

Sockets (BSD)



Programação Avançada

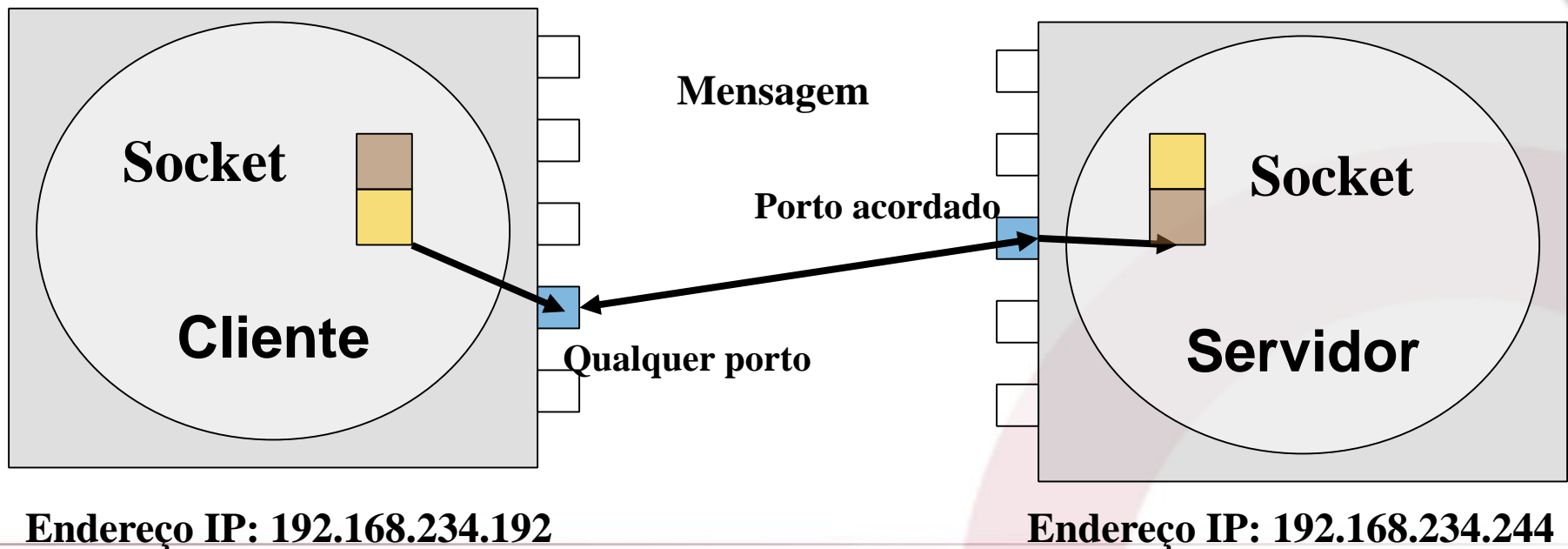
gdb threads tree ponteiro ciclogcc for char *ptr: #include sockets (c) Patricio Domingues doxygen lock/unlock #define malloc IPL++ mutex i++ linked list

