ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 22 — Сокеты AF_INET

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2023

2023.06.07

Оглавление

AF_INET — реализация протокола IPv4 в Linux

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> // включает предыдущий

tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
raw_socket = socket(AF_INET, SOCK_RAW, protocol);
```

В Linux реализован протокол Интернета (Internet Protocol, IP) версии 4, описанный в RFC 791 и RFC 1122. Реализован второй уровень групповых многоадресных сообщений, которая соответствует RFC 1112. Кроме того, в нём имеется маршрутизатор IP с фильтрацией пакетов.

Программный интерфейс совместим с интерфейсом сокетов BSD (**socket()**, **bind()**, **listen()**, **connect()**, **accept()**, **gethostbyname()**, **gethostbyaddr()**).

Сокет IP создаётся с помощью системного вызова **socket()**:

```
socket(AF_INET, socket_type, protocol);
```

socket_type — возможные типы сокета:
SOCK_STREAM — для открытия сокета **tcp(7)**;
SOCK_DGRAM — для открытия сокета **udp(7)**;
SOCK_RAW — для открытия сокета **raw(7)** с прямым доступом к протоколу IP.

protocol — задаётся протокол IP, который указывается в IP-заголовке принимаемых или отправляемых пакетов. Допустимые значения для параметра protocol:

0 и **IPPROTO_TCP** — для сокетов TCP,

0 и **IPPROTO_UDP** — для сокетов UDP.

Для **SOCK_RAW** можно указать любой из IP-протоколов, описанных ранее[1] в RFC 1700 и зарегистрированных в IANA (Internet Assigned Numbers Authority).

1) Если процесс хочет принимать новые входящие пакеты или соединения, то он должен связать сокет с адресом локального интерфейса с помощью **bind(2)**.

Каждый IP-сокет может быть связан только с одной задаваемой локальной парой (адрес, порт).

2) Если в вызове **bind(2)** в качестве адреса указать **INADDR_ANY**[2], то сокет будет связан со всеми локальными интерфейсами.

3) При вызове **listen(2)** для непривязанного сокета происходит автоматическая привязка к произвольно выбранному свободному порту с локальным адресом **INADDR_ANY**.

4) При вызове **connect(2)** для непривязанного сокета происходит автоматическая привязка к произвольно выбранному свободному порту или используемому общему порту с локальным адресом **INADDR_ANY**.

5) После закрытия привязанного локального TCP-сокета его адрес будет недоступен в течение некоторого времени, чего можно избежать, установив флаг **SO_REUSEADDR** (**setsockopt(2)**).

Однако, следует проявлять осторожность при использовании этого флага, поскольку это делает TCP менее надежным.

---

1) С 1994 года документ RFC 1700 заменен онлайновой базой данных, доступной через веб-страницу (в настоящее время www.iana.org). RFC 1700 устарел, его значения неполны и в некоторых случаях могут быть неправильными.

2) **INADDR_ANY** (0.0.0.0) означает любой адрес для связывания.

Класификация IP-сетей

Для адресации IP-сетей используется 4 байта.

Существуют два типа адресации IP-сетей — классовая и бесклассовая.

Классовая адресация

При классовой адресации сетевой адрес IPv4 разделяется на узловой и сетевой компоненты по байтовой границе следующим образом:

Класс A

На данный тип адреса указывает 0 на месте старшего бита (сетевой порядок байтов) адреса.

Адрес сети (сетевой адрес) содержится в самом старшем байте, а адреса узлов занимают оставшиеся три байта.

Класс B

На данный тип адреса указывает двоичное значение 10 на месте двух самых старших битов (сетевой порядок байтов) адреса.

Адрес сети содержится в двух старших байтах, а адреса узлов занимают оставшиеся два байта.

Класс C

На данный тип адреса указывает двоичное значение 110 на месте самых трех старших битов (сетевой порядок байтов) адреса.

Адрес сети содержится в первых трёх старших байтах, а адреса узлов занимают оставшийся байт.

Классовая адресация в настоящее время устарела и была заменена на бесклассовую адресацию (CIDR), при которой компоненты сети и узла в адресе могут занимать произвольное число битов (а не байтов).

Бесклассовая адресация

Бесклассовая адресация (Classless Inter-Domain Routing, CIDR - бесклассовая междоменная маршрутизация) — метод IP-адресации, позволяющий гибко управлять пространством IP-адресов, не используя жёсткие рамки классовой адресации. Использование этого метода позволяет экономно использовать ограниченный ресурс IP-адресов, поскольку возможно применение различных масок подсетей к различным подсетям.

IP-адрес трактуется, как массив бит. Маска подсети задаёт какие биты в IP-адресе являются адресом сети.

Блок адресов задаётся указанием начального адреса и маски подсети.

Бесклассовая адресация основывается на переменной длине маски подсети (англ. variable length subnet mask, VLSM), в то время, как в классовой адресации длина маски подсети имела всего лишь 3 фиксированных значения.

Пример подсети 192.0.2.16/28 с применением бесклассовой адресации:

Октеты IP-адреса	192								0								2								16							
Биты IP-адреса	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**	0	0	0	0
Биты маски подсети	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
Октеты маски подсети	255								255								255								240							

В маске подсети 28 бит слева — единицы. В таком случае говорят о длине префикса подсети в 28 бит и указывают через косую черту (символ /) после базового адреса.

192.0.2.16 — адрес подсети

192.0.2.17…30 — адреса хостов (14 штук)

192.0.2.31 — широковещательный адрес

192.0.2.32 — адрес следующей подсети

Формат адреса IP-сокета

Адрес IP-сокета определяется как комбинация IP-адреса интерфейса и номера порта.

В самом протоколе IP нет номеров портов, они реализуются протоколами более высокого уровня, например **udp(7)** и **tcp(7)**.

У неструктурированных (raw) сокетов в **sin_port** указывается номер протокола IP.

```
struct sockaddr_in {
    sa_family_t    sin_family; // семейство адресов: AF_INET
    in_port_t      sin_port;   // порт сокета в сетевом порядке байт
    struct in_addr sin_addr;   // Интернет-адрес
};

// Интернет-адрес
struct in_addr {
    uint32_t s_addr;           // адрес в сетевом порядке байт
};
```

Значение `sin_family` всегда устанавливается в AF_INET. Это обязательно.

В **sin_port** указывается номер порта в сетевом порядке байт.

Порты, номера которых меньше 1024, называются привилегированными портами (зарезервированными портами). Только привилегированные процессы могут быть связаны с этими сокетами с помощью **bind(2)**.

Исходный протокол IPv4 как таковой не имеет понятия порта — они реализуются только более высокими протоколами, такими как tcp(7) и udp(7).

```
struct sockaddr_in {
    sa_family_t    sin_family; // семейство адресов: AF_INET
    in_port_t      sin_port;   // порт сокета в сетевом порядке байт
    struct in_addr sin_addr;   // Интернет-адрес
};
```

В **sin_addr** указывается IP-адрес узла. В поле **s_addr** структуры **struct in_addr** содержится адрес интерфейса узла в сетевом порядке байт.

Значение в **in_addr** может быть одним из **INADDR_*** (например, **INADDR_LOOPBACK**) установленное **htonl(3)**, или с помощью библиотечных функций **inet_aton(3)**, **inet_addr(3)**, **inet_makeaddr(3)** или напрямую с помощью преобразователя имён (**gethostbyname(3)**).

Адреса

Адрес IPv4 (чаще его называют просто IP-адрес) представляет собой 32-битное число, обозначающее адрес сетевого интерфейса.

Для записи этого числа чаще всего используется точечная нотация, в которой байты числа записываются отдельно друг от друга и отделяются символом точки, например `80.94.171.236`.

Но и такой способ записи адресов зачастую оказывается неудобным, поэтому, как правило, используются описательные имена, например `lsi.bas-net.by`.

Порты

Каждый IP-адрес может быть связан с набором служб (сервисов), каждая из которых имеет свой собственный номер порта. Значительная часть портов привязана к определенным службам.

Например, HTTP-серверы работают с портом 80, FTP-серверы — с портом 21, серверы ssh — с портом 22, и т. д.

Соответствие служб и номеров портов находится в файле **/etc/services**.

Для номера порта и IP-адреса определены типы **in_pcrt_t** и **in_addr_t**, которые фактически являются типами **uint16_t** и **uint32_t** соответственно. Порядок следования байт в них должен соответствовать сетевому порядку. Это означает необходимость использования функций **htons()** и **htonl()** примерно таким образом:

```
struct scckaddr_in sa;
sa.sin.family      = AF_INET;
sa.sin.pcrt        = htons(80);
sa.sin.addr.s_addr = htonl((80UL << 24) + (94UL << 16) +(171UL << 8) + 236UL);
```

Но для адресов, записываемых в точечной нотации, существует более удобные функции, которые преобразуют строку с адресом в целое число с сетевым порядком следования байт.

Адреса IPv4 делятся на
- однозначные (unicast);
- широковещательные (broadcast);
- многоадресные (multicast).

Однозначный адрес указывает на один интерфейс узла, широковещательный адрес указывает на все узлы в сети, а многоадресный указывает на все узлы многоадресной группы (multicast group).

Дейтаграммы могут посылаться или приниматься по широковещательным адресам только, если для сокета установлен флаг **SO_BROADCAST**.

В текущей реализации сокетам, ориентированным на соединения, разрешено иметь только однозначные адреса.

Значение адреса и порта всегда хранится в сетевом порядке байт. В частности, это означает, что требуется вызывать **`htons(3)`** для числа, обозначающего порт.

Все функции из стандартной библиотеки, используемые для работы с адресами/портами, используют сетевой порядок байт.

Есть несколько специальных адресов:

INADDR_LOOPBACK (127.0.0.1) всегда ссылается на локальный узел через интерфейс обратной петли;

INADDR_ANY (0.0.0.0) означает любой адрес для связывания;

INADDR_BROADCAST (255.255.255.255) означает любой узел и, по историческим причинам, при связывании подобен **INADDR_ANY**.

Следует быть осторожным при использовании параметра **SO_BROADCAST** — в Linux он не является привилегированным. Если небрежно относиться к широковещательным сообщениям, то можно легко перегрузить сеть. В новых протоколах для приложений лучше использовать многоадресные рассылки вместо широковещательных. **Не используйте широковещание**.

Значение **INADDR_ANY** (0.0.0.0) и **INADDR_BROADCAST** (255.255.255.255) указываются с нейтральным порядком байт.

Это означает, что **`htonl(3)`** на них не действует.

