ненарушение законов, отсутствие скрытых или иных дефектов, точность, наличие или отсутствие обнаруживаемых или не обнаруживаемых ошибок.

За исключением области действия применимого законодательства, автор не несёт ответственности ни за какие события, происходящие в силу правовых предположений по любым особым, случайным, косвенным, штрафным или характерным сбоям, возникшим при использовании данной работы, особенно если автор предупреждал о возможности возникновения таких сбоев.

Этот документ может свободно распространяться под лицензией Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License. Смотрите раздел *Копирайт и распространение*.

Copyright © 2012 Brian “Beej Jorgensen” Hall

[1] Перевод книги подготовлен на системе Mac OS X с использованием стандартных инструментов.

# Содержание


**1. Введение 1**

1.1. Для кого 1

1.2. Платформа и компилятор 1

1.3. Официальная страница и книги 1

1.4. Для программистов Solaris/SunOS 1

1.5. Для программистов Windows 2

1.6. Политика Email 3

1.7. Зеркалирование 3

1.8. Замечания для переводчиков 3

1.9. Копирайт и распространение 3

**2. Что такое сокет? 5**

2.1. Два типа интернет сокетов 5

2.2. Низкоуровневый Вздор и Теория сетей 6

**3. IP адреса, структуры и повреждение данных 9**

3.1. IP адреса, версии 4 и 6 9

3.2. Порядок байт 11

3.3. Структуры 12

3.4. IP адреса, Часть Вторая 15

**4. Прыжок из IPv4 в IPv6 17**

**5. Системные вызовы или Облом 18**

5.1. getaddrinfo() - К старту - товсь! 18

5.2. soket() - Получи дескриптор файла 21

5.3. bind() - На каком я порте? 22

5.4. connect() - Эй, вы там! 23

5.5. listen() - Позвони мне, позвони… 24

5.6. accept() - “Спасибо за звонок на порт 3490.” 25

5.7. send() and recv() - Поговори со мною, бэби! 26

5.8. sendto() и recvfrom() - Поговори со мной, DGRAM-стиль 27

5.9. close() и shutdown() - Прочь с глаз моих! 28

5.10.getpeername() - Кто вы? 28

5.11.gethostname() - Кто Я? 29

**6. Архитектура Клиент-Сервер 30**

6.1. Простой потоковый сервер 30

6.2. Простой потоковый клиент 33

6.3. Дейтаграммные сокеты 34

**7. Немного продвинутая техника 38**

7.1. Блокировка 38

7.2. select()—Мультиплексирование синхронного ввода/вывода 38

7.3. Обработка незавершённых send() 44

7.4. Сериализация - Как упаковать данные 45
7.5. Дитя Инкапсуляции Данных 53
7.6. Широковещательные пакеты - Hello, world! 55

**8. Общие вопросы 59**

**9. Man Страницы 65**
**9.1. accept() 66**
**9.2. bind() 68**
**9.3. connect() 70**
**9.4. close() 72**
**9.5. getaddrinfo(), freeaddrinfo(), gai_strerror() 73**
**9.6. gethostname() 76**
**9.7. gethostbyname(), gethostbyaddr() 77**
**9.8. getnameinfo() 80**
**9.9. getpeername() 81**
**9.10. errno 82**
**9.11. fcntl() 83**
**9.12. htons(), htonl(), ntohs(), ntohl() 84**
**9.13. inet_ntoa(), inet_aton(), inet_addr 86**
**9.14. inet_ntop(), inet_pton() 87**
**9.15. listen() 89**
**9.16. perror(), strerror() 90**
**9.17. poll() 91**
**9.18. recv(), recvfrom() 93**
**9.19. select() 95**
**9.20. setsockopt(), getsockopt() 97**
**9.21. send(), sendto() 99**
**9.22. shutdown() 101**
**9.23. socket() 102**
**9.24. struct sockaddr сотоварищи 103**

**10. Дополнительные ссылки 105**
10.1.Книги 105
10.2.Web ссылки 105
10.3.RFC 106

**Предметный указатель 108**

# 1. Введение

Люди! Программирование сокетов вас убивает? Слишком трудно выискать что-то в **man** страницах? Вы хотите писать крутые Интернет программы, но у вас нет времени продираться сквозь завалы **struct**-ур пытаясь определить нужно ли вам вызывать **bind()** перед **connect()** и т.д. и т.п.?

Догадываетесь! Я уже сделал эту грязную работу и умираю от желания поделиться со всеми! Вы попали в нужное место. Этот документ должен дать C-программисту средней руки край, за который ему(ей) можно ухватиться в этом сетевом сумбуре.

И поверьте, я полностью увлечен будущим (и настоящим тоже!) и обновил это руководство для IPv6! Наслаждайтесь!

## 1.1. Для кого

Этот документ был написан как учебное пособие, а не как полное описание. Может быть лучше всего его читать людям, только начинающим программировать с сокетами, чтобы обрести точку опоры. В любом случае это не *полное* и *исчерпывающее* описание программирования сокетов.

Надеюсь, однако, что этого будет достаточно, чтобы начать чувствовать… :-)

## 1.2. Платформа и компилятор

Код, приведенный в этом документе, проверен на Linux PC с компилятором GNU **gcc**. Однако он должен строиться на любой платформе, использующей **gcc**. Естественно, он неприменим, если вы программируете для Windows. Смотрите раздел *1.5. Для программистов Windows.*

## 1.3. Официальная страница и книги

Официальное расположение этого документа http://beej.us/guide/bgnet/. Здесь вы также найдёте примеры кода и переводы на разные языки.

Чтобы купить прекрасно переплетённые печатные копии (их называют “книгами”) посетите http://beej.us/guide/url/bgbuy. Я буду признателен, поскольку это поможет поддержать мой книгописательский стиль жизни.

## 1.4. Для программистов Solaris/SunOS

Компилируя для Solaris или Sun OS вам нужно указать дополнительные ключи в командную строку для включения правильных библиотек. Для этого просто добавьте

“`-lnsl -lsocket -lresolv`” в конец команды компилятора. Как здесь:

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

Если всё равно получаете ошибки, можете попробовать добавить ещё “`-lxnet`” в конец этой строки. Я точно не знаю, что это делает, но некоторым людям кажется, что так надо.

Ещё одно место, где могут обнаружиться проблемы, это вызов **setsockopt()**. Прототип отличается от моего на Linux-е, так что вместо

```
int yes=1;
```

введите

```
char yes='1';
```

Поскольку у меня нет Sun, я приведённую выше информацию не проверял. Это люди просто сказали мне по email.

## 1.5. Для программистов Windows

Вот здесь, исторически, я с удовольствием немного попеняю на Windows просто потому что я её очень не люблю. Но я должен быть честным и сказать вам, что у Windows громадная база установок и она совершенно прелестная операционная система.

Говорят разлука заставляет сердце любить сильнее и я верю, что это так. (А может это возраст). Но после десяти с гаком лет неиспользования Windows в моей работе я могу сказать. Я намного счастливее! Так что я могу присесть и спокойно сказать "Конечно, пользуйтесь Windows!" Ладно, стисну зубы и скажу.

Так что я до сих пор поддерживаю ваши попытки использовать вместо этого Linux[2], BSD[3] или какую-нибудь Unix.

Но люди любят то, что они любят, и Windows народ с удовольствием узнает, что в общем виде эта информация пригодна и для них, конечно с небольшими изменениями.

Что вы можете круто сделать, это установить Cygwin[4], что есть набор Unix инструментов для Windows. За бокалом вина я слышал, что это позволяет всем нашим программам компилироваться незменёнными.

Но некоторые из вас могут возжелать делать всё в Чисто Windows Стиле. Это очень отважный поступок и вам надо немедленно бежать и брать Unix. Нет, нет, я шучу! Я хочу немного побыть дружественным(…нее) к Windows…

Вот что вам нужно сделать (кроме установки Cygwin!): первое, игнорировать значительное множество упомянутых мною системных заголовочных файлов. Вам нужно включить только

```
#include <winsock.h>
```

Подождите! Вам также нужно вызвать **`WSAStartup()`** перед тем как использовать что-либо в библиотеке сокетов. Код может выглядеть примерно так:

```
#include <winsock.h>
{
    WSADATA wsaData;            // если это не работает
    //WSAData wsaData;          // попробуйте так
    // MAKEWORD(1,1) for Winsock 1.1, MAKEWORD(2,0) для Winsock 2.0:
    if (WSAStartup(MAKEWORD(1,1), &wsaData) != 0) {
    fprintf(stderr, "WSAStartup failed.\n");
    exit(1);
}
```

Вам также нужно сказать компилятору компоновать с библиотекой Winsock, обычно называемой *`wsock32.lib`*, или *`winsock32.lib`*, или *`ws2_32.lib`* для Winsock 2.0. Под VC++ это можно сделать в меню **Project** под **Settings**…. Кликните пункт **Link** и ищите окошко "**Object/library modules**". Добавьте в этот список "*`wsock32.lib`*" (или что вы там предпочитаете).

Или я так слышал?

В конце, после всей работы с библиотекой сокетов вам нужно вызвать **`WSACleanup()`**. Подробности смотрите в **online help**.

Когда вы это сделали, то остальные примеры в этом пособии в общем должны работать, за некоторыми исключениями. Первое, вместо **`close()`** нужно использовать **`closesocket()`**. Кроме того, **`select()`** работает только с дескриптором сокета, а не с дескриптором файла (типа **`0`** для **`stdin`**).

---

[2] http://www.linux.com/

3 http://www.bsd.org/

[4] http://www.cygwin.com/

Также вы можете использовать класс **CSocket**. За информацией обращайтесь в страницы помощи компилятора.

Информацию по Winsock читайте в Winsock FAQ[5] и выходите отсюда.

Последнее, я слышал, что в Windows нет системного вызова **fork()**, который я, к сожалению, использовал в некоторых примерах. Может быть нужно подключить библиотеку POSIX или что-то ещё чтобы он работал, или вы можете использовать **CreateProcess()**. У вызова **fork()** нет аргументов, а **CreateProcess()** имеет около 48 миллиардов аргументов. Если вы до этого не доросли, то **CreateThread()** немного легче переваривать, но, к сожалению дискуссия о многопоточности выходит за рамки этого документа. Вы же понимаете, я могу только разглагольствовать об этом!

## 1.6. Политика Email

Вообще-то я могу помочь с вопросами по email, так что пишите, не стесняйтесь, но я не могу гарантировать ответа. Я живу весьма насыщенной жизнью и бывают времена, когда я не могу отвечать на ваши вопросы. В этом случае я удаляю послания. Ничего личного, просто у меня не всегда есть время для детального ответа на ваш запрос.

Как правило, чем сложней вопрос, тем маловероятней я отвечу. Если вы сумеете сузить вопрос и включить в него дополнительную информацию (как платформа, компилятор, полученные сообщения об ошибках и всё, что по вашему мнению может мне помочь), то тем вероятнее получите ответ. Подсказки прочтите в ESR документе How To Ask Questions The Smart Way[6].

Если вы не получили ответа поковыряйтесь ещё немного, попробуйте найти решение и если оно до сих пор не найдено, напишите мне снова, приложив найденную информацию, и надеюсь, её будет достаточно чтобы я вам помог.

И когда я уже извёл вас как писать и не писать мне, хотелось бы сказать, что я очень благодарен за всю хвалу документу, полученную мной за эти годы. Это настоящая моральная поддержка и мне радостно услышать, что он был использован на дело добра! :-) Спасибо!

## 1.7. Зеркалирование

Зеркалирование этого сайта более чем приветствуется, и публичное и приватное. Если вы захотите публично зеркалировать сайт и иметь ссылку на него на главной странице, черкните мне на beej@beej.us.

## 1.8. Замечания для переводчиков

Если вы хотите перевести руководство на другой язык напишите мне на beej@beej.us и я поставлю ссылку на ваш перевод на главной странице. Не смущайтесь добавить ваше имя и контакты в перевод.

Пожалуйста, обратите внимание на лицензионные ограничения в разделе *1.9. Копирайт и распространение* ниже.

Если вы хотите разместить перевод у меня, просто попросите. Я также поставлю на него ссылку. Любой способ замечателен.

## 1.9. Копирайт и распространение

Beej's Guide to Network Programming is Copyright © 2012 Brian “Beej Jorgensen” Hall.

---

[5] http://tangentsoft.net/wskfaq/

[6] http://www.catb.org/~esr/faqs/smart-questions.html

За особыми исключениями для исходного кода ниже и переводов эта работа лицензирована Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License.

Чтобы посмотреть копию этой лицензии посетите http://creativecommons.org/licenses/by-nc-nd/3.0/ или напишите в Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Одно особое исключение для части “No Derivative Works” лицензии таково: это руководство может быть свободно переведено на любой язык при соблюдении точности и полноты перевода. Одинаковые ограничения налагаются и на перевод и на исходное руководство. Перевод может также включать имя и контактную информацию переводчика.

Представленный в этом документе исходный С код предоставляется в публичное пользование и полностью свободен от каких-либо лицензионных ограничений.

Преподаватели могут свободно рекомендовать или предоставлять копии этого руководства своим студентам. Для информации свяжитесь с beej@beej.us.

# 2. Что такое сокет?

Вы всё время слышите разговоры о “сокетах” и наверное даже удивляетесь что же они такое? А они это способ общения с другими программами с использованием стандартных дескрипторов файлов Unix.

Что?

Хорошо, может быть вы слышали утверждение некоторых хакеров от Unix, “Здорово! В Unix *всё* - это файл!”. Всё, о чём они говорят, это факт, когда Unix-программа выполняет любую операцию ввода/вывода, она делает это чтением или записью в дескриптор файла. Дескриптор файла это обыкновенное целое число, связанное с открытым файлом. Но (и здесь прикол), этим файлом может быть сетевое подключение, FIFO, конвейер, терминал, реальный файл на диске и всё что угодно. В Unix всё *есть* файл! Так что лучше поверьте, если вы захотите связаться с другой программой через Интернет, это надо делать через дескриптор файла.

“Где мне взять этот дескриптор файла, мистер Умник?” сейчас это, наверное, последний вопрос у вас на уме, но я на него отвечу. Вы вызываете системную программу **`socket()`**. А она возвращает дескриптор сокета и вы связываетесь через него, используя специальные вызовы сокета **`send()`** и **`recv()`** (**man send**, **man recv**).

Вы можете воскликнуть - “Постойте! Если это дескриптор файла, то почему, именем Нептуна, я не могу воспользоваться для связи нормальными вызовами **`read()`** и **`write()`**?” Короткий ответ - “Можете!”. Ответ подлиннее - “Можете, но **`send()`** и **`recv()`** предоставляют намного больше возможностей управления передачей данных”.

Что дальше? Как насчёт такого: сокеты бывают самые разные. Есть DARPA Интернет Адреса (Internet Sockets), путевые имена в локальном узле (Unix Sockets), CCITT X.25 адреса (X.25 Sockets которые вы можете спокойно игнорировать) и множество других в зависимости от вида используемой Unix. Этот документ касается только первого вида: интернет сокетов.

## 2.1. Два типа интернет сокетов

Что такое? Есть два типа интернет сокетов? Да. Ладно, нет, вру. Существует множество типов, но я не хочу вас пугать. Я собираюсь здесь говорить только о двух из них. Помимо этого я хочу сказать, что “Сырые Сокеты” (Raw Sockets) также очень мощные и вы должны их просмотреть.

Хорошо, начинаем. Что за два типа? Один это “Потоковые сокеты” (Stream Sockets), другой - “Дейтаграммные сокеты” (Datagram Sockets), на которые можно ссылаться как “`SOCK_STREAM`” и “`SOCK_DGRAM`”, соответственно. Дейтаграммные сокеты иногда называют “неподключаемыми” (без установки логического соединения), хотя если очень хочется их можно подключить **`connect()`**-ом. (См. ниже).

Потоковые сокеты это надёжные подключаемые двунаправленные потоки связи. Если вы отправите в сокет два послания в порядке “1, 2”, то на другой стороне они также появятся в порядке “1, 2”. Они также будут свободными от ошибок (error-free). Я в этом так уверен, что могу заткнуть себе уши пальцами и напевать *ля-ля-ля* если кто-нибудь будет утверждать обратное.

Что использует потоковые сокеты? Ну, вы наверное слышали о **telnet** приложениях, да? Они используют потоковые сокеты. Все напечатанные символы должны появиться в том же порядке, правда? Также web браузеры используют HTTP протокол, который пользуется потоковыми сокетами для загрузки страниц. Действительно, если

подключиться к web сайту по **telnet**-у, напечатать “GET / HTTP/1.0” и дважды нажать RETURN, он вывалит вам HTML!

Как потоковые сокеты достигают столь высокого качества передачи данных? Они используют протокол, называемый “The Transmission Control Protocol”, иначе известный как “TCP” (см. RFC 793[7] с полной информацией по TCP). TCP обеспечивает появление ваших данных последовательно и без ошибок. Может быть ранее вы встречали “TCP” как первую часть “TCP/IP”, где “IP” означает “Internet Protocol” (см. RFC 791[8]). IP изначально работает с Интернет маршрутизацией и обычно не отвечает за целостность данных.

Круто. Как насчёт Дейтаграммных сокетов? Почему их называют неподключаемыми? В чём в конце концов дело? Почему они ненадёжны? Ну, есть несколько фактов: если вы посылаете дейтаграмму, она может и появиться. Она может появиться не в нужном порядке, но если появилась, то данные в пакете будут переданы без ошибок.

Дейтаграммные сокеты тоже используют IP маршрутизацию, но они не используют TCP. Они используют “User Datagram Protocol” или “UDP” (см. RFC 768[9]).

Почему они неподключаемые? Ну, это в основном потому что вы не открываете точку подключения как при потоковых сокетах. Вы только строите пакет, пришлёпываете к нему IP заголовок с адресом назначения и отсылаете. Подключения не требуются. Они обычно нужны когда недоступен TCP стек или когда потеря нескольких пакетов здесь и там не означают конца Вселенной. Примеры приложений: **tftp** (trivial file transfer protocol, младший брат FTP), **dhcpcd** (DHCP клиент), командные игры, потоковый звук, видеоконференции и т.д.

“Минуточку! **tftp** и **dhcpcd** используются для передачи двоичных приложений от одного хоста к другому! Данные не могут быть утеряны если хотите получить работающее приложение! Что за тёмная магия?”

Ладно, друзья мои человеки, **tftp** и подобные приложения имеют свой собственный протокол поверх UDP. Например, **tftp** протокол говорит приёмнику, что на каждый посылаемый пакет он должен послать обратно пакет, говорящий “Я получил!” (“ACK” пакет). Если отправитель не получит ответа, скажем, течении пяти секунд, он повторяет передачу пока в конце концов не получит ACK. Эта процедура подтверждения очень важна при создании надёжных SOCK_DGRAM приложений.

Для ненадёжных приложений как игры, звук, видео вы просто игнорируете потерянные пакеты или возможно пытаетесь по-умному компенсировать их. (Quake игроки знают этот эффект под техническим термином *проклятая задержка*. Слово “проклятая” в этом случае имеет чрезвычайно богохульное значение).

Почему мы должны использовать ненадёжный протокол нижнего уровня? Причины две: скорость и скорость. Этот способ “выстрелил-забыл” быстрее, чем отслеживать, что прибыло, в каком порядке и всё такое. Если вы посылаете в чат сообщение, TCP великолепен, но если по 40 обновлений местоположения персонажей в игровом мире в секунду, то не всё ли равно если один или два потеряются. Здесь UDP хороший выбор.

## 2.2. Низкоуровневый Вздор и Теория сетей

До этого я только упомянул об уровнях протоколов, теперь пора поговорить о том как сети в действительности работают и показать несколько примеров как построены UDP пакеты. Практически вы можете пропустить этот раздел, однако это хорошая основа.

---

[7] http://tools.ietf.org/html/rfc793

[8] http://tools.ietf.org/html/rfc791

[9] http://tools.ietf.org/html/rfc768

**Инкапсуляция данных.**

Ребята, пора узнать об *Инкапсуляции Данных*! Это очень, очень важно. Это настолько важно, что вы сможете узнать о ней только если пройдёте сетевой курс здесь, в Chico State ;-). По существу это означает следующее: пакет рождён, пакет завёрнут ("инкапсулирован") в заголовок (реже и хвостик) протоколом первого уровня (скажем, TFTP), затем вся штучка (включая TFTP заголовок) инкапсулируется следующим протоколом (скажем, UDP), затем снова следующим (IP), затем снова протоколом аппаратного (физического) уровня (скажем, Ethernet).

Когда другой компьютер принимает пакет, аппаратура обрывает Ethernet заголовок, ядро обрывает IP и UDP заголовки, TFTP программа обрывает TFTP заголовок и в итоге он имеет данные.

Наконец я могу рассказать о печально известной *Layered Network Model* (aka "ISO/OSI"). Эта Многоуровневая Сетевая Модель описывает систему сетевых функций, которая имеет множество преимуществ перед другими моделями. Например, вы можете написать такую же точно программу совершенно не заботясь о том, как данные физически передаются (последовательно, тонкий Ethernet, AUI, что угодно) потому что программа на нижнем уровне сделает это за вас. Реальное оборудование сети и топология прозрачны для программиста сокетов.

Без излишних хлопот я представлю уровни полнофункциональной модели. Запомните их для сетевых примеров:

- Приложений
- Представления
- Сессии
- Транспортный
- Сетевой
- Данных
- Физический

Физический Уровень это аппаратура (последовательный, Ethernet и т.д.) Уровень Приложений так далёк от физического как только можно себе представить - на этом уровне пользователь работает с сетью.

Эта модель так широка, что, если захотите, можете использовать её в ремонте автомобилей. Более совместимая с Unix модель уровней может быть такой:

- Уровень приложений (*telnet, ftp и т. д.*)
- Уровень транспорта хост-хост (*TCP, UDP*)
- Уровень интернета (*IP и маршрутизация*)
- Уровень доступа к сети (*Ethernet, wi-fi и т.п.*)

Сейчас вы, наверное, видите, как эти уровни соотносятся с оригинальными данными.

Видите как много труда уходит на построение простого пакета? Вот так! И вам нужно впечатать в заголовок пакета себя используя "**cat**"! Да шутя! Всё, что вам нужно с потоковыми сокетами это послать (**`send()`**) данные. С дейтаграммными сокетами вам нужно инкапсулировать пакет выбранным вами методом и послать (**`sendto()`**) его. Ядро построит Транспортный и Интернет уровни, а аппаратура исполнит Уровень доступа к сети. Ах, современная технология.

Так заканчивается моё краткое вторжение в теорию сетей. Да, я забыл сказать вам всё, что я хотел сказать о маршрутизации: ничего! Правильно, я не собираюсь говорить о ней вообще. Маршрутизатор обрывает пакет до IP адреса, консультируется со своей таблицей, бла-бла-бла. Посмотрите IP RFC[10] если вам совсем уж надо. А если вы никогда об этом не узнаете, прекрасно, будете жить.

---

[10] http://tools.ietf.org/html/rfc791

# 3. IP адреса, структуры и повреждение данных

В этой части игры мы поговорим о коде для пересадки.

Но сначала давайте ещё обсудим не-код! Так! Во-первых, я хочу чуточку поговорить об IP адресах и портах, чтобы всё разрешить. Затем мы поговорим о том как API сокетов хранит и обрабатывает IP адреса и данные.

## 3.1. IP адреса, версии 4 и 6

В старые добрые времена, когда Ben Kenobi ещё звался Obi Wan Kenobi, существовала чудная система маршрутизации сетей, именуемая The Internet Protocol Version 4, также названная IPv4. В ней были адреса, состоящие из четырёх байт (А.К.А. четырёх "октетов"), как правило записанных "цифрами и точками" вот так 192.0.02.111. Вам она, наверное, встречалась.

В действительности, фактически все сайты используют IPv4.

Все, включая Obi Wan-а, были счастливы. Всё было прекрасно, пока один ниспровергатель основ по имени Vint Cerf не предупредил всех, что мы почти исчерпали IPv4 адреса! (Помимо предупреждения о наступающем Апокалипсисе Рока и Мрака IPv4 Vint Cerf[11] также широко известен как Отец Интернета. Так что я не в том положении, чтобы критиковать его суждение.)

Исчерпали адреса? Как это может быть? Я имею в виду, что в 32-битном пространстве IPv4 существуют миллиарды IP адресов. У нас действительно есть миллиарды компьютеров?

Да.

Также, в начале, когда компьютеров было мало и все думали, что миллиард это невозможно большое число, некоторым крупным организациям были щедро выделены миллионы IP адресов для собственных нужд. (Таких как Xerox, MIT, Ford, HP, IBM, GE, AT&T, одной маленькой компании по названию Apple и далее по списку…)

Действительно, если бы не ряд временных мер, мы бы исчерпали его давным давно.

Но сейчас мы живём в эпоху, когда мы считаем, что лучше бы каждому человеку, каждому компьютеру, каждому калькулятору, каждому телефону, каждому парковочному счётчику и (почему бы и нет) каждому щенку иметь свой IP адрес.

Итак, IPv6 рождён. Поскольку Vint Cerf по всей вероятности бессмертен (даже если его физическая форма, небеса упаси, исчезнет, он, наверняка, уже существует как некий гипер-разум программы ELIZA[12] в глубинах Интернета 2), то никто не хочет снова услышать от него "Я же вам говорил", если у нас не будет хватать адресов в следующей версии Интернет Протокола.

Что это вам подсказывает?

Что нам нужно *намного* больше адресов. Что нам нужно не вдвое, не в миллиард, не в тысячу триллионов, а в 79 МИЛЛИОНОВ МИЛЛИАРДОВ ТРИЛЛИОНОВ раз больше возможных адресов! Мы им покажем!

Вы скажете: "Это правда? У меня есть все причины не верить большим числам." Хорошо, разница между 32 и 128 битами не звучит как *много*; это всего на 96 бит больше, правда? Но помните, мы обсуждаем степени: 32 бита представляют 4 миллиарда чисел ($2^{32}$), тогда как 128 бит представляют около 340 триллионов триллионов

[11] http://en.wikipedia.org/wiki/Vinton_Cerf

[12] http://en.wikipedia.org/wiki/ELIZA

триллионов чисел (в действительности $2^{128}$). Это как по миллиону IPv4 Интернетов для *каждой отдельной звезды во Вселенной*.

Забудьте также как выглядят цифры-и-точки в IPv4; теперь у нас шестнадцатиричное представление с разделёнными двоеточиями двухбайтовыми кусками, как здесь

```
2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551.
```

Это ещё не всё! Много раз у вас будут IP адреса с обилием нулей внутри. Их можно сжать между двумя двоеточиями, и вы можете опустить ведущие нули в каждой паре байт. Например, эти пары адресов эквивалентны:

```
2001:0db8:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51
```

```
2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::
```

```
0000:0000:0000:0000:0000:0000:0000:0001
::1
```

Адрес ::1 это *loopback* адрес. Он всегда означает “эта машина, на которой я сейчас работаю”. В IPv4 *loopback* адрес равен 127.0.0.1.