Autor: Patricio Domingues

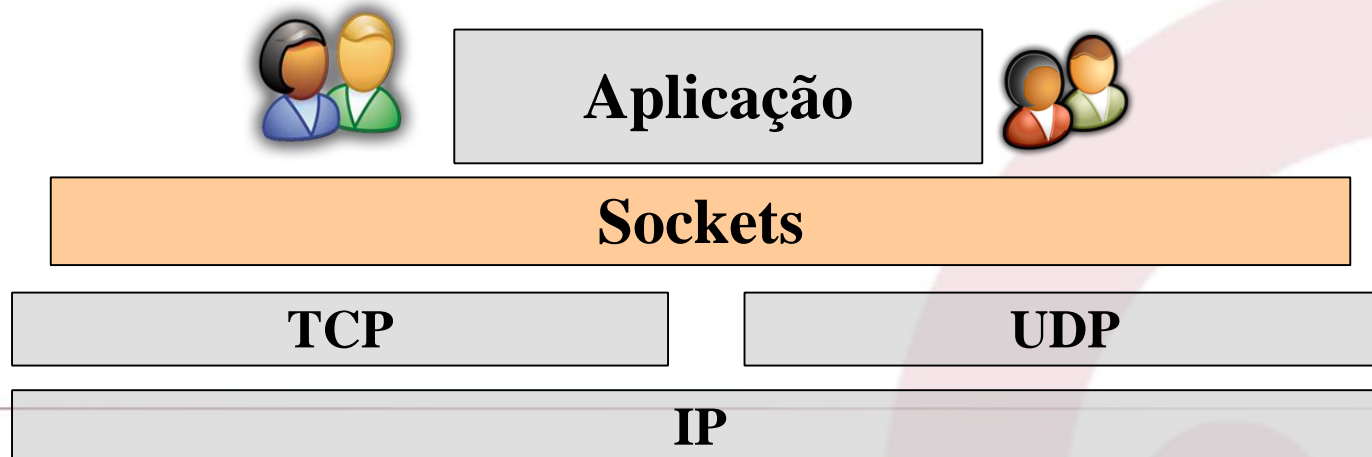
(c) Patricio Domingues, Vitor Carreira

Camada de transporte

- **Nível de processos**
 - Dois processos em máquinas distintas trocam pacotes (segmentos TCP ou datagramas UDP)
 - IP + porto origem ← – protocolo – → IP + porto destino



- APIs de programação da camada de transporte
 - Interface entre a camada da aplicação e a camada de transporte
 - Possibilita o acesso e uso dos serviços da camada de transporte (TCP e UDP) às aplicações
 - Consiste num conjunto de funções com comportamento bem definido (parâmetros, valores de retorno, etc.)





- O TCP/IP não inclui nenhuma definição de *Application Programming Interface* (API)
- Existem várias API para a programação de aplicações sobre a pilha protocolar TCP/IP
 - Sockets (UNIX Sockets, Java Sockets, Winsock)
 - XTI (Evolução TLI)
 - XTI – X/Open Transport Interface
 - TLI – Transport Layer Interface
 - MacTCP (Apple)
 - Obsoleto, substituída pela API OpenTransport (1995)
 - OpenTransport (Apple)
 - Obsoleto, substituído pela API BSD

Interface Socket BSD (1)

- Socket
 - Encaixe (tradução)
- No que respeita à comunicação distribuída
 - Socket
 - Mecanismo de comunicação entre processos sejam eles do mesmo sistema (comunicação local) ou de diferentes sistemas (comunicação remota)
 - Canal de comunicação bidireccional entre dois ou mais processos
 - Encaixa no nível 4 do modelo OSI da ISO (camada de transporte)
- Interface Socket BSD (Berkeley Software Distribution)
 - API *standard* para programar a pilha protocolar TCP/IP
 - Vantagens da interface Socket BSD
 - Portabilidade das aplicações
 - Simplificação do desenvolvimento de aplicações



Wall socket



- Genérica
 - Suporte para múltiplas famílias de protocolos (TCP, UDP, etc.; IPv4, IPV6, etc.)
 - Representação genérica de endereços (*struct sockaddr*)
- Promove a utilização da interface de programação de Entrada/Saída (I/O)
 - Funções
 - write
 - read
 - close
 - ...