```
$ ifconfig
enp0s31f6: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.121  netmask 255.255.255.0  broadcast 192.168.100.255
        inet6 fe80::e270:eaff:fe53:b1e8  prefixlen 64  scopeid 0x20<link>
        ether e0:70:ea:53:b1:e8  txqueuelen 1000  (Ethernet)
...
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
$
```

getsockopt(), setsockopt() — получение и установка параметров сокетов

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int getsockopt(int sockfd,         // файловый дескриптор сокета
               int level,          // уровень протокола (IPPROTO_IP для IP)
               int optname,        // параметр сокета
               void *optval,       // значение
               socklen_t *optlen); // размер

int setsockopt(int sockfd,         // файловый дескриптор сокета
               int level,          // уровень протокола
               int optname,        // параметр сокета
               const void *optval, // значение
               socklen_t optlen);  // размер
```

Функции **getsockopt()** и **setsockopt()** манипулируют параметрами сокета, на который ссылается файловый дескриптор **sockfd**.

Опции могут существовать на нескольких уровнях протокола и они всегда присутствуют на самом верхнем уровне сокета.

При манипулировании параметрами сокета необходимо указать уровень, на котором находится параметр, и имя параметра.

Чтобы управлять параметрами на уровне API сокетов, уровень указывается как **SOL_SOCKET**.

Для управления опциями на любом другом уровне предоставляется номер соответствующего протокола, управляющего опцией. Например, чтобы указать, что параметр должен интерпретироваться протоколом TCP, уровень должен быть установлен на номер протокола TCP.

Номер протокола можно получить с помощью функций **getproto*()**, которые берут эту информацию из базы протоколов (protocols(5)), либо взять его из самой базы (**/etc/protocols**).

```
$ cat /etc/protocols | grep -i TCP
tcp 6   TCP     # transmission control protocol
```

Для доступа к значениям параметров в случае **setsockopt()** используются аргументы **optval** и **optlen**. Для **getsockopt()** они идентифицируют буфер, в котором должно быть возвращено значение запрошенного параметра (опции).

Для **getsockopt()** аргумент **optlen** — это аргумент значение/результат, изначально содержащий размер буфера, на который указывает **optval**, и измененный при возврате для указания фактического размера возвращаемого значения.

Если значение параметра не должно предоставляться или возвращаться, **optval** может иметь значение **NULL**.

Имя параметра и любые указанные параметры передаются без интерпретации в соответствующий модуль протокола, где и будут проинтерпретированы.

Включаемый файл **<sys/socket.h>** содержит определения параметров уровня сокета. Опции на других уровнях протокола различаются по как формату, так и по имени.

Большинство параметров уровня сокета используют аргумент типа **int** для **optval**.

Для **setsockopt()** аргумент должен быть ненулевым, чтобы включить логическую опцию, или нулем, если опция должна быть отключена.

Описание доступных параметров сокета можно найти **socket (7)** и на страницах руководства по соответствующему протоколу.

Параметры сокета

IP поддерживает некоторые параметры сокета, относящиеся к протоколу, которые могут быть установлены с помощью **`setsockopt(2)`** и прочитаны с помощью **`getsockopt(2)`**.

Значением уровня (**`level`**) параметров сокета для IP является **IPPROTO_IP**. Логический флаг в виде целого числа со значением ноль означает «ложь», другие значения —«истина».

Если сокету передается неправильный параметр, то **`getsockopt(2)`** и **`setsockopt(2)`** завершаются с ошибкой **ENOPROTOOPT**.

IP_ADD_MEMBERSHIP

Присоединиться к многоадресной (multicast) группе.

Группа многоадресной рассылки (multicast group) — это набор систем, которым присвоен IP-адрес многоадресной рассылки из диапазона класса D. Члены группы продолжают использовать собственные IP-адреса, однако они имеют возможность принимать данные, посланные в многоадресной рассылке. Любая система может принадлежать нескольким группам многоадресной рассылки или ни одной из них.

Адреса класса D для многоадресных рассылок имеют старшие биты **`1110`** и находятся в диапазоне номеров от 224 до 239. Некоторые IP-адреса многоадресных рассылок являются постоянными (они перечислены в RFC о присвоенных номерах Интернета). К таким адресам относятся:

224.0.0.1 Все хосты локальной подсети

224.0.0.2 Все маршрутизаторы локальной подсети

224.0.0.5 Все маршрутизаторы, поддерживающие протокол Open Shortest Path First (OSPF)

Многоадресные рассылки могут применяться для временной группы систем, создаваемой или ликвидируемой по мере надобности, например для аудио- или видеоконференций.

Многоадресные рассылки не ограничиваются только локальными сетями. Маршрутизаторы со специальным программным обеспечением для таких рассылок способны распространять дата-граммы IP среди систем в интернете.

Хост должен поддерживать несколько определенных функций, чтобы участвовать в одной или нескольких группах многоадресных рассылок:

- реализацию команды для объединения с многоадресной группой и идентификации интер-фейса, который будет отслеживать соответствующие адреса;
- распознавание на уровне IP многоадресную рассылку для входящих и исходящих датаграмм;
- должна существовать команда, позволяющая хосту исключить себя из группы многоадресной рассылки.

Для более эффективного выполнения рассылки маршрутизатор должен знать, принадлежит ли хост локальной сети одной из многоадресных групп. Кроме того, маршрутизаторам необходимо обмениваться информацией между собой для определения многоадресных групп в удаленных се-тях, куда должны направляться датаграммы.

Хосты используют протокол обслуживания групп Интернета (Internet Group Management Protocol — IGMP) для отчета о своем членстве в группе перед ближайшим маршрутизатором, под-держивающим многоадресные рассылки. Такой отчет посылается по IP-адресу многоадресной рас-сылки, присвоенному данной группе. Маршрутизатор не транслирует такой отчет вне пределов ло-кальной сети, поэтому он будет услышан только маршрутизаторами и другими членами локальной группы.

Так как протокол IGMP предполагает полноту информации о членстве в группе, то он разреша-ет маршрутизаторам периодически опрашивать хосты о членстве в различных текущих группах. Опрос проводится по IP-адресу многоадресной рассылки 224.0.0.1 на все хосты.

В аргументе параметра **IP_ADD_MEMBERSHIP** указывается структура **ip_mreqn**.

```
struct ip_mreqn {
    struct in_addr imr_multiaddr; // IP-адрес группы
    struct in_addr imr_address;   // IP локального интерфейса
    int            imr_ifindex;   // индекс интерфейса
};
```

В **imr_multiaddr** содержится адрес многоадресной группы, в которую приложение хочет войти или выйти. Это должен быть правильный адрес многоадресной рассылки (иначе **setsockopt(2)** завершится с ошибкой **EINVAL**).

В **imr_address** указывается адрес локального интерфейса, через который система должна войти в многоадресную группу; если указано значение **INADDR_ANY**, то нужный интерфейс выбирается системой самостоятельно.

В **imr_ifindex** указывается индекс интерфейса, через который нужно войти/выйти в группу **imr_multiaddr**, или 0, если интерфейс может быть любым.

IP_ADD_MEMBERSHIP допустим только для **setsockopt(2)**.

IP_ADD_SOURCE_MEMBERSHIP

Присоединиться к многоадресной группе и разрешить принимать данные только из указанного источника. Аргументом является структура **ip_mreq_source**.

```
struct ip_mreq_source {
    struct in_addr imr_multiaddr;  // IP многоадресной группы
    struct in_addr imr_interface;  // IP-адрес локального интерфейса
    struct in_addr imr_sourceaddr; // IP-адрес многоадресного источника
};
```

Структура **ip_mreq_source** похожа на **ip_mreqn**, которая описана в **IP_ADD_MEMBERSIP**.

imr_multiaddr — адрес многоадресной группы, к которой приложение хочет подключиться или выйти.

imr_interface — адрес локального интерфейса, с которого система должна подключаться к многоадресной группе.

imr_sourceaddr — адрес источника, из которого приложение хочет получать данные.

Для приёма данных из нескольких источников этот параметр можно использовать несколько раз.

IP_BLOCK_SOURCE

Прекратить приём многоадресных данных из указанного источника заданной группы.

Это допустимо, если приложение подписывалось на многоадресную группу с помощью **IP_ADD_MEMBERSHIP** или **IP_ADD_SOURCE_MEMBERSHIP**.