Для концовки, у IPv6 адресов есть мостик совместимости с IPv4, по которому можно пройти. Если вы, например, хотите представить IPv4 адрес 192.0.2.33 как IPv6 адрес используйте следующую нотацию:”::ffff:192.0.2.33”.

Мы тут серьёзно шутим.

Действительно, это такая серьёзная шутка, что Создатели IPv6 весьма благородно урезали триллионы и триллионы адресов на резервные нужды, но у нас их так много, что, откровенно, кто их сочтёт. Их осталось с избытком на каждого мужчину, женщину, ребёнка, щенка и паркомата на каждой планете в галактике. И поверьте мне, на каждой планете галактики есть паркоматы. Это правда.

### 3.1.1.Подсети

По организационным причинам иногда удобно объявить, что “эта первая часть IP адреса вплоть до этого бита является *сетевой частью* IP адреса, а остальная это *часть хоста*”.

Например с IPv4 адресом 192.0.2.12 можно сказать, что первые три байта это сеть, а последний это хост, Или другими словами мы имеем дело хостом 12 в сети 192.0.2.0 (заметьте, что мы обнулили байт хоста).

А теперь добавим устаревшей информации! Готовы? В Стародавние Времена были “классы” подсетей, где один, два или три байта были сетевой частью. И если вы были достаточно удачливы чтобы иметь один байт для сети и три для хостов, вы могли иметь 24 бит-значное число хостов с вашей сети (24 миллиона или около того). Это была сеть класса A. С другого конца был класс C с тремя байтами сети и одним байтом хостов (256 хостов минус парочка резервных).

Как видите, было немного классов A, громадная куча классов C и сколько-то классов B посерёдке.

Сетевая порция определяется так называемой *сетевой маской*, которой вы поразрядным И с IP адресом выделяете номер сети. Обычно сетевая маска выглядит

подобным образом 255.255.255.0. (Например, если ваш IP равен 192.0.2.12, то 192.0.2.12 & 255.255.255.0 дают 192.0.2.0).

К сожалению оказалось, что этого недостаточно для повседневных нужд Интернета на низком уровне. Мы очень быстро исчерпали сети класса С, почти исчерпали класс А и даже негде попросить. Чтобы исцелить от этого Силы Небесные позволили сетевой маске иметь переменное число бит, а не только 8, 16 или 24. Так что вы можете иметь маску, скажем, 255.255.255.252, в которой 30 бит это сеть, а 2 бита номера хоста позволяют иметь до 4 хостов в сети. (Заметьте, что сетевая маска это ВСЕГДА группа 1 с последующей группой 0).

Но использовать длинную строку чисел как 255.192.0.0 несколько неуклюже. Во-первых, люди интуитивно не могут понять сколько это бит, и, во-вторых, это действительно не компактно. И пришёл Новый Стиль, и было это много удачней. Просто добавьте в конец IP адреса через косую черту десятичное число бит номера сети. Вот так: 192.0.2.12/30.

Или для IPv6 примерно так: 2001:db8::/32 или 2001:db8:5413:4028::9db9/64.

### 3.1.2. Номера портов

Если вы благожелательно вспомните, ранее я показал вам Многоуровневую Сетевую Модель, где уровень Интернета (IP) отделён от транспортного уровня хост-хост (TCP, UDP). Разогрейтесь перед следующим параграфом.

Получается, что кроме IP адреса (использованного IP уровнем) есть дополнительный адрес, которым пользуются TCP (потоковые сокеты) и по случайному стечению обстоятельств UDP (дейтаграммные сокеты). Это *номер порта*. Это 16-ти разрядное число, что-то вроде местного адреса для соединений.

Думайте об IP как об адресе отеля, а о номере порта, как номере комнаты. Это неплохая аналогия, может быть позже я приведу что-нибудь из автомобильной промышленности.

Говорите вы хотите чтобы компьютер обрабатывал входящую почту и web сервисы? Как различить их на компьютере с одним IP адресом?

Что ж, различные интернет сервисы имеют различные хорошо известные номера портов. Их можно найти в Big IANA Port List[13] или на Unix в файле */etc/services.* HTTP отведён 80, telnet - 23, SMTP - 25, игра DOOM[14] использует порт 666 и т.д. и т.п. Порты до 1024 часто рассматриваются как особые и обычно требуют привилегий ОС.

Вот так!

## 3.2. Порядок байт

По Указу Королевства! Да будет двухбайтовый порядок, отныне именуемый как Великий и Хромой!

Я шучу, но один ведь лучше другого. :-)

В действительности сказать это непросто, и я лишь сболтнул. Ваш компьютер может тайком хранить байты в другом порядке. Я то знаю, но никто не захотел вам это сказать!

Дело в том, что все в Интернет мире сообща договорились, что если вы хотите представить двухбайтное шестнадцатиричное число, скажем b34f, вы храните его в двух последовательных байтах b3 и следом 4f. Складывается ощущение, что, как сказал Wilford

[13] http://www.iana.org/assignments/port-numbers

[14] http://en.wikipedia.org/wiki/Doom_(video_game)

Brimley[15], Так Надо Делать. Это число, сохраняемое сначала старшими цифрами называется *Big-Endian*.

К сожалению, некоторое число разбросанных по миру компьютеров, а именно, имеющих Intel или Intel-совместимые процессоры, хранят данные наоборот, так что число b34f, будет храниться в памяти сначала 4f, затем b3. Этот метод хранения называется *Little-Endian*.

Подождите, я ещё не закончил с терминологией! Более вменяемый *Big-Endian* также называется Порядком Байтов Сети (*Network Byte Order*) потому что такой порядок используется сетью.

Ваш компьютер хранит числа в Порядке Байтов Хоста (*Host Byte Order*). Если у вас Intel 80x86, то это *Little-Endian*. Если у вас Motorola 68k, то это *Big-Endian.* Если у вас *PowerPC*, то порядок байт… ладно, зависит!

Каждый раз, строя пакет или заполняя структуры вам нужно быть уверенным, что ваши двух и четырёх байтовые числа построены с Порядке Байтов Сети. Но как это сделать, если вы не знаете порядка байт вашего хоста?

Хорошие вести! Вы просто предполагаете, что порядок байт хоста неправильный, и всегда прогоняете данные через функции установки порядка байт сети. Эта функция делает волшебное преобразование, если делает, и таким образом ваш код становится переносимым на машины с другим порядком байт.

Чудненько. Есть два типа чисел, которые вы можете преобразовать, короткие (два байта) и длинные (четыре байта). Эти функции также работают с беззнаковыми числами. Скажем, вы хотите преобразовать короткое целое (short) из Порядка Байтов Хоста (Host Byte Order) в Порядок Байтов Сети (Network Byte Order). Начинаем с “h” для “host”, дополняем “to”, затем “n” для “network”, и “s” для “short”: h-to-n-s, или **`htons()`** (читаем: “**H**ost **to** **N**etwork **S**hort”).

Это почти слишком легко…

Можно использовать любую комбинацию “n”, “h”, “s”, и “l” кроме действительно глупых. Например, функции `stolh()` (“Short to Long Host”) нет, ни здесь, ни где-нибудь ещё.

Но есть:

```
htons()    host to network short
htonl()    host to network long
ntohs()    network to host short
ntohl()    network to host long
```

В основном, вам захочется преобразовать числа в Порядок Байтов Сети перед тем, как послать их по проводам, и в Порядок Байтов Хоста, когда они оттуда придут.

Я не знаю, как насчёт 64-разрядных вариантов, извините. И если вам захочется поработать с плавающей запятой, обратитесь в раздел *Сериализации* много ниже.

Подразумевается, что числа в этом документе имеют Порядок Байтов Хоста, если не скажу обратного.

## 3.3. Структуры

Наконец-то мы здесь. Пора поговорить о программировании. В этом разделе я охвачу различные типы данных, применяемых в интерфейсе сокетов, поскольку некоторые их них по-настоящему тяжелы для описания.

Сначала простой: дескриптор сокета. Он имеет тип

```
int
```

[15] http://en.wikipedia.org/wiki/Wilford_Brimley

Просто обычный int.

Отсюда всё становится таинственным, так что просто читайте и терпите вместе со мной.

Моя Первая Структура™—struct addrinfo.

Эта структура более позднее изобретение и используется для подготовки адресных структур сокета для дальнейшего использования. Она также используется для поиска имён хоста и службы. Большее понимание придёт позже, когда мы подойдём к реальному использованию, а сейчас просто знайте, что это одна из первых вещей, вызываемых при создании соединения.

```
struct addrinfo {
    int              ai_flags;       // AI_PASSIVE, AI_CANONNAME,  т.д.
    int              ai_family;      // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;    // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;    // используйте 0 для"any"
    size_t           ai_addrlen;     // размер ai_addr в байтах
    struct sockaddr  *ai_addr;       // struct sockaddr_in или _in6
    char             *ai_canonname;  // полное каноническое имя хоста
    struct addrinfo  *ai_next;       // связанный список, следующий
};
```

Вы немного загружаете эту структуру и вызываете **getaddrinfo()**. Она возвращает указатель на новый связанный список этих структур, содержащих всё, что вам надо.

Вы можете приказать использовать IPv4 или IPv6 в поле ai_family или оставить AF_UNSPEC чтобы использовать любой. Это круто, поскольку ваш код сможет быть независимым от версии IP.

Заметьте, что это связанный список: ai_next указывает на следующий элемент - результатов может быть несколько, есть из чего выбирать. Я использую первый рабочий, а у вас могут быть другие нужды. Я же всего не знаю, чувак!

Как видите поле ai_addr в структуре addrinfo это указатель на структуру sockaddr. Вот мы и полезли во внутренности структур IP адреса.

Обычно вам нет нужды заполнять эти структуры, гораздо чаще достаточно вызвать **getaddrinfo()** и всё нужное там. Однако, вы *будете* всматриваться во внутренности этих структур, чтобы получить их значения, так что я их здесь представляю.

(Кроме того, весь код, написанный до изобретения структуры addrinfo, вставлял все эти принадлежности вручную, так что вы обнаружите много дурного кода IPv4, который именно так и поступает. И вы знаете, и в старых версиях этого руководства и в других!)

Некоторые структуры относятся к IPv4, некоторые к IPv6, некоторые к обоим. Я отмечу что есть что. В любом случае, структура sockaddr содержит адресную информацию для многих типов сокетов.

```
struct sockaddr {
    unsigned short sa_family;      // семейство адресов, AF_xxx
    char           sa_data[14];    // 14 байт адреса протокола
};
```

*sa_family* может быть разнообразной, но будет либо AF_INET (IPv4) или AF_INET6 для всего, что мы делаем в этом документе.

*sa_data* содержит адрес назначения и номер порта для сокета. Это весьма громоздко, но ведь вы не хотите утомительно упаковывать адрес в *sa_data* вручную.

Для работы со структурой sockaddr программисты создают параллельную структуру struct sockaddr_in (“in” это “Internet”) для использования с IPv4.

И *что важно*, указатель на структуру sockaddr_in может быть приведен к указателю на структуру sockaddr и наоборот. Так что даже хоть **connect()** и требует struct

sockaddr* вы всё равно можете пользоваться структурой sockaddr_in и привести её в последний момент!

```
// (Только для IPv4 – для IPv6 смотри struct sockaddr_in6)

struct sockaddr_in {
    short int           sin_family;     // Семейство адресов, AF_INET
    unsigned short int  sin_port;       // Номер порта
    struct in_addr      sin_addr;       // Интернет адрес
    unsigned char       sin_zero[8];    // Размер как у struct sockaddr
};
```

Эта структура позволяет обращаться элементам адреса сокета. Обратите внимание, что *sin_zero*, который включён для расширения длины структуры до длины sockaddr, должен быть обнулён функцией **memset()**. Также заметьте, что *sin_family* соответствует *sa_family* структуры sockaddr и должен быть установлен в “AF_INET”. Напоследок, должен быть в *Порядке Байтов Сети* (используйте **htons()**!)

Копнём глубже! Поле *sin_addr* это структура in_addr. Что это значит? Не будем излишне драматичны, но это одна из самых пугающих структур **union** всех времён:

```
// (Только для IPv4 – для IPv6 смотри struct sockaddr_in6_addr)

// Интернет адрес (исторически обоснованная структура)
struct in_addr {
    uint32_t        s_addr;         // это 32-битный int (4 байта)
};
```

Тпру! Ей положено быть union, но, похоже, эти дни прошли. Скатертью дорожка. Так что, если вы объявили *ina* типа struct sockaddr_in, то *ina.sin_addr.s_addr* ссылается на 4-байтный IP адрес (в Порядке Байтов Сети). Отметим, что даже если ваша система до сих пор использует богопротивный union для структуры in_addr, вы всё равно можете ссылаться на 4-байтный IP адрес так как я сделал это выше (это благодаря #define-ам).

Как насчёт IPv6? Подобные структуры существуют и для него:

```
// (Только для IPv6 –- для IPv4 смотри struct sockaddr_in и struct in_addr)

struct sockaddr_in6 {
    u_int16_t           sin6_family;        // семейство адресов, AF_INET6
    u_int16_t           sin6_port;          // номер порта,  Порядок Байтов Сети
    u_int32_t           sin6_flowinfo;      // потоковая информация IPv6
    struct in6_addr sin6_addr;              // адрес IPv6
    u_int32_t       sin6_scope_id;          // Scope ID
};
struct in6_addr {
    unsigned char       s6_addr[16];        // адрес IPv6
}
```

Отметим, что IPv6 имеет адрес и номер порта также, как IPv4.

Также я пока не собираюсь говорить о полях *flow information* и *Scope ID* IPv6… это ведь пособие для начинающих. :-)

И последнее, но не совсем, есть ещё одна простая структура *struct sockaddr_storage*, созданная достаточно большой, чтобы содержать обе IPv4 и IPv6 структуры. Судя по некоторым вызовам вы не знаете наперёд каким адресом загружать вашу структуру sockaddr: IPv4 или IPv6. Так передайте эту параллельную структуру, подобную struct sockaddr, только больше, и приведите к нужному типу:

```
struct sockaddr_storage{
    sa_family_tss_family;       // семейство адресов
```

```
    // это выравнивание длины, зависит от реализации, проигнорируйте
    char        __ss_pad1[SS_PAD1SIZE];
    int64_t     __ss_align;
    char        __ss_pad2[SS_PAD2SIZE];
};
```

Важно, что в поле *ss_family* вы можете посмотреть это `AF_INET` или `AF_INET6`, и когда нужно привести её к `sockaddr_in` или `sockaddr_in6`.

## 3.4. IP адреса, Часть Вторая

К счастью для вас существует пачка функций, позволяющих манипулировать IP адресами. Рисовать их вручную и упаковывать в `long` оператором “<<“ нет нужды.

Допустим, у вас есть структура `struct sockaddr_in ina` и IP адрес “10.12.110.57” или “2001:db8:63b3:1::3490” который вы хотите в неё поместить. Вам нужно воспользоваться функцией **`inet_pton()`**, которая преобразует IP адрес в записи “цифры-точки” в структуру `struct in_addr` либо `struct in6_addr` в зависимости от указанных `AF_INET` или `AF_INET6`. (“pton” означает “presentation to network” или можете называть “printable to network”, если так проще запомнить. Преобразование можно сделать так:

```
struct sockaddr_in    sa;                         // IPv4
struct sockaddr_in6   sa6;                        // IPv6
inet_pton(AF_INET, “192.0.2.1”, &(sa.sin_addr));       // IPv4
inet_pton(AF_INET6, “2001:db8:63b3:1::3490”, &(sa6.sin6_addr));   // IPv6
```

(Быстрое примечание: старый способ использования функций **`inet_addr()`** и **`inet_aton()`** устарел и не работает с IPv6.)

И ещё, приведённый выше отрезок кода не очень надёжен, потому что здесь нет проверки на ошибки. Видите ли, **`inet_pton()`** возвращает -1 при ошибке и 0 если произошла какая-то путаница. Так что перед использованием убедитесь, что результат больше нуля!

Хорошо, теперь вы можете преобразовывать строковые IP адреса в их двоичное представление. Как насчёт других способов? Что если у вас есть `struct in_addr` и вы хотите напечатать её в представлении цифр-и-точек? (Или `struct in6_addr`, которая нужна в, ух, “шестнадцатирично-с-двоеточием” представлении.) В этом случае вам нужно воспользоваться функцией **`inet_ntop()`** (“ntop” означает “network to presentation” или можете называть “network to printable”, если так проще запомнить, как здесь:

```
// IPv4

char    ip4[INET_ADDRSTRLEN];          // место для строки IPv4
struct  sockaddr_in    sa;             // предположительно чем-то загружено

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

printf(“The IPv4 address is: %s\n”, ip4);


// IPv6

char    ip6[INET6_ADDRSTRLEN];         // место для строки IPv6
struct  sockaddr_in6   sa6;            // предположительно чем-то загружено

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf(“The IPv6 address is: %s\n”, ip6);
```

При вызове вы передаёте тип адреса (IPv4 или IPv6), адрес, указатель на строку для результата, максимальную длину этой строки. (Два макроса удобно определяют максимальный размер строки для адреса: `INET_ADDRSTRLEN` и `INET6_ADDRSTRLEN`.)

(Ещё одно быстрое замечание к упомянутым старым методам: историческая функция для такого преобразования **`inet_ntoa()`**. Она также устарела и не работает с IPv6.)

И напоследок, эти функции работают только с цифровыми IP адресами, но не с именами хостов для DNS серверов, как “www.example.com”. Для этого надо использовать **`getaddrinfo()`**, но об этом позднее.

### 3.4.1.Частные (или отключённые) сети

Множество мест имеют брандмауэры, скрывающие сети от остального мира своей собственной защитой. Очень часто брандмауэр транслирует “внутренние” IP адреса во “внешние” (известные всему миру) IP адреса с помощью процесса именуемого *Network Address Translation* или NAT.

Вы ещё нервничаете? “Куда он забрёл со всеми этими странными штуками?”

Ладно, расслабьтесь и купите себе безалкогольный (или алкогольный) напиток, поскольку, как начинающему, вам не нужно беспокоиться об NAT, он для вас прозрачен. Но я хотел поговорить о сетях за брандмауэром если вас начнут смущать увиденные сетевые номера.

Например, у меня дома есть брандмауэр. Я имею два статичных IPv4 адреса, выделенных мне DSL компанией, и ещё семь компьютеров в сети. Как это возможно? Два компьютера не могут иметь одинаковый адрес иначе данные не будут знать к какому им направляться!

Вот ответ: они не разделяют один адрес. Они находятся в частной сети с выделенными для неё 24 миллионами IP адресов. Они все только для меня. Вот так, они все для меня и больше никого не касаются. Вот что происходит:

Если я вхожу в удалённый компьютер, он говорит мне, что я вошёл с 192.0.2.33, публичного IP, который мне выделил мой провайдер. Но если я спрашиваю у моего локального компьютера его адрес, он отвечает 10.0.0.5. Кто транслирует один IP адрес в другой? Правильно, брандмауэр! Это делает NAT!

10.x.x.x одна из немногих зарезервированных сетей, которые используются в полностью отключённых сетях, либо за брандмауэрами. Доступные вам номера частных сетей описаны в RFC 1918[16], но наиболее часто вам встретятся 10.x.x.x и 192.168.x.x, где *x* принимает значения от 0 до 255. Менее распространены 172.y.x.x, где *y* стоит между 16 и 31.

Сетям за брандмауэром *не нужно* быть одной из этих зарезервированных сетей, но обычно так и есть.

(Забавный факт! Мой внешний IP адрес в действительности не 192.0.2.33. Сеть 192.0.2.x зарезервирована в качестве воображаемых “настоящих” IP адресов для использования в документации, такой как это пособие!)

IPv6 до известной степени тоже имеет частные сети. Они будут начинаться с fd*xx*: (или может быть в будущем fc*xx*:), как в RFC 4193[17]. NAT и IPv6 как правило не смешиваются, однако, (если только вы не делаете шлюз между IPv6 и IPv4, что лежит вне области рассмотрения этого документа) в теории у вас будет столько адресов, что NAT больше не понадобится. Но если вы хотите выделить для себя адреса в сети, которая не будет доступна снаружи, то вот как это делается.

---

[16] http://tools.ietf.org/html/rfc1918

[17] http://tools.ietf.org/html/rfc4193

# 4. Прыжок из IPv4 в IPv6

Но я просто хотел сказать вам что изменить в моём коде, чтобы он работал с IPv6! Скажите! Окей! Окей!

Почти всё здесь я проходил ранее, но это краткая версия для нетерпеливых. (Конечно, то больше, чем это, зато это применимо в данном пособии.)

1. Прежде всего, попробуйте воспользоваться **`getaddrinfo()`**, чтобы получить всю информацию структуры `sockaddr`, вместо того, чтобы заполнять её вручную. Это сделает вас независимым от версии IP и устранит множество последующих шагов.
2. Попробуйте любое написанное место, относящееся к версии IP, исполнить во вспомогательной функции.
3. Измените `AF_INET` на `AF_INET6`.
4. Измените `PF_INET` на `PF_INET6`.
5. Измените `INADDR_ANY` на `in6addr_any` , что несколько отличается:

```
struct sockaddr_in sa;
struct sockaddr_in6 sa6;
sa.sin_addr.s_addr = INADDR_ANY;   // используйте мой IPv4 адрес
sa6.sin6_addr = in6addr_any;       // используйте мой IPv6 адрес
```

   Также, используйте значение `IN6ADDR_ANY_INIT` как инициализатор при объявлении `struct in6_addr` :

```
struct in6_addr ia6 = IN6ADDR_ANY_INIT;
```

6. Вместо `struct sockaddr_in` используйте `struct sockaddr_in6`, обязательно добавив “6” в соответствующие поля (см. *3.3 Структуры* выше). Поля *`sin6_zero`* нет.
7. Вместо `struct in_addr` используйте `struct in6_addr`, обязательно добавив “6” в соответствующие поля (см. `struct`-s выше).
8. Вместо **`inet_aton()`** или **`inet_addr()`**, используйте **`inet_pton()`**.
9. Вместо **`inet_ntoa()`**, используйте **`inet_ntop()`**.
10. Вместо **`gethostbyname()`**, используйте лучшую **`getaddrinfo()`**.
11. Вместо **`gethostbyaddr()`**, используйте лучшую **`getnameinfo()`** (хотя **`gethostbyaddr()`** до сих пор работает с IPv6).
12. `INADDR_BROADCAST` больше не работает. Используйте широковещание IPv6.

*Et voila*!

# 5. Системные вызовы или Облом

В этом разделе мы приступаем к системным вызовам (и вызовам других библиотек), которые позволяют вам достичь сетевых возможностей Unix или любых иных систем, поддерживающих этот API сокетов (BSD, Windows, Linux, Mac и что-там-у-вас). Когда вы вызываете одну из этих функций, ядро берет власть и выполняет всю работу за вас автомагически.

Место, где большинство людей зависает, это порядок вызова. Как вы уже, наверное, обнаружили, **man** страницы бесполезны. Хорошо, чтобы помочь в этой ужасной ситуации, я попытался в последующих разделах расположить системные вызовы *точно* (примерно) в том порядке, в каком к ним надо обращаться в ваших программах.

Это, на пару с несколькими кусками примеров кода здесь и там, немного молока с печеньем (которым, я боюсь, вам надо будет себя снабжать), немного сырых потрошков и мужества и вы будете излучать данные в Интернет как Сын Джона Постела!

(Пожалуйста, заметьте, что во многие отрывки кода для краткости не включена необходимая проверка на ошибки. Подразумевается, что вызов **getaddrinfo()** успешен и возвращена правильная ссылка на связанный список. Так что используйте их как модель, хотя в отдельных программах они применены правильно.)

## 5.1. getaddrinfo() - К старту - товсь!

Это настоящая рабочая лошадка функций со множеством опций, хотя использовать её очень просто.

Крохотный кусочек истории. Обычно вам нужно было вызывать функцию **gethostbyname()** для DNS поиска. Затем вы вручную записывали полученную информацию в `struct sockaddr_in` и использовали её в вызовах.

Слава Богу, этого больше не требуется. (И даже нежелательно, если вы хотите писать код, работающий и с IPv4 и с IPv6!) В наши продвинутые времена у вас есть функция **getaddrinfo()**, которая делает для вас много хорошего, включая поиск имён DNS и служб и сверх того заполняет нужные вам структуры!

Давайте взглянем!