- Uma ligação socket é plenamente descrita através de 5 parâmetros
 - Protocolo
 - Endereço local
 - Porto local
 - Endereço remoto
 - Porto remoto
- Exemplo
(TCP, 192.168.234.21,4500,192.168.234.7,22)



■ Porto

- Identificador numérico (inteiro) com 16 bits
 - 16 bits → existem 2^{16} (65536) portos diferentes
- Deste modo, uma aplicação é identificada pelo:
 - endereço IP
 - porto (1 a 65535)
- Um computador pode manter várias ligações em simultâneo com outro computador
 - IP + porto

COMMON PORTS packetlife.net

TCP/UDP Port Numbers			
7 Echo	554 RTP	2745 Apple II	6891-6901 Windows Live
19 Chargen	546-547 DHCPv6	2957 Symantec AV	6970 QuickTime
20-21 FTP	560 monitor	3050 Interbase DB	7212 GhostSurf
22 SSH/SCP	563 SMTP over SSL	3074 XBOX Live	7648-7649 CU-SeeMe
23 Telnet	587 SMTP	3124 HTTP Proxy	8000 Internet Radio
25 SMTP	591 FileMaker	3127 MySQL	8080 HTTP Proxy
42 WINS Replication	593 Microsoft DCOM	3128 HTTP Proxy	8086-8087 Kaspersky AV
43 WINS	631 Internet Printing	3222 GUP	8118 Privacy
49 TACACS	636 SSH over SSL	3260 iSCSI Target	8200 VMware Server
53 DNS	639 MSDP (PM)	3306 MySQL	8500 Adobe ColdFusion
67-68 DHCP/BOOTP	646 LDP (MPLS)	3389 Terminal Server	8767 TeamSpeak
69 TFTP	691 MS Exchange	3689 iTunes	8866 Apple II
70 Gopher	860 iSCSI	3690 Subversion	9100 HP JetDirect
79 Finger	873 rsync	3724 World of Warcraft	9101-9103 Bacula
80 HTTP	902 VMware Server	3784-3785 Veritas	9119 RDP
88 Kerberos	989-990 FTP over SSL	4333 mSQL	9800 WebDAV
102 MS Exchange	993 IMAP over SSL	4444 Master	9898 Outlook
110 POP3	995 POP3 over SSL	4664 Google Desktop	9988 RemoteApp
113 Ident	1025 Microsoft RPC	4672 iMule	9999 Urchin
119 NNTP (Usenet)	1026-1029 Windows Messenger	4899 Radmin	10000 Webmin
123 NTP	1080 SOCKS Proxy	5000 UPnP	10000 BackupExec
135 Microsoft RPC	1080 Syslog	5001 Sslbox	10113-10116 NetIQ
137-139 NetBIOS	1194 OpenVPN	5001 iperf	11371 OpenPGP
143 IMAP	1214 Kazaa	5004-5005 RTP	12035-12036 Second Life
144 IMAP	1241 Nessus	5050 Yahoo! Messenger	12345 NetBus
161-162 SNMP	1311 Dell OpenManage	5060 SIP	13720-13721 NetBackup
177 XDMCP	1337 iKSEE	5190 iVNC	14567 NetWitness
179 BGP	1433-1434 Microsoft SQL	5222-5223 Jitsi/Jabber	15118 Symantec DiskMon
201 AppleTalk	1512 WINS	5432 PostgreSQL	19226 AdminSecure
264 BGMF	1589 Cisco VQP	5500 VNC Server	19638 Ensim
318 TSP	1701 L2TP	5554 Ssher	20000 Usermin
381-383 HP Openview	1723 MS PFTP	5631-5632 pcAnywhere	24800 Synergy
389 LDAP	1725 iVNC	5800 VNC over HTTP	25999 Xfile
411-412 Direct Connect	1741 CiscoWorks 2000	5999+ VNC Server	27015 RealLife
443 HTTP over SSL	1755 MS Media Server	6000-6001 X11	27374 Sub7
445 Microsoft DS	1812-1813 RADIUS	6112 Battle.net	28960 Call of Duty
464 Kerberos	1863 MSN	6129 DameWare	31337 Back Office
465 SMTP over SSL	1985 Cisco HSRP	6257 WinMX	33434+ Traceroute
497 Retrospect	2000 Cisco SCCP	6346-6347 Gnutella	
500 ISAKMP	2002 Cisco ACS	6566 SANE	
512 ronc	2049 NFS	6588 Analogt	
513 rlogin	2082-2083 cPanel	6665-6669 IRC	
514 syslog	2100 Oracle XDB	6679/6697 IRC over SSL	
515 LPD/LPR	2222 DirectAdmin	6881-6899 BitTorrent	
520 RIP	2302 iSCSI		
521 RIPng (IPv6)	2453-2464 Oracle DB		
540 UUCP			