Аргументом является структура **ip_mreq_source**, описанная в разделе про **IP_ADD_SOURCE_MEMBERSHIP**.

IP_DROP_MEMBERSHIP

Выйти из многоадресной группы.

Аргументом является структура **ip_mreqn** или **ip_mreq**, описана в **IP_ADD_MEMBERSHIP**.

IP_DROP_SOURCE_MEMBERSHIP

Выйти из указанной группы — то есть прекратить приём данных указанной многоадресной группы, которые поступают из указанного источника. Если приложение подписано на несколько источников одной группы, то данные из оставшихся источников продолжат поступать. Чтобы прекратить приём данных из всех источников сразу, следует использовать **IP_DROP_MEMBERSHIP**.

Аргументом является структура **ip_mreq_source**, описанная в **IP_ADD_SOURCE_MEMBERSHIP**.

IP_FREEBIND

Этот логический параметр позволяет привязаться (если значение равно «истина») к IP-адресу, который не является локальным или (пока) не существует. Это позволяет прослушивать сокет, не имея нижележащего сетевого интерфейса или назначенного динамического IP-адреса, которых может ещё не быть, когда приложение пытается связаться с ним.

Этот параметр имеет эквивалентный интерфейс **ip_nonlocal_bind** (описан далее) в **/proc** на каждый сокет.

IP_HDRINCL

Если значение равно «истина», то это означает, что пользователь добавил заголовок IP в начало своих данных.

Допустим только для сокетов **SOCK_RAW**; более подробную информацию следует смотреть в **raw(7)**. Если этот флаг установлен, то значения, заданные параметрами **IP_OPTIONS**, **IP_TTL** и **IP_TOS**, игнорируются.

IP_MSFILTER

Этот параметр предоставляет доступ к расширенному программному интерфейсу фильтрации. Аргументом является структура **ip_msfilter**.

```
struct ip_msfilter {
    struct in_addr imsf_multiaddr; // IP-адрес многоадресной группы
    struct in_addr imsf_interface; // IP-адрес локального интерфейса
    uint32_t       imsf_fmode;     // Режим фильтрации

    uint32_t       imsf_numsrc;    // Количество источников в следующем массиве
    struct in_addr imsf_slist[1];  // Массив адресов источников
};
```

Для задания режима фильтрации существует два макроса — **MCAST_INCLUDE** и **MCAST_EXCLUDE**. Также, существует макрос **IP_MSFILTER_SIZE(n)**, которым можно определить количество памяти, требуемой для хранения структуры **ip_msfilter** с **n** источниками в списке источников. Полное описание фильтрации многоадресных групповых источников следует смотреть в RFC 3376.

IP_MTU

Возвращает известное в данный момент значение MTU маршрута текущего сокета. Возвращается целое число. Параметр **IP_MTU** допускается только для **getsockopt(2)** и может использоваться только для подключённого сокета.

IP_MTU_DISCOVER

Устанавливает или возвращает значение Path MTU Discovery (обнаружение значения MTU маршрута) для сокета. Если он установлен, то Linux будет производить обнаружение значения MTU маршрута для сокетов **SOCK_STREAM** согласно RFC 1191. Для сокетов не **SOCK_STREAM** при значении **IP_PMTUDISC_DO** у всех исходящих пакетов будет устанавливаться флаг запрета фрагментации.

Ответственность за разбивку данных на пакеты согласно размеру MTU, и за выполнение, при необходимости, повторной передачи данных, лежит на пользователе. Ядро будет отвергать пакеты (с ошибкой EMSGSIZE), размер которых больше текущего значения MTU у маршрута. При значении **IP_PMTUDISC_WANT** дейтаграмма будет фрагментироваться по размеру MTU, если требуется, иначе устанавливается флаг запрета фрагментации.

Системное значение по умолчанию можно переключать между **IP_PMTUDISC_WANT** и **IP_PMTUDISC_DONT**, записывая, соответственно, нулевое и ненулевое значение в файл **/proc/sys/net/ipv4/ip_no_pmtu_disc**.

Значение MTU маршрута	Значение
IP_PMTUDISC_WANT	Использовать для каждого маршрута своё значение.
IP_PMTUDISC_DONT	Никогда не выполнять обнаружение значения MTU маршрута.
IP_PMTUDISC_DO	Всегда выполнять обнаружение значения MTU маршрута.
IP_PMTUDISC_PROBE	Установить DF, но игнорировать маршрут MTU.

Если задано значение Path MTU Discovery, то ядро автоматически следит за MTU маршрута для каждого удалённого узла. Когда с некоторым узлом установлено соединение с помощью **connect(2)**, текущее значение MTU маршрута можно легко получить через параметр сокета **IP_MTU** (например, после возникновения ошибки **EMSGSIZE**). Значение MTU может со временем меняться. Для сокетов без установления соединения, которые имеют несколько узлов-получателей, новое значение MTU для заданного узла назначения может быть получено с помощью очереди ошибок (см. **IP_RECVERR**). При каждом входящем сообщении об обновлении MTU в очередь будет добавляться новая ошибка.

Во время процесса обнаружения MTU начальные пакеты от дейтаграмных сокетов могут быть отброшены. Приложения, использующие UDP, должны учитывать это и не думать, что эти пакеты будут переданы повторно.

Чтобы запустить процесс обнаружения MTU маршрута для сокетов без установления соединения сначала можно установить большой размер дейтаграммы (с размером заголовка до 64 килобайт) и сокращать его при изменении MTU маршрута.

Чтобы получить начальную оценку MTU маршрута, соедините дейтаграмный сокет с адресом назначения, используя connect(2), и узнайте значение MTU путем вызова **getsockopt(2)** с параметром IP_MTU.

IP_MULTICAST_ALL

Может использоваться для изменения политики доставки многоадресных сообщений в сокеты, подсоединённые к шаблонному (wildcard) адресу **INADDR_ANY**.

Аргументом является логическое целое (по умолчанию 1). Если значение равно 1, то сокет будет принимать сообщения от всех групп, к которым было выполнено присоединение глобально всей системы. В противном случае будут доставляться сообщения от групп, к которым было выполнено присоединение явным образом(например, с помощью **IP_ADD_MEMBERSHIP**) на этом сокете.

IP_BIND_ADDRESS_NO_PORT

Информирует ядро, что не требуется резервировать динамический (эфемерный) порт при использовании **bind(2)** с номером порта 0. Порт будет автоматически выбран позднее при вызове **connect(2)** — это позволяет использовать общий исходящий порт.

IP_MULTICAST_IF

Назначает локальное устройство для многоадресного группового сокета (multicast socket). Аргументом для **setsockopt(2)** является структура **ip_mreqn** или **ip_mreq**, подобная **IP_ADD_MEMBERSHIP** или структуре **in_addr** (при передаче ядро определяет нужную структуру исходя из размера, переданного в optlen). Для **getsockopt(2)** аргументом является структура **in_addr**.

IP_MULTICAST_TTL

Устанавливает или возвращает значение времени существования (time-to-live) для многоадресных исходящих из этого сокета пакетов, использующих многоадресную адресацию. Для подобных пакетов очень важно установить наименьшее возможное значение TTL. По умолчанию оно равно 1, это значит, что многоадресные пакеты не выйдут за пределы локальной сети, если только пользовательская программа явно не попросит этого. Значением аргумента является целое число.

IP_NODEFRAG

Если установлен (аргумент не равен нулю), то на уровне netfilter запрещается выполнять переборку (reassembly) исходящих пакетов. Значением аргумента является целое число.

Этот параметр допускается только для сокетов с типом SOCK_RAW.

IP_MULTICAST_LOOP

Устанавливает или возвращает логический флаг в виде целого числа, в зависимости от того, будут ли пакеты, использующие многоадресную адресацию, закольцовываться на локальные сокеты.

IP_PKTINFO (начиная с Linux 2.2)

Передает вспомогательное (ancillary) сообщение **IP_PKTINFO** с структурой **pktinfo**, которая содержит некоторую информацию о входящем пакете.

Аргументом является флаг, который сообщает сокету, нужно ли посылать сообщение **IP_PKTINFO** или нет. Само сообщение может быть послано/получено только в виде управляющего сообщения с пакетом, используя **recvmsg(2)** или **sendmsg(2)**.

```
struct in_pktinfo {
    unsigned int   ipi_ifindex;  // индекс интерфейса
    struct in_addr ipi_spec_dst; // локальный адрес
    struct in_addr ipi_addr;     // заголовок адреса назначения
};
```

ipi_ifindex — это уникальный индекс интерфейса, из которого был получен этот пакет. **ipi_spec_dst** это локальный адрес пакета, а **ipi_addr** это адрес назначения, указанный в заголовке пакета.

Если для **sendmsg(2)** передаётся параметр **IP_PKTINFO** и **ipi_spec_dst** не равно нулю, то **ipi_spec_dst** будет использован как локальный адрес источника при просмотре таблицы маршрутизации и для установки IP-параметров маршрутизации от источника. Если значение **ipi_ifindex** не равно нулю, то при поиске в таблице маршрутизации вместо значения **ipi_spec_dst** используется первичный локальный адрес интерфейса с указанным индексом.

IP_RECVERR

Делает передачу сообщений об ошибках более надёжной. Если для дейтаграмного сокета установлен этот параметр, то все возникающие ошибки будут поставлены в очередь ошибок, свою для каждого сокета.