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
    int getaddrinfo(const char *node,       // например, "www.example.com" или IP
    const char *service,                    // например, "http" или номер порта
    const struct addrinfo *hints,
    struct addrinfo **res);
```

Вы передаёте этой функции три входных параметра и она возвращает указатель на связанный список результатов, *res*.

Параметр *node* это имя или IP адрес хоста, к которому надо подключиться.

Следующий параметр *service* может быть номером порта, вроде “80”, или именем отдельной службы (приведены в The IANA Port List[18] или файле */etc/services* вашей Unix машины) как “http”, “ftp”, “telnet”, “smtp” или любой другой.

Наконец, параметр *hints* указывает на `struct addrinfo`, которую вы уже заполнили нужной информацией.

Вот пример вызова если вы сервер, который хочет слушать порт 3490 вашего IP адреса. Заметим, что в действительности “слушания” или установки сети не происходит, просто заполняются структуры, которые мы используем позднее.

[18] http://www.iana.org/assignments/port-numbers

```
    int  status;
    struct addrinfo hints;
struct addrinfo *servinfo;                    // укажет на результат

    memset(&hints, 0, sizeof hints);          // очистка структуры
    hints.ai_family = AF_UNSPEC;              // IPv4 либо IPv6
    hints.ai_socktype = SOCK_STREAM;          // потоковый сокет TCP
    hints.ai_flags = AI_PASSIVE;              // записать мой IP для меня

    if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
        exit(1);
    }

    // servinfo теперь указывает на связанный список из 1 или более struct addrinfo
    // ... работайте пока не исчезнет надобность в servinfo ....
    freeaddrinfo(servinfo);                   // освободить связанный список
```

Заметьте, что я установил *ai_family* в AF_UNSPEC, указывая, что мне всё равно IPv4 или IPv6. Вы можете установить AF_INET или AF_INET6 если хотите использовать их отдельно.

Также вы видите флаг AI_PASSIVE, он говорит **getaddrinfo()**, что структурам сокета нужно назначить адрес моего локального хоста. Это прекрасно, покольку отныне вам не нужно его жёстко определять. (Или вы можете специальный адрес в первый параметр **getaddrinfo()**, где у меня сейчас NULL.)

Затем мы делаем вызов. Если есть ошибка (**getaddrinfo()** возвращает не-ноль), мы можем распечатать её, используя функцию **gai_strerror()**. Если всё работает правильно, *servinfo* будет указывать на связанный список структур struct addrinfo, каждая из которых содержит struct sockaddr определённого типа, которые мы можем использовать позднее! Ловко!

В итоге, когда мы наконец-то закончим работать со связанным списком, который **getaddrinfo()** так любезно нам предоставила, мы можем (и должны) освободить всё это, вызвав **freeaddrinfo()**.

Вот пример вызова если вы клиент, который хочет подсоединиться к определённому серверу, скажем, “www.example.net” порт 3490. Опять же это не настоящее подключение, а заполнение структур, которые мы используем позднее:

```
    int status;
    struct addrinfo hints;
    struct addrinfo *servinfo;                // укажет на результат

    memset(&hints, 0, sizeof hints);          // очистка структуры
    hints.ai_family = AF_UNSPEC;              // IPv4 либо IPv6
    hints.ai_socktype = SOCK_STREAM;          // потоковый сокет TCP

    // готовьтесь к соединению
    status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

    // servinfo теперь указывает на связанный список из 1 или более struct addrinfo
    // и т.д.
```

Я продолжаю говорить, что  это связанный список со всеми видами адресной информации. Давайте напишем быструю демо программу вывода этой информации. Эта

программа[19] будет печатать IP адрес любого хоста, который вы укажете в командной строке:

```
/*
** showip.c -- выводит IP адреса заданного в командной строке хоста
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
    struct addrinfo hints, *res, *p;
    int status;
    char ipstr[INET6_ADDRSTRLEN];
    if (argc != 2) {
        fprintf(stderr,"usage: showip hostname\n”);

        return 1;
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;            // AF_INET или AF_INET6 если требуется
    hints.ai_socktype = SOCK_STREAM;
    if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));

    return 2;
    }

printf("IP addresses for %s:\n\n", argv[1]);
for(p = res;p != NULL; p = p->ai_next) {
    void *addr;
    char *ipver;
            // получить,
            // в IPv4 и IPv6 поля разные:
    if (p->ai_family == AF_INET) {          // IPv4
        struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
        addr = &(ipv4->sin_addr);
        ipver = "IPv4";
    } else { // IPv6

        struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
        addr = &(ipv6->sin6_addr);
        ipver = "IPv6";
    }
            // перевести IP в строку и распечатать:
    inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
    printf("  %s: %s\n", ipver, ipstr);
}
freeaddrinfo(res);                          // освободить связанный список
return 0;
}
```

19 http://beej.us/guide/bgnet/examples/showip.c

Как видите, код передаёт **getaddrinfo()** всё, что вы укажете в командной строке, она заполняет связанный список, на который указывает *res*, и мы можем пройтись по нему, распечатывая содержимое или что-то ещё.

(Есть некоторое уродство в том, что нам нужно копаться в различных типах struct sockaddr в зависимости версии IP. Простите за это! Я не уверен, что можно лучше.)

Пример работает! Всем нравится распечатка:

```
$ showip www.example.net
 IP addresses for www.example.net:
  IPv4: 192.0.2.88
$ showip ipv6.example.com
 IP addresses for ipv6.example.com:
  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

Теперь у нас всё под контролем и мы передадим полученные от **getaddrinfo()** результаты другой функции сокета и, в конце концов, установим сетевое соединение! Продолжайте читать!

## 5.2. soket() - Получи дескриптор файла

Полагаю, откладывать больше нельзя, мне нужно рассказать о системном вызове **socket()**. Вот схема:

```
#include <sys/types.h>
#include <sys/socket.h>

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

Что это за аргументы? Они позволяют указать какой тип сокета вам нужен (IPv4 или IPv6, потоковый или дейтаграммный и TCP или UDP). Люди обычно жёстко устанавливают эти значения и вы можете поступить абсолютно так же. (*domain* ставится PF_INET или PF_INET6, *type* это SOCK_STREAM или SOCK_DGRAM и *protocol* может быть установлен в 0 для выбора правильного протокола для заданного типа. Или вы можете вызвать **getprotobyname()** и выбрать нужный протокол, “tcp” or “udp”).

(PF_INET это близкий родственник AF_INET, который вы используете при инициализации поля *sin_family* в вашей структуре sockaddr_in. Они настолько близкие родственники, что имеют одинаковое значение и многие программисты при вызове передают **socket()** в первом аргументе AF_INET вместо **PF_INET**. Теперь возьмите молока и печенья, поскольку настало время сказания. Однажды, давным-давно, людям представилось, что может быть семейство адресов (“AF” в “AF_INET”) сможет поддерживать несколько протоколов, определяемых их семейством протоколов (“PF” в “PF_INET”). Этого не случилось. И жили они долго и счастливо. Конец. Так что правильней всего использовать AF_INET в вашей struct sockaddr_in и PF_INET в вашем вызове **socket()**.)

В любом случае, хватит об этом. Что вам действительно нужно, так это взять данные из результатов вызова **getaddrinfo()** и передать их **socket()**, прямо как здесь:

```
int s;
struct addrinfo hints, *res;

// поиск
// [полагаем, что структура “hints" уже заполнена]
getaddrinfo("www.example.com", "http", &hints, &res);

// [нужно проверить выход getaddrinfo() на ошибки и просмотреть
// связанный список "res" на действительный элемент, не полагаясь
// на то, что это первый (как во многих других примерах.)]
```

```
// [Примеры смотри в разделе Клиент-Сервер.]

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

**socket()** просто возвращает вам *дескриптор сокета*, который вы можете позже использовать в системных вызовах или -1 в случае ошибки. Глобальная переменная ***errno*** содержит код ошибки (см. подробности в **man** странице ***errno*** и кратких заметкиах по использованию ***errno*** в многопоточных программах.)

Прекрасно, прекрасно, но что хорошего в этих сокетах? Само по себе - ничего, и вам нужно читать дальше и делать больше системных вызовов, чтобы хоть что-нибудь почувствовать.

## 5.3. bind() - На каком я порте?

Коли у вас есть сокет, вам может понадобиться связать его с портом на вашей локальной машине. (Так обычно делается если вы хотите слушать (**listen()**) входные подключения на специальном порте - сетевые игры делают так, когда говорят вам "подключитесь к 192.168.5.10 порт 3490".) Номер порта используется ядром при сравнении входящего пакета с дескриптором сокета конкретного процесса. Если вы собираетесь выполнить только **connect()** (поскольку вы клиент, а не сервер), это может быть ненужным. В любом случае читайте, просто для забавы.

Вот форма системного вызова **bind()**:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

*sockfd* это файловый дескриптор сокета, возвращенный **socket()**-ом.

*my_addr* это указатель на struct sockaddr, которая содержит информацию о вашем адресе, а именно, порт и IP address.

*addrlen* длина этого адреса в байтах.

Вот так так! Это же глотается одним куском. Давайте рассмотрим пример, в котором привяжем сокет к порту 3490 хоста, на котором выполняется программа:

```
struct addrinfo hints, *res;
int sockfd;

// сначала заполнить адресные структуры с помощью getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;          // использовать либо IPv4 либо IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;          // заполнить мой IP для меня

getaddrinfo(NULL, "3490", &hints, &res);

// создать сокет:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// связать с портом, полученным из getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

Указав флаг AI_PASSIVE, я сказал программе привязаться к IP хоста, на котором выполняется. Если вы хотите соединиться с отдельным локальным IP адресом, опустите AI_PASSIVE и укажите IP адрес в первом аргументе **getaddrinfo()**.

В случае ошибки **bind()** также возвращает -1 и устанавливает в ***errno*** код ошибки.

Много старого кода перед вызовом **bind()** заполняет struct sockaddr_in вручную. Это явно специфично для IPv4, но ничто не удерживает вас от того чтобы делать так и с IPv6, разве что использование **getaddrinfo()**, в общем-то, проще. В любом случае, старый код выглядит примерно так:

```
// !!! ЭТО СТАРЫЙ СПОСОБ !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);                // short, порядок байтов сети
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

В коде выше вы можете присвоить INADDR_ANY полю *s_addr* если хотите подключиться к локальному IP адресу (подобно флагу AI_PASSIVE выше.)

IPv6 версия INADDR_ANY это глобальная переменная *in6addr_any*, записанная в поле *sin6_addr* вашей struct sockaddr_in6. (Также есть макрос IN6ADDR_ANY_INIT, который вы можете использовать при инициализации переменной.)

Ещё одна вещь, за которой нужно следить при вызове **bind()**: не опускайте номера ваших портов ниже планки. Все порты ниже 1024 ЗАРЕЗЕРВИРОВАНЫ (если только вы не суперпользователь)! Вы можете обладать любым номером порта аж до 65535 (при условии, что он не используется другой программой.)

Иногда вы могли заметить, вы перезапускаете сервер, и **bind()** сбоит, заявляя “Адрес уже занят”. Что это значит? Это кусочек подключавшегося сокета до сих пор висит в ядре и это загаживает порт. Вы можете подождать, пока он очистится (минуту или около того), или добавить в программу код, позволяющий использовать порт повторно, вот как здесь:

```
int yes=1;
    //char yes='1';           // Это для пользователей Solaris
    // устранить противное сообщение "Address already in use"

if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
}
```

Ещё одно финальное замечание по **bind()**: бывают времена, когда вы абсолютно не можете её вызвать. Если вы подключаетесь (**connect()**) к уделённой машине и вас не заботит номер вашего локального порта (как в случае с **telnet**, где вам нужен только удалённый порт) вы можете просто вызвать **connect()**, он проверит, подключён ли порт и, если необходимо, вызовет **bind()** и подключит свободный локальный порт.

## 5.4. connect() - Эй, вы там!

Давайте на несколько минут притворимся, что вы **telnet** приложение. Ваш пользователь приказал вам (как в кино *TRON*) получить файловый дескриптор сокета. Вы подчинились и вызываете **socket()**. Потом пользователь говорит вам подключиться к “10.12.110.57” на порт “23” (стандартный **telnet** порт.) Ой! Что вам делать?

К счастью для вас, программа, вы сейчас внимательно читаете раздел по **connect()** - как подключиться к удалённому хосту. Так что яростно вперёд! Не терять времени!

**connect()** вызывается вот так:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

*sockfd* это файловый дескриптор сокета нашего доброго соседа, возвращённый вызовом **socket()**,

*serv_addr* это struct sockaddr, содержащая порт и IP адрес назначения,

*addrlen* это длина этой структуры в байтах.

Всю эту информацию можно наскрести из результатов вызова **getaddrinfo()**.

Вы уже начали чувствовать? Я отсюда вас не слышу и просто надеюсь, что это так. Давайте для примера подключим сокет к "www.example.com", порт 3490:

```
struct addrinfo hints, *res;
int sockfd;

// сначала заполнить адресные структуры с помощью getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// создать сокет:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// подключить!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

И снова, программы старой школы заполнены своими собственными struct sockaddr_in для передачи в **connect()**. Если хотите, можете делать так. Смотрите подобное замечание в разделе **bind()** выше.

Не забудьте проверить возвращаемое значение **connect()**, в случае ошибки она вернёт -1 и установит переменную ***errno***.

Отметьте также, что мы не вызывали **bind()**. В основном, нас заботит не наш локальный порт, а тот, куда мы подсоединяемся (удалённый порт). Ядро подберёт для нас локальный порт и подключённый сайт будет получать от нас информацию. Не тревожьтесь.

## 5.5. listen() - Позвони мне, позвони...

О'кей, пора сменить ритм. Что если вы не хотите подключаться к удалённому хосту. Скажем, для забавы вы хотите дождаться входящих подключений и затем их как-то обрабатывать. Это двухшаговый процесс: сначала вы слушаете - **listen()**, затем принимаете - **accept()** (см. ниже.)

Вызов **listen()** очаровательно прост, но требует некоторого разъяснения:

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

*sockfd* это обычный файловый дескриптор сокета из системного вызова **socket()**.

*backlog* это число разрешённых входных подключений во входной очереди. Что это значит? Хорошо, входящие подключения будут ждать в очереди пока вы их не примете (**accept()** см. ниже) и это предел сколько их там может быть. Большинство систем молчком ограничивает их числом порядка 20, наверное вы можете остановиться на 5 или 10.

И снова, как обычно, **listen()** возвращает -1 и устанавливает ***errno***.

Как вы, наверное, поняли нам нужно вызвать **bind()** до вызова **listen()**, так что сервер работает на определенном порте. (Вы должны знать, как сказать вашим приятелям, к какому порту подключаться!) Так что, если вы собираетесь слушать входящие подключения, то вам нужно выполнить следующую последовательность вызовов:

```
    getaddrinfo();
    socket();
    bind();
    listen();
/* accept() будет тут */
```

Я просто поставил строку в код, поскольку это само всё объясняет. (Код в разделе **accept()** ниже более полный.) Наиболее мудрёная часть этого предприятия это вызов **accept()**.

## 5.6. accept() - “Спасибо за звонок на порт 3490.”

Приготовьтесь, вызов **accept()** весьма причудлив. Что произойдёт в таком случае: некто очень далёкий будет пытаться подключиться вызовом **connect()** к вашей машине на порт, который вы слушаете вызовом **listen()**. Это соединение будет поставлено в очередь ждать **accept()**-а. Вы вызываете **accept()** и говорите ему принять ожидающие подключения. Он вернёт *совершенно новый файловый дескриптор сокета* для использования с одним подключением! Всё верно, внезапно у вас появилось *два файловых дескриптора сокета* по цене одного! Исходный до сих пор слушает новые подключения, а вновь созданный полностью готов к **send()** и **recv()**. Вот так!

Вызов выглядит так:

```
    #include <sys/types.h>
    #include <sys/socket.h>

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

*sockfd* это дескриптор слушающего сокета. Достаточно просто.

*addr* обычно будет указателем на локальную структуру sockaddr_storage. Сюда приходит информация о входящих подключениях (с её помощью вы можете определить какой хост вызывает вас и с какого порта).

*addrlen* is это локальная целая переменная, которая должна содержать размер struct sockaddr_storage до того как её адрес будет передан **accept()**. **accept()** больше, чем указано, байтов в *addr* не запишет. Если запишет меньше указанного, то изменит значение *addrlen*.

Догадались? **accept()** в случае ошибки возвращает -1 и устанавливает ***errno***. Спорим, и не снилось.

Как и раньше, здесь есть много чему внимать за раз, так что этот фрагмент кода смотрите внимательно:

```
    #include <string.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>

    #define MYPORT "3490"    // номер моего порта для подключения пользователей
    #define BACKLOG 10       // размер очереди ожидающих подключений
    int main(void)
    {
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
```

```
struct addrinfo hints, *res;
int sockfd, new_fd;

    // !! не забудьте проверить ошибки для этих вызовов !!
    // сначала заполнить адресные структуры с помощью getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // использовать либо IPv4 либо IPv6
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // заполнить мой IP для меня

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // создать сокет, связать и слушать:

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // принять входящие подключения:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

    // связываемся по дескриптору сокета new_fd!
...
```

Опять отметим, что мы будем использовать дескриптор сокета *new_fd* для всех вызовов **send()** и **recv()**. Если вы всегда только слушаете одно подключение, вы можете закрыть (**close()**) слушающий *sockfd*, чтобы остановить входящие подключения, если вам так уж хочется.

### 5.7. send() и recv() - Поговори со мною, бэби!

Эти две функции обеспечивают связь по потоковым и подключённым дейтаграммным сокетам. Если вы хотите использовать обычные неподключаемые дейтаграммные сокеты, обратитесь в раздел об **sendto()** и **recvfrom()** ниже.

Вызов **send()**:

```
int send(int sockfd, const void *msg, int len, int flags);
```

*sockfd* это дескриптор сокета, куда вы хотите отправить данные (возможно он был возвращён **socket()** или получен **accept()**.)

*msg* это указатель на посылаемые данные.

*len* это длина этих данных в байтах.

*flags* просто установите в 0. (См. **man** страницу вызова **send()**, там есть информация относительно *flags*.)

Пример кода может быть таким:

```
char *msg = "Beej was here!";
int len, bytes_sent;
...
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
...
```

**send()** возвращает количество действительно посланных байтов - *это может быть меньше числа байтов, указанного для передачи!* Видите ли, иногда вы посылаете такую кучу данных , что она не может их обработать. Она выпалит столько данных, сколько сможет, и поверит, что вы пошлёте оставшиеся позже. Помните, если количество байтов,

возвращённых **send()** не совпадает с *len*, то вам надо послать остаток строки. Хорошая новость такова: если пакет невелик (меньше 1K или около того), он, возможно будет послан одним куском. Опять же, в случае ошибки **send()** возвращает -1 и устанавливает ***errno***.

Вызов **recv()** во многом подобен:

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

*sockfd* это дескриптор сокета для чтения,

*buf* это буфер куда читать,

*len* это максимальная длина буфера,

*flags* can опять может быть установлен в 0 (см. **man** страницу **recv()**.)

**recv()** возвращает действительное количество записанных в буфер байтов или -1 при ошибке ( и соответственно установив ***errno***).

Погодите! **recv()** может возвращать 0. Это означает, что удалённая сторона закрыла для вас подключение! Это способ сказать вам об этом.

Это было просто, не правда ли? Теперь вы можете гонять данные туда-сюда по потоковым сокетам! Чудо! Вы теперь Сетевой Программист Unix!

## 5.8. sendto() и recvfrom() - Поговори со мной, DGRAM-стиль

Слышу, как вы говорите:" Это всё чудесно и прелестно, но что мне делать с неподключаемыми сокетами?" No problemo, amigo. У нас с собой было.

Поскольку дейтаграммные сокеты не подключены к удалённому хосту, угадайте, какую информацию мы должны задать до посылки пакета? Правильно! Адрес назначения! Сенсационная новость:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);
```

Как видите, этот вызов в основном подобен вызову **send()** с добавлением двух крупиц информации.

*to* это указатель на `struct sockaddr` (которая может быть `struct sockaddr_in`, `struct sockaddr_in6` или `struct sockaddr_storage`, приведёнными в последний момент), содержащую IP адрес назначения и порт.

*tolen*, в глубине души целое число, можно просто установить в `sizeof *to` или `sizeof(struct sockaddr_storage)`.

Структуру с адресом назначения можно получить из **getaddrinfo()** или **recvfrom()** ниже, или заполняйте сами.

Как и **send()**, **sendto()** возвращает число действительно отправленных байт (которое, опять же, может быть меньше, чем вы указали!) или -1 при ошибке.

Точно также **recvfrom()** подобна **recv()**. Вызов **recvfrom()**:

```
int  recvfrom(int sockfd, void *buf, int len, unsigned int flags,
              struct sockaddr *from, int *fromlen);
```

И опять, он такой же, как и **recv()** с добавлением пары полей.

*from* это указатель на локальную `struct sockaddr_storage`, которая будет заполнена IP адресом машины отправителя.

*fromlen* это указатель на локальную целую, которая будет инициализирована из `sizeof *from` или `sizeof(struct sockaddr_storage)`. Когда функция возвращает управление, *fromlen* будет содержать действительную длину сохранённого в *from*.

**recvfrom()** возвращает число принятых байтов или -1 в случае ошибки (соответственно установив ***errno***).

Теперь вопрос: почему мы используем `struct sockaddr_storage` как тип сокета? Почему не `struct sockaddr_in`? Потому что, видите ли, мы не хотим привязывать

себя к IPv4 или IPv6. Так что мы используем общую `struct sockaddr_storage`, которой, как мы знаем, хватает для обеих.

(Так… Ещё вопрос: почему `struct sockaddr` самой недостаточно для любого адреса? Мы даже приводим `struct sockaddr_storage` общего назначения к `struct sockaddr` общего назначения! Кажется чуждым и чрезмерным, ха! Ответ в том, что она недостаточно велика и изменение её сейчас будет весьма проблематичным. Так что они сделали новую.)

Помните, если вы подключаете дейтаграммный сокет **connect()**-ом, то можете просто использовать **send()** и **recv()** для транзакций. Сам сокет остаётся дейтаграммным и пакеты используют UDP, но интерфейс сокета автоматически добавит информацию об источнике и назначении.

## 5.9. close() и shutdown() - Прочь с глаз моих!

Уф! Вы гоняли **send()** и **recv()** с данными весь день и закончили. Вы готовы закрыть соединение на вашем дескрипторе сокета. Это легко. Можно использовать обычную функцию закрытия файлового дескриптора Unix **close()**:

```
close(sockfd);
```

Это предотвратит дальнейшее чтение и запись в сокет. Любой, попытавшийся читать или писать в этот сокет на удалённом конце получит ошибку.

Если вы хотите получить немного больше управления над закрытием сокета, можно использовать функцию **shutdown()**. Она позволяет разорвать связь в определенном направлении или в обоих (как **close()**). Пишется так:

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

*sockfd* это файловый дескриптор выключаемого сокета, *how* принимает следующие значения:

| | |
|---|---|
| 0 - | Дальнейший приём запрещён |
| 1 - | Дальнейшая отправка запрещена |
| 2 - | Дальнейшие приём и отправка запрещены (как **close()**) |

**shutdown()** возвращает 0 в случае успеха и -1 при ошибке (***errno*** установлен соответственно).

Если вы соизволите использовать **shutdown()** с неподключённым дейтаграммным сокетом, он просто делает его недоступным для последующих вызовов **send()** и **recv()** (помните, что вы можете их использовать, если вы **connect()**ите дейтаграммный сокет).

Важно отметить, что **shutdown()** в действительности не закрывает файловый дескриптор, а только изменяет его использование. Для освобождения дескриптора сокета используйте **close()**.

Вот и всё.

(Разве что нужно помнить, если вы используете Windows и Winsock, вам должно вызывать **closesocket()** вместо **close()**).

## 5.10. getpeername() - Кто вы?

Эта функция очень проста.

Так проста, что я чуть было не оставил её без своего раздела. Но всё равно вот она.

Функция скажет вам кто находится на другом конце подключённого потокового сокета. Вот запись:

```
#include    <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

*sockfd* это дескриптор подключённого потокового сокета, *addr* это указатель на struct sockaddr (или struct sockaddr_in), которая будет содержать информацию о другой стороне соединения и *addrlen* это указатель на целое, которое должно быть инициализировано sizeof *addr или sizeof(struct sockaddr).

В случае ошибки функция возвращает -1 и соответственно устанавливает ***errno***.

Как только у вас есть их адрес, вы можете использовать **inet_ntop()**, **getnameinfo()** или **gethostbyaddr()**, чтобы распечатать или получить дополнительную информацию. Нет, их логин вы получить не можете. (Ладно, ладно. Если на другом компьютере выполняется демон **ident**, это возможно. Однако, это выходит за рамки данного документа. Подробнее см. RFC 1413[20]).

## 5.11.gethostname() - Кто Я?

Функция **gethostname()** даже проще, чем **getpeername()**. Она возвращает имя компьютера, на котором запущена ваша программа. Это имя затем может быть использовано в **gethostbyname()** для определения IP адреса вашей локальной машины.

Что может быть ещё веселее? Я подумал о многих вещах, но они не относятся к программированию сокетов.

В любом случае, вот запись:

```
#include    <unistd.h>

int gethostname(char *hostname, size_t size);
```

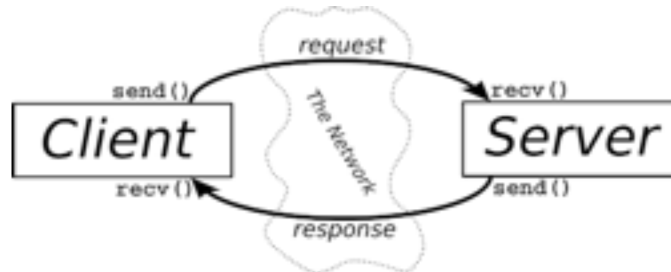
Аргументы просты: *hostname* это указатель на массив символов, который по возвращении из функции будет содержать имя хоста, и *size* это длина массива *hostname* в байтах.

При успешном завершении функция возвращает 0 и -1 в случае ошибки, ***errno*** устанавливается как обычно.

[20] http://tools.ietf.org/html/rfc1413

# 6. Архитектура Клиент-Сервер

Это клиент-серверный мир, крошка. Почти всё в сети это разговор клиентского процесса с серверным и наоборот. Возьмём к примеру **telnet**. Когда вы подключаетесь к удалённому хосту на порт 23 с **telnet**-ом (клиент), программа на хосте (именуемая **telnetd**, сервер) оживает. Она обрабатывает входное **telnet** подключение, выставляет вам запрос на логин и т.д.




**Взаимодействие Клиент-Сервер**

Обмен информацией между клиентом и сервером приведён на диаграмме выше.

Заметим, что пара клиент-сервер могут говорить на `SOCK_STREAM`, `SOCK_DGRAM` или чём угодно, если они говорят на одном языке. Хорошие примеры пар клиент-сервер это **telnet**/**telnetd**, **ftp**/**ftpd** или **Firefox**/**Apache**. Каждый раз, когда вы используете **ftp** есть удалённая программа **ftpd**, которая обслуживает вас.

Часто на машине будет работать только один сервер, и этот сервер будет обслуживать множество клиентов используя **`fork()`**. Основная программа такова: сервер будет ждать подключения, примет его (**`accept()`**), и запустит процесс-потомок для его обслуживания (**`fork()`**). Именно так работает пример сервера в следующем разделе.

## 6.1. Простой потоковый сервер

Этот сервер просто посылает строку “`Hello, World!\n`” по потоковому соединению. Вам нужно только запустить его в одном окне и **telnet**-нуть ему из другого вот так:

```
$ telnet remotehostname 3490
```

где `remotehostname` это имя машины на которой вы работаете.

Серверный код[21]:

```
/*
** server.c -- пример сервера потокового сокета
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT "3490"        // порт для подключения пользователей

#define BACKLOG 10         // размер очереди ожидающих подключений
```

[21] http://beej.us/guide/bgnet/examples/server.c

```
void sigchld_handler(int s)
{
   while(waitpid(-1, NULL, WNOHANG) > 0);
}

// получить sockaddr, IPv4 или IPv6:

void *get_in_addr(struct sockaddr *sa)
{
      if (sa->sa_family == AF_INET) {
            return &(((struct sockaddr_in*)sa)->sin_addr);
      }

      return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
      int sockfd, new_fd;                    // слушать на sock_fd, новое подключение на new_fd
      struct addrinfo hints, *servinfo, *p;
      struct sockaddr_storage their_addr;    // адресная информация подключившегося
      socklen_t sin_size;
      struct sigaction sa;
      int yes=1;
      char s[INET6_ADDRSTRLEN];
      int rv;

      memset(&hints, 0, sizeof hints);
      hints.ai_family = AF_UNSPEC;
      hints.ai_socktype = SOCK_STREAM;
      hints.ai_flags = AI_PASSIVE;           // использовать мой IP

      if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
            fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
            return 1;
            }

      // цикл по всем результатам и связывание с первым возможным
      for(p = servinfo; p != NULL; p = p->ai_next) {
            if ((sockfd = socket(p->ai_family, p->ai_socktype,
                  p->ai_protocol)) == -1) {
            perror("server: socket");
                  continue;
                  }

            if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
           sizeof(int)) == -1) {
                  perror("setsockopt");
                              exit(1);
                  }

            if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("server: bind");
                        continue;
                  }

                  break;
            }

      if (p == NULL)  {
```

```
        fprintf(stderr, "server: failed to bind\n");
        return 2;
        }

    freeaddrinfo(servinfo);                 // со структурой закончили

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
        }

    sa.sa_handler = sigchld_handler;        // жатва всех мёртвых процессов
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
            exit(1);
        }

    printf("server: waiting for connections…\n");

    while(1) {                              //  главный цикл accept()
        sin_size = sizeof their_addr;
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        if (new_fd == -1) {
        perror("accept");
            continue;
            }

        inet_ntop(their_addr.ss_family,
        get_in_addr((struct sockaddr *)&their_addr),
        s, sizeof s);

        printf("server: got connection from %s\n", s);

        if (!fork()) {                      // это порождённые процесс
        close(sockfd);                      // его слушать не нужно
        if (send(new_fd, "Hello, world!", 13, 0) == -1)
            perror("send");
        close(new_fd);
                    exit(0);
            }

        close(new_fd);                      // родителю это не нужно
    }

    return 0;
}
```

На случай, если вы любопытны. Весь мой код расположен в одной большой функции **`main()`** для синтаксической ясности (мне так кажется). Можете свободно разделить его на меньшие функции если вам от этого станет лучше.