IANA port assignments published at <http://www.iana.org/assignments/port-numbers>

by Jeremy Stretch v1.1

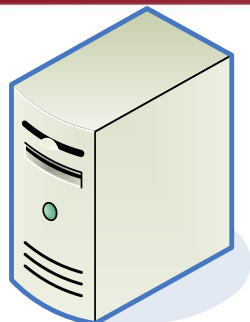
■ Analogia

- “Porto” é como uma extensão de uma central telefónica
 - IP é como o número de telefone central

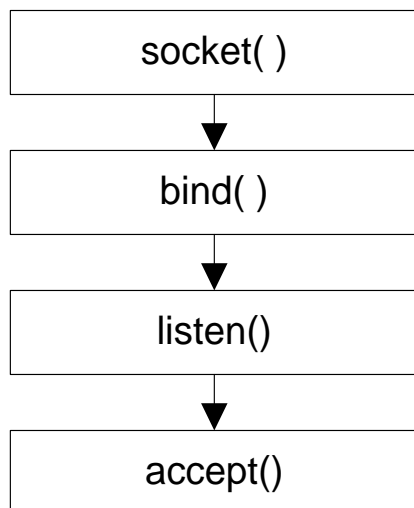
Tipos de *socket*

- A API socket define três tipos de sockets
 1. socket *stream*
 - Interface para o protocolo de transporte TCP
 2. socket *datagram*
 - Interface para o protocolo de transporte UDP
 3. socket *raw*
 - Interface para o protocolo de rede IP
 - Empregues para a construção de pacotes da camada IP
 - Exemplo: wireshark, nmap, etc.
 - Em sistemas Unix e Windows a criação de um socket do tipo *raw* requer privilégios de administração
 - Em sistemas MacOS X e FreeBSD deve ser empregue uma biblioteca (e.g., libpcap)

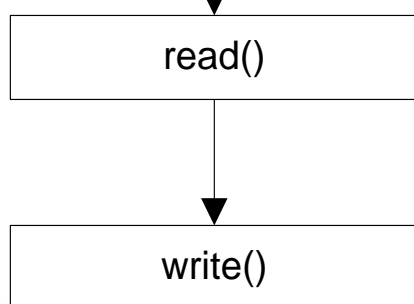
Funções API Socket: TCP



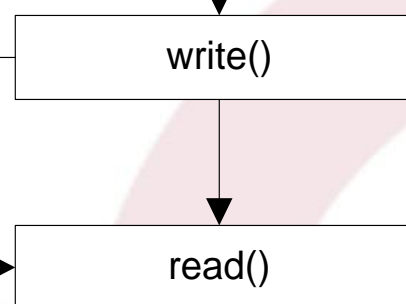
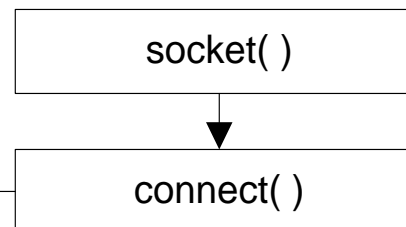
Servidor TCP