Для получения ошибки при операции с сокетом пользователь может воспользоваться вызовом **recvmsg(2)** с установленным флагом **MSG_ERRQUEUE**. Структура **sock_extended_err**, описывающая ошибку, будет передана в вспомогательном сообщении с типом **IP_RECVERR** и уровнем **IPPROTO_IP**. Этот параметр полезен для надежной обработки ошибок для сокетов без установления соединения. В пакете с ошибкой из очереди ошибок, также содержится порция полученных данных. Вспомогательное сообщение **IP_RECVERR** содержит структуру **sock_extended_err**:

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3

struct sock_extended_err {
    uint32_t ee_errno;   // номер ошибки
    uint8_t  ee_origin;  // где возникла ошибка
    uint8_t  ee_type;    // тип
    uint8_t  ee_code;    // код
    uint8_t  ee_pad;
    uint32_t ee_info;    // дополнительная информация
    uint32_t ee_data;    // другие данные
    /* Дальше могут следовать данные */
};
struct sockaddr *SO_EE_OFFENDER(struct sock_extended_err *);
```

В **ee_errno** содержится номер ошибки в очереди.

В **ee_origin** содержится код источника ошибки. Значение остальных полей зависит от протокола.

Макрос **SO_EE_OFFENDER** возвращает указатель на адрес сетевого объекта, в котором возникла ошибка, согласно указанному указателю на вспомогательное сообщение. Если адрес неизвестен, то поле **sa_family** структуры **sockaddr** будут содержать **AF_UNSPEC**, и остальные поля **sockaddr** будут не определены.

Для IP структура **sock_extended_err** используется следующим образом — значение поля **ee_origin** устанавливается в **SO_EE_ORIGIN_ICMP**, если ошибка получена из пакета **ICMP**, или в **SO_EE_ORIGIN_LOCAL**, если возникла локальная ошибка. Неизвестные значения следует игнорировать. Значения полей **ee_type** и **ee_code** устанавливаются исходя из значений полей типа и кода заголовка ICMP.

При ошибках **EMSGSIZE** поле **ee_info** содержит обнаруженную величину MTU. Сообщение также содержит структуру **sockaddr_in** узла, вызвавшего ошибку, которая доступна через макрос **SO_EE_OFFENDER**. Если источник неизвестен, то поле **sin_family** адреса, возвращённого макросом **SO_EE_OFFENDER**, содержит значение **AF_UNSPEC**. Если ошибка возникла в сети, то все параметры IP (**IP_OPTIONS**, **IP_TTL** и т.д.), которые используются сокетом и содержатся в пакете с описанием ошибки, передаются в управляющих сообщениях. Данные пакета, вызвавшего ошибку, возвращаются как нормальные данные.

Следует отметить, что у TCP нет очереди ошибок, соответственно, флаг **MSG_ERRQUEUE** нельзя использовать для сокетов типа **SOCK_STREAM**. Параметр **IP_RECVERR** допустим для TCP, но все ошибки возвращаются только через функцию сокета или через параметр **SO_ERROR**.

Для неструктурированных сокетов, параметром **IP_RECVERR** включается передача в приложение всех получаемых ошибок ICMP, иначе сообщается только об ошибках в сокетах, ориентированных на соединение. Этот параметр устанавливается или возвращается как логический флаг в виде целого числа. По умолчанию, параметр IP_RECVERR выключен.

IP_RECVOPTS

Передает пользователю все входящие параметры IP с помощью управляющего сообщения **IP_OPTIONS**. Для локального узла заполняется заголовок маршрутизации и другие параметры. Не поддерживается сокетами типа **SOCK_STREAM**.

IP_RETOPTS

Идентичен параметру **IP_RECVOPTS**, но возвращает необработанные параметры, причём без заполненных временных меток и записи о маршрутизации до этой точки (hop).

IP_RECVTOS

Если включён, то вместе с входящими пакетами передаётся вспомогательное сообщение **IP_TOS**. В нём содержится байт, в котором указано поле типа сервиса/приоритета из заголовка пакета. Ожидается логическое значение в виде целого числа.

IP_TOS

Устанавливает или получает значение поля Type-Of-Service (TOS, тип сервиса) каждого IP-пакета, который отсылается с этого сокета. Это поле используется для указания приоритета пакета в сети. Значение TOS хранится в одном байте. Существует несколько стандартных флагов TOS:

IPTOS_LOWDELAY — для минимизации задержки передаваемого трафика;

IPTOS_THROUGHPUT — для оптимизации пропускной способности;

IPTOS_RELIABILITY — для увеличения надёжности;

IPTOS_MINCOST — при пересылки данных, для которых неважна скорость передачи.

Может быть указано не более одного из этих значений TOS. Все другие биты являются недействительными и должны быть обнулены. По умолчанию, Linux посылает дейтаграммы с IPTOS_LOWDELAY первыми, но точное поведение зависит от настроенного порядка очередности (queueing discipline).

IP_RECVTTL

Если указан этот флаг, то передаётся управляющее сообщение **IP_TTL** с байтом значения поля времени существования из полученного пакета в виде 32-битного целого. Не поддерживается сокетами типа **SOCK_STREAM**.

IP_TTL

Устанавливает или получает текущее значение поля времени существования (time to live), которое указывается в каждом пакете, отсылаемом с этого сокета.

IP_RECVORIGDSTADDR

Данный логический параметр включает вспомогательное сообщение **IP_ORIGDSTADDR** в **recvmsg(2)**, в котором ядро возвращает первоначальный адрес назначения полученной дейтаграммы. Вспомогательное сообщение содержит структуру **struct sockaddr_in**.

IP_ROUTER_ALERT

Передаёт этому сокету все пересылаемые (forwarded) пакеты с установленным параметром IP Router Alert. Этот параметр используется для «raw» сокетов. Он может быть полезен, например, для служб RSVP, запущенных в пространстве пользователя. Перехваченные пакеты дальше ядром не пересылаются — ответственность за их отсылку лежит на пользователе. Привязка сокета игнорируется, так как пакеты фильтруются по протоколу. Ожидается логическое значение в виде целого числа.

IP_UNBLOCK_SOURCE

Разблокировать ранее заблокированный многоадресный источник. Возвращает EADDRNOTAVAIL, если указанный источник не заблокирован.

Аргументом является структура ip_mreq_source, описанная в разделе о IP_ADD_SOURCE_MEMBERSHIP.

IP_TRANSPARENT

Установка этого логического параметра включает прозрачное проксирование на заданный сокет. Данный параметр сокета позволяет вызвавшему приложению привязаться к нелокальному IP-адресу и работать клиентом и сервером с внешним адресом как с локальной конечной точкой.

ЗАМЕЧАНИЕ: требуется настройка маршрутизации пакетов для внешнего адреса через TProxy (то есть, системы, на которой находится приложение, применяющее параметр сокета IP_TRANSPARENT).

Для установки данного параметра сокета требуются права суперпользователя (мандат CAP_NET_ADMIN).

Также, для установки данного параметра на перенаправляемый сокет требуется перенаправление TProxy с помощью цели TPROXY в iptables.

IP_OPTIONS

Устанавливает или возвращает параметры IP, которые посылаются с каждым пакетом из данного сокета. Аргументами являются указатель на буфер памяти с этими параметрами и размер параметра. Системный вызов **`setsockopt(2)`** устанавливает параметры IP, связанные с сокетом. Для IPv4 максимальный размер параметра IPv4 равен 40 байтам. Все возможные параметры перечислены в RFC 791. Если пакет, устанавливающий соединение с сокетом типа SOCK_STREAM, содержит параметры IP, то эти параметры IP (с инвертированными заголовками маршрутизации) будут использоваться в этом сокете. После установления соединения изменять параметры входящими пакетами запрещено. По умолчанию, обработка всех параметров, связанных с маршрутизацией от источника, отключена, но её можно включить через интерфейс accept_source_route в /proc.

Другие параметры, например связанные с временными отметками (timestamp), продолжают обрабатываться. Для дейтаграмных сокетов параметры IP могут быть установлены только локальным пользователем. Вызов getsockopt(2) с параметром IP_OPTIONS помещает в указанный буфер текущие параметры IP, используемые при отправке.

Интерфейсы в /proc

Настройку глобальных параметров протокола IP можно осуществлять через интерфейс **/proc**. Все параметры доступны посредством чтения или записи файлов из каталога **/proc/sys/net/ipv4/**.

Для логических (Boolean) параметров значения указываются в виде целых чисел — ненулевое значение («истина») означает включает параметра, а нулевое значение («ложь») — выключение.

ip_always_defrag

[Появился в ядре версии 2.2.13; в ранних версиях это свойство контролировалось с помощью флага **CONFIG_IP_ALWAYS_DEFRAG** времени компиляции; данный параметр убран в 2.4.x]

Если этот флаг включён (не равен 0), то входящие фрагменты (части IP-пакетов, которые образуются, если некоторый узел, находящийся между отправителем и получателем, решает, что пакеты слишком велики и разделяет их на кусочки) будут снова собраны (дефрагментированны) перед дальнейшей обработкой, даже если они должны быть пересланы дальше.

Включать этот параметр следует только на межсетевом экране, который является единственной связью с вашей сетью, или на прозрачном прокси. Если включить его на обычном маршрутизаторе или узле, соединение может быть нарушено, если фрагменты передаются по различным линиям. Дефрагментация также требует много памяти и процессорного времени.

Этот параметр включается автоматически при настройке маскарадинга или прозрачного проксирования.

ip_dynaddr (Boolean; по умолчанию: выключен)

Включает динамическую адресацию сокета и подмену (masquerading) при изменении адреса интерфейса. Это полезно для интерфейсов коммутируемых соединений (dialup) с изменяющимися IP-адресами. Значение 0 означает не подменять, 1 включает подмену и 2 включает режим подробностей работы.

ip_default_ttl (integer; по умолчанию: 64)

Устанавливает значение time-to-live по умолчанию для исходящих пакетов. Это значение может быть изменено для каждого отдельного сокета с помощью параметра IP_TTL.

ip_forward (Boolean; по умолчанию: выключен)

Включает/выключает пересылку (forwarding) IP-пакетов. Пересылка IP также может быть включена для каждого интерфейса в отдельности.

ip_local_port_range (32768 60999)

В этом файле содержатся два целых числа, определяющие диапазон локальных портов по умолчанию, выделенных для сокетов, у которые нет явно привязанного номера порта — то есть диапазон эфемерных портов. Эфемерный порт выделяется сокету в следующих случаях:

- при вызове **bind(2)** в номере порта адреса сокета указан 0;
- вызов **listen(2)** вызван для потокового сокета, который ещё не привязан;
- вызов **connect(2)** вызван для сокета, который ещё не привязан;
- вызов **sendto(2)** вызван для дейтаграмного сокета, который ещё не привязан.

Выделение эфемерных портов начинается с первого числа в **ip_local_port_range** и заканчивается вторым числом.

Если диапазон эфемерных портов закончился, то соответствующий системный вызов вернёт ошибку.

Следует отметить, что диапазон портов в ip_local_port_range не должен конфликтовать с портами, используемыми для маскарадинга (хотя это проверяется). Также, произвольные значения могут вызвать проблемы с некоторыми пакетными фильтрами межсетевых экранов, которые делают предположения об используемых локальных портах. Первое число должно быть не менее 1024, или лучше более 4096, чтобы не пересекаться с всем известными портами и минимизировать проблемы с межсетевыми экранами.

ip_no_pmtu_disc (Boolean; по умолчанию: выключен)

Если включён, то, по умолчанию, не производится обнаружение значения MTU у маршрута для TCP сокетов. Обнаружение MTU маршрута может завершиться с ошибкой из-за встретившихся на пути неверно настроенных межсетевых экранов (которые отбрасывают все пакеты ICMP) или из-за неверно настроенных интерфейсов (например, соединение точка-точка, у которого оба конца не договорились о MTU). Лучше исправить встреченные на пути неисправные маршрутизаторы, чем глобально отключать обнаружение MTU маршрута, потому что это отключение приведёт к высокой нагрузке на сеть.

ip_nonlocal_bind (Boolean; по умолчанию: выключен)

Если установлен, то это позволяет процессам привязываться (bind(2)) к нелокальным IP-адресам, что полезно, но может привести к неработоспособности некоторых приложений.

ip6frag_time (integer; по умолчанию: 30)
Время в секундах, на которое фрагмент IPv6 остаётся в памяти.

ip6frag_secret_interval (integer; по умолчанию: 600)
Интервал регенерации (в секундах) контрольной суммы секрета (hash secret) (или время существования контрольной суммы секрета) фрагментов IPv6.

ipfrag_high_thresh (integer), ipfrag_low_thresh (integer)
Если количество фрагментов IP, стоящих в очереди, достигает значения ipfrag_high_thresh, то очередь укорачивается до значения ipfrag_low_thresh. Содержит целое число, означающее количество байт.

Ошибки

EACCES — Пользователь попытался выполнить действие, не имея на это необходимых полномочий. Примеры таких действий: посылка пакета по широковещательному адресу без предварительной установки флага SO_BROADCAST; посылка пакета по запрещённому маршруту; изменение настроек межсетевого экрана не имея прав суперпользователя (мандата CAP_NET_ADMIN); связывание сокета с зарезервированным портом, не имея прав суперпользователя (мандата CAP_NET_BIND_SERVICE).

EADDRINUSE — Попытка связать сокет с уже используемым адресом.

EADDRNOTAVAIL — Был запрошен несуществующий интерфейс или запрошенный исходящий адрес не является локальным.

EAGAIN — Действие над неблокирующим сокетом привело бы к его блокировке.

EALREADY — Операция соединения на неблокирующем сокете уже находится в процессе выполнения.

ECONNABORTED — Соединение закрыто во время accept(2).

EHOSTUNREACH — В таблице маршрутизации нет допустимых записей, соответствующих адресу назначения. Эта ошибка может возникнуть из-за ICMP-сообщения от удалённого маршрутизатора или из-за локальной таблицы маршрутизации.

EINVAL — Передан недопустимый аргумент. При операциях отправки эта ошибка может возникнуть из-за передачи по маршруту чёрная дыра (blackhole).

EISCONN — Вызов connect(2) запущен для сокета, уже установившего соединение.

EMSGSIZE — Дейтаграмма больше значения MTU на маршруте, и она не может быть фрагментирована.

ENOBUFS, **ENOMEM** — Недостаточно свободной памяти. Часто это означает, что выделение памяти ограничено не размером системной памяти, а границами буфера сокета, но это не всегда так.

ENOENT — Для сокета вызван SIOCGSTAMP, но он ещё не получил ни одного пакета.

ENOPKG — Не настроена подсистема ядра.

ENOPROTOOPT и **EOPNOTSUPP** — Передан недопустимый параметр сокета.

ENOTCONN — Операция определена только для сокета, установившего соединение, а этот сокет не соединён.

EPERM — У пользователя нет достаточных полномочий, чтобы повысить приоритет, изменить настройку или послать сигнал запрашиваемому процессу или группе процессов.

EPIPE — Соединение неожиданно закрылось или завершено (shut down) другой стороной.

ESOCKTNOSUPPORT — Сокет не настроен или запрошен неизвестный тип сокета.

Протоколами более высокого уровня могут генерироваться другие ошибки.

inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp); // 1.2.3.4 -> struct in_addr
in_addr_t inet_addr(const char *cp);                // 1.2.3.4 -> in_addr_t
in_addr_t inet_network(const char *cp);             // 1.2.3.4 -> in_addr_t

char *inet_ntoa(struct in_addr in);                 // in_addr_t -> 1.2.3.4

in_addr_t inet_lnaof(struct in_addr in);            // in_addr -> host#
in_addr_t inet_netof(struct in_addr in);            // in_addr -> net#
struct in_addr inet_makeaddr(in_addr_t net, in_addr_t host); // net#+host#->in_addr
```