(И ещё, **`sigaction()`** может быть совсем новинкой для вас, это нормально. Этот код здесь отвечает за уборку зомби-процессов, которые появляются при завершении процесса-потомка после **`fork()`**. Если вы создадите множество зомби-процессов и не сожнёте их, ваш системный администратор очень разволнуется).

Данные от этого сервера можно получить с помощью описанного в следующем разделе клиента.

## 6.2. Простой потоковый клиент

Этот хлопец даже проще, чем сервер. Всё, что этот клиент делает, это подключается к хосту, который вы указываете в командной строке, порт 3490. Он принимает строку, которую высылает сервер.

Исходник клиента[22]:

```
/*
** client.c -- пример клиента потокового сокета
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#include <arpa/inet.h>

#define PORT "3490"                 // порт для подключения клиентов

#define MAXDATASIZE 100             // максимальная длина принимаемых за раз данных

// получить sockaddr, IPv4 или IPv6:
void *get_in_addr(struct sockaddr *sa)
{
   if (sa->sa_family == AF_INET) {
      return &(((struct sockaddr_in*)sa)->sin_addr);
      }

   return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
   int sockfd, numbytes;
   char buf[MAXDATASIZE];
   struct addrinfo hints, *servinfo, *p;
   int rv;
   char s[INET6_ADDRSTRLEN];

   if (argc != 2) {
      fprintf(stderr,"usage: client hostname\n");
      exit(1);
      }

   memset(&hints, 0, sizeof hints);
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;
   if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
      return 1;
      }

   // цикл по всем результатам и связывание с первым возможным
```

[22] http://beej.us/guide/bgnet/examples/client.c

```
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
             p->ai_protocol)) == -1) {
           perror("client: socket");
                    continue;
               }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("client: connect");
                     continue;
               }

               break;
        }

    if (p == NULL) {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
        }

    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
            s, sizeof s);
    printf("client: connecting to %s\n", s);

    freeaddrinfo(servinfo);                  // с этой структурой закончили

    if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
             perror("recv");
                 exit(1);
        }

    buf[numbytes] = '\0';

    printf("client: received '%s'\n",buf);

    close(sockfd);

        return 0;
}
```

Заметим, что если вы не запустите сервер до клиента, **`connect()`** вернёт “Connection refused” (“Подключение отвергнуто”). Очень полезно.

## 6.3. Дейтаграммные сокеты

Мы уже рассмотрели основы дейтаграммных сокетов UDP обсуждая **`sendto()`** и **`recvfrom()`** выше, так что я просто представлю пару примеров программ: *`talker.c`* и *`listener.c`*.

**listener** сидит на машине, ожидая входящие пакеты на порт 4950. **talker** посылает пакет в этот порт на указанной машине, который содержит всё, что пользователь введёт в командной строке.

Вот исходник listener.c[23]:

```
/*
** listener.c -- пример “сервера” дейтаграммного сокета
*/

#include <stdio.h>
```

[23] http://beej.us/guide/bgnet/examples/listener.c

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MYPORT "4950"
#define MAXBUFLEN 100

// порт для подключающихся пользователей
// получить sockaddr, IPv4 или IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;
    struct sockaddr_storage their_addr;
    char buf[MAXBUFLEN];
    socklen_t addr_len;
    char s[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;            // установить AF_INET для выбора IPv4
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE;            // использовать мой IP
    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // цикл по всем результатам и связывание с первым возможным
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
            perror("listener: socket”);
            continue;
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("listener: bind");
            continue;
        }

        break;
    }
```

```
    if (p == NULL) {
        fprintf(stderr, "listener: failed to bind socket\n");
        return 2;
    }

    freeaddrinfo(servinfo);

    printf("listener: waiting to recvfrom...\n");

    addr_len = sizeof their_addr;
    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN - 1 , 0,
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
        perror("recvfrom");
        exit(1);
    }

    printf("listener: got packet from %s\n",
        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s));
    printf("listener: packet is %d bytes long\n", numbytes);
    buf[numbytes] = '\0';
    printf("listener: packet contains \"%s\"\n", buf);

    close(sockfd);

    return 0;
}
```

Отметим, что в нашем вызове **`getaddrinfo()`** мы в итоге используем `SOCK_DGRAM`. Также отметим, что использовать **`listen()`** и **`accept()`** нет нужды. Этим можно выпендриваться используя дейтаграммные сокеты.

Исходник talker.c[24]:

```
/*
** talker.c -- пример дейтаграммного “клиента”
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT “4950"

// порт для подключающихся пользователей
int main(int argc, char *argv[])
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;
```

[24] http://beej.us/guide/bgnet/examples/talker.c

```
    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;

    if ((rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // цикл по всем результатам и создание сокета
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("talker: socket");
            continue;
        }
        break;
    }

    if (p == NULL) {
        fprintf(stderr, "talker: failed to bind socket\n");
        return 2;
    }
    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
                p->ai_addr, p->ai_addrlen)) == -1) {
        perror("talker: sendto");
        exit(1);
    }

    freeaddrinfo(servinfo);
    printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);

    close(sockfd);
    return 0;
}
```

И это всё! Запустите **listener** на одной машине, затем **talker** на другой. Как они общаются! Радости на штуку баксов для всей ядерной семейки!

В этот раз вам даже не нужно запускать сервер! Вы можете запустить один **talker** и он с удовольствием выстрелит пакеты во тьму, где они исчезнут если никто не ждёт их с **`recvfrom()`** на другой стороне. Помните: данные, посланные в дейтаграммный UDP сокет не обязательно прибывают!

За исключением одной крохотной детали, о которой я много раз говорил в прошлом: подключённые дейтаграммные сокеты. Должет сказать об этом и здесь, поскольку мы в дейтаграммном разделе документа. Ведь **talker** вызывает **`connect()`** и задаёт адрес **listener**-а. С этого момента **talker** может посылать и принимать данные только с адреса, определённого **`connect()`**-ом. Поэтому вам не нужно использовать **`sendto()`** и **`recvfrom()`**; вы можете просто пользоваться **`send()`** и **`recv()`**.

# 7. Немного продвинутая техника

Эти приёмы не являются *действительно* продвинутыми, но они выходят за рамки уже пройденных более общих уровней. Действительно, если вы дошли так далеко, то можете считать себя весьма успешным в основах сетевого программирования Unix! Поздравляю!

Отныне мы отважно вступаем мир более эзотерических знаний, которые вам захочется узнать о сокетах! Налетайте!

## 7.1. Блокировка

Блокировка. Вы о ней слыхали - что это за чертовщина? В двух словах, “блокировка” на жаргоне технарей это “сон”. Вы, наверное, заметили, что когда вы запускаете **listener**, выше, он просто сидит там до появления пакета. Происходит так, что она вызывает **recvfrom()**, а данных нет, и **recvfrom()** сказано “заблокироваться” (в данном случае спать) пока не появятся данные.

Множество функций блокируются. **accept()** блокируется. Все **recv()** функции блокируются. Они могут это делать потому что им разрешено. Когда вы создаёте сокет функцией **socket()**, ядро устанавливает его на блокировку. Если вы не хотите, чтобы сокет блокировался, вызовите **fcntl()**:

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
.
```

Установив сокет на не-блокировку, вы можете эффективно его опрашивать. Если вы попытаетесь читать из неблокируемого сокета, а данных нет, он, неблокируемый, вернёт -1 и установит ***errno*** в EWOULDBLOCK.

Вообще-то говоря, этот тип опроса это плохая идея. Если вы запустите вашу программу в постоянном цикле опроса данных от сокета, она будет жрать процессорное время, как свинья помои. Более элегантное решение узнать есть ли данные на чтение дано в следующем разделе по **select()**.

## 7.2. select()—Мультиплексирование синхронного ввода/вывода

Эта функция это нечто странное, но очень полезное. Возьмём такую ситуацию: вы сервер и хотите слушать входящие подключения и одновременно читать уже имеющиеся.

Вы скажете - без проблем, один **accept()** и парочка **recv()**. Не так быстро, приятель! Что если вы заблокированы на вызове **accept()**? Как вы собираетесь в это же время принимать данные от **recv()**? “Использовать неблокируемые сокеты!” Да никогда! Вы же не хотите быть процессорной свиньёй. И что дальше?

**select()** даёт вам возможность следить за несколькими сокетами одновременно. Она скажет вам какие готовы для чтения, какие для записи, а какие возбудили исключение, если вы действительно хотите это знать.

Как было сказано, в настоящее время **select()**, хоть и очень переносимый, но и один из самых медленных способов мониторинга сокетов. Возможная альтернатива libevent[25]

[25] http://www.monkey.org/~provos/libevent/

или что-либо подобное, объединяющее системо-зависимые вещи, связанные с получением нотификации сокетов.

Без дальнейших хлопот я предлагаю запись **`select()`**:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
                    fd_set *exceptfds, struct timeval *timeout);
```

Функция мониторит “массивы” файловых дескрипторов, в частности *`readfds`*, *`writefds`* и *`exceptfds`*. Если вы хотите знать можно ли читать со стандартного ввода и сокета *`sockfd`*, просто добавьте 0 и *`sockfd`* в массив *`readfds`*. Параметр *`numfds`* должен содержать значение наибольшего файлового дескриптора плюс один. В этом примере он должен быть установлен в *`sockfd+1`*, поскольку он заведомо больше стандартного ввода (0).

Когда **`select()`** возвращает управление, *`readfds`* будет модифицирован чтобы отражать какой из выбранных файловых дескрипторов готов к чтению. Вы можете проверить их с помощью макроса **`FD_ISSET()`**.

Перед тем как продолжить, я расскажу, как работать с этими массивами. Каждый массив имеет тип `fd_set`. С ним работают следующие макросы:

| | |
|---|---|
| **`FD_SET(int fd, fd_set *set);`** | Добавляет *`fd`* в *`set`*. |
| **`FD_CLR(int fd, fd_set *set);`** | Удаляет *`fd`* из *`set`*. |
| **`FD_ISSET(int fd, fd_set *set);`** | Возвращает `true` если *`fd`* есть в *`set`*. |
| **`FD_ZERO(fd_set *set);`** | Очищает *`set`*. |

Наконец, что странного в `struct timeval`? Хорошо, иногда вы не хотите вечно ждать когда кто-нибудь пришлёт вам какие-нибудь данные. Может вы хотите каждые 96 секунд печатать на терминале “Ещё работаю…”, хотя ничего не происходит. Эта структура с временем позволяет вам определить период таймаута. Если время истекло и **`select()`** не нашёл готового файлового дескриптора он возвращает управление и вы можете продолжить работу.

`struct timeval` имеет следующие поля:

```
struct timeval {
    int tv_sec;         // секунды
    int tv_usec;        // микросекунды
};
```

Просто установите в *`tv_sec`* число секунд и в *`tv_usec`* число микросекунд ожидания. Да, это микросекунды, а не миллисекунды. В миллисекунде 1000 микросекунд, и в секунде 1000 миллисекунд. Таким образом в секунде 1 000 000 микросекунд. Почему она “usec”? Предполагается, что “u” выглядит как греческая буква μ (мю), которую мы используем для обозначения “микро”. Кроме того, когда функция **`select()`** возвращает управление, *`timeout`* *может* быть изменена чтобы показать сколько времени ещё осталось. Это зависит от вида вашей Unix.

Ура! У нас есть таймер с микросекундным разрешением! На это не рассчитывайте. Вероятно вы будете ждать некоторую часть стандартного времени квантования Unix, вне зависимости от того, насколько малую величину вы записали в вашу `struct timeval`.

Другие интересные вещи: Если установите поля вашей `struct timeval` в 0, **`select()`** вернётся немедленно, опросив все файловые дескрипторы в ваших массивах. Если вы установите параметр *`timeout`* в `NULL`, она будет ждать пока первый файловый

дескриптор не станет готовым. Наконец, если вы не хотите ждать определённого массива дескрипторов, при вызове **`select()`** установите его в `NULL`.

Следующий отрывок кода[26] ждёт 2.5 секунды чтобы что-то появилось на стандартном вводе:

```
/*
** select.c -- пример select()
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0                 // файловый дескриптор стандартного ввода

int main(void)
{
   struct timeval tv;
   fd_set readfds;

   tv.tv_sec = 2;
   tv.tv_usec = 500000;

   FD_ZERO(&readfds);
   FD_SET(STDIN, &readfds);

   // writefds и exceptfds не нужны:
   select(STDIN+1, &readfds, NULL, NULL, &tv);

   if (FD_ISSET(STDIN, &readfds))
      printf("A key was pressed!\n");
   else
      printf("Timed out.\n”);

      return 0;
}
```

Если у вас терминал со строковой буферизацией, то нажимать нужно клавишу `RETURN`, иначе не дождётесь.

Некоторые из вас могут подумать, что это великолепный способ ожидания данных от дейтаграммных сокетов, и вы правы - это *может* быть. Некоторые Unix-ы могут для этого использовать **`select()`**, а некоторые нет. Посмотрите, что скажут ваши **man** страницы по этому поводу.

Некоторые Unix-ы обновляют время в вашей `struct timeval` для обозначения времени, оставшегося до истечения таймаута, а некоторые нет. Не полагайтесь на это, если хотите писать переносимые программы. (Используйте **`gettimeofday()`** если вам надо отследить затраченное время. Это глупость, я знаю, но всё-таки выход.)

Что происходит когда сокет в массиве чтения закрывает соединение? В этом случае **`select()`** возвращается с этим дескриптором сокета, как “готовым для чтения”. И когда вы действительно вызываете **`recv()`** с ним, **`recv()`** возвращает 0. Так вы узнаёте, что клиент закрыл соединение.

[26] http://beej.us/guide/bgnet/examples/select.c

Ещё одно интересное замечание о **`select()`**, если у вас есть сокет с запущенным **`listen()`**, вы можете проверить, есть ли новое подключение, установив файловый дескриптор сокета в массив *`readfds`*.

Вот, друзья мои, быстрый обзор всемогущей функции **`select()`**.

Но по требованию народа ниже приведён углублённый пример. К сожалению, разница между простым примером выше и приведённым здесь весьма значительна. Но всё равно взгляните и прочтите последующее описание.

Эта программа[27] работает как простой многопользовательский сервер чата. Запустите её в одном окне и затем отправьте ей по **telnet**-у “**telnet hostname 9034**” из многих других окон. Когда вы печатаете что-нибудь в одном окне, это должно появляться во всех остальных.

```
/*
** selectserver.c -- убогий многопользовательский сервер чата
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define PORT "9034"             // этот порт мы слушаем

// получить sockaddr, IPv4 или IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    fd_set master;              // главный список файловых дескрипторов
    fd_set read_fds;            // временный список файловых дескрипторов для select()
    int fdmax;                  // максимальный номер файлового дескриптора

    int listener;               // дескриптор слушаемого сокета
    int newfd;                  // новопринятый дескриптор сокета
    struct sockaddr_storage remoteaddr; // адрес клиента
    socklen_t addrlen;

    char buf[256];              // буфер для данных клиента
    int nbytes;

    char remoteIP[INET6_ADDRSTRLEN];

    int yes=1;                  // для setsockopt() SO_REUSEADDR, ниже
    int i, j, rv;
```

[27] http://beej.us/guide/bgnet/examples/selectserver.c

```
    struct addrinfo hints, *ai, *p;

    FD_ZERO(&master);         // очистка главного и временного массивов
    FD_ZERO(&read_fds);

    // получить сокет и связать
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (listener < 0) {
            continue;
        }

        // убрать мерзкое сообщение "address already in use"
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }

        break;
    }

    // если мы здесь, значит не связались
    if (p == NULL) {
        fprintf(stderr, "selectserver: failed to bind\n");
        exit(2);
    }

    freeaddrinfo(ai);             // с этим закончили

    // слушаем
    if (listen(listener, 10) == -1) {
        perror("listen");
        exit(3);
    }

    // добавить слушателя в главный массив
    FD_SET(listener, &master);

    // сохранить наибольший файловый дескриптор
    fdmax = listener;             // вот он

    // главный цикл
    for(;;) {
        read_fds = master;        // копируем
        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(4);
        }

        // ищем готовые для чтения данные в существующих подключениях
        for(i = 0; i <= fdmax; i++) {
```

```
            if (FD_ISSET(i, &read_fds)) {       // Есть!!
                if (i == listener) {

                    // обрабатываем новые подключения
                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                                        (struct sockaddr *)&remoteaddr,
                                        &addrlen);

                    if (newfd == -1) {
                        perror("accept");
                    } else {
                        FD_SET(newfd, &master);      // добавить в главный массив
                        if (newfd > fdmax) {         // отслеживаем максимальный номер
                            fdmax = newfd;
                        }
                        printf("selectserver: new connection from %s on "
                                "socket %d\n",
                                inet_ntop(remoteaddr.ss_family,
                                get_in_addr((struct sockaddr*)&remoteaddr),
                                remoteIP, INET6_ADDRSTRLEN),
                                newfd);
                    }
                } else {
                    // обработка данных от клиента
                    if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                        // ошибка или соединение закрыто клиентом
                        if (nbytes == 0) {
                            // соединение закрыто
                            printf("selectserver: socket %d hung up\n", i);
                        } else {
                            perror("recv");
                            }
                        close(i);                    // Пока!
                        FD_CLR(i, &master);          // удалить из главного массива
                    } else {
                        // от клиента что-то получили
                        for(j = 0; j <= fdmax; j++) {
                            // посылаем всем!
                            if (FD_ISSET(j, &master)) {
                                // кроме слушателя и себя
                                if (j != listener && j != i) {
                                    if (send(j, buf, nbytes, 0) == -1) {
                                        perror("send");
                                    }
                                }
                            }
                        }
                    }
                } // END обработка данных от клиента
            } // END есть новое входящее подключение
        } // END цикл по файловым дескрипторам
    } // END for(;;)–и вы думаете, что это не закончится!

    return 0;
}
```

Заметьте, что у меня два массива файловых дескрипторов: `master` и `read_fds`. Первый, `master`, содержит как уже подключённые, так и прослушиваемые для новых подключений дескрипторы.

Я завёл массив *master* потому что **select()** реально *изменяет* переданный массив для отображения готовых к чтению сокетов. Поскольку мне нужно сохранять подключения от одного вызова **select()** до другого, в последний момент перед вызовом **select()** я копирую *master* в *read_fds* и затем вызываю **select()**.

Означает ли это, что получив новое соединение я должен добавить его в массив *master*? Ага! И каждый раз, когда соединение закрывается, мне нужно удалить его из массива *master*. Да, это так!

Заметьте, я проверяю сокет *listener* на готовность чтения. Когда он готов, это означает, что есть ожидающее подключение, я принимаю его **accept()**-ом и добавляю в массив *master*. Точно так же, когда клиентское подключение готово к чтению и **recv()** возвращает 0, я знаю, что клиент закрыл подключение и я должен удалить его из массива *master*.

Если **recv()** клиента всё таки возвращает не-ноль, я знаю, что были приняты какие-то данные. Я получаю их и по списку *master* раздаю остальным подключённым клиентам.

Вот, друзья мои, наипростейший обзор всемогущей функции **select()**.

В довесок, запоздалый бонус: есть функция, именуемая **poll()**, которая ведёт себя почти также как **select()**, но с отличающейся системой управления массивом файловых дескрипторов. Посмотрите её!

## 7.3. Обработка незавершённых send()

Помните в прошлом разделе по **send()** я сказал, что **send()** может не выслать всех указанных байт? То есть, вы хотит послать 512 байт, а она возвращает число 412. Что случилось с оставшимися 100 байтами?

Они до сих пор в вашем маленьком буфере ждут отсылки. По независящим от вас обстоятельствам ядро решило не посылать их одним куском и теперь, мой друг, вам пора взять их оттуда.

Для этого вы тоже можете написать функцию, подобную этой:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendall(int s, char *buf, int *len)
{
   int total = 0;              // сколько байт мы послали
   int bytesleft = *len;       // сколько байт осталось послать
   int n;

   while(total < *len) {
      n = send(s, buf+total, bytesleft, 0);
      if (n == -1) { break; }
      total += n;
      bytesleft -= n;
     }

   *len = total;               // здесь количество действительно посланных байт

   return n==-1?-1:0;          // вернуть -1 при сбое, 0 при успехе
}
```

В этом примере *s* это сокет, куда вы хотите послать данные, *buf* буфер с данными и *len* указатель на целое, содержащее количество байт в буфере.

В случае ошибки функция возвращает -1 (***errno*** установлена **send()**). Число действительно посланных байт возвращается в *len*. Это будет число, указанное вами для

отсылки, если не было ошибки. **sendall()**, пыхтя и вздыхая, пошлёт данные наилучшим образом, но если случится ошибка, она просто вернётся к вам.

Для полноты вот пример вызова функции:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
   perror("sendall");
   printf("We only sent %d bytes because of the error!\n", len);
}
```

Что происходит когда часть пакета появляется на стороне приёмника? Если пакеты переменной длины, то откуда приёмник узнаёт, где кончается один пакет и начинается другой? Да, сценарии реального мира - вещь беспокойная. Может быть вам нужно *инкапсулировать* (помните об этом говорилось в начале?) Читайте дальше, там подробности!

## 7.4. Сериализация - Как упаковать данные

Посылать по сети текстовые данные достаточно легко, не правда ли, но что происходит если вы хотите послать некие двоичные данные, целочисленные или с плавающей запятой? Оказывается, у вас есть несколько вариантов.

1. Преобразовать числа в текст функцией типа **sprintf()** и послать текст. Приёмник преобразует текст обратно в цифру функцией типа **strtol()**.
2. Просто послать необработанные данные, передав указатель в **send()**.
3. Закодировать данные в переносимую двоичную форму. Приёмник их декодирует.

Закрытый просмотр! Только сегодня!
[*Занавес открывается*]
Бидж говорит:”Я предпочитаю Метод Три, выше!”
[*КОНЕЦ*]

(Прежде чем серьёзно начать этот раздел я должен сказать, что где-то существуют библиотеки, которые это делают. Но и упаковать, и оставить переносимым, и безошибочным, это весьма серьёзно. Так что сходите на охоту, сделайте домашнее задание прежде чем решиться делать это самому. Для любопытных я включил информацию о том как такие штуки работают.)

В действительности все эти методы имеют свои достоинства и недостатки, но, как я сказал, в общем-то, я предпочитаю третий метод. Но всё таки сначала поговорим о достоинствах и недостатках двух других.

Первый метод, кодирование чисел в текст перед посылкой выгоден тем, что легко можете распечатать и прочесть приходящие по проводам данные. Иногда читаемый протокол превосходен при использовании в не-широковещательно-интенсивной ситуации, как Internet Relay Chat (IRC)[28]. Однако он невыгоден тем, что преобразование медленно и результат почти всегда занимает больше места, чем исходное число!

Метод два: передача необработанных данных. Этот метод существенно легче (но опасней!): просто берёте указатель на посылаемые данные и вызываете с ним **send()**.

```
double  d = 3490.15926535;
```

[28] http://en.wikipedia.org/wiki/Internet_Relay_Chat

```
send(s, &d, sizeof d, 0);  /* Опасно - не портируется! */
```

Приёмник получает их так:

```
double d;

recv(s, &d, sizeof d, 0);  /* Опасно - не портируется! */
```

Быстро, просто - что не нравится? Оказывается, не все архитектуры представляют числа с плавающей запятой (или даже целые) с тем же расположением бит или даже порядком байт! Код вообще решительно не переносим. (А может переносимость вам не нужна, в таком случае он быстр и замечательно подходит.)

Упаковывая целочисленные типы мы уже видели как функции класса **htons()** могут помочь сохранить переносимость преобразовывая числа в Порядок Байтов Сети и как это Надо Делать. К сожалению, для плавающих типов таких функций нет. Все надежды пропали?

Боюсь, что нет! (Вы этого хоть на секунду опасались? Нет? Даже чуть-чуть?) Мы можем кое-что сделать: мы можем упаковать (или “маршализировать” или “ сериализировать” или любое из тысяч миллионов других названий) данные в известный двоичный формат, который приёмник может распаковать на удалённой стороне.

Что я подразумеваю под “известным двоичным форматом”? Мы уже видели пример функции **htons()**, правда? Она преобразует (или “перекодирует”, если так вам хочется думать) число из любого формата хоста в Порядок Байтов Сети. Для обратного преобразования (“раскодирования”) числа приёмник вызывает **ntohs()**.

Но разве я только что не говорил, что таких функций для других не-целых форматов нет? Да, говорил. Это немножко прискорбно, поскольку в C стандартного способа для этого нет (бесплатный каламбур для фанатов **Python**).

Что нужно сделать, так это упаковать данные в известный формат и послать их по проводам для раскодирования. Например, для упаковки плавающих есть кое-что быстрое и убогое с изобилием мест для улучшения[29]:

```
#include <stdint.h>

uint32_t htonf(float f)
{
   uint32_t p;
   uint32_t sign;

   if (f < 0) { sign = 1; f = -f; }
   else { sign = 0; }

   p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31);            // целое и знак
   p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff;         // дробная часть

      return p;
}

float ntohf(uint32_t p)
{
   float f = ((p>>16)&0x7fff);                                // целое и знак
   f += (p&0xffff) / 65536.0f;                                // дробная часть

   if (((p>>31)&0x1) == 0x1) { f = -f; }                      // установка бита знака
      return f;
}
```

[29] http://beej.us/guide/bgnet/examples/pack.c

Это бесхитростная программка, которая сохраняет плавающее в 32-битном числе. Старший бит (31) используется для хранения знака (“1” означает отрицательное), следующие пятнадцать бит (30-16) используются для хранения целой части числа с плавающей запятой. Оставшаяся часть битов (15-0) используется для хранения дробной части числа.

Использование довольно просто:

```
#include <stdio.h>

int main(void)
{
   float f = 3.1415926, f2;
   uint32_t netf;

   netf = htonf(f);                          // преобразовать в “сетевую” форму””
   f2 = ntohf(netf);                         // обратно для теста

   printf("Original: %f\n", f);              // 3.141593
   printf(" Network: 0x%08X\n", netf);       // 0x0003243F
   printf("Unpacked: %f\n", f2);             // 3.141586

     return 0;
}
```

Из плюсов, она маленькая, простая и быстрая, Из минусов, неэффективное использование пространства и очень ограниченный диапазон - попробуйте сохранить здесь число больше 32767 и вы не будете очень счастливы! Также в этом примере видно, что две последние десятичные цифры сохраняются неправильно.

Что можно сделать взамен? Хорошо, *конкретно* Стандарт хранения чисел с плавающей запятой известен как IEEE-754[30]. Большинство компьютеров используют этот формат для выполнения вычислений с плавающей запятой, так что, строго говоря, в данном случае преобразование не нужно. Но если вы хотите добиться переносимости вашего исходного кода, то полагаться на это нельзя. (С другой стороны, если хотите добиться скорости, вы должны оптимизировать код для платформы, на которой этого не требуется! Как поступают **htons()** и её семейство.)

Теперь немного кода, который преобразует числа с плавающей запятой одинарной и двойной точности в формат IEEE-754. (Обычно он не преобразует NaN и Неопределённость, но может быть модифицирован для этого.) Вот он[31]:

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

uint64_t pack754(long double f, unsigned bits, unsigned expbits)
{
   long double fnorm;
   int shift;
   long long sign, exp, significand;
   unsigned significandbits = bits - expbits - 1;       // -1 для бита знака

   if (f == 0.0) return 0;                              // особый случай, нельзя!

   // проверить знак и начать нормализацию
   if (f < 0) { sign = 1; fnorm = -f; }
```

---

[30] http://en.wikipedia.org/wiki/IEEE_754

[31] http://beej.us/guide/bgnet/examples/ieee754.c

```
    else { sign = 0; fnorm = f; }

    // получить нормализованную форму f и отследить экспоненту
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // вычислить двоичную (не-плавающую) форму мантиссы
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // получить смещённую экспоненту
    exp = shift + ((1<<(expbits-1)) - 1);              // сдвиг + смещение

    // вернуть ответ
    return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) | significand;
}

long double unpack754(uint64_t i, unsigned bits, unsigned expbits)
{
    long double result;
    long long shift;
    unsigned bias;
    unsigned significandbits = bits - expbits - 1;     // -1 для бита знака

    if (i == 0) return 0.0;

    // извлечь мантиссу
    result = (i&((1LL<<significandbits)-1));           // маска
    result /= (1LL<<significandbits);                  // обратно в плавающую
    result += 1.0f;                                    // добавить назад единицу

    // работа с экспонентой
    bias = (1<<(expbits-1)) - 1;
    shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
    while(shift > 0) { result *= 2.0; shift--; }
    while(shift < 0) { result /= 2.0; shift++; }

    // установить знак
    result *= (i>>(bits-1))&1? -1.0: 1.0;
    return result;
}
```

Сверху я поставил макросы для упаковки и распаковки 32-битных (возможно `float`) и 64-битных (возможно `double`) чисел, но **`pack754()`** может быть вызвана непосредственно с указанием битовой значимости данных (*`expbits`* которых зарезервировано для нормализованного порядка числа).