ligação estabelecida



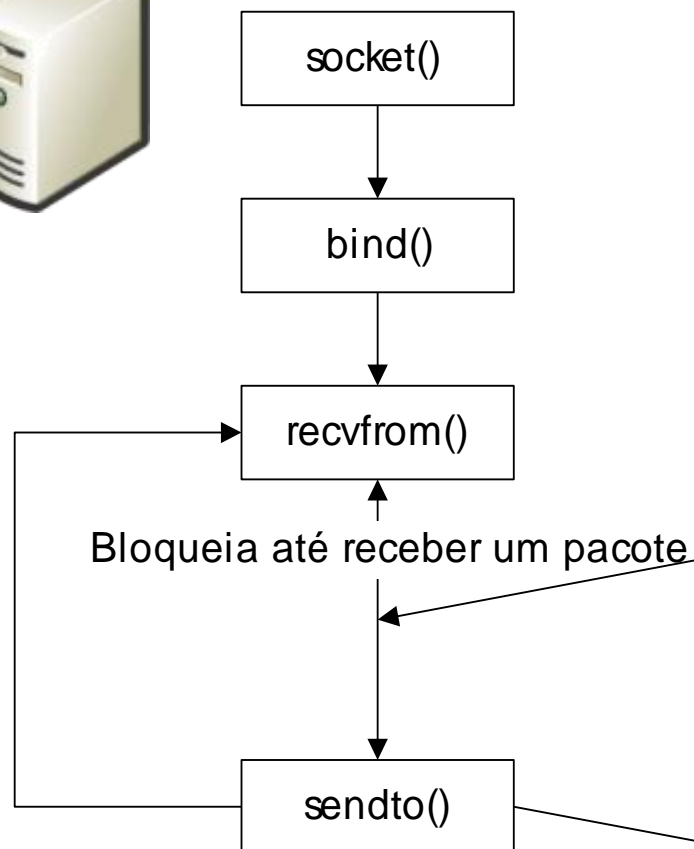
Cliente TCP



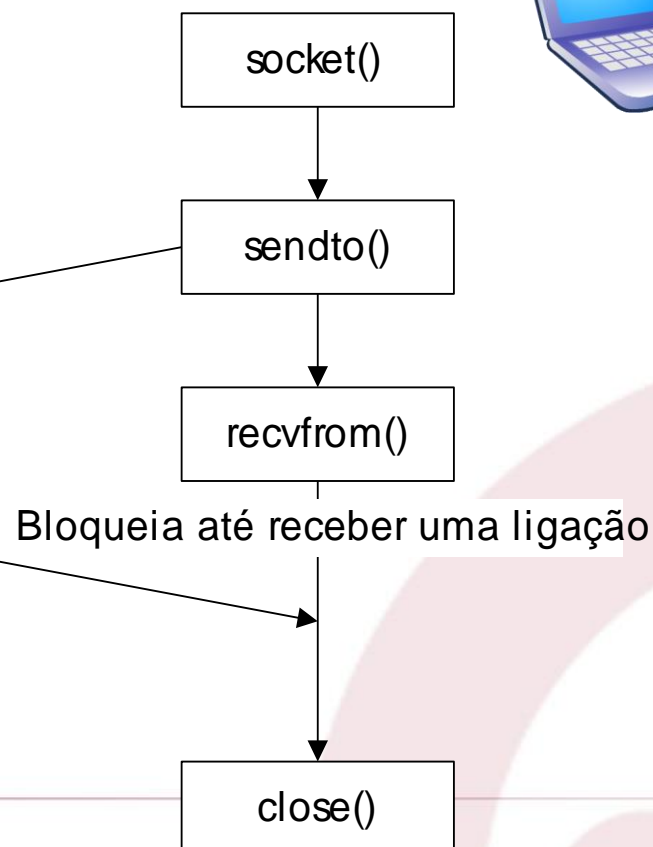
Funções API Socket: UDP



UDP Server



UDP Client



Funções da API Socket: socket (1)

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

- Criação de um descritor de socket
- Apenas envolve a tabela de descritores do processo da máquina local
- Parâmetros:
 - Domain
 - Define o domínio de comunicação (AF_INET, etc)
 - Type
 - Define o tipo de comunicação (Stream, Datagram, etc)
 - Protocol
 - Define o protocolo específico a utilizar
 - » Na maioria das situações, apenas existe um protocolo por tipo de comunicação
 - » Normalmente passa-se o valor 0 (zero)



stdin
stdout
stderr
x
Socket 1
(...)