inet_aton()

Преобразует адрес интернет-узла **cp** из числовой формы записи IPv4 с точками (IPv4 numbers-and-dots) в двоичную форму (с сетевым порядком байт) и сохраняет её в структуре, на которую ссылается **inp**. Вызов **inet_aton()** возвращает ненулевое значение, если адрес правильный, и ноль, если нет. Адрес, указанный в **cp**, может принимать одну из следующих форм:

a.b.c.d — каждая из четырёх групп чисел представляет байт адреса. Байты назначаются слева направо.

a.b.c — части **a** и **b** задают первые два байта двоичного адреса. Часть **c** расценивается как 16- разрядное значение, определяющее два самых правых байта двоичного адреса. Такая запись совместима с сетевыми адресами (устаревшими) класса B.

a.b — часть **a** определяет первый байт двоичного адреса. Часть **b** расценивается как 24-разрядное значение, определяющее три самых правых байта двоичного адреса. Данная форма записи совместима с сетевыми адресами (устаревшими) класса A.

a — значение a расценивается как 32-разрядное значение, которое хранится в двоичном формате без какой либо перегруппировки байтов.

Во всех вышеперечисленные формах записи адресов числа могут быть указаны в десятичной, восьмеричной (с префиксом 0) или шестнадцатеричной (с префиксом 0X) системе счисления.

Адреса, записанные в любой из этих форм, называются числовой формой записи IPv4 с точками.

Форма записи, в которой используются только четыре десятичных числа, называется десятично-точечной записью IPv4 (IPv4 dotted-decimal notation) или иногда IPv4 dotted-quad notation.

Функция **inet_aton()** возвращает 1, если переданная строка была обработана успешно, или 0, если строка имеет некорректные данные (в **errno** ошибка не записывается).

Функция **inet_aton()** не определена в POSIX.1, однако доступна в большинстве систем.

inet_addr()

```
in_addr_t inet_addr(const char *cp);                // 1.2.3.4 -> in_addr_t
```

Функция **inet_addr()** преобразует адрес интернет-узла **cp** из числовой формы записи IPv4 с точками в двоичную форму с сетевым порядком байт.

Если адрес, подаваемый на вход, неверный, функция возвращает **INADDR_NONE** (обычно -1).