Вот пример использования:

```
#include <stdio.h>
#include <stdint.h>         // определяет типы uintN_t
#include <inttypes.h>       // определяет макросы PRIx

int main(void)
{
    float f = 3.1415926, f2;
    double d = 3.14159265358979323, d2;
    uint32_t fi;
    uint64_t di;

    fi = pack754_32(f);
    f2 = unpack754_32(fi);
```

```
    di = pack754_64(d);
    d2 = unpack754_64(di);

    printf("float before : %.7f\n", f);
    printf("float encoded: 0x%08" PRIx32 "\n", fi);
    printf("float after  : %.7f\n\n", f2);

    printf("double before : %.20lf\n", d);
    printf("double encoded: 0x%016" PRIx64 "\n", di);
    printf("double after  : %.20lf\n", d2);

        return 0;
}
```

Этот код выдаёт вот это:

```
float before : 3.1415925
float encoded: 0x40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double after  : 3.14159265358979311600
```

У вас может возникнуть вопрос, как вы упаковываете структуры? К несчастью для вас, компилятор волен размещать всё в структурах сам, и, значит, вы не можете переносимо послать её по проводам одним куском. ( Вы ещё не устали слышать “нельзя то”, “нельзя это”? Извините! Процитирую друга: “Какая бы неприятность ни случилась, я всегда виню Microsoft.” Надо сказать, что может быть в данном случае это вина не Microsoft, но утверждение моего друга полная правда.)

Вернёмся к нашим баранам: лучший способ посылать структуру по проводам, это упаковать каждое поле независимо и распаковать их в структуру на другой стороне.

Вы думаете, это же много работы. Да, это так. Но вы можете написать вспомогательную функцию, которая поможет упаковать данные. Это будет весело! Действительно!

В книге “Практика Программирования[32]” Кернигана и Пайка (K&P) они привели **printf()**-подобные функции **pack()** и **unpack()**, которые выполняют то же самое. Я заходил на этот сайт, но, вероятно, этих функций, как и остального исходного кода из книги, там нет.

(“Практика Программирования” превосходное чтиво. Каждый раз, когда я рекомендую эту книгу, Зевс спасает котёнка.)

Здесь я хочу указать на лицензированную BSD книгу Typed Parameter Language C API[33], достойную уважения, но которую я никогда не использовал. Программисты Python и Perl захотят проверить функции **pack()** и **unpack()** своих языков, выполняющих то же самое. И Ява имеет весьма внушительный Сериализуемый Интерфейс, который может быть использован подобным образом.

Но если вы захотите написать свою утилиту упаковки на C, для построения пакета воспользуйтесь трюком от K&P - переменным списком аргументов для создания **printf()**-подобных функций. Основываясь на этом я состряпал свою собственную версию[34], которой, надеюсь, будет достаточно, чтобы дать вам представление как это работает.

---

[32] //cm.bell-labs.com/cm/cs/tpop/

[33] http://tpl.sourceforge.net/

[34] http://beej.us/guide/bgnet/examples/pack2.c

(Этот код обращается к описанным выше функциям **pack754()**. Функции **packi*()** действуют подобно знакомому семейству **htons()**, разве что они упаковывают в символьный массив вместо другого целого.)

```
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>

// распределение битов в плавающих типах
// отличается в разных архитектурах
typedef float float32_t;
typedef double float64_t;

/*
** packi16() -- сохранить 16-бит int в char буфере (как htons())
*/
void packi16(unsigned char *buf, unsigned int i)
{
   *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() -- сохранить 32-бит int в char буфере (как htonl())
*/
void packi32(unsigned char *buf, unsigned long i)
{
   *buf++ = i>>24; *buf++ = i>>16;
   *buf++ = i>>8;  *buf++ = i;
}

/*
** unpacki16() -- распаковать 16-бит int из char буфера (как ntohs())
*/
unsigned int unpacki16(unsigned char *buf)
{
   return (buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- распаковать 32-бит int из char буфера (как ntohl())
*/
unsigned long unpacki32(unsigned char *buf)
{
   return (buf[0]<<24) | (buf[1]<<16) | (buf[2]<<8) | buf[3];
}

/*
** pack() -- упаковать данные в буфер согласно формату
**
**  h - 16-бит              l - 32-бит
**  c - 8-бит char          f - плавающее, 32-бит
**  s - строка (начинается с 16-бит длины)
*/
int32_t pack(unsigned char *buf, char *format, ...)
{
   va_list ap;
   int16_t h;
   int32_t l;
   int8_t c;
```

```
    float32_t f;
    char *s;
    int32_t size = 0, len;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-бит
            size += 2;
            h = (int16_t)va_arg(ap, int); // продвинуто
            packi16(buf, h);
            buf += 2;
            break;

        case 'l': // 32-bit
            size += 4;
            l = va_arg(ap, int32_t);
            packi32(buf, l);
            buf += 4;
            break;

        case 'c': // 8-бит
            size += 1;
            c = (int8_t)va_arg(ap, int); // продвинуто
            *buf++ = (c>>0)&0xff;
            break;

        case 'f': // плавающее
            size += 4;
            f = (float32_t)va_arg(ap, double); // продвинуто
            l = pack754_32(f); // преобразовать в IEEE 754
            packi32(buf, l);
            buf += 4;
                    break;

        case 's': // строка
            s = va_arg(ap, char*);
            len = strlen(s);
            size += len + 2;
            packi16(buf, len);
            buf += 2;
            memcpy(buf, s, len);
            buf += len;
            break;
             }
        }

        va_end(ap);

    return size;
}

/*
** unpack() -- распаковать данные в буфер согласно формату
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int16_t *h;
    int32_t *l;
```

```
    int32_t pf;
    int8_t *c;
    float32_t *f;
    char *s;
    int32_t len, count, maxstrlen=0;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-бит
            h = va_arg(ap, int16_t*);
            *h = unpacki16(buf);
            buf += 2;
            break;

        case 'l': // 32-бит
            l = va_arg(ap, int32_t*);
            *l = unpacki32(buf);
            buf += 4;
            break;

        case 'c': // 8-бит
            c = va_arg(ap, int8_t*);
            *c = *buf++;
            break;

        case 'f': // плавающее
            f = va_arg(ap, float32_t*);
            pf = unpacki32(buf);
            buf += 4;
            *f = unpack754_32(pf);
            break;

        case 's': // строка
            s = va_arg(ap, char*);
            len = unpacki16(buf);
            buf += 2;
            if (maxstrlen > 0 && len > maxstrlen) count = maxstrlen - 1;
            else count = len;
            memcpy(s, buf, count);
            s[count] = '\0';
            buf += len;
            break;

        default:
            if (isdigit(*format)) { // отследить максимальную длину строки
                maxstrlen = maxstrlen * 10 + (*format-'0');
              }
            }

        if (!isdigit(*format)) maxstrlen = 0;
    }

    va_end(ap);
}
```

Вот демонстрационная программа[35], использующая этот код, которая упаковывает какие-то данные в *buf* и затем распаковывает их в переменные. Обратите внимание на

[35] http://beej.us/guide/bgnet/examples/pack2.c

вызов **unpack()** со строковым аргументом (спецификатор “s”), очень мудро ставить в начале счётчик максимальной длины, во избежание переполнения буфера, например “96s”. Будьте осторожны при распаковке полученных по сети данных - злокозненный пользователь может послать вредно построенные пакеты в попытке атаковать вашу систему!

```
#include <stdio.h>

// распределение битов в плавающих типах
// отличается в разных архитектурах
typedef float float32_t;
typedef double float64_t;

int main(void)
{
   unsigned char buf[1024];
   int8_t magic;
   int16_t monkeycount;
   int32_t altitude;
   float32_t absurdityfactor;
   char *s = "Great unmitigated Zot!  You've found the Runestaff!";
   char s2[96];
   int16_t packetsize, ps2;

   packetsize = pack(buf, "chhlsf", (int8_t)'B', (int16_t)0, (int16_t)37,
         (int32_t)-5, s, (float32_t)-3490.6677);

   packi16(buf+1, packetsize);                         // запомнить размер пакета

   printf("packet is %" PRId32 " bytes\n", packetsize);

   unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount, &altitude, s2,
      &absurdityfactor);

   printf("'%c' %" PRId32" %" PRId16 " %" PRId32
         " \"%s\" %f\n", magic, ps2, monkeycount,
         altitude, s2, absurdityfactor);

     return 0;
}
```

Прокручиваете вы свой собственный код или что-либо ещё, неплохо иметь общий набор программ упаковки данных, чем каждый раз упаковывать каждый бит вручную.

Какой формат годится для упаковки данных? Прекрасный вопрос. К счастью, RFC 4506[36], Стандарт Представления Внешних Данных, уже определил двоичные форматы для связывания данных различных типов, как то, с плавающей запятой, целочисленных, массивов, произвольных данных и т.д. Я предлагаю придерживаться его если вы собираетесь делать всё вручную. Но вы не обязаны. Полиция Пакетов не стоит за вашей дверью. По крайней мере, я *не думаю*, что они там.

В любом случае, так или эдак закодировать данные перед посылкой, это правильно!

## 7.5. Дитя Инкапсуляции Данных

Что, в конце концов, означает инкапсуляция данных? В простейшем случае, это значит, что вы подклеиваете к ним заголовок, содержащий либо идентифицирующую информацию, либо длину, либо обе.

[36] http://tools.ietf.org/html/rfc4506

Как должен выглядеть ваш заголовок? Ну, это просто некоторые двоичные данные, представляющие нечто, по вашему мнению необходимое для завершения проекта.

Ух. Туманно.

Ладно. Например, скажем у вас программа многопользовательского чата, которая использует SOCK_STREAM. Когда пользователь что-то печатает (“говорит”) серверу надо передавать два вида информации: кто сказал и что сказал.

Пока что всё хорошо? Вы спросите: “В чём проблема?”

Проблема в том, что сообщения могут быть переменной длины. Некто по имени “tom” может сказать “Hi”, и другой, по имени “Benjamin” может сказать “Hey guys what is up?”

И вы посылаете всё это клиенту так как оно пришло. Ваш исходящий поток выглядит вот так:

```
t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

И так далее. Как клиент узнаёт, где заканчивается одно сообщение и начинается другое? Вы можете, если хотите, сделать все сообщения одной длина и просто вызвать ранее сделанную **sendall()**. Но это растрата полосы пропускания! Мы не хотим посылать 1024 байта чтобы “tom” мог сказать “Hi”.

И мы *инкапсулируем* данные в крохотный заголовок и структуру пакета. И клиент и сервер знают, как упаковать и распаковать (“маршализировать” и “размаршализировать”) эти данные. Теперь не смотрите, а мы начинаем определять *протокол*, по которому клиент и сервер общаются!

Для этого случая давайте примем, что имя пользователя фиксированной длины 8 байт, дополненное ’\0’. И затем примем, что данные у нас переменной длины до 128 байт. Взглянем на структуру пакета, которую мы можем использовать в этой ситуации:

1. len (1 байт, без знака) - Общая длина пакета, включая 8-байтное имя пользователя и данные чата.
2. name (8 байт) - имя пользователя, при необходимости дополненное нулями.
3. chatdata (*n*-байт) - сами данные, не более 128 байт. Длина пакета вычисляется как сумма этих данных плюс 8 (длина поля name).

Почему я выбрал 8 и 128 байтовые пределы для полей? Я вытащил их на свет божий полагая, что они достаточно длинны. Хотя, если 8 байт может быть маловато для ваших нужд, вы можете иметь 30-байтовое поле имени или ещё как. Выбор за вами.

При использовании этого определения первый пакет будет содержать следующую информацию ( в шестнадцатиричном и символьном виде):

```
   0A      74 6F 6D 00 00 00 00 00      48 69
(length)  T  o  m    (padding)          H  i
```

И второй пакет подобен:

```
   18      42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(length)  B  e  n  j  a  m  i  n        H  e  y     g  u  y  s     w ...
```

(Конечно же, длина хранится в Порядке Байтов Сети. В данном случае она имеет только один байт, так что это не имеет значения, но вообще-то говоря вы захотите хранить ваши целые в пакете в Порядке Байтов Сети.)

Посылая данные вы должны быть предусмотрительны и использовать команду, подобную **sendall()** выше чтобы знать, что все данные посланы, даже если пришлось использовать множество вызовов **send()**.

Кроме того, принимая эти данные, вы должны сделать ещё кое-что. Вы должны предусматривать, что можете принять часть пакета (как “18 42 65 6E 6A” из Benjamin-а

выше, но это всё, что мы приняли от этого вызова **recv()**.) Нам нужно вызывать **recv()** снова и снова, пока не примем весь пакет полностью.

Но как? Нам известно общее количество байт, которое нужно принять для завершения пакета, поскольку это число пришито к пакету спереди. Также мы знаем, что максимальный размер пакета равен 1+8+128, или 137 байт ( мы так его определили).

В действительности мы можем сделать пару вещей. Поскольку вы знаете, что каждый пакет начинается с длины, вы можете вызвать **recv()** и принять только длину пакета. Затем, зная длину пакета, вы можете вызвать её снова, указав точную длину оставшейся части пакета (возможно повторяя вызов, чтобы получить весь пакет). Преимущество этого метода в том, что вы можете иметь только один буфер длиной в один пакет, а недостаток в том, что вам придётся вызывать **recv()** по меньшей мере дважды.

Другой вариант это указать **recv()**, что вы хотите принять пакет максимальной длины, подклеить то что пришло к хвосту буфера и, наконец, проверить, завершён ли пакет. Разумеется, вы можете получить кое-что из следующего пакета, так что вам нужно иметь для этого место.

Вы должны объявить достаточно большой массив, чтобы вместить два пакета. Это ваш рабочий массив, в котором вы будете перестраивать пакеты по мере их появления.

Каждый раз, приняв данные вы добавите их в рабочий буфер и проверите пакет на завершение. То есть, если количество байт в буфере больше или равно указанной в заголовке (+1, потому что длина в заголовке не содержит её саму.) Если число байт в буфере меньше 1, то, ясно, что пакет не завершён. Для этого случая вам нужно предусмотреть особую обработку, поскольку на то, что первый байт содержит правильную длину полагаться нельзя.

Как только пакет завершён, можете делать с ним всё что вам угодно. Используйте и удалите из рабочего буфера.

Ну что, в голове ещё шумит? Ладно, вот второй из парочки ударов: за один вызов **recv()** вы могли прочесть конец одного пакета и перейти на следующий. Это означает, что в рабочем буфере у вас один завершённый пакет и незавершённая часть следующего! Чёрт побери. (Вот поэтому вы и сделали ваш рабочий буфер достаточно большим, чтобы вмещать *два* пакета - на случай если это произойдёт!)

Поскольку вы теперь знаете из заголовка длину первого пакета и следите за количеством байт в буфере вы можете вычислить количество байт, принадлежащих второму (незавершённому) пакету. После обработки первого пакета вы можете удалить его из буфера и сдвинуть часть второго пакета в начало буфера, приготовив всё для следующего **recv()**.

(Некоторые из читателей заметят, что перемещение части пакета в начало рабочего буфера занимает время и этого не потребуется если написать программу с использованием закольцованного буфера. К несчастью для остальных из вас, обсуждение закольцованных буферов выходит за рамки этой статьи. Если вам до сих пор интересно, хватайте книгу по структуре данных и начинайте оттуда.)

Я никогда не говорил, что это легко. Ладно, я говорил, что это легко. И это так; просто вам нужна практика и, естественно, я очень скоро приду к вам. Клянусь Эскалибуром!

## 7.6. Широковещательные пакеты - Hello, world!

До сих пор этот документ рассказывал о том, как посылать данные от одного хоста другому. А я утверждаю, что, имея соответствующие полномочия, можно посылать данные многим хостам *одновременно*!

Используя UDP (только UDP, не TCP) в стандартном IPv4 это делается через механизм, называемый *широковещанием*. В IPv6 широковещание не поддерживается и вам нужно

прибегнуть к часто превосходящей технике *мультивещания*, которую, к прискорбию, я в этот раз обсуждать не буду. Но хватит о звёздноглазом будущем - мы застряли в 32-битном настоящем.

Подождите! Вы не можете просто выскочить и вещать во всю ивановскую. Прежде чем посылать широковещательный пакет в сеть вам нужно установить сокету опцию `SO_BROADCAST`. Это как маленькая пластиковая крышечка, которую устанавливают на кнопку запуска баллистической ракеты! Вот сколько мощи вы держите в своих руках!

Тем не менее, если серьёзно, опасность широковещательных пакетов в том, что каждая система, принявшая широковещательный пакет должна очистить шелуху слоёв инкапсуляции, чтобы добраться до порта, которому это предназначено. И затем применить или отбросить это.

В любом случае, это много ненужной работы для каждой машины в локальной сети, а они все принимают широковещательные пакеты. Когда игра Doom впервые вышла в свет, было много жалоб на её сетевой код.

На свете существуют больше одного способа посылки широковещательных пакетов. Так что объединим картошку с мясом: как вам указать адрес назначения для широковещательного послания? Есть два общепринятых способа:

1. Послать данные по широковещательному адресу отдельной подсети. Это сетевой номер подсети со всеми установленными битами номера хоста. Например, моя домашняя сеть имеет номер 192.168.1.0, моя сетевая маска 255.255.255.0, значит последний байт адреса это номер хоста (потому что первые три байта, соответственно маске, это номер сети). Так что мой широковещательный адрес 192.168.1.255. Под Unix, команда **ifconfig** действительно выдаст вам все эти данные. (Выражение в двоичной логике - *сетевой_номер* OR (NOT *сетевая_маска*), если вам интересно.) Вы можете послать этот тип широковещательного пакета и в удалённую сеть тоже, но при этом рискуете, что пакет будет отброшен маршрутизатором сети назначения.(Если он этого не сделает, то какой-нибудь случайный смёрфер может утопить его сеть в широковещательном трафике.)

2. Послать данные по “глобальному” широковещательному адресу. Это 255.255.255.255, aka `INADDR_BROADCAST`. Многие машины логическим И с вашим сетевым номером автоматически преобразуют его в сетевой широковещательный адрес, а некоторые нет. Это переменчиво. По иронии судьбы, маршрутизаторы не выпускают этот тип широковещательных пакетов за пределы вашей локальной сети.

Что происходит, если вы пытаетесь послать данные по широковещательному адресу без установки опции `INADDR_BROADCAST`? Давайте запустим старые добрые `talker` и `listener` и посмотрим, что произойдёт.

```
$ talker 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied
$ talker 255.255.255.255 foo
sendto: Permission denied
```

Да, неудачно, и всё потому что мы не установили опцию `SO_BROADCAST`. Установите её и вызывайте **`sendto()`** где захотите!

В действительности, существует только *одна разница* между UDP приложениями, которые могут и не могут посылать широковещательные сообщения. Давайте возьмём

старое приложение `talker` и добавим участок, который устанавливает опцию `SO_BROADCAST`. Назовём её broadcaster.c[37]:

```
/*
** broadcaster.c -- дейтаграммный “клиент” подобный talker.c, но
**              этот может вещать
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950             // порт для подключения пользователей

int main(int argc, char *argv[])
{
   int sockfd;
   struct sockaddr_in their_addr;          // адресная информация подключившегося
   struct hostent *he;
   int numbytes;
   int broadcast = 1;

  //char broadcast = '1';                     // если то не работает, попробуйте так
   if (argc != 3) {
       fprintf(stderr,"usage: broadcaster hostname message\n");
       exit(1);
       }
   if ((he=gethostbyname(argv[1])) == NULL) {  // получить информацию хоста
       perror("gethostbyname");
       exit(1);
   }

   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
       perror("socket");
       exit(1);
       }

   // этот вызов позволяет посылать широковещательные пакеты:
   if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
       sizeof broadcast) == -1) {
       perror("setsockopt (SO_BROADCAST)");
       exit(1);
       }

   their_addr.sin_family = AF_INET;                    // порядок байтов хоста
   their_addr.sin_port = htons(SERVERPORT);      // short, порядок байтов сети
   their_addr.sin_addr = *((struct in_addr *)he->h_addr);
   memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);
   if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
            (struct sockaddr *)&their_addr, sizeof their_addr)) == -1) {
       perror("sendto");
       exit(1);
```

[37] http://beej.us/guide/bgnet/examples/broadcaster.c

```
    }

    printf("sent %d bytes to %s\n", numbytes,
        inet_ntoa(their_addr.sin_addr));

    close(sockfd);

        return 0;
}
```

Какая разница между этой и “нормальной” ситуацией UDP клиент/сервер? Никакой! (За исключением того, что клиенту в данном случае разрешено посылать широковещательные сообщения.) Так что, вперёд, запускайте старую UDP программу `listener` в одном окне и `broadcaster` в другом. Теперь вы можете сделать то, что ранее отказывало.

```
$ broadcaster 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ broadcaster 192.168.1.255 foo
sent 3 bytes to 192.168.1.255
$ broadcaster 255.255.255.255 foo
sent 3 bytes to 255.255.255.255
```

Вы видите, `listener` отвечает, что он пакеты получает. (Если `listener` не отвечает, то может быть потому что он подключён к IPv6 адресу. Попробуйте изменить в *`listener.c`* `AF_UNSPEC` на `AF_INET`, чтобы включить IPv4.)

Это нечто возбуждающее. Попробуйте запустить `listener` на соседней машине в той же сети, чтобы иметь две копии, по одной на каждой машине, и снова запустите `broadcaster` с вашим широковещательным адресом… Оба `listener`-а получают пакет, даже если вы вызываете **`sendto()`** единожды! Круто!

Если **listener** получает данные, которые вы посылаете непосредственно ему, и не получает по широковещательному адресу, то может быть на вашей машине установлен брандмауэр, который блокирует эти пакеты. (Да, спасибо вам, Пэт и Баппер, что вы раньше меня поняли почему мой пример не работает. Я сказал, что упомяну вас в этом руководстве, я сделал. Вот так.)

Опять же, будьте осторожны с широковещательными пакетами. Поскольку все машины в локальной сети будут вынуждены иметь дело с пакетами, независимо от того, есть для них **`recvfrom()`** или нет, это может представлять собой значительную нагрузку для всей вычислительной сети. Определённо, их нужно использовать скупо и подобающим образом.

# 8. Общие вопросы

**Где взять заголовочные файлы?**

Если на вашей системе их нет, то возможно, они вам не нужны. Справьтесь в мануале по вашей платформе. Если вы работаете на Windows, вам нужен только `#include <winsock.h>`.

**Что делать когда `bind()` выдаёт "Address already in use"?**

Вам нужно использовать **`setsockopt()`** с опцией `SO_REUSEADDR` на слушающем сокете. Для примера посмотрите разделы по **`bind()`** и **`select()`**.

**Как получить список открытых сокетов в системе?**

Используйте **netstat**. За деталями обратитесь в **man** страницы, но вы должны получить некоторые полезные данные напечатав:

```
$ netstat
```

Это единственный способ определить, какой сокет ассоциирован с какой программой. :-)

**Как посмотреть таблицу маршрутизации?**

Задайте команду **route** (в */sbin* на большинстве Linux-ов) или команду **netstat -r.**

**Как запустить клиент и сервер если у меня только один компьютер? Нужна ли сеть для написания сетевых программ?**

К счастью, фактически все машины имеют закольцованное (*loopback*) сетевое "устройство", которое сидит в ядре и претендует на звание сетевой карты. (Это интерфейс, именуемый как "lo" в таблице маршрутизации.)

Положим вы вошли в машину под именем "**goat**". Запустите клиент в одном окне и сервер в другом. Или запустите сервер в фоновом режиме ("**server &**") и клиент в том же окне. В итоге с *loopback*-устройством вы можете задавать **client goat** или **client localhost** (поскольку "`localhost`", вероятно определён в вашем файле */etc/hosts*) и у вас будет клиент, разговаривающий с сервером без сети!

Кратко, вносить изменения в программу, чтобы она работала на отдельной, не подключённой к сети машине, не нужно! Ура!

**Как мне узнать, что удалённая сторона закрыла соединение?**

Вы узнаете, потому что **`recv()`** вернёт 0.

**Как выполнять утилиту "ping"? Что такое ICMP? Где найти сведения по сырым сокетам и `SOCK_RAW`?**

Ответы на все ваши вопросы по сырым сокетам есть в книгах W. Richard Stevens' UNIX Network Programming. Также посмотрите в подоглавлении *ping/* в Stevens' UNIX Network Programming source code, доступной онлайн[38].

[38] http://www.unpbook.com/src.html

### Как изменить или сократить таймаут вызова connect()?

Вмасто того, чтобы давать тот же ответ, что и W. Richard Stevens, я просто сошлюсь на *lib/connect_nonb.c* в UNIX Network Programming source code[39].

Суть в том, что вы создаёте дескрипор сокета вызовом **socket()**, делаете его неблокируемым, вызываете **connect()** и, если всё идёт хорошо, **connect()** немедленно возвратит -1 и установит ***errno*** в EINPROGRESS. Затем вызваете **select()** с нужным таймаутом, передавая дескриптор в обоих массивах, чтения и записи. Если таймаут не сработал, значит вызов **connect()** завершён. В этом месте вам нужно использовать **getsockopt()** с опцией SO_ERROR чтобы получить возврат из функции **connect()**, который должен быть нулевым, если не было ошибки.

В конце, возможно, вам захочется до начала передачи данных через сокет опять сделать его блокируемым.

Заметьте, что в вашу программу была добавлена возможность делать что-либо ещё, пока она подключается. Например, вы можете установить таймаут маленьким, вроде 500 ms, обновлять индикатор на экране каждый таймаут и вызывать **select()** снова. Когда таймаут вызовов истечёт, скажем, 20 раз, вы будете знать, что пора отказаться от соединения.

Как я сказал, посмотрите у Stevens-а исходный код совершенно превосходных примеров.

### Как писать для Windows?

Первым делом удалите Windows и установите Linux или BSD. };-). Нет, действительно, посмотрите раздел по программированию для Windows во введении.

### Как писать для Solaris/SunOS? Когда я компилирую, то продолжаю получать ошибки компоновщика!

Ошибки компоновщика происходят потому что на платформе Sun библиотеки сокетов не подключаются автоматически. Посмотрите примеры в разделе по программированию для Solaris/SunOS во введении.

### Почему select() не ладит с сигналами?

Сигналы имеют склонность принуждать заблокированные системные вызовы возвращать -1, устанавливая ***errno*** в EINTR. Когда вы устанавливаете обработчик сигналов функцией **sigaction()**, вы можете установить флаг SA_RESTART, который предполагает перезапуск системного вызова после того как он был прерван.

Естественно, это не всегда работает.

Моё любимое решение привлекать оператор goto. Как вы знаете, это до бесконечности раздражает ваших профессоров, так что соглашайтесь!

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
   if (errno == EINTR) {
      // какой-то сигнал нас прервал, перезапуск
      goto select_restart;
   }
     // обработка настоящих ошибок
   perror("select");
}
```

[39] http://www.unpbook.com/src.html

Уверен, применять goto в данном случае *необходимости* нет, вы можете использовать другие структуры. Но, по-моему, goto намного понятнее.