**Tabela
descritores**

- `int socket(int domain, int type, int protocol);`
 - Domínios mais comuns
 - `AF_UNIX/AF_LOCAL`
 - Socket local
 - `AF_INET/AF_INET6`
 - Socket “Internetwork” IPv4/IPv6
 - Outros (depende da implementação)
 - `AF_ISO`
 - Socket ISO (OSI)
 - `AF_NS`
 - Socket XNS (*Xerox Network System*)
 - `AF_PACKET, AF_IMPLNK`
 - Socket baixo nível
 - ...

- `int socket(int domain, int type, int protocol);`
 - Domínios (duas notações para as constantes)
 - AF_xxx vs PF_xxx
 - O prefixo AF é o acrónimo de “Address Family”
 - O prefixo PF é o acrónimo de “Protocol Family”
 - AF_UNIX, AF_LOCAL Local communication
 - AF_INET IPv4 Internet protocols
 - AF_INET6 IPv6 Internet protocols
 - AF_IPX IPX - Novell protocols
 - AF_NETLINK Kernel user interface device
 - AF_X25 ITU-T X.25 / ISO-8208 protocol
 - AF_AX25 Amateur radio AX.25 protocol
 - AF_ATMPVC Access to raw ATM PVCs
 - AF_APPLETALK Appletalk
 - AF_PACKET Low level packet interface
 - **Nota:** consultar a página de manual: `man socket`

- `int socket(int domain, int type, int protocol);`
 - Tipo de comunicação a utilizar
 - `SOCK_STREAM`
 - Comunicação bidireccional (estabelece ligação, é fiável e sequencial)
 - Exemplo: TCP (em `AF_INET` ou `AF_INET6`)
 - `SOCK_DGRAM`
 - Comunicação por datagrams (não estabelece ligação e não é fiável)
 - Exemplo: UDP (em `AF_INET` ou `AF_INET6`)
 - `SOCK_RAW`
 - Acesso à camada de rede (socket baixo nível)
 - `SOCK_SEQPACKET`
 - Comunicação bidireccional por datagramas com estabelecimento de ligação (fiável e sequencial)
 - » Não está implementado para o domínio `AF_INET`
 - » Empregue para protocolos X.25 e AX.25

- `int socket(int domain, int type, int protocol);`
 - Protocolo a utilizar (usualmente a combinação <domínio, tipo> define o protocolo)
 - `(AF_INET, SOCK_STREAM)` => TCP
 - `(AF_INET, SOCK_DGRAM)` => UDP

- Exemplo:

```
int serverfd;  
  
if ( (serverfd=socket(AF_INET, SOCK_STREAM, 0))== -1 ) {  
    ERROR(-1, "Nao criou o socket");  
}
```

Socket *raw* >>

- `int socket(int domain, int type, int protocol);`
 - Criação de um socket *raw*
 - Lembrete – *socket raw*
 - Interação com a camada IP
 - type: SOCK_RAW
 - protocol: IPPROTO_RAW

- Exemplo:

```
int socket_fd;  
  
if ( (socket_fd=socket(AF_INET, SOCK_RAW, IPPROTO_RAW))== -1 ){  
    ERROR(-1, "Nao criou o socket");  
}
```

Funções da API Socket: bind(1)

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

- Registo do socket no sistema
 - Associa um endereço ao socket
 - endereço IP local + porto local
 - Os pacotes que o sistema recebe no porto especificado são associados ao processo que registou o socket (processo que efectuou o *bind*)
- Parâmetros
 - **sockfd**
 - Descritor do socket a registar no sistema
 - **my_addr**
 - Endereço a associar ao socket (contém o porto)
 - **addrlen**
 - Tamanho do endereço

Funções da API Socket: bind(2)

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

- Estrutura genérica de um endereço

```
struct sockaddr {  
    uint8_t      sa_len;  
    sa_family_t  sa_family;  
    char         sa_data[14];  
};
```