Использование этой функции проблематично, т.к. значение -1 эквивалентно корректному адресу (255.255.255.255). Поэтому следует избегать использования этой функции, вместо нее нужно использовать **inet_aton()**, **inet_pton(3)** или **getaddrinfo(3)**, которые используют правильный способ указания на ошибку.

inet_network()

```
in_addr_t inet_network(const char *cp);              // 1.2.3.4 -> in_addr_t
```

Функция **inet_network()** преобразует строку **cp**, записанную в числовой форме записи IPv4 с точками, в число (порядок байт узла), пригодное для использования в качестве сетевого адреса интернета. В случае успешного выполнения возвращается преобразованный адрес. В случае указания некорректной строки, возвращается -1.

inet_ntoa()

```
char *inet_ntoa(struct in_addr in);                  // in_addr_t -> 1.2.3.4
```

Функция **inet_ntoa()** преобразует адрес Интернет-узла **in**, заданного в сетевом порядке байтов, в строку в числовой форме записи IPv4 с точками. Строка возвращается в статически выделяемом буфере, который перезаписывается при последующих вызовах.

inet_lnaof(), inet_netof()

```
in_addr_t inet_lnaof(struct in_addr in);             // in_addr -> host#
in_addr_t inet_netof(struct in_addr in);             // in_addr -> net#
```

Функция **inet_lnaof()** возвращает номер узла из интернет-адреса **in**.
Функция **inet_netof()** возвращает номер сети из интернет-адреса **in**.
Возвращаемые значения имеют порядок байт узла.

inet_makeaddr()

```
struct in_addr
inet_makeaddr(in_addr_t net, in_addr_t host); // net# + host# -> in_addr
```

Функция **inet_makeaddr()** противоположна **inet_netof()** и **inet_lnaof()**. Она возвращает адрес интернет-узла в сетевом порядке байт, создавая его путём объединения номера сети **net** с номером узла **host** (оба задаются в порядке байт узла).

Структура **in_addr**, используемая в **inet_ntoa()**, **inet_makeaddr()**, **inet_lnaof()** и **inet_netof()**, определена в **<netinet/in.h>**:

```
typedef uint32_t in_addr_t;

struct in_addr {
    in_addr_t s_addr;
};
```

inet_lnaof(), **inet_netof()** и **inet_makeaddr()** является устаревшими функциями, которые предполагают, что используется классовая сетевая адресация.

В x86 порядок байтов узла таков, что младший байт является первым (little endian), а в сетевом порядке байт, который используется в интернет, старший байт является первым (big endian).

Пример использования **inet_aton()** и **inet_ntoa()**.

```
$ ./a.out 226.000.000.037  # Последний байт в восьмеричной сс
226.0.0.31
$ ./a.out 0x7f.1           # Первый байт в шестнадцатеричной сс (тип адреса a.b)
127.0.0.1
```

```
#define _BSD_SOURCE
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    struct in_addr addr;
    if (argc != 2) {
        fprintf(stderr, "%s <dotted-address>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (inet_aton(argv[1], &addr) == 0) {
        fprintf(stderr, "Invalid address\n");
        exit(EXIT_FAILURE);
    }
    printf("%s\n", inet_ntoa(addr));
    exit(EXIT_SUCCESS);
}
```

gethostbyname, gethostbyaddr, sethostent, gethostent, endhostent, h_errno, herror, hstrerror

Данные функции позволяют получить записи о сетевом узле

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);

#include <sys/socket.h>       /* для AF_INET */

struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
void            sethostent(int stayopen);
void            endhostent(void);
void            herror(const char *s);
const char     *hstrerror(int err);

/* расширение System V/POSIX */
struct hostent *gethostent(void);
```

Функции: **gethostbyname(), gethostbyaddr(), herror() , hstrerror()** являются устаревшими.

Вместо них в приложениях следует использовать **getaddrinfo(3)**, **getnameinfo(3)** и **gai_strerror(3)**.

gethostbyname()

```
struct hostent *gethostbyname(const char *name);
```

Функция **gethostbyname()** возвращает структуру типа **hostent** для узла **name**.

Значением name может быть или имя узла, или адрес IPv4 в стандартной точечной записи, как в **inet_addr(3)**. Структура **hostent** определена в **<netdb.h>** следующим образом:

```
struct hostent {
    char  *h_name;            // официальное имя узла
    char **h_aliases;         // массив псевдонимов
    int    h_addrtype;        // тип адреса узла
    int    h_length;          // длина адреса
    char **h_addr_list;       // массив адресов
}
#define h_addr h_addr_list[0] // для обратной совместимости
```

h_name — официальное имя узла.

h_aliases — массив альтернативных имён узла, заканчивается указателем **null**.

h_addrtype — тип адреса — AF_INET или AF_INET6.

h_length — длина адреса в байтах.

h_addr_list — массив указателей сетевых адресов узла (в сетевом порядке байт), заканчивается указателем **null**.

h_addr — первый адрес из **h_addr_list**, для обратной совместимости.

Если **name** — адрес IPv4, то поиск не выполняется и **gethostbyname()** просто копирует **name** в поле **h_name**, а его эквивалент **struct in_addr** — в поле **h_addr_list[0]** возвращаемой структуры **hostent**.

Если **name** не оканчивается точкой и установлена переменная окружения **HOSTALIASES**, то **name** сначала ищется в файле псевдонимов, указанном в **HOSTALIASES** (формат файла описан в **hostname(7)**).

Если **name** не оканчивается точкой, то поиск производится в текущем домене и в его предках.

gethostbyaddr()

```
struct hostent *gethostbyaddr(const void *addr,
                              socklen_t   len,
                              int         type);
```

Функция **gethostbyaddr()** возвращает структуру типа **hostent** для адреса узла **addr** длиной **len** и типом адреса **type**. Допустимые типы адресов — **AF_INET** и **AF_INET6**.

Аргумент адреса узла — указатель на структуру с типом, зависящим от типа адреса, например для типа адреса **AF_INET** используется **struct in_addr *** (возможно, полученная из вызова **inet_addr(3)**).

Запросы доменного имени, выполняемые **gethostbyname()** и **gethostbyaddr()**, полагаются на настроенные источники диспетчера службы имён (**nsswitch.conf(5)**) или локальный сервер имён (**named(8)**). Действием по умолчанию является запрос к настроенным источникам диспетчера службы имён (**nsswitch.conf(5)**), при ошибке — к локальному серверу имён (**named(8)**).

Современным способом управления порядком поиска узлов является файл **nsswitch.conf(5)**.

/etc/nsswitch.conf — файл настроек диспетчера службы имён.

Возвращаемое значение

Функции **gethostbyname()** и **gethostbyaddr()** возвращают структуру **hostent** или указатель **NULL** при ошибке. При ошибке переменная **h_errno** содержит номер ошибки. Если получен не **NULL**, то возвращаемое значение может указывать на статические данные.

sethostent()

```
void sethostent(int stayopen);
```

Функция **sethostent()** задаёт (при **stayopen** равным истине (1)), что для опроса сервера имён должен использоваться подключённый сокет TCP и что соединение должно остаться открытым для последующих запросов. В противном случае для опроса сервера имён будут использоваться дейтаграммы UDP.

endhostent()

```
void endhostent(void);
```

Функция **endhostent()** закрывает использованное для опросов сервера имён соединение TCP.

herror()/hstrerror()