### Как применять таймаут при вызове **recv()**?

Используйте **select()**! Она позволяет вам указывать параметр таймаута для дескриторов сокетов, через которые вы намереваетесь читать. Или можете объединить все действия в одной функции, как здесь:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

int recvtimeout(int s, char *buf, int len, int timeout)
{
   fd_set fds;
   int n;
   struct timeval tv;

   // подготовка массива файловых дескрипторов
   FD_ZERO(&fds);
   FD_SET(s, &fds);

   // подготовка struct timeval для таймаута
   tv.tv_sec = timeout;
   tv.tv_usec = 0;

   // ждём таймаут или данные
   n = select(s+1, &fds, NULL, NULL, &tv);
   if (n == 0) return -2; // таймаут!
   if (n == -1) return -1; // ошибка

   // это данные, так что нормальный recv()
   return recv(s, buf, len, 0);
}
.
.
.
    // recvtimeout() извлекаем таймаут:
    n = recvtimeout(s, buf, sizeof buf, 10); // 10 секунд таймаут

    if (n == -1) {
   // ошибка случилась
   perror("recvtimeout");
     }
     else if (n == -2) {
   // таймаут пришёл
     } else {
   // в буфере есть данные
     }
.
.
.
```

Заметьте, что **recvtimeout()** в случае таймаута возвращает -2. Почему не ноль? Потому что **recv()** возвращает 0, если удалённая сторона закрыла соединение. Так что это значение уже занято и я выбрал -2, как мой индикатор таймаута.

**Как кодировать или сжимать данные перед посылкой через сокет?**

Простой способ кодирования - использовать SSL (secure sockets layer), но это выходит за рамки этого руководства. (Смотрите OpenSSL project[40].)

Полагая, что вы хотите подключить или внедрить вашу собственную систему сжатия или кодирования данных, самое время подумать о том, что они пройдут через последовательность шагов между обоими концами. Каждый шаг как-то изменяет данные.

1. сервер читает данные из файла (или ещё откуда-то)
2. сервер кодирует/сжимает данные (это добавляете вы)
3. сервер посылает кодированные данные (**send()**)

Теперь наоборот:

1. клиент принимает кодированные данные (**recv()**)
2. клиент декодирует/разворачивает данные (это добавляете вы)
3. клиент пишет данные в файл(или ещё куда-то)

Если вы собираетесь и сжимать и кодировать, помните, что сначала сжимают. :-)

Как только клиент совершит действия, обратные действиям сервера, данные будут в порядке и в итоге не важно, сколько шагов вы добавили.

Так что при использовании моего кода вам достаточно найти место между чтением данных и посылкой их в сеть и вклеить некоторый код, выполняющий кодирование.

**Я постоянно вижу "PF_INET", что это? Это связано с AF_INET?**

Да, это так. Смотрите раздел по **socket()**.

**Как написать серверу, чтобы принял от клиента команды и выполнил их?**

Давайте для простоты примем, что клиент подключается (**connect()**), посылает данные (**send()**) и закрывает соединение (**close()**) (т.е. для последующих вызовов клиент подключается вновь).

Процесс для клиента такой:

1. подключиться к серверу (**connect()**)
2. послать (**send("/sbin/ls > /tmp/client.out")**)
3. закрыть соединение (**close()**)

Тогда как сервер принимает данные и выполняет их:

1. принимает соединение от клиента (**accept()**)
2. получает командную строку (**recv(str)**)
3. закрывает соединение (**close()**)
4. запускает команду (**system(str)**)

Осторожно! Позволить серверу выполнять команды клиента - это позволить ему вытворять с вашим аккаунтом всё что угодно когда он подключится к серверу. Например,

---

[40] http://www.openssl.org/

что если клиент пришлёт команду “`rm -rf ~`”? Это удалит всё в вашем аккаунте, вот что это!

Так что будьте мудрее и не позволяйте клиенту ничего исполнять кроме парочки утилит, которые, вы знаете, безопасны, как утилита **foobar**:

```
if (!strncmp(str, "foobar", 6)) {
   sprintf(sysstr, "%s > /tmp/server.out", str);
   system(sysstr);
}
```

К сожалению, это до сих пор небезопасно: что если клиент пришлёт “`foobar; rm -rf ~`”? Самая безопасное это написать маленькую программку, которая будет вставлять эскейп-символ (“\”) перед всеми не алфавитно-цифровыми символами (включая если можно пробелы) в аргументах команды.

Как видите, безопасность это весьма большая проблема, если вы позволяете серверу выполнять команды клиента.

### Я посылаю массу данных, но за раз приходит только 536 или 1460 байт. Но если запуститься на моей локальной машине, то данные принимаются за один раз. В чём дело?

Вы превысили MTU - максимальный размер пакета, который может обработать физическая среда. На локальной машине вы используете *loopback* устройство, которое без проблем может обрабатывать 8K или больше. А на Ethernet, который может обрабатывать только 1500 байт с заголовком, вы этот предел превысили. С модемом с 576 MTU (опять же, с заголовком), вы превысили этот нижний предел.

Прочтите раздел *Дитя Инкапсуляции Данных* о том как принимать целые пакеты данных многократным вызовом **`recv()`**.

### На платформе Windows у меня нет системного вызова fork() и любого вида struct sigaction. Что делать?

Если они где и есть, так это в библиотеках POSIX, которые могут поставляться с вашим компилятором. Поскольку платформы Windows у меня нет, я не могу ответить, но, помнится, у Microsoft есть уровень совместимости с POSIX, где **`fork()`** должен быть. (И может быть даже **`sigaction`**.)

Поищите в системе помощи VC++ “fork” или “POSIX” и, может быть, он даст вам какую-нибудь нить.

Если это совсем не работает, выбросьте **`fork()/sigaction`** на помойку и замените их эквивалентом из Win32: **CreateProcess()**. Я не знаю как **CreateProcess()** работает - она принимает базиллион аргументов, но это должно быть в документации, поставляемой с VC++.

### Я под брандмауэром - как указать людям за ним мой IP адрес, чтобы они подключились к моей машине?

К сожалению, цель брандмауэра - предотвратить подключение людей за брандмауэром к машинам под ним, так что разрешение такого подключения повсеместно рассматривается как брешь в безопасности.

Но нельзя сказать, что всё потеряно. Одна вещь, вы до сих пор можете подключаться через брандмауэр, даже если он использует какой-нибудь маскарадинг или NAT или что-то вроде этого. Просто напишите программу так, чтобы инициатором соединения всегда были вы, и будет вам счастье.

Если это не удовлетворяет, попросите вашего системного администратора проткнуть дыру в брандмауэре, чтобы люди могли к вам подключаться. Брандмауэр может пробираться к вам либо через свои NAT программы либо через прокси либо через нечто подобное.

Но знайте, что дыра в брандмауэре так просто не обнаруживается. Вы должны быть уверены, что не даёте плохим людям доступа во внутреннюю сеть, если вы новичок, запомните, сделать программы безопасными намного труднее, чем это можно представить.

Не сводите с ума вашего системного администратора вместе со мной. ;-)

### Как написать анализатор пакетов? Как переключить мой Ethernet интерфейс в беспорядочный режим?

Для тех кто не знает, когда сетевая карта находится в "беспорядочном режиме", она пропускает в операционную систему ВСЕ пакеты, а не только адресованные этой конкретной машине. (Здесь мы говорим не об IP адресах, а об адресах Ethernet-уровня, и, поскольку уровень Ethernet ниже, чем IP, все IP адреса замечательно передаются. Смотрите раздел *Низкоуровневый Вздор и Теория сетей*.)

Это основа того, как работает анализатор пакетов. Он переключает интерфейс в беспорядочный режим и ОС получает все ходящие по проводам пакеты. Вам нужно иметь сокет некоторого типа, через который вы можете читать эти данные.

К сожалению, ответ на этот вопрос меняется в зависимости от платформы, но если вы дадите Google запрос, например, "windows promiscuous ioctl", то может что-то и получите. Кроме того, достойно выглядит статья в Linux Journal[41].

### Как установить своё значение таймаута для TCP или UDP сокета?

Это зависит от вашей системы. Можете поискать в сети `SO_RCVTIMEO` и `SO_SNDTIMEO` (для использования с **`setsockopt()`**), чтобы узнать поддерживает ли ваша система такие операции.

**`Man`** страницы Linux предлагают вместо этого использовать **`alarm()`** или **`setitimer()`**.

### Как определить доступные порты? Есть ли список "официальных" номеров портов?

Обычно это не проблема. Если вы пишете, скажем, web сервер, то неплохо бы использовать широко известный порт 80. Если же вы пишете свой специализированный сервер, выберите случайный порт (но больше 1023) и попробуйте.

Если порт уже занят, то **`bind()`** выдаст ошибку "Address already in use". Выберите другой порт. (Неплохая идея позволить пользователю самому определить альтернативный порт в *config* файле либо ключом в командной строке.)

Есть список официальных номеров портов[42], определяемый Internet Assigned Numbers Authority (IANA). То что какой-либо порт (старше 1023) находится в этом списке не означает, что вы не можете его использовать. Например, DOOM от Id Software использует такой порт как "**mdqs**", каким бы он ни был. Важно лишь чтобы в это время никто кроме вас не использовал этот порт *на этой же машине*.

---

[41] http://interactive.linuxjournal.com/article/4659

[42] http://www.iana.org/assignments/port-numbers

# 9. Man Страницы

В мире Unix много мануалов. В них есть маленькие разделы, которые описывают имеющиеся в вашем распоряжении функции.

Конечно, мануал обычно слишком велик чтобы набирать его. Я имею ввиду, никто из мира Unix, включая меня, не любит так много печатать. В действительности я могу до бесконечности печатать о том, как сильно я предпочитаю быть кратким, но вместо этого я буду кратким и не стану докучать вам пространными пламенными речами о том каким чрезвычайно удивительно кратким я предпочитаю быть практически во всех случаях во всей их полноте.

[*Аплодисменты*]

Спасибо. Я добрался до того, что в мире Unix эти страницы называются "**man** страницами" и я включил сюда мои собственные личные усечённые варианты для вашего читательского удовольствия. Вся штука в том, что многие из этих функций суть более общего назначения, чем я описываю, но я намереваюсь представить только ту часть, которая соответствует Программированию Интернет Сокетов.

Погодите! Это ещё не все недостатки моих **man** страниц:

- Они не полные и показывают только основы из руководства.
- Они даже более **man** страницы, чем в реальном мире.
- Они отличаются от таких же в вашей системе.
- Заголовочные файлы могут различаться для определённых функций в вашей системе.
- Параметры функций могут быть другими для определённых функций в вашей системе.

Если вам нужна настоящая информация, посмотрите **man** страницы вашей Unix, напечатав **man whatever**, где "whatever" означает что-то необычайно вам интересное, вроде "`accept`". (Я уверен, в Microsoft Visual Studio есть нечто подобное в их секции помощи. Но "man" лучше, потому что на один байт короче, чем "help". Unix побеждает снова!)

Но если они такие ущербные, зачем вообще включать их в это Руководство? Есть несколько причин, но лучшие из них то, что (a) эти версии специально заточены под сетевое программирование, и (b) эти версии содержат примеры!

И говоря о примерах, я не намерен пускаться во все проверки ошибок потому что это по-настоящему увеличивает длину кода. Но вы должны выполнять проверку ошибок преогромное количество раз при каждом системном вызове пока не будете на 100% уверены, что программа не собирается сбоить и, возможно, даже после этого!

## 9.1. accept()

Принимает входные подключения на слушающем сокете.

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

### Описание

Как только вы пошли сквозь хлопоты получения SOCK_STREAM сокета и поставили его для получения входящих соединений через **listen()**, вы вызываете **accept()** чтобы получить новый дескриптор сокета для последующего общения с только что подключённым клиентом.

Старый сокет, использовавшийся для прослушивания, ещё здесь и будет использован для дальнейшего приёма **accept()**-ом новых вызовов по мере их поступления.

| | |
|---|---|
| *s* | Дескриптор слушаемого сокета. |
| *addr* | Заполняется адресом подключающегося сайта. |
| *addrlen* | Заполняется размером (sizeof()) структуры, возвращённой в параметре *addr*. Можно спокойно игнорировать, если полагаете, что получили назад struct sockaddr_in, потому что именно такой тип вы передавали в параметре *addr*. |

**accept()** обычно блокируется и вы можете использовать **select()**, чтобы заранее взглянуть готов ли он к чтению. Если готов, значит новое подключение ждёт **accept()**-а! Йес! По другому, с помощью **fcntl()** вы можете установить на слушающем сокете флаг O_NONBLOCK, выбирая возврат -1 и установку ***errno*** в EWOULDBLOCK.

Возвращаемый **accept()**-ом дескриптор определяет уже добросовестно открытый и подключённый к удалённому хосту сокет. Вам нужно вызвать **close()** по завершении работы с ним.

### Возвращаемое значение

**accept()** возвращает дескриптор только что подключённого сокета, или -1 при ошибке, при этом соответствующим образом установив ***errno***.

### Пример

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// сначала заполняем адресные структуры с помощью getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;          // использовать либо IPv4 либо IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;          // заполнить мой IP для меня
getaddrinfo(NULL, MYPORT, &hints, &res);

// создать сокет, связать и слушать:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
```

```
listen(sockfd, BACKLOG);

// теперь принять входящие подключения:

addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

// можно беседовать по дескриптору сокета new_fd!
```

### Смотри также

**socket(), getaddrinfo(), listen(),** struct sockaddr_in

## 9.2. bind()

Связывает сокет с IP адресом и номером порта.

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

### Описание

Когда удалённая машина хочет связаться с вашей серверной программой, ей для этого нужны IP адрес и номер порта. А позволяет ей это сделать вызов **bind()**.

Сначала вы вызываете **getaddrinfo()** и заполняете struct sockaddr адресом назначения и информацией порта. Затем вы вызываете **socket()** чтобы получить дескриптор сокета и передаёте сокет и адрес в **bind()**, и вот IP адрес и порт волшебным образом (используя настоящее волшебство) привязан к сокету!

Если вы не знаете своего IP адреса, или вам известно, что у вашей машины только один IP адрес, или вам безразлично, какие IP адреса использованы на вашей машине, просто установите флаг AI_PASSIVE в параметре *hints* при вызове **getaddrinfo()**. При этом в часть IP адреса в struct sockaddr записывается специальное значение, которое указывает **bind()**, что ей нужно автоматически заполнить этот IP адрес хоста.

Что, что? Что за специальное значение записывается в IP адрес struct sockaddr чтобы автоматически установить адрес текущего хоста? Я скажу, но помните, что это происходит только при заполнении struct sockaddr вручную, иначе воспользуйтесь результатом **getaddrinfo()**, как указано выше. В IPv4, поле *sin_addr.s_addr* структуры struct sockaddr_in устанавливается INADDR_ANY. В IPv6, в поле *sin6_addr* структуры sockaddr_in6 записывается значение глобальной переменной *in6addr_any*. Или, если вы объявляете новую struct in6_addr, вы можете инициализировать её IN6ADDR_ANY_INIT.

Напоследок, параметр *addrlen* должен быть установлен в sizeof my_addr.

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
// современный способ работы с getaddrinfo()

struct addrinfo hints, *res;
int sockfd;

// сначала заполняем адресные структуры с помощью getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;          // использовать либо IPv4 либо IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;          // заполнить мой IP для меня

getaddrinfo(NULL, "3490", &hints, &res);

// создать сокет:

// (вам нужно прогуляться по связанному списку "res" и проверить на ошибки!)
```

```
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// связать с портом, переданным getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

```
// пример упаковки структуры вручную, IPv4

struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// можете указать IP адрес:

inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

// или позволить выбрать его автоматически:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

### Смотри также

**getaddrinfo()**, **socket()**, `struct sockaddr_in`, `struct in_addr`

## 9.3. connect()

Подключает сокет к серверу.

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>

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

### Описание

После того, как вы создали дескриптор сокета вызовом **socket()**, вы можете подключить его к удалённому серверу системным вызовом **connect()**. Вам нужно только передать ему дескриптор сокета и адрес сервера, с которым вам захотелось познакомиться поближе. (Да, и длину адреса, которую принято передавать таким функциям.)

Обычно эту информацию получают как результат вызова **getaddrinfo()**, но, если хотите, можете заполнить свою собственную `struct sockaddr` сами.

Если вы ещё не вызывали **bind()** с этим дескриптором сокета, он автоматически привязывается к вашему IP адресу и случайному локальному порту. Обычно это просто замечательно для вас, если вы не сервер, поскольку вам безразличен номер вашего локального порта. Если номер удалённого порта важен, то укажите его в параметре в *serv_addr*. Вы *можете* вызвать **bind()** если действительно хотите, чтобы сокет вашего клиента был привязан к определённому IP адресу и порту, но это бывает весьма редко.

Как только сокет подключён (**connect()**), вы можете спокойно посылать (**send()**) и принимать (**recv()**) данные от него согласно велению вашего сердца.

Отдельное примечание: если вы подключили **connect()** `SOCK_DGRAM` UDP сокет к уделённому хосту, то можете использовать **send()** и **recv()** также как **sendto()** и **recvfrom()**. Если хотите.

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
// соединиться с www.example.com порт 80 (http)
struct addrinfo hints, *res;
int sockfd;

// сначала заполняем адресные структуры с помощью getaddrinfo():
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;          // использовать либо IPv4 либо IPv6
hints.ai_socktype = SOCK_STREAM;

// в это строке можно указать "80" вместо "http":
getaddrinfo("www.example.com", "http", &hints, &res);

// создать сокет:
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// соединить с адресом и портом, переданным getaddrinfo():
connect(sockfd, res->ai_addr, res->ai_addrlen);
```

### Смотри также
**`socket(), bind()`**

## 9.4. close()

Закрывает дескриптор сокета.

### Прототип

```
#include    <unistd.h>

int  close (int s);
```

### Описание

После того как вы закончили использовать сокет в любой состряпанной вами сумасбродной затее и больше не хотите посылать (**send()**) или принимать (**recv()**) данные, и вообще *ничего* с ним не делать, вы можете закрыть (**close()**) его и он будет освобождён, дабы более никогда не использоваться.

Удалённая сторона может узнать об этом одним из двух способов. Первый: Если удалённая сторона вызывает **recv()**, он возвращает 0. Второй: удалённая сторона вызывает **send()**, он примет сигнал SIGPIPE и вернёт -1, ***errno*** будет установлен в EPIPE.

**Пользователям Windows**: функция, которую вам нужно использовать называется **closesocket()**, а не **close()**. Если вы попробуете использовать **close()**, то, возможно, Windows рассердится… И вам не понравится когда она сердится.

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// куча всего...*BRRRONNNN!*
.
.
.
close(s);  // действительно, не очень много.
```

### Смотри также

**socket(), shutdown()**

## 9.5. getaddrinfo(), freeaddrinfo(), gai_strerror()

Получает информацию об имени хоста и/или сервисе и записывает результат в `struct sockaddr`.

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
            const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

const char *gai_strerror(int ecode);

struct addrinfo {
    int    ai_flags;                // AI_PASSIVE, AI_CANONNAME, ...
    int    ai_family;               // AF_xxx
    int    ai_socktype;             // SOCK_xxx
    int    ai_protocol;             // 0 (авто) или IPPROTO_TCP, IPPROTO_UDP

    socklen_t  ai_addrlen;          // длина ai_addr
    char   *ai_canonname;           // каноническое имя
    struct sockaddr  *ai_addr;      // двоичный адрес
    struct addrinfo  *ai_next;      // следующая структура в связанном списке
};
```

### Описание

**`getaddrinfo()`** это превосходная функция, которая возвращает информацию об имени отдельного хоста (такую как IP адрес) и заполняет `struct sockaddr`, заботится о мелких деталях (типа это IPv4 или IPv6.) Она заменяет старые функции **`gethostbyname()`** и **`getservbyname()`**. Описание ниже содержит много информации, которая, возможно, немного устрашает, но реальное использование достаточно просто. Может быть стоит сначала посмотреть примеры.

Имя интересующего вас хоста передаётся в параметре *`nodename`*. Адрес может быть именем хоста, как “www.example.com”, либо IPv4 или IPv6 адрес (передаваемый как строка). Этот параметр также может быть `NULL` если вы используете флаг `AI_PASSIVE` (см. ниже.)

Обычно параметр *`servname`* это номер порта. Он может быть номером (передаваемый строкой, как “80”), или он может быть именем сервиса, как “http” или “tftp” или “smtp” или “pop”, и т.д. Распространённые имена сервисов можно найти в IANA Port List[43] или в вашем системном файле *`/etc/services`*.

Напоследок, во входных параметрах есть *`hints`*. Именно здесь вы определяете что функции **`getaddrinfo()`** нужно делать. Перед использование обнулите всю структуру целиком функцией **`memset()`**. Давайте взглянем на поля, которые вам нужно заполнить до использования.

Поле *`ai_flags`* может содержать множество флагов, вот два из них. (Много флагов указывается поразрядным ИЛИ оператором “`|`”). Полный список флагов приведён в вашей man странице.

[43] http://www.iana.org/assignments/port-numbers

AI_CANONNAME заставляет записать в поле *ai_canonname* результата каноническое (настоящее) имя хоста. AI_PASSIVE приводит к записи в IP адрес INADDR_ANY (IPv4) или *in6addr_any* (IPv6); из-за этого последует вызов **bind()** чтобы автоматически записать в IP адрес структуры struct sockaddr адрес текущего хоста. Это превосходно для запуска сервера если вы не хотите использовать постоянно установленный адрес.

Если вы используете флаг AI_PASSIVE, то в *nodename* можно указать NULL (поскольку **bind()** заполнит его позднее.)

Продолжаем с входными параметрами. Вам лучше всего установить в *ai_family* AF_UNSPEC чтобы **getaddrinfo()** искала и IPv4 и IPv6 адреса. Хотя вы можете ограничить себя одним или другим, установив AF_INET или AF_INET6.

Следующее, в поле *ai_socktype* нужно установить SOCK_STREAM или SOCK_DGRAM, в зависимости от того, какой тип сокета вам нужен.

Наконец, просто оставьте в *ai_protocol* 0 чтобы автоматически выбрать тип вашего протокола.

Теперь, когда всё установки сделаны, вы *наконец-то* можете вызвать **getaddrinfo()**!

Конечно, здесь веселье только начинается. *res* теперь указывает на связаный список struct addrinfo и вы можете пройтись по нему, посмотреть адреса, удовлетворяющие тому, что вы передали в *hints*.

Теперь стало возможным определить какие из адресов по той или иной причине не работают, и, как в Linux man странице, циклически пройти по нему вызывая **socket()** и **connect()** (или **bind()** если вы запустили сервер с флагом AI_PASSIVE) до самого конца.

Наконец, когда со связанным списком покончено, вам нужно вызвать **freeaddrinfo()** чтобы освободить память, иначе она затеряется (утечёт) и Кое-Кто расстроится.

### Возвращаемое значение

При успехе возвращает ноль или не-ноль при ошибке. В таком случае вы можете воспользоваться функцией **gai_strerror()** чтобы получить печатную версию кода возврата.

### Пример

```
// код для подключения клиента к серверу,
// а именно потокового сокета к www.example.com на порт 80 (http)
// IPv4 или IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;            // для задания IPv6 используйте AF_INET6
hints.ai_socktype = SOCK_STREAM;

if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) {
   fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
   exit(1);
}

// цикл по всем результатам и подключение к первому возможному
for(p = servinfo; p != NULL; p = p->ai_next) {
   if ((sockfd = socket(p->ai_family, p->ai_socktype,
         p->ai_protocol)) == -1) {
```

```
            perror("socket");
                continue;
        }
    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("connect");
            continue;
        }
    break;          // здесь мы подключились удачно
}
if (p == NULL) {
    // цикл закончился, а подключения нет
    fprintf(stderr, "failed to connect\n");
    exit(2);
}
freeaddrinfo(servinfo);       // со структурой закончили
```

```
// сервер, ожидающий подключений,
// а именно потоковый сокет порт 3490, IP этого хоста
// IPv4 либо IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;          // для выбора IPv6 используйте AF_INET6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;          // использовать мой IP адрес

if ((rv = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// цикл по всем результатам и подключение к первому возможному
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
            continue;
        }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("bind");
                continue;
        }
    break; // здесь мы подключились удачно
}

if (p == NULL) {
    // цикл закончился, а подключения нет
    fprintf(stderr, "failed to bind socket\n");
    exit(2);
}
freeaddrinfo(servinfo);               // со структурой закончили
```

**Смотри также**

**`gethostbyname(), getnameinfo()`**

## 9.6. gethostname()

Возвращает имя системы.