- Porquê uma estrutura genérica para o endereço?
 - Existem várias famílias de endereços (AF_INET, AF_INET6, AF_UNIX, ...)
 - Algumas funções da API sockets recebem como parâmetros endereços
 - A API sockets existe antes do ANSI C.
- Solução
 - as funções recebem um ponteiro para uma estrutura genérica de endereços
 - O campo sa_family é preenchido com o código correspondente à família de endereço efetivamente colocado na estrutura



Funções da API Socket: representação de endereços (1)



IPv4

`sockaddr_in{}`

length	AF_INET
16-bit port#	
32-bit IPv4 address	
Unused	

fixed length (16 bytes)

IPv6

`sockaddr_in6{}`

length	AF_INET6
16-bit port#	
32-bit flow label	
128-bit IPv6 address	

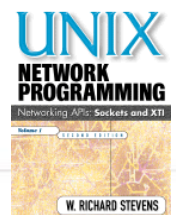
fixed length (24 bytes)

Unix

`sockaddr_un{}`

length	AF_LOCAL
pathname (up to 104 bytes)	

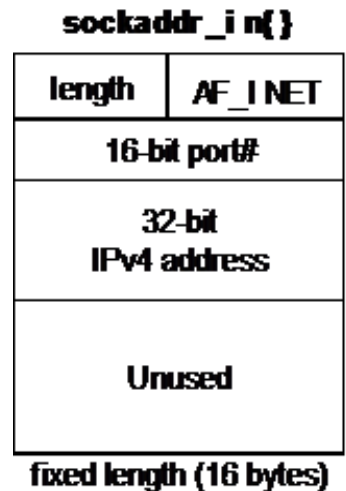
variable length





- Para cada família de endereços existe uma estrutura específica

- Para os endereços IPv4 (família AF_INET) existe a seguinte estrutura



```
struct sockaddr_in {  
    uint8_t    sin_len;  
    sa_family_t sin_family;  
    in_port_t   sin_port; // número do porto  
    struct in_addr sin_addr; // endereço IP 32 bits  
    char sin_zero[8]; // Padding  
};
```



- Um endereço IP da família AF_INET é representado pela seguinte estrutura

```
struct in_addr {  
    in_addr_t s_addr;    // endereço IP (32 bits)  
};
```

- Porquê uma estrutura?

*The reason the **sin_addr** member is a structure, and not just an **in_addr_t**, is historical. Earlier releases (4.2BSD) defined the **in_addr** structure as a union of various structures, to allow access to each of the 4 bytes and to both of the 16-bit values contained within the 32-bit IPv4 address. This was used with class A, B, and C addresses to fetch the appropriate bytes of the address. But with the advent of subnetting and then the disappearance of the various address classes with classless addressing, the need for the union disappeared. Most systems today have done away with the union and just define **in_addr** as a structure with a single **in_addr_t** member. **Fonte:** Addison Wesley : UNIX Network Programming Volume 1, Third Edition:*

IPv6

sockaddr_in6{}

length	AF_INET6
16-bit port#	
32-bit flow label	
128-bit IPv6 address	

fixed length (24 bytes)

- Endereços IPV6 (família AF_INET6 ou PF_INET6)

```
struct sockaddr_in6 {
    sa_family_t sin6_family;    /* AF_INET6 */
    in_port_t sin6_port;        /* porto */
    uint32_t sin6_flowinfo;     /* info fluxo */
    struct in6_addr sin6_addr;  /* endereço IPv6 */
    uint32_t sin6_scope_id;     /* conjunto
                                interfaces (escopo) */
};
```

```
struct in6_addr {
    uint8_t s6_addr[16];        /* endereço IPv6 */
};
```