```
void herror(const char *s);
const char *hstrerror(int err);
```

Функция **herror()** (устарела) печатает в **stderr** сообщение об ошибке в соответствии с текущим значением **h_errno**.

Функция **hstrerror()** (устарела) принимает номер ошибки (обычно, h_errno) и возвращает соответствующую строку с сообщением об ошибке.

getaddrinfo(), freeaddrinfo(), gai_strerror() — трансляция сетевого адреса и службы

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

int getaddrinfo(const char              *nodename, // узел интернета
                const char              *servname, // служба (порт)
                const struct addrinfo  *hints,    // селектор возврата
                struct addrinfo       **res);     // возвращаемая информация

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```

Функция **getaddrinfo()** преобразует имя местоположения службы[3] (например, имя хоста) и/или имя службы (порт) и возвращает набор адресов сокетов и связанную с ними информацию, которые могут использоваться для создании сокета, с помощью которого можно адресовать указанную службу.

Функция **getaddrinfo()** объединяет в одном интерфейсе возможности, предоставляемые функциями **gethostbyname(3)** и **getservbyname(3)**, но в отличие от этих функций, **getaddrinfo()** реентерабельна и позволяет программам не зависеть от IPv4 или IPv6. Во многих случаях это реализуется системой доменных имен, как описано в RFC 1034, RFC 1035 и RFC 1886.

3) service location

Функция **freeaddrinfo()** освобождает одну или несколько структур **addrinfo**, возвращаемых функцией **getaddrinfo()**, вместе с любой дополнительной памятью, связанной с этими структурами. Если поле **ai_next** структуры не равно нулю, освобождается весь список структур.

Функции **freeaddrinfo()** и **getaddrinfo()** являются потокобезопасными (thread-safe).

Аргументы **nodename** и **servname** являются либо нулевыми указателями, либо указателями на строки с завершающим нулем. Один или оба из этих двух аргументов должны быть ненулевыми указателями.

Реальный формат имени зависит от семейства или семейств адресов. Если конкретное семейство не задано и имя может быть интерпретировано как действительное в пределах нескольких поддерживаемых семейств, реализация будет пытаться разрешить имя во всех поддерживаемых семействах, и при отсутствии ошибок будет возвращен один или несколько результатов.

nodename

Если **nodename** не равен **null**, он может быть описательным именем или строкой адреса.

Если указанное в **hints** семейство адресов — **AF_INET**, **AF_INET6**, или **AF_UNSPEC**, допустимые описательные имена включают имена хостов.

Если указанное семейство адресов — **AF_INET** или **AF_UNSPEC**, допустимы строки адресов, использующие стандартную для Интернета точечную нотацию, как указано в **inet_addr()**.

Если указано семейство адресов **AF_INET6** или **AF_UNSPEC**, допустимы стандартные текстовые формы IPv6, описанные в **inet_ntop()**.

Если **nodename** не равно **null**, запрошенная локация службы идентифицируется по **nodename**.

Если **nodename** равно **null**, запрошенное расположение службы является локальным для вызывающего абонента.

servname

Если **servname** равно **null**, вызов возвращает адреса сетевого уровня для указанного имени узла.

Если **servname** не равно **null**, это должна быть символьная строка с завершающим нулем, идентифицирующая запрошенную службу. Также это может быть либо описательное имя, либо числовое представление, подходящее для использования с семейством или семействами адресов.

Если указано семейство адресов AF_INET, AF_INET6, или AF_UNSPEC, служба может быть указана в виде строки, указывающей десятичный номер порта.

hints

Если аргумент **hints** не **null**, он ссылается на структуру **addrinfo**, содержащую входные значения, которые могут управлять операцией, предоставляя опции и ограничивая возвращаемую информацию определенным типом сокета, семейством адресов и/или протоколом.

Структура **addrinfo** содержит следующие поля:

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;    // AF_INET, AF_INET6 или AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM или SOCK_DGRAM
    int              ai_protocol;  //
    socklen_t        ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

Программист должен гарантировать, что каждый из членов в **ai_flags**, **ai_family**, **ai_socktype** и **ai_protocol**, а также каждый из нестандартных дополнительных элементов, если таковые имеются, этой структуры подсказок инициализированы соответствующим образом.

Члены, не являющиеся **ai_flags**, **ai_family**, **ai_socktype** и **ai_protocol** должны быть установлены в ноль или в нулевой указатель (делаем **memset( )** перед инициализацией).

Если какой-либо из этих элементов имеет значение, отличное от значения, которое было бы результатом инициализации по умолчанию, поведение определяется реализацией.

ai_family — это поле определяет предпочитаемое семейство адресов для возвращаемых адресов. Значениями для данного поля могут быть **AF_INET**, **AF_INET6** и **AF_UNSPEC**.

Значение **AF_UNSPEC** для **ai_family** означает, что вызывающую сторону интересует любое семейство адресов.

ai_socktype — это поле определяет предпочитаемый тип сокета, например, **SOCK_STREAM** или **SOCK_DGRAM**. Нулевое значение для **ai_socktype** означает, что вызывающую сторону интересует любой тип **сокета** и это означает, что **getaddrinfo()** может вернуть адреса сокета любого типа.

ai_protocol — это поле определяет протокол для возвращаемых адресов сокета. Нулевое значение для **ai_protocol** означает, что вызывающую сторону интересует любой протокол и это означает, что **getaddrinfo()** может вернуть адрес сокета с любым протоколом.

Если **hints** является нулевым указателем, поведение должно быть таким, как если бы **hints** ссылался на структуру, содержащую нулевое значение для полей **ai_flags**, **ai_socktype** и **ai_protocol**, а также **AF_UNSPEC** для поля **ai_family**.

Поле **ai_flags**, на которое указывает параметр **hints**, должно быть установлено равным нулю или представлять собой побитовое ИЛИ одного или нескольких значений **AI_PASSIVE**, **AI_CANONNAME**, **AI_NUMERICHOST**, **AI_NUMERICSERV**, **AI_V4MAPPED**, **AI_ALL** и **AI_ADDRCONFIG**.

AI_PASSIVE

Если указан флаг **AI_PASSIVE**, возвращаемая информация об адресе будет пригодна для привязки сокета для приема входящих соединений к указанной службе.

В этом случае, если аргумент **nodename** имеет значение **null**, IP-адрес в структуре адреса сокета для IPv4 будет установлен в **INADDR_ANY** или в **IN6ADDR_ANY_INIT** для адреса IPv6.

Если аргумент **nodename** не равен нулю, флаг **AI_PASSIVE** игнорируется.

Если флаг **AI_PASSIVE** не указан, возвращаемая адресная информация будет пригодна для вызова **connect()** для протокола с установлением соединения, или для вызова **connect()**, **sendto()** или **sendmsg()** для протокол без установления соединения.

Если аргумент **nodename** в этом случае имеет значение **null**, то IP-адреса в структуре адреса сокета будет установлен на петлевой адрес (**127.0.0.1**).

AI_CANONNAME

Если указан флаг **AI_CANONNAME** и аргумент **nodename** не равен нулю, функция попытается определить каноническое имя, соответствующее **nodename** (например, если **nodename** является псевдонимом или сокращенной записью для полного имени).

Примечание:

Поскольку разные реализации используют разные концептуальные модели, термины «каноническое имя» и «псевдоним» для общего случая точно определены быть не могут. Однако ожидается, что реализации системы доменных имен будут интерпретировать их так, как они используются в RFC 1034[4].

Строка числового адреса хоста не является «именем» и, следовательно, не имеет формы «канонического имени», соответственно, в этом случае преобразование адреса в имя хоста не выполняется.

AI_NUMERICHOST

Если указан флаг **AI_NUMERICHOST**, то предоставленная ненулевая строка **nodename** должна быть числовой строкой адреса хоста. В противном случае возвращается ошибка **EAI_NONAME**. Этот флаг должен препятствовать запуску службы разрешения имен любого типа (например, DNS[5]).

---

4) RFC-1034 Domain names – Concepts and Ficilities. Nov. 1987.

5) DNS — Domain Name System («система доменных имён») — распределенная система, использующаяся, в том числе, для получения IP-адреса по имени хоста (компьютера или устройства), получения информации о маршрутизации. Распределённая база данных DNS поддерживается с помощью иерархии DNS-серверов, взаимодействующих по определённому протоколу.

AI_NUMERICSERV

Если указан флаг **AI_NUMERICSERV**, то предоставленная ненулевая строка **servname** должна быть числовой строкой с номером порта. В противном случае будет возвращена ошибка **EAI_NONAME**. Этот флаг должен препятствовать запуску службы разрешения имен любого типа (например, NIS+[6]).

Если в **ai_family** указано с **AF_INET6** и при этом указан флаг **AI_V4MAPPED**, то при отсутствии совпадающих IPv6-адресов **getaddrinfo()** будет возвращать IPv6-адреса, соответствующие IPv4, (**ai_addrlen** должно быть равно 16).

Если же **ai_family** не равен **AF_INET6**, флаг **AI_V4MAPPED** будет игнорироваться Если с флагом **AI_V4MAPPED** используется флаг **AI_ALL**, то **getaddrinfo()** будет возвращать все подходящие IPv6 и IPv4 адреса. Флаг **AI_ALL** без флага **AI_V4MAPPED** игнорируется.

AI_ADDRCONFIG

Если указан флаг **AI_ADDRCONFIG**, IPv4 адреса будут возвращаться только в том случае, если адрес IPv4 в локальной системе настроен, а адреса IPv6 будут возвращаться, только если в локальной системе настроен адрес IPv6.

---

6) NIS+ — служба каталогов, разработанная корпорацией Sun Microsystems. портирована с Solaris на другие Unix-подобные системы, в том числе Linux.

Поле **ai_socktype**, на которое указывает аргумент **hint**, указывает тип сокета для службы, определенный в **socket()**.

Если конкретный тип сокета не указан (например, значение равно нулю) и имя службы может быть интерпретировано как действительное для нескольких поддерживаемых типов сокетов, реализация будет попытаться разрешить имя службы для всех поддерживаемых типов сокетов и, в отсутствие ошибок, будут возвращены все возможные результаты. Ненулевое значение типа сокета должно ограничивать возвращаемую информацию значениями с указанным типом сокета.

Если поле **ai_family**, на которое указывает **hints**, имеет значение **AF_UNSPEC**, будут возвращены адреса, подходящие для использования с любым семейством адресов, которое может использоваться с указанным именем узла и/или именем сервера. В противном случае будут возвращены адреса, подходящие ля использования только с указанным семейством адресов.

Если **ai_family** не равно **AF_UNSPEC** и **ai_protocol** не равно нулю, то возвращаются адреса, пригодные для использования только с указанным семейством адресов и протоколом. Значение **ai_protocol** в этом случае должно интерпретироваться так же, как и в случае вызова функции **socket()** с соответствующими значениями **ai_family** и **ai_protocol**.

Существует несколько причин того, почему связный список может содержать более одной структуры **addrinfo**:

- сетевой узел имеет несколько адресов, доступен по нескольким протоколам (например, AF_INET и AF_INET6);

- служба доступна через несколько типов сокетов (например, один её адрес — **SOCK_STREAM**, а второй — **SOCK_DGRAM**).

Обычно, приложение должно стараться использовать адреса в том порядке, в котором они получены.

Расширения getaddrinfo() для интернациональных доменных имён

Начиная с glibc 2.3.4, **getaddrinfo()** был расширен для выборочного прозрачного разрешения исходящих и входящих адресов в формате интернациональных доменных имен (IDN, см. RFC 3490, Internationalizing Domain Names in Applications (IDNA)). Было определено четыре новых флага:

AI_IDN

Если указан этот флаг, то, в случае необходимости, имя узла, указанного в **nodename**, будет преобразовано в IDN-формат. Исходной кодировкой будет текущая локаль.

Если имя на входе содержит символы не-ASCII, то будет задействовано кодирование IDN.

Части имени узла (разделенные точками), которые содержат не-ASCII символы, будут закодированы с помощью ASCII Compatible Encoding (ACE) прежде, чем будут переданы функциям преобразования имен.

AI_CANONIDN

При указанном флаге **AI_CANONNAME** после успешного преобразования имени **getaddrinfo()** вернет каноничное имя узла согласно значению структуры **addrinfo**. Возвращаемое значение будет точной копией значения, возвращенного функцией разрешения имени.

Если имя закодировано с помощью ACE, то оно будет содержать префикс **xn--** для каждого из закодированных компонентов имени. Чтобы преобразовать эти компоненты в читаемый вид, вместе с флагом **AI_CANONNAME** следует использовать **AI_CANONIDN**. Итоговая строка будет закодирована при помощи текущей локали.

AI_IDN_ALLOW_UNASSIGNED, AI_IDN_USE_STD3_ASCII_RULES

Установка этих флагов включает **IDNA_ALLOW_UNASSIGNED** (разрешать не назначенные кодовые точки Юникода) и **IDNA_USE_STD3_ASCII_RULES** (проверять вывод на соответствие имени узла STD3) соответственно для возможности работы с IDNA.

Возвращаемое значение

Нулевое возвращаемое значение для **getaddrinfo()** указывает на успешное завершение, ненулевое возвращаемое значение указывает на сбой. Возможные значения ошибок перечислены в разделе Ошибки.

После успешного возврата **getaddrinfo()** местоположение, на которое указывает **res**, будет ссылаться на связанный список структур **addrinfo**, каждая из которых будет указывать адрес сокета и информацию, которую можно использовать при создании сокета, с которым будет использоваться данный адрес. Список будет включать как минимум одну структуру **addrinfo**.

Содержимое поля **ai_flags** в возвращаемых структурах не определено.

Поле **ai_next** каждой структуры содержит указатель на следующую структуру в списке или нулевой указатель, если это последняя структура в списке.

Каждая структура в списке будет включать значения для использования в вызове функции **socket()** и адрес сокета для использования с функцией **connect()** или, если был указан флаг **AI_PASSIVE**, для использования с функцией **bind()**.

Поля **ai_family**, **ai_socktype** и **ai_protocol** можно использовать в качестве аргументов функции **socket()** для создания сокета, подходящего для использования с возвращенным адресом.

Поля **ai_addr** и **ai_addrlen** можно использовать в качестве аргументов функций **connect()** или **bind()** с таким сокетом в соответствии с флагом **AI_PASSIVE**.

Если **nodename** не является нулем, и если при этом указан флаг **AI_CANONNAME**, поле **ai_canonname** первой возвращаемой структуры **addrinfo** будет указывать на строку с завершающим нулем, содержащую каноническое имя, соответствующее исходному значению **nodename.**

Если каноническое имя недоступно, то **ai_canonname** будет ссылаться на аргумент **nodename** или строку с тем же содержимым.

Все поля в структурах адресов сокетов, возвращаемых функцией **getaddrinfo()**, которые не заполнены явным значением аргумента (например, **sin6_flowinfo**[7]), будут установлены в ноль.

Ошибки

Функция **getaddrinfo()** завершится ошибкой и вернет значение, соответствующее ошибке:

EAI_ADDRFAMILY — у указанного сетевого узла нет сетевых адресов в запрашиваемом семействе адресов.

EAI_AGAIN – в настоящее время не удалось разрешить имя. Будущие попытки могут быть успешными.

EAI_BADFLAGS – параметр **flags** имеет недопустимое значение.

EAI_FAIL — при попытке разрешения имени произошла неисправимая ошибка.

EAI_FAMILY — семейство адресов не распознано.

EAI_MEMORY — при попытке выделить память для возвращаемого значения произошла ошибка выделения памяти.

EAI_NONAME — имя не разрешается для предоставленных параметров. Ни имя узла, ни имя сервера не были указаны. Должен быть предоставлен хотя бы один из них.

EAI_SERVICE — переданная служба не была распознана для указанного типа сокета.

EAI_SOCKTYPE — предполагаемый тип сокета не был распознан. Такая ошибка может возникнуть, если **hints.ai_socktype** и **hints.ai_protocol** противоречат друг другу (например, **SOCK_DGRAM** и **IPPROTO_TCP** соответственно).

---

7) В структуре sockaddr_in6 определяет класс IPv6 трафика.

EAI_SYSTEM — произошла системная ошибка и код ошибки можно найти в **`errno`**.

EAI_OVERFLOW — буфер аргументов переполнен.

Функция **`gai_strerror()`** транслирует эти коды ошибок в читаемый формат, подходящий для сообщений об ошибке.

Пример

Использование **getaddrinfo()**, **gai_strerror()**, **freeaddrinfo()** и **getnameinfo(3)**.
Это программы эхо-сервера и клиента UDP-дейтаграмм.

Серверная программа

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>

#define BUF_SIZE 500

int main(int argc, char *argv[]) {

    struct addrinfo  hints;
    struct addrinfo *res, *rp;
    int sfd, s;                          // дескриптор сокета
    struct sockaddr_storage peer_addr; // адрес источника
    socklen_t peer_addr_len;           // и его длина
    ssize_t   nread;
    char buf[BUF_SIZE];
```

```
    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family    = AF_UNSPEC;   // Разрешены IPv4 и IPv6
    hints.ai_socktype  = SOCK_DGRAM;  // Сокет для дейтаграмм
    hints.ai_flags     = AI_PASSIVE;  // Для wildcard IP-адреса
    hints.ai_protocol  = 0;           // Любой протокол
    hints.ai_canonname = NULL;
    hints.ai_addr      = NULL;
    hints.ai_next      = NULL;

    s = getaddrinfo(NULL,    // локалхост
                    argv[1], // порт, на котором будем получать
                    &hints,
                    &res);
    if (s != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
        exit(EXIT_FAILURE);
    }
```

```
// getaddrinfo() возвратил список структур адресов. Делаем проверку каждого
// адреса до успешного bind(2). Если socket(2) или bind(2) терпят неудачу,
// мы закрываем сокет и пробуем следующий.

for (rp = res; rp != NULL; rp = rp->ai_next) {
    sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
    if (sfd == -1) {
        continue;
    }
    if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0) {
        break;                  // Успех
    }
    close(sfd);
}

if (rp == NULL) {               // Нет успешных адресов
    fprintf(stderr, "Could not bind\n");
    exit(EXIT_FAILURE);
}

freeaddrinfo(result);           // Больше не нужен
```

```
    // Читаем дейтаграмму и пересылаем ее назад отправителю
    for (;;) {
        peer_addr_len = sizeof(struct sockaddr_storage); // длина буфера
        nread = recvfrom(sfd,                            // дескриптор сокета
                         buf, BUF_SIZE,                  // буфер
                         0,                              // флаги
                         (struct sockaddr *)&peer_addr,  // адрес источника
                         &peer_addr_len);                // и его длина
        if (nread == -1)
            continue;     // Игнорируем запрос с ошибкой
        char host[NI_MAXHOST], service[NI_MAXSERV];

        s = getnameinfo((struct sockaddr *) &peer_addr, peer_addr_len, host,
                        NI_MAXHOST, service, NI_MAXSERV, NI_NUMERICSERV);
        if (s == 0)
            printf("Получено %zd байт из %s:%s\n", nread, host, service);
        else
            fprintf(stderr, "getnameinfo: %s\n", gai_strerror(s));
        // Отсылаем назад
        if (sendto(sfd, buf, nread, 0, (struct sockaddr *) &peer_addr,
                   peer_addr_len) != nread)
            fprintf(stderr, "Ошибка отправки ответа\n");
    }
} // main()
```