### Прототип

```
#include <sys/unistd.h>

int gethostname(char *name, size_t len);
```

### Описание

У вашей системы есть имя. У всех оно есть. Это даже несколько более Unix-овая вещь, чем все остальные сетевые причиндалы, о которых мы говорили, но она до сих пор используется.

Например, вы можете получить имя вашего хоста и затем вызвать **gethostbyname()**, чтобы получить ваш IP адрес.

Параметр *name* должен указывать на буфер, который будет содержать имя хоста, а *len* это размер этого буфера в байтах. **gethostname()** не выйдет за пределы этого буфера (он может возвратить ошибку или просто остановить запись) и вернёт строку с “\0” в конце если в буфере хватило места для строки.

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("My hostname: %s\n", hostname);
```

### Смотри также

**gethostbyname()**

## 9.7. gethostbyname(), gethostbyaddr()

Получают IP адрес хоста по имени или наоборот.

### Прототип

```
#include <sys/socket.h>
#include <netdb.h>

struct hostent *gethostbyname(const char *name); // УСТАРЕЛО!
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

### Описание

ЗАМЕТЬТЕ ПОЖАЛУЙСТА: эти две функции заменены на **getaddrinfo()** и **getnameinfo()**! В частности, **gethostbyname()** не очень хорошо работает с IPv6.

Эти функции выполняют преобразование имён хостов в IP адреса и обратно. Например, если у вас есть“www.example.com”, вы можете использовать **gethostbyname()** чтобы получить IP адрес и сохранить его в `struct in_addr`.

И наооборот, если у вас есть `struct in_addr` или `struct in6_addr`, можете воспользоваться **gethostbyaddr()** чтобы получить назад имя хоста. **gethostbyaddr()** *совместима* с IPv6, но пользоваться нужно **getnameinfo()** поновее и поярче.

(Если у вас есть строка с IP адресом в формате цифр-и-точек для которой вы хотите узнать имя хоста, то вам лучше воспользоваться **getaddrinfo()** с флагом `AI_CANONNAME`.)

**gethostbyname()** принимает строку вроде “www.yahoo.com” и возвращает `struct hostent`, которая содержит тонны информации, включая IP адрес. (Другая информация это официальное имя хоста, список псевдонимов, тип адреса и список адресов. Это структура общего назначения и вы увидели как очень просто использовать её в наших особых целях.)

**gethostbyaddr()** принимает `struct in_addr` или `struct in6_addr` и выдаёт соответствующее имя хоста (если оно есть), так что это функция, обратная **gethostbyname()**. Насчёт параметров, даже хотя *addr* меет тип `char*`, в действительности вам надо передавать указатель на `struct in_addr`. *len* должна быть `sizeof(struct in_addr)`, и *type* должен быть `AF_INET`.

Что это за `struct hostent`, которую нам возвращают? Она содержит множество полей, содержащих информацию о запрошенном хосте.

| | |
|---|---|
| *char *h_name* | Настоящее каноническое имя хоста |
| *char **h_aliases* | Список псевдонимов, к нему можно обращаться, как к массиву, последний элемент содержит `NULL` |
| *int h_addrtype* | Тип адреса результата, в нашем случае должен быть `AF_INET` |
| *int length* | Длина адресов в байтах (4 для IPv4) |
| *char **h_addr_list* | Список IP адресов этого хоста. Хоть он и `char**`, в действительности это массив переодетых `struct in_addr*`. Последний элемент массива равен `NULL`. |
| *h_addr* | Псевдоним *h_addr_list[0]*. Если вам нужен любой старый IP адрес этого хоста (да, у них может быть несколько) используйте это поле. |

### Возвращаемое значение

Возвращает указатель на получившуюся `struct hostent` или `NULL` при ошибке.

Вместо нормальной **`perror()`** и всего прилагающегося для выдачи сообщений об ошибке, эти функции пишут результат в переменную *`h_errno`*, которую можно распечатать функциями **`herror()`** или **`hstrerror()`**. Это подобно классической ***`errno`***, **`perror()`**, и **`strerror()`**.

### Пример

```
// ЭТО УСТАРЕВШИЙ МЕТОД ПОЛУЧЕНИЯ ИМЁН ХОСТА
// взамен используйте getaddrinfo()!

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
   int i;
   struct hostent *he;
   struct in_addr **addr_list;

   if (argc != 2) {
      fprintf(stderr,"usage: ghbn hostname\n");
      return 1;
     }

   if ((he = gethostbyname(argv[1])) == NULL) {  // получить информацию хоста
      herror("gethostbyname");
      return 2;
     }

   // распечатать информацию об этом хосте:
   printf("Official name is: %s\n", he->h_name);
   printf("    IP addresses: ");
   addr_list = (struct in_addr **)he->h_addr_list;
   for(i = 0; addr_list[i] != NULL; i++) {
      printf("%s ", inet_ntoa(*addr_list[i]));
   }
   printf(“\n");
     return 0;
}
```

```
// ЭТО ЗАМЕНЕНО
// взамен используйте getnameinfo()!

struct hostent *he;
struct in_addr ipv4addr;
struct in6_addr ipv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
printf("Host name: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6);
```

```
printf("Host name: %s\n", he->h_name);
```

### Смотри также

**getaddrinfo()**, **getnameinfo()**, **gethostname()**, ***errno***, **perror()**, **strerror()**, struct in_addr

## 9.8. getnameinfo()

Ищет информацию об имени хоста и сервиса по заданной `struct sockaddr`.

### Прототип

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
            char *host, size_t hostlen,
            char *serv, size_t servlen, int flags);
```

### Описание

Эта функция противоположна **`getaddrinfo()`**, что означает, она принимает уже заполненную `struct sockaddr` и по ней выполняет поиск имён хоста и сервиса. Она заменяет старые функции **`gethostbyaddr()`** и **`getservbyport()`**.

Вам надо передать указатель на `struct sockaddr` (которая в действительности приведённая вами `struct sockaddr_in` либо `struct sockaddr_in6`) в параметре *`sa`* и длину структуры в *`salen`*.

Имена хоста и сервиса в результате будут записаны в области, указанные параметрами *`host`* и *`serv`*. Конечно, максимальную длину этих буферов нужно указать в *`hostlen`* и *`servlen`*.

Последнее, вы можете передать несколько флагов, но есть парочка хороших. При установленном `NI_NOFQDN` *`host`* будет содержать только имя хоста, а не полное имя домена. `NI_NAMEREQD` вызовет отказ функции, если имя не будет найдено DNS поиском (если флаг не указан и имя не найдено, **`getnameinfo()`** заполнит *`host`* строковой версией IP адреса)

Как всегда, полный обзор ищите в ваших локальных **man** страницах.

### Возвращаемое значение

При успехе возвращает ноль и не-ноль при ошибке. Ненулевое значение можно передать **`gai_strerror()`**, чтобы получить читаемую строку. (Смотри **`getaddrinfo`**.)

### Пример

```
struct sockaddr_in6 sa; // если хотите может быть IPv4
char host[1024];
char service[20];

// положим, что sa наполнена доброй информацией о хосте и порте…

getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0);

printf("   host: %s\n", host);             // например, "www.example.com"
printf("service: %s\n", service);       // например, "http"
```

### Смотри также

**`getaddrinfo(), gethostbyaddr()`**

## 9.9. getpeername()

Возвращает адресную информацию об удалённой стороне соединения.

### Прототип

```
#include <sys/socket.h>

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

### Описание

Как только вы приняли (**accept()**) удалённое соединение или подключились (**connect()**) к серверу, у вас появляется некто, известный как “ровня” (пир, *peer*). Ваш пир это просто компьютер, к которому вы подключились через IP адрес и порт. Итак...

**getpeername()** просто возвращает struct sockaddr_in, заполненную информацией о подключённой машине.

Почему она зовётся “именем” (name)? Ну, поскольку существует много типов сокетов, а не только Интернет Сокеты, которые мы рассматриваем в данном руководстве, то “имя” прекрасный общий термин для всех случаев. Хотя в нашем случае имя пира это IP адрес и порт.

Пусть функция возвращает размер полученного адреса в *len*, но вам нужно предварительно записать в *len* длину *addr*.

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
// подразумеваем, что s это подключённый сокет

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
int port;

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);

// работаем с обоими: IPv4 и IPv6:
if (addr.ss_family == AF_INET) {
   struct sockaddr_in *s = (struct sockaddr_in *)&addr;
   port = ntohs(s->sin_port);
   inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof ipstr);
} else {                                // AF_INET6
   struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr;
   port = ntohs(s->sin6_port);
   inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof ipstr);
}

printf("Peer IP address: %s\n", ipstr);
printf("Peer port      : %d\n", port);
```

### Смотри также

**gethostname(), gethostbyname(), gethostbyaddr()**

## 9.10. *errno*

Содержит код ошибки последнего системного вызова.

### Прототип

```
#include <errno.h>

int errno;
```

### Описание

Эта переменная содержит информацию об ошибках для множества системных вызовов. Если при вызове, например, **`socket()`** или **`listen()`** происходит ошибка, то возвращается -1 и в ***`errno`*** устанавливается код, позволяющий точно определить, что случилось.

В заголовочном файле *`errno.h`* перечислены все символические имена ошибок, как `EADDRINUSE`, `EPIPE`, `ECONNREFUSED` и т.д. Ваши местные **man** страницы скажут какие коды могут быть возвращены как ошибки, и вы сможете использовать их во время исполнения, чтобы обрабатывать разные ошибки по разному.

Или, как правило, вы вызываете **`perror()`** или **`strerror()`** чтобы получить читаемую версию ошибки.

Ещё одно замечание для энтузиастов многопоточности, в большинстве систем ***`errno`*** определена потокобезопасным способом. (То есть, в действительности она не глобальная переменная, но ведёт себя так, как должна вести себя глобальная переменная в однопотоковой среде.)

### Возвращаемое значение

Значение переменной это код последней произошедшей ошибки, но может означать “успех” если последнее действие завершилось удачно.

### Пример

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
   perror("socket"); // или используйте strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
   // ошибочка вышла!!

   // если мы просто прерваны, просто перезапуск вызовом select():
   if (errno == EINTR) goto tryagain;  // AAAA!  goto!!!

   // иначе это ошибка посерьёзней:
   perror("select");
   exit(1);
}
```

### Смотри также

**`perror()`, `strerror()`**

## 9.11. fcntl()

Управляет дескрипторами сокетов.

### Прототип

```
#include <sys/unistd.h>
#include <sys/fcntl.h>

int fcntl(int s, int cmd, long arg);
```

### Описание

Обычно эта функция используется для блокировки файла и других файл-ориентированных вещей, но у неё есть пара относящихся к сокетам функций, которые вы можете посмотреть или время от времени использовать.

Параметр ***s*** это дескриптор сокета, с которым вы хотите работать, ***cmd*** должен быть установлен в `F_SETFL` и *arg* может быть одной из следующих команд. (Как я сказал, здесь больше **fcntl()** чем я себе позволяю, но я пытаюсь оставаться сокет-ориентированным.)

| | |
|---|---|
| `O_NONBLOCK` | Делает сокет неблокируемым. См. раздел по блокировке. |
| `O_ASYNC` | Переключает сокет на асинхронный ввод/вывод. Когда данные на сокете готовы к чтению возбуждается сигнал `SIGIO`. Это встречается редко и выходит за рамки данного документа. И я думаю, это доступно только на некоторых системах. |

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

Различные вызовы **fcntl()** возвращают различные значения, но я их здесь не рассматриваю, поскольку они не относятся к сокетам. Смотри **fcntl() man** страницы.

### Пример

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);      // установить не-блокировку
fcntl(s, F_SETFL, O_ASYNC);         // установит асинхронный ввод/вывод
```

### Смотри также

**Блокировка**, **send()**

## 9.12. htons(), htonl(), ntohs(), ntohl()

Преобразуют многобайтные целые типы из порядка байтов хоста в порядок байтов сети и наоборот.

### Прототип

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

### Описание

Чтобы сделать вас по-настоящему несчастными, различные компьютеры используют внутри различный порядок байт для хранения многобайтных целых (т.е. больше `char`.) В итоге, когда вы посылаете двухбайтное `short int` из Intel машины в Mac (я имею ввиду до того как они тоже стали Intel), то если один компьютер думает, что это число 1, то другой думает, что это число 256, и наоборот.

Способ обойти эту проблему это всем отложить разногласия и договориться как сделали Motorola и IBM, и Intel тоже сделала это загадочным способом, так что мы все перед посылкой преобразуем наш порядок байт в “big-endian”. Поскольку Intel это “*little-endian*” машины, то намного политкорректнее будет называть наш предпочитаемый порядок байт “Порядком Байтов Сети”. Вот эти функции и выполняют преобразование из вашего врождённого порядка в порядок байтов сети и наоборот.

(Это означает, что на Intel эти функции меняют все байты местами, а на PowerPC они ничего не делают поскольку байты уже в Порядке Байтов Сети. Но в своих программах вы должны использовать их всегда, потому что кто-нибудь может захотеть скомпилировать их на Intel машине и иметь правильно работающий вариант.)

Заметьте, что все использованные типы это 32-битные (4 байта, возможно `int`) и 16-битные (2 байта, очень похоже на `short`) числа. 64-битные машины возможно имеют **`htonll()`** для 64-битных `int`, но я таких не видел. Вам нужно просто написать свои собственные.

В любом случае, что этим функциям делать решаете вы, преобразовывать ли **из** порядка байт хоста (ваша машина) или из порядка байтов сети. Если “host”, то первая буква функции, которую вы собираетесь вызвать будет “h”. Иначе это “n” для “network”. В середине имени функции всегда стоит “to” (“**в**”) потому что вы всегда преобразуете **из** одного **в** другое, предпоследняя буква показывает во что вы преобразуете. Последняя буква это размер данных, “`s`” для `short`, или “`l`” для `long`. Таким образом:

```
htons()    host to network short
htonl()    host to network long
ntohs()    network to host short
ntohl()    network to host long
```

### Возвращаемое значение

Каждая функция возвращает преобразованное значение.

### Пример

```
uint32_t some_long = 10;
uint16_t some_short = 20;
uint32_t network_byte_order;
```

```
// преобразовать и послать
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);
some_short == ntohs(htons(some_short)); // это выражение истинно
```

## 9.13. inet_ntoa(), inet_aton(), inet_addr

Преобразуют строковые IP адреса в формате цифр-и-точек в `struct in_addr` и назад.

### Прототип

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
// ЭТО ВСЁ УСТАРЕЛО!  Взамен используйте inet_pton() или inet_ntop()!!
char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

### Описание

Эти функции непригодны потому что они не работают с IPv6! Вместо них используйте **`inet_ntop()`** или **`inet_pton()`**! Они включены здесь потому что их до сих пор можно встретить в природе.

Все эти функции преобразуют `struct in_addr` (вероятней всего часть вашей `struct sockaddr_in`) в строковый формат цифр-и-точек (например, “192.168.5.10”) и наоборот. Если вы получили IP адрес в командной строке или ещё как, то это самый простой способ получения `struct in_addr` для **`connect()`** или любых других нужд. Если вам нужно больше власти, испробуйте какую-нибудь DNS функцию, вроде **`gethostbyname()`** или устройте в вашей стране *coup d'État* (госпереворот).

Функция **`inet_ntoa()`** преобразует сетевой адрес из `struct in_addr` в строку формата цифр-и-точек. По историческим причинам буква “**n**” в “**ntoa**” означает сеть, и “**a**” означает ASCII (так что это “Network To ASCII” - у суффикса “**toa**” есть похожий друг в библотеке C, именуемый **`atoi()`**, который преобразует строку ASCII в целое.)

Функция **`inet_aton()`** обратная, она преобразует строку с цифрами-и-точками в `in_addr_t` (это тип поля `s_addr` в `struct in_addr`.)

Напоследок, функция **`inet_addr()`** более старая функция, делающая практически то же самое, что и **`inet_aton()`**. Теоретически она непригодна, но вы будете многократно её встречать и полиция вас не арестует, если вы её используете.

### Возвращаемое значение

**`inet_aton()`** возвращает не-ноль, если адрес действительный, и ноль, если нет.

**`inet_ntoa()`** возвращает строку цифр-и-точек в статическом буфере, который при каждом вызове функции перезаписывается.

**`inet_addr()`** возвращает адрес в формате `in_addr_t`, или -1 при ошибке. (Тот же результат вы получите, если попытаетесь преобразовать строку “255.255.255.255”, которая является действительным IP адресом. Вот почему **`inet_aton()`** лучше.)

### Пример

```
struct sockaddr_in antelope;
char *some_addr;
inet_aton("10.0.0.1", &antelope.sin_addr);       // запомнить IP в antelope

some_addr = inet_ntoa(antelope.sin_addr);      // возвращаем IP
printf("%s\n", some_addr);                     // печатает “10.0.0.1"

// этот вызов подобен вызову inet_aton() выше:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

### Смотри также

**`inet_ntop()`, `inet_pton()`, `gethostbyname()`, `gethostbyaddr()`**

## 9.14. inet_ntop(), inet_pton()

Преобразуют IP адреса в читаемую форму и назад.

### Прототип

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);
int inet_pton(int af, const char *src, void *dst);
```

### Описание

Эти функции предназначены для работы с читаемыми IP адресами и преобразования их в двоичное представление для последующего использования с различными функциями и системными вызовами. Буква “n” означает “network” (сеть), и “p” - “presentation” (представление). Или “text presentation” (текстовое представление). Но вы можете думать о нём, как “printable” (печатное). “ntop” это “network to printable” (сеть в печатное). Видите?

Иногда, глядя на IP адрес, вы не хотите смотреть на кучу цифр. Вы хотите видеть чудную печатную строку, как 192.0.2.180 или 2001:db8:8714:3a90::12. В этом случае, **inet_ntop()** к вашим услугам.

В параметре *af* **inet_ntop()** принимает семейство адресов (AF_INET или AF_INET6). Параметр *src* должен быть указателем на struct in_addr либо struct in6_addr, содержащую адрес, который вы хотите преобразовать в строку. Наконец, *dst* и *size* это указатели на строку назначения и длина этой строки.

Какой должна быть максимальная длина строки *dst*? Какова максимальная длина IPv4 и IPv6 адресов? К счастью есть пара макросов, которые вам помогут. Максимальные длины это INET_ADDRSTRLEN and INET6_ADDRSTRLEN.

В другой раз, у вам может быть строка, содержащая IP в читаемой форме, и вы хотите упаковать её в struct sockaddr_in или struct sockaddr_in6. В этом случае обратная функция **inet_pton()** это то, что вы ищете.

**inet_pton()** также принимает семейство адресов (AF_INET или AF_INET6) в параметре *af*. Параметр *src* это указатель на строку, содержащую IP адрес в печатной форме. Наконец, параметр *dst* указывает где нужно сохранить результат, это может быть struct in_addr или struct in6_addr.

Эти функции не выполняют DNS поиск - для этого нужна **getaddinfo()**.

### Возвращаемое значение

**inet_ntop()** возвращает параметр *dst* при успехе или NULL в случае отказа (***errno*** установлена).

**inet_pton()** возвращает 1 при успехе и -1 если была ошибка (***errno*** установлена), либо 0 если задан недействительный IP адрес.

### Пример

```
// пример inet_ntop() и inet_pton() с IPv4
struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];
// запомнить этот IP адрес в sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));
// извлекаем и печатаем
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // печатает "192.0.2.33"
```

```
// пример inet_ntop() и inet_pton() с IPv6
// (в основном то же самое кроме кучи разбросанных 6-ок)

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

// запомнить этот IP адрес в sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// извлекаем и печатаем
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // печатает “2001:db8:8714:3a90::12"
```

```
// Полезная вспомогательная функция:

// Преобразует адрес из struct sockaddr в строку, IPv4 и IPv6:

char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
   switch(sa->sa_family) {
      case AF_INET:
         inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr),
               s, maxlen);
                    break;

      case AF_INET6:
         inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr),
               s, maxlen);
         break;

      default:
         strncpy(s, "Unknown AF", maxlen);
         return NULL;
      }

      return s;
}
```

**Смотри также**

**`getaddrinfo()`**

## 9.15. listen()

Говорит сокету слушать входящие подключения.

### Прототип

```
#include <sys/socket.h>

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

### Описание

Вы можете взять ваш дескриптор сокета (созданный системным вызовом **socket()** и сказать ему слушать входящие соединения. Это и отличает серверы от клиентов, ребята.

Параметр *backlog* может означать пару различных вещей в зависимости от вашей системы, но приближённо он означает резерв - сколько ожидающих соединений вы можете иметь до того, как ядро начнёт отбрасывать новые. Так что, как только придёт новое соединение, вам нужно быстро принять (**accept()**) его, чтобы не переполнять резерв. Попробуйте установить 10 или около того и если при высокой нагрузке ваши клиенты начнут получать “Connection refused” (“Соединение отвергнуто”), установите побольше.

Перед вызовом **listen()** ваш сервер должен вызвать **bind()** чтобы подключиться к определённому номеру порта. Клиенты будут подключаться к этому порту на IP адресе сервера.

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
struct addrinfo hints, *res;
int sockfd;

// сначала заполняем адресные структуры с помощью getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // использовать либо IPv4 либо IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // заполнить мой IP для меня

getaddrinfo(NULL, "3490", &hints, &res);

// создать сокет:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// связать с портом, переданным getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);

listen(sockfd, 10); // теперь сервер слушает

// где-то дальше есть цикл accept()
```

### Смотри также

**accept(), bind(), socket()**

## 9.16. perror(), strerror()

Распечатывают ошибку как читаемую строку.

### Прототип

```
#include <stdio.h>
#include <string.h>   // для strerror()

void perror(const char *s);
char *strerror(int errnum);
```

### Описание

Очень много функций при ошибке возвращают -1 и записывают в ***errno*** некоторое число, и было бы замечательно если бы вы могли легко распечатать его в некоторой понятной форме.

И **perror()** благосклонно это делает. Если вы хотите добавить описание перед сообщением об ошибке, передайте указатель на него в параметре *s* (или оставьте *s* как NULL и ничего дополнительно не напечатается.)

Коротко, эта функция берёт значение ***errno***, вроде ECONNRESET, и любезно печатает его как “Connection reset by peer.”

Функция **strerror()** подобна **perror()**, за исключением того, что она возвращает указатель на строку с сообщением об ошибке для заданного параметром *errnum* значения (обычно вы передаёте переменную ***errno***.)

### Возвращаемое значение

**strerror()** возвращает указатель на строку с сообщением об ошибке.

### Пример

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // ошибка вышла
   // печатает "socket error: " + сообщение об ошибке:
   perror("socket error");
}

// подобно:
if (listen(s, 10) == -1) {
   // это печатает "an error: " + сообщение об ошибке из errno:
   printf("an error: %s\n", strerror(errno));
}
```

### Смотри также

***errno***

## 9.17. poll()

Одновременно проверяет события на множестве сокетов.

### Прототип

```
#include <sys/poll.h>

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

### Описание

Эта функция подобна **select()** в том, что обе отслеживают события в массивах дескрипторов, такие как: входные данные готовы к **recv()**, сокет готов к **send()**, данные сокета внеполосного (*out-of-band*) канала готовы к **recv()**, ошибки и т.д.

Основная идея в том, что вы передаёте массив из *nfds* структур `struct pollfd` в параметре *ufds*, вместе с таймаутом в миллисекундах (1000 миллисекунд это 1 секунда.) *timeout* может быть отрицательным если вы хотите ждать вечно. Если за время таймаута ни с одним дескриптором событий не происходит, **poll()** возвращает управление.

Каждый элемент массива структур `struct pollfd` представляет один дескриптор сокета и состоит из следующих полей:

```
struct pollfd {
    int fd;           // дескриптор сокета
    short events;     // биткарта интересующих событий
    short revents;    // биткарта произошедших событий при возврате из poll()
};
```

Перед вызовом **poll()** запишите дескриптор сокета в поле *fd* (если записать в *fd* отрицательное число, то эта `struct pollfd` игнорируется и в поле *revents* записывается ноль) и заполните поле *events* поразрядным ИЛИ следующих макросов:

| | |
|---|---|
| POLLIN | Известить меня о готовности данных к **recv()** на этом сокете. |
| POLLOUT | Известить меня, что я могу вызвать **send()** на этом сокете без блокировки. |
| POLLPRI | Известить меня о готовности *out-of-band* данных к **recv()**. |

По возвращению из **poll()** поле *revents* будет содержать поразрядное ИЛИ этих полей, показывая, какое событие произошло с этим дескриптором. Дополнительно могут быть установлены следующие поля:

| | |
|---|---|
| POLLERR | На этом сокете произошла ошибка. |
| POLLHUP | Удалённая сторона соединения зависла. |
| POLLNVAL | Что-то не то с дескриптором сокета *fd* - может он не инициализирован? |

### Возвращаемое значение

Возвращает количество элементов массива *ufds* для которых обнаружены события; оно может быть равно нулю, если истёк таймаут. Также при ошибке возвращает -1 (***errno*** будет установлена соответственно.)

### Пример

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
```

```
s2 = socket(PF_INET, SOCK_STREAM, 0);

// полагаем, что здесь оба подключены к серверу
//connect(s1, ...)...
//connect(s2, ...)...
// заполняем массив файловых дескрипторов.
//
// хотим узнать когда обычные или out-of-band
// готовы к приёму (recv())…

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI;           // обычные или out-of-band
ufds[1] = s2;
ufds[1].events = POLLIN;                     // только обычные данные

// ждём события на сокете, таймаут 3.5 секунды
rv = poll(ufds, 2, 3500);

if (rv == -1) {
   perror("poll");                           // в poll() произошла ошибка
} else if (rv == 0) {
   printf("Timeout occurred!  No data after 3.5 seconds.\n");
} else {
   // проверка события на s1:
   if (ufds[0].revents & POLLIN) {
      recv(s1, buf1, sizeof buf1, 0);        // принимаем обычные данные
   }
   if (ufds[0].revents & POLLPRI) {
      recv(s1, buf1, sizeof buf1, MSG_OOB);  // out-of-band данные
}

   // проверка события на s2:
   if (ufds[1].revents & POLLIN) {
      recv(s1, buf2, sizeof buf2, 0);
   }
}
```

### Смотри также

**`select()`**

## 9.18. recv(), recvfrom()

Принимают данные из сокета.

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

### Описание

Как только вы создали и подключили сокет, вы можете принимать из него данные вызовами **recv()** (для TCP SOCK_STREAM сокетов) и **recvfrom()** (для UDP SOCK_DGRAM сокетов).

Обе функции принимают дескриптор сокета *s*, указатель на буфер *buf*, длину буфера в байтах *len*, и набор флагов *flags*, определяющих работу функций.