Клиентская программа

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUF_SIZE 500

int main(int argc, char *argv[]) {

    struct addrinfo  hints;
    struct addrinfo *res, *rp;
    int sfd, s, j;
    size_t len;
    ssize_t nread;
    char buf[BUF_SIZE];

    if (argc < 3) {
        fprintf(stderr, "Usage: %s host port msg...\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    // Получаем адрес(а), соответствующие узлу/порту
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family   = AF_UNSPEC;  // Разрешены IPv4 и IPv6
    hints.ai_socktype = SOCK_DGRAM; // Сокет дейтаграммный
    hints.ai_flags    = 0;
    hints.ai_protocol = 0;          // Любой протокол

    s = getaddrinfo(argv[1],  // хост
                    argv[2],  // порт
                    &hints, &result);
    if (s != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
        exit(EXIT_FAILURE);
    }

    // getaddrinfo() возвратил список структур адресов. Делаем проверку каждого
    // адреса до успешного bind(2). Если socket(2) или connect(2) терпят неудачу,
    // мы закрываем сокет и пробуем следующий.

    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
            continue;
        if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
            break;                  // Успех
        close(sfd); // закрываем сокет и пробуем следующий
    }
```

```
    if (rp == NULL) {               // Нет успешных адресов
        fprintf(stderr, "Could not connect\n");
        exit(EXIT_FAILURE);
    }
    freeaddrinfo(result);           /* Больше не нужен */

    // Отправляем оставшиеся аргументы командной строки
    // в виде отдельных дейтаграмм и ждем ответа от сервера
    for (j = 3; j < argc; j++) {
        len = strlen(argv[j]) + 1; // +1 для завершающего null-байта
        if (len > BUF_SIZE) {
            fprintf(stderr, "Игнорируем длинное сообщение в аргументе %d\n", j);
            continue;
        }
        if (write(sfd, argv[j], len) != len) {
            fprintf(stderr, "partial/failed write\n");
            exit(EXIT_FAILURE);
        }
        nread = read(sfd, buf, BUF_SIZE);
        if (nread == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        printf("Получено %zd байт: %s\n", nread, buf);
    }
    exit(EXIT_SUCCESS);
}
```