Дополнительно, **recvfrom()** принимает struct sockaddr* *from*, указывающую откуда принимать данные и запишет в *fromlen* размер struct sockaddr. (Вы тоже можете инициализировать *fromlen* размером *from* или struct sockaddr.)

Так что же за дивные флаги вы можете передавать в эту функцию? Вот некоторые из них, но вам нужно осведомиться в свои **man** страницах, какие из них действительно поддерживаются вашей системой. Вы объединяете их поразрядным ИЛИ либо просто устанавливаете *flags* в 0 если хотите получить обычный скучный **recv()**.

| | |
|---|---|
| MSG_OOB | Принять *Out of Band* данные. То есть принять данные посланные вам **send()** с MSG_OOB флагом. Как у принимающей стороны у вас будет возбуждён сигнал SIGURG , говорящий о том, что появились срочные данные. В обработчике сигнала вы можете вызвать **recv()** с флагом MSG_OOB. |
| MSG_PEEK | Если вы хотите вызвать **recv()** “просто для вида”, можете указать этот флаг. Это покажет вам, что в буфере что-то есть для вызова **recv()** “по- настоящему” (т.е. *без* флага MSG_PEEK.) Это что-то вроде закрытого просмотра перед последующим вызовом **recv()**. |
| MSG_WAITALL | Говорит **recv()** не возвращаться пока не будут получены все указанные в параметре *len* данные. Однако, в экстремальных ситуациях, вроде прерывания вызова сигналом, возникновения ошибки или закрытия соединения удалённой стороной, ваши пожелания будут проигнорированы. Не сумасбродствуйте с этим. |

Когда вы вызываете **recv()** он блокируется до появления каких-либо данных. Если вы не хотите блокировки, установите сокет в неблокируемый режим или проверьте есть ли данные с помощью **select()** или **poll()** до вызова **recv()** или **recvfrom()**.

### Возвращаемое значение

Возвращает число действительно принятых данных (что может быть меньше затребованного в параметре *len*), или -1 при ошибке (***errno*** будет установлен соответственно.)

Если удалённая сторона закрыла соединение, то **recv()** вернёт 0. Это нормальный способ определения того, что удаленная сторона закрыла соединение. Нормально это хорошо, бунтарь!

### Пример

```
// потоковые сокеты и recv()

struct addrinfo hints, *res;
int sockfd;
char buf[512];
int byte_count;

// получить информацию хоста, создать сокет и подключиться
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // использовать либо IPv4 либо IPv6
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
connect(sockfd, res->ai_addr, res->ai_addrlen);

// Прекрасно!  Мы подключены и можем принимать данные!
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes of data in buf\n", byte_count);
```

```
// дейтаграммные сокеты и recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

// получить информацию хоста, создать сокет и подключиться к порту 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // использовать либо IPv4 либо IPv6
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, "4950", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);

// accept() не нужен, только recvfrom():

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes of data in buf\n", byte_count);
printf("from IP address %s\n",
    inet_ntop(addr.ss_family,
        addr.ss_family == AF_INET?
            ((struct sockadd_in *)&addr)->sin_addr:
            ((struct sockadd_in6 *)&addr)->sin6_addr,
        ipstr, sizeof ipstr);
```

### Смотри также

**send(), sendto(), select(), poll(), Блокировка**

## 9.19. select()

Проверяет готовы ли дескрипторы сокетов к чтению/записи.

### Прототип

```
#include <sys/select.h>

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
                  struct timeval *timeout);

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

### Описание

Функция **select()** предоставляет способ одновременной проверки множества сокетов на предмет ожидания **recv()**, готовности к передаче данных через **send()** без блокирования или возникновения исключения.

Вы заселяете ваш массив дескрипторов сокетов используя макросы вроде **FD_SET()**. Получив массив, вы передаёте его функции как один из следующих параметров: *readfds* если хотите знать, готов ли какой-нибудь сокет из массива к **recv()**, *writefds* если какой-либо сокет готов к **send()**, и/или *exceptfds* если вам нужно узнать произошло ли на каком-нибудь сокете исключение. Любой их этих параметров может быть NULL если этот тип событий вам неинтересен. После возврата из **select()** значения в массиве будут изменены, чтобы показать, какие сокеты готовы к чтению/записи и какие имеют исключения.

Первый параметр, *n* это наивысший номер дескриптора сокета (они просто int, помните?) плюс один.

Напоследок, struct timeval **timeout* в конце позволяет сказать **select()** как долго проверять эти массивы. Она вернёт управление при истечении таймаута или при возникновении события, смотря что раньше. В struct timeval есть два поля: *tv_sec* это количество секунд, к которому добавляется *tv_usec*, количество микросекунд

(1 000 000 микросекунд в секунде.)

Вспомогательные макросы делают следующее:

| | |
|---|---|
| **FD_SET(int *fd*, fd_set **set*);** | Добавляет *fd* в *set*. |
| **FD_CLR(int *fd*, fd_set **set*);** | Удаляет *fd* из *set*. |
| **FD_ISSET(int *fd*, fd_set **set*);** | Возвращает true если *fd* есть в *set*. |
| **FD_ZERO(fd_set **set*);** | Очищает *set*. |

### Возвращаемое значение

Возвращает количество дескрипторов с событиями в массиве, 0 если таймаут истёк и -1 при ошибке (***errno*** устанавливается соответственно.) Кроме того, массивы изменяются чтобы показать готовые сокеты.

### Пример

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// полагаем, что здесь оба подключены к серверу
```

```
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// заранее очищаем массив
FD_ZERO(&readfds);

// добавляем наши дескрипторы в массив
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// поскольку s2 создан вторым, он “больше” и его используем
// в параметре n в select()
n = s2 + 1;

// ждём появления данных на каком-либо сокете (таймаут 10.5 секунд)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // в select() произошла ошибка
      } else if (rv == 0) {
    printf("Timeout occurred!  No data after 10.5 seconds.\n");
      } else {
    // на одном или обоих дескрипторах есть данные
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s1, buf2, sizeof buf2, 0);
      }
}
```

### Смотри также

**`poll()`**

## 9.20. setsockopt(), getsockopt()

Устанавливает для сокета различные опции.

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval,
                              socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
                              socklen_t optlen);
```

### Описание

Сокеты это весьма конфигурируемые звери. В действительности они настолько конфигурируемые, что я даже не собираюсь здесь обо всём говорить. Возможно это зависит от системы. Но я поговорю об основах.

Заметно, что эти функции получают и устанавливают определённые опции сокета. На Linux, вся информация по сокетам находится в **man** странице, секция 7. (Напечатайте: “**man 7 socket**” , чтобы получить все эти прелести.)

Насчёт параметров, *s* это сокет, о котором вы говорите, *level* должен быть установлен в SOL_SOCKET. Затем, в *optname* установите интересующее вас имя. Опять же, насчёт всех опций посмотрите вашу **man** страницу, а здесь только несколько самых забавных:

| | |
|---|---|
| SO_BINDTODEVICE | Связывает сокет с символическим именем устройства вроде eth0 вместо использования **bind()** для привязки к IP адресу. Под Unix напечатайте команду **ifconfig** чтобы увидеть имена устройств. |
| SO_REUSEADDR | Позволяет другим сокетам связываться (**bind()**) с этим портом, несмотря на то, что уже существует активный сокет, слушающий этот порт. Это позволяет обойти сообщения “Address already in use”, когда вы пытаетесь перезапустить ваш сервер после обрушения. |
| SO_BROADCAST | Позволяет UDP дейтаграммным (SOCK_DGRAM) сокетам посылать пакеты по широковещательным адресам и принимать и с них. На TCP потоковых сокетах ничего - *НИЧЕГО!!* - не делает! Ха-ха-ха! |

Насчёт параметра *optval*, обычно это указатель на int, показывающую значение запроса. Для булевых переменных, ноль это ложь и не-ноль это истинно. И это абсолютный факт, несмотря на отличия вашей системы. Если передавать нечего, то *optval* может быть NULL.

Последний параметр, *optlen*, заполняется **getsockopt()** и вам нужно указать его для **setsockopt()**, возможно он будет sizeof(int).

**Предупреждение**: на некоторых системах (особенно Sun и Windows), эта опция может быть char вместо int, и содержать, например, символьное значение '1' вместо целого 1. Опять же, подробности смотрите в ваших **man** страницах командами “**man setsockopt**” и “**man 7 socket**”!

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
int optval;
int optlen;
char *optval2;

// установить SO_REUSEADDR на сокете в ИСТИННО (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// связать сокет с именем устройства (может не работать на некоторых системах):
optval2 = "eth1"; // 4 байта длины, итого 4,ниже:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);

// посмотреть установлен ли флаг SO_BROADCAST:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
    print("SO_BROADCAST enabled on s3!\n");
}
```

### Смотри также

**fcntl()**

## 9.21. send(), sendto()

Посылают данные через сокет.

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len,
           int flags, const struct sockaddr *to,
           socklen_t tolen);
```

### Описание

Эти функции посылают данные в сокет. В общем случае **send()** используется для TCP SOCK_STREAM подключённых сокетов, а **sendto()** для UDP SOCK_DGRAM неподключённых дейтаграммных сокетов. Каждый раз посылая пакет по неподключённому сокету вы должны указывать место назначения, поэтому последние параметры **sendto()** задают куда пакет направляется.

В обоих, **send()** и **sendto()**, параметр *s* это сокет, *buf* указатель на данные, которые вы хотите послать, *len* число посылаемых байт и *flags* позволяет определить дополнительную информацию как посылать данные. Установите *flags* в ноль, если хотите иметь "нормальные" данные. Вот несколько из обычно используемых флагов, но за подробностями загляните в ваши местные **man** страницы по **send()**:

| | |
|---|---|
| MSG_OOB | Посылает "*out of band*" данные. TCP поддерживает этот способ сказать принимающей стороне, что у этих данных приоритет выше, чем у нормальных. Приёмник получит сигнал SIGURG и примет эти данные без предварительной выборки нормальных данных из очереди. |
| MSG_DONTROUTE | Не посылать эти данные через маршрутизатор, они местные. |
| MSG_DONTWAIT | Если **send()** должна блокироваться из-за загруженности внешнего трафика, она вернёт EAGAIN. Это вроде "Разрешить не-блокирование для этой посылки". Детали описаны в разделе по блокированию. |
| MSG_NOSIGNAL | **send()** на удалённый хост, который больше не принимает (**recv()**) данные, обычно возбуждает сигнал SIGPIPE. Этот флаг предотвращает возбуждение такого сигнала. |

### Возвращаемое значение

Возвращает число действительно посланных байт или -1 при ошибке (***errno*** устанавливается соответственно.) Заметьте, что это число может быть меньше затребованного! Вспомогательная функция в разделе по **send()** поможет обойти это.

Также, если сокет был закрыт на противной стороне, процесс, вызвавший **send()**, получит сигнал SIGPIPE. (Если только **send()** не был вызван с флагом MSG_NOSIGNAL.)

### Пример

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;
```

```
// сначала с потоковым сокетом TCP:

// полагаем, что сокеты созданы и подключены
//stream_socket = socket(...
//connect(stream_socket, …
// преобразовать в порядок байтов сети
temp = htonl(spatula_count);

// послать данные нормально:
send(stream_socket, &temp, sizeof temp, 0);

// послать секретное out of band сообщение:
send(stream_socket, secret_message, strlen(secret_message)+1, MSG_OOB);

// теперь с дейтаграммным сокетом UDP:
//getaddrinfo(...
//dest = ...  // полагаем, что "dest" содержит адрес назначения
//dgram_socket = socket(…

// послать секретное послание нормально:
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
      (struct sockaddr*)&dest, sizeof dest);
```

### Смотри также

**`recv()`, `recvfrom()`**

## 9.22. shutdown()

Останавливает обмен по сокету.

### Прототип

```
#include <sys/socket.h>

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

### Описание

Вот так! Получилось! На этом сокете **send()** больше не разрешены, но хочу всё ещё принимать (**recv()**) с него данные! Или наоборот! Как это сделать?

Когда вы закрываете (**close()**) дескриптор сокета закрываются обе стороны, для чтения и для записи, а дескриптор освобождается. Если вы хотите закрыть только одну или другую сторону, вам нужно вызвать **shutdown()**.

Насчёт параметров, понятно, что *s* это сокет с которым вы работаете, а что с ним делать определяет параметр *how*. Это может быть SHUT_RD для предотвращения дальнейших **recv()**, SHUT_WR для запрещения дальнейших **send()**, или SHUT_RDWR для обоих.

Заметьте, что **shutdown()** не освобождает дескриптор сокета и в итоге вам нужно вызвать **close()** чтобы закрыть его полностью.

Этот системный вызов используется редко.

### Возвращаемое значение

Возвращает 0 при успехе или -1 в случае ошибки (***errno*** устанавливается соответственно).

### Пример

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...посылаем и обрабатываем здесь...

// и когда всё сделано запрещаем дальнейшие посылки:
shutdown(s, SHUT_WR);
```

### Смотри также

**close()**

## 9.23. socket()

Создаёт дескриптор сокета

### Прототип

```
#include <sys/types.h>
#include <sys/socket.h>

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

### Описание

Возвращает дескриптор сокета, с которым вы можете творить всё ему присущее. Обычно это первый шаг в этом ужасном процессе написания программ с сокетами и результат можно использовать в последующих вызовах **listen()**, **bind()**, **accept()** или множества других функций.

Обычно вы получаете значения этих параметров из вызова **getaddrinfo()**, как в примере ниже, но можете и заполнять их вручную, если вам так уж хочется.

| | |
|---|---|
| *domain* | определяет тип нужного вам сокета. Поверьте, их может быть множество, но поскольку это руководство по сокетам, он будет PF_INET для IPv4 и PF_INET6 для IPv6. |
| *type* | Хотя параметр *type* может принимать множество значений, вы установите его в SOCK_STREAM для надёжных TCP сокетов (**send()**, **recv()**) либо SOCK_DGRAM для ненадёжных быстрых UDP сокетов (**sendto()**, **recvfrom()**.)<br>(Другой интересный тип сокетов это SOCK_RAW, который позволяет строить пакеты вручную. Это очень круто.) |
| *protocol* | Последнее, параметр *protocol* указывает какой протокол использовать для этого типа сокетов. Как я уже сказал, например, SOCK_STREAM использует TCP. К счастью для вас, используя SOCK_STREAM или SOCK_DGRAM, вы можете просто установить *protocol* в 0, и он автоматически использует правильный протокол. Иначе вы можете использовать **getprotobyname()** для выбора номера нужного протокола. |

### Возвращаемое значение

Дескриптор нового сокета для последующих вызовов или -1 при ошибке (и ***errno*** будет установлен соответственно.)

### Пример

```
struct addrinfo hints, *res;
int sockfd;

// сначала заполняем адресные структуры с помощью getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;            // AF_INET, AF_INET6, или AF_UNSPEC
hints.ai_socktype = SOCK_STREAM;        // SOCK_STREAM или SOCK_DGRAM
getaddrinfo("www.example.com", "3490", &hints, &res);

// создаём сокет с помощью информации, которую наскребла getaddrinfo():
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

### Смотри также

**accept(), bind(), getaddrinfo(), listen()**

# 9.24. struct sockaddr сотоварищи

Структуры для обработки интернет адресов.

### Прототип

```
include <netinet/in.h>

// Все указатели на адресные структуры сокетов часто приводятся
// к этому типу перед их использованием в различных функциях и вызовах:

struct sockaddr {
    unsigned short sa_family;       // семейство адресов, AF_xxx
    char           sa_data[14];     // 14 байт адреса протокола
};

// IPv4 AF_INET сокеты:

struct sockaddr_in {
  short            sin_family;      // например, AF_INET, AF_INET6
  unsigned short   sin_port;        // например, htons(3490)
  struct in_addr   sin_addr;        // смотри struct in_addr, ниже
  char             sin_zero[8]      // обнулите, если хочется
};

struct in_addr {
    unsigned long s_addr;           // заполнить с помощью inet_pton()
};


// IPv6 AF_INET6 сокеты:

struct sockaddr_in6 {
    u_int16_t       sin6_family;    // семейство адресов, AF_INET6
    u_int16_t       sin6_port;      // номер порта, Порядок Байтов Сети
    u_int32_t       sin6_flowinfo;  // IPv6 flow information
    struct in6_addr sin6_addr;      // IPv6 адрес
    u_int32_t       sin6_scope_id;  // Scope ID
};

struct in6_addr {
    unsigned char s6_addr[16];      // заполнить с помощью inet_pton()
};

// Общая структура хранения адреса сокета достаточно велика для хранения
// данных struct sockaddr_in или struct sockaddr_in6:

struct sockaddr_storage {
    sa_family_t ss_family;          // семейство адресов

  // всё это расширение зависит от реализации, проигнорируйте:

  char    __ss_pad1[_SS_PAD1SIZE];
  int64_t __ss_align;
  char    __ss_pad2[_SS_PAD2SIZE];
};
```

### Описание

Это базовые структуры для всех системных вызовов и функций, работающих с интернет адресами. Для заполнения этих структур вы часто будете использовать **getaddinfo()** и затем по мере надобности читать их.

В памяти struct sockaddr_in и struct sockaddr_in6 начинаются с одинаковой struct sockaddr, и вы можете спокойно приводить один тип к другому без какого-либо ущерба за исключением возможного конца света.

Пошутим над концом света… если вселенная прекратит своё существование в то время когда вы приводите struct sockaddr_in* к struct sockaddr*, обещаю, это будет чистейшее совпадение, и вам заботиться об этом не нужно.

Так что, помня об этом, знайте, что если функция говорит, что принимает struct sockaddr* вы спокойно и безопасно можете привести к этому типу ваши struct sockaddr_in*, struct sockaddr_in6* или struct sockadd_storage*.

Структура struct sockaddr_in используется с IPv4 адресами (вроде “192.0.2.10”). Она содержит семейство адресов (AF_INET), порт в *sin_port* и IPv4 адрес в *sin_addr*.

Кроме того в struct sockaddr_in есть поле *sin_zero* , которое по утверждению некоторых людей должно содержать нули. Другие ничего об этом не утверждают (документация Linux вообще об этом ничего не упоминает) и установка его в ноль не кажется действительно необходимой. Так что, если вы согласны, обнулите её функцией **memset()**.

Да и struct in_addr это таинственный зверь на разных системах. Иногда это сумасшедший union со всеми видами #define-ов и прочей чушью. Но вам нужно использовать только поле *s_addr*, поскольку многие системы реализуют только его.

struct sockadd_in6 очень похожа на struct in6_addr , но используется для IPv6.

struct sockaddr_storage передаётся в **accept()** или **recvfrom()** когда вы пытаетесь написать код, не зависящий от версии IP, и вы не знаете каким будет новый адрес - IPv4 или IPv6. Структура struct sockaddr_storage достаточно велика, чтобы содержать оба типа, в отличие от оригинальной маленькой struct sockaddr.

### Пример

```
// IPv4:
struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);
s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);
```

```
// IPv6:
struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);
s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

### Смотри также

**accept(), bind(), connect(), inet_aton(), inet_ntoa()**

# 10. Дополнительные ссылки

Вы добрались так далеко и теперь возопили о большем! Где ещё можно узнать обо всём этом великолепии?

## 10.1.Книги

Для старой школы актуальны осязаемые дешёвые бумажные книги. Вот некоторые из превосходных книг. Обычно я связан с очень популярными интернет книготорговцами, но их нынешняя система обслуживания потребителей несовместима с печатными документами. Так что, откатов у меня больше нет. И если вы сочувствуете моим устремлениям, пожертвуйте, пожалуйста, *paypal* на beej@beej.us. :-)

*Unix Network Programming, volumes 1-2* by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 0131411551[44], 0130810819[45].

*Internetworking with TCP/IP, volumes I-III* by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0131876716[46], 0130319961[47], 0130320714[48].

*TCP/IP Illustrated, volumes 1-3* by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3 (and a 3-volume set): 0201633469[49], 020163354X[50], 0201634953[51], (0201776316[52]).

*TCP/IP Network Administration* by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 0596002971[53].

*Advanced Programming in the UNIX Environment* by W. Richard Stevens. Published by Addison Wesley. ISBN 0201433079[54].

## 10.2.Web ссылки

В Сети:

BSD Sockets: A Quick And Dirty Primer[55] (Программирование Unix систем тоже!)
The Unix Socket FAQ[56]
Intro to TCP/IP[57]
TCP/IP FAQ[58]
The Winsock FAQ[59]

---

44 http://beej.us/guide/url/unixnet1
45 http://beej.us/guide/url/unixnet2
46 http://beej.us/guide/url/intertcp1
47 http://beej.us/guide/url/intertcp2
48 http://beej.us/guide/url/intertcp3
49 http://beej.us/guide/url/tcpi1
50 http://beej.us/guide/url/tcpi2
51 http://beej.us/guide/url/tcpi3
52 http://beej.us/guide/url/tcpi123
53 http://beej.us/guide/url/tcpna
54 http://beej.us/guide/url/advunix
55 http://www.frostbytes.com/~jimf/papers/sockets/sockets.html
56 http://www.developerweb.net/forum/forumdisplay.php?f=70
57 http://pclt.cis.yale.edu/pclt/COMM/TCPIP.HTM
58 http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/
59 http://tangentsoft.net/wskfaq/

И есть несколько серьёзных страниц в Википедии:

Berkeley Sockets[60]
Internet Protocol *(IP)*[61]
Transmission Control Protocol (TCP)[62]
User Datagram Protocol *(UDP)*[63]
Client-Server[64]
Serialization[65] (упаковка и распаковка данных)

## 10.3.RFC

Все RFC[66] - это настоящий отвал пустой породы! Эти документы описывают назначенные номера, программные API и протоколы, используемые в Интернете. Я включил немного ссылок на них для вашего удовольствия. Так что хватайте бадью попкорна и надевайте вашу думательную шапку:

RFC 1[67] - Первый RFC; он даёт понимание того, каким “Интернет” был, когда он родился и взгляд на то, как он возрос. (Понятно, что этот RFC полностью устарел!)
RFC 768[68] - The User Datagram Protocol (UDP)
RFC 791[69] - The Internet Protocol (IP)
RFC 793[70] - The Transmission Control Protocol (TCP)
RFC 854[71] - The Telnet Protocol
RFC 959[72] - File Transfer Protocol (FTP)
RFC 1350[73] - The Trivial File Transfer Protocol (TFTP)
RFC 1459[74] - Internet Relay Chat Protocol (IRC)
RFC 1918[75] - Address Allocation for Private Internets
RFC 2131[76] - Dynamic Host Configuration Protocol (DHCP)
RFC 2616[77] - Hypertext Transfer Protocol (HTTP)

[60] http://en.wikipedia.org/wiki/Berkeley_sockets
[61] http://en.wikipedia.org/wiki/Internet_Protocol
[62] http://en.wikipedia.org/wiki/Transmission_Control_Protocol
[63] http://en.wikipedia.org/wiki/User_Datagram_Protocol
[64] http://en.wikipedia.org/wiki/Client-server
[65] http://en.wikipedia.org/wiki/Serialization
[66] http://www.rfc-editor.org/
[67] http://tools.ietf.org/html/rfc1
[68] http://tools.ietf.org/html/rfc768
[69] http://tools.ietf.org/html/rfc791
[70] http://tools.ietf.org/html/rfc793
[71] http://tools.ietf.org/html/rfc854
[72] http://tools.ietf.org/html/rfc959
[73] http://tools.ietf.org/html/rfc1350
[74] http://tools.ietf.org/html/rfc1459
[75] http://tools.ietf.org/html/rfc1918
[76] http://tools.ietf.org/html/rfc2131
[77] http://tools.ietf.org/html/rfc2616

RFC 2821[78] - Simple Mail Transfer Protocol (SMTP)
RFC 3330[79] - Special-Use IPv4 Addresses
RFC 3493[80] - Basic Socket Interface Extensions for IPv6
RFC 3542[81] - Advanced Sockets Application Program Interface (API) for IPv6
RFC 3849[82] - IPv6 Address Prefix Reserved for Documentation
RFC 3920[83] - Extensible Messaging and Presence Protocol (XMPP)
RFC 3977[84]—Network News Transfer Protocol (NNTP)
RFC 4193[85]—Unique Local IPv6 Unicast Addresses
RFC 4506[86]—External Data Representation Standard (XDR)

В IETF есть прекрасный онлайн инструмент для поиска и просмотра RFC[87].

[78] http://tools.ietf.org/html/rfc2821
[79] http://tools.ietf.org/html/rfc3330
[80] http://tools.ietf.org/html/rfc3493
[81] http://tools.ietf.org/html/rfc3542
[82] http://tools.ietf.org/html/rfc3849
[83] http://tools.ietf.org/html/rfc3920
[84] http://tools.ietf.org/html/rfc3977
[85] http://tools.ietf.org/html/rfc4193
[86] http://tools.ietf.org/html/rfc4506
[87] http://tools.ietf.org/rfc/

# Предметный указатель


**accept()** 25, 66

**bind()** 22, 68

**close()** 28, 72
**closesocket()** 2
**CreateProcess()** 3
**CreateThread()** 3

F_SETFL 83
**fcntl()** 83
**FD_CLR()** 39
**FD_ISSET()** 39
**FD_SET()** 39
**FD_ZERO()** 39
**fork()** 3

**getaddrinfo()** 18, 73
**gethostbyaddr()** 77
**gethostbyname()** 77
**gethostname()** 29, 76
**getnameinfo()** 80
**getpeername()** 28, 81
**getprotobyname()** 102
**getsockopt()** 97

**herror()** 78
**hstrerror()** 78
**htonl()** 84
**htons()** 84

INADDR_ANY 17
INADDR_BROADCAST 17
**inet_addr()** 15
**inet_aton()** 15
**inet_ntoa()** 16
**inet_ntop()** 87
**inet_pton()** 87

**listen()** 24, 89

MSG_DONTROUTE 99
MSG_DONTWAIT 99
MSG_NOSIGNAL 99
MSG_OOB 99
MSG_PEEK 93
MSG_WAITALL 93
**ntohl()** 84
**ntohs()** 84

**perror()** 90
PF_INET 17
**poll()** 91

**read()** 5
**recv()** 26, 93
**recvfrom()** 27, 93

SA_RESTART 60
**select()** 38, 95
**send()** 26
**sendall()** 44
**sendto()** 27
**setsockopt()** 97
**shutdown()** 28, 101
**sigaction()** 32
SIGIO 83
SIGPIPE 72
SIGURG 93
SO_BINDTODEVICE 97
SO_BROADCAST 97
SO_RCVTIMEO 64
SO_REUSEADDR 97
SO_SNDTIMEO 64
SOCK_DGRAM 5
SOCK_RAW 59
SOCK_STREAM 5
**socket()** 21, 102
SOL_SOCKET 97
**strerror()** 90
struct addrinfo 13
struct hostent 77
struct in_addr 103
struct pollfd 91
struct sockaddr 103
struct sockaddr_in 103
struct timeval 39

**write()** 5
**WSACleanup()** 2
**WSAStartup()** 2