- O valor de endereços e portos devem estar no formato de rede
 - Os campos **sin_port** e **sin_addr** devem ser especificados no formato de rede
 - Formato de rede = BIG endian
 - RFC 1700, 1994 (<http://tools.ietf.org/html/rfc1700>)
 - «The convention in the documentation of Internet Protocols is to express numbers in decimal and to picture data in "big-endian" order. That is, fields are described left to right, with the most significant octet on the left and the least significant octet on the right.»
 - **Erro frequente**
 - Esquecer a conversão para o formato de rede (e vice-versa)





■ Funções de conversão (tradicionais)

```
uint16_t htons(uint16_t) ;
```

```
uint16_t ntohs(uint16_t) ;
```

```
uint32_t htonl(uint32_t) ;
```

```
uint32_t ntohl(uint32_t) ;
```

– Legenda:

- **n** – ordem da rede (**n**etwork)
- **h** – ordem do “hospedeiro” (**h**ost)
- **s** – inteiro de 16 bits (**s**hort)
- **l** – inteiro de 32 bits (**l**ong)



■ Funções de conversão disponíveis na glibc

- Não fazem parte da norma da linguagem C
- Requerem a existência da constante préprocessador `_DEFAULT_SOURCE`

`#include <endian.h>`

```
uint16_t htobe16(uint16_t host_16bits);  
uint16_t htole16(uint16_t host_16bits);  
uint16_t be16toh(uint16_t big_endian_16bits);  
uint16_t le16toh(uint16_t little_endian_16bits);  
uint32_t htobe32(uint32_t host_32bits);  
uint32_t htole32(uint32_t host_32bits);  
uint32_t be32toh(uint32_t big_endian_32bits);  
uint32_t le32toh(uint32_t little_endian_32bits);  
uint64_t htobe64(uint64_t host_64bits);  
uint64_t htole64(uint64_t host_64bits);  
uint64_t be64toh(uint64_t big_endian_64bits);  
uint64_t le64toh(uint64_t little_endian_64bits);
```

Macros de *byteswap*

- Macros de troca de blocos de octetos
 - Devolvem os octetos do parâmetro **X** por ordem invertida

```
#include <byteswap.h>

bswap_16(x);
bswap_32(x);
bswap_64(x);
```

```
#include <stdio.h>
#include <byteswap.h>
#include <stdint.h>

int main(void)
{
    uint32_t u32 = 0x11223344;
    printf("u32=0x%X=> 0x%X\n",
           u32, bswap_32(u32));
    return 0;
}
```



u32=0x11223344 ==> 0x44332211



- Um endereço IPv4 é representado por um inteiro de 32 bits. Como especificar este valor?

int **inet_aton**(char *in, struct in_addr *out);

- Converte uma string no formato **ASCII-dotted-decimal** ("192.168.234.243") para um endereço IPv4 no formato de rede
- Retorno:
 - 0 Erro; <> 0 OK

char* **inet_ntoa**(struct in_addr *in);

- Converte um endereço IPv4 especificado no formato de rede para uma string no formato ASCII-dotted-decimal
- Retorna endereço de string no formato *dotted decimal*
- Função simétrica da função **inet_aton**

Perigos do *inet_ntoa* >>

- `char* inet_ntoa(struct in_addr *in);`
 - Converte um endereço IPv4 especificado no formato de rede para uma string no formato ASCII-dotted-decimal
- Mas de onde vêm o espaço em memória empregue para devolver a string?
 - O que diz o manual? (`man inet_ntoa`)
 - “The string is returned in a **statically** allocated buffer, which subsequent calls will overwrite”
 - A função não é reentrante, nem thread-safe!
- Versão reentrante

```
char* inet_ntoa_r(struct in_addr in, char *buf, socklen_t size);
```

`inet_ntoa` / `inet_aton`: uso desaconselhado



- Então e os endereços IPv6?
 - funções **inet_pton** e **inet_ntop** (IPv4 e IPv6)
 - As funções são reentrantes
- `int inet_pton(int family, const char *src, void *dst);`
 - Converte uma string no formato ASCII-dotted-decimal (“192.168.234.243” ou “0:0:0:0:0:0:0:1”) para endereço no formato de rede
 - A família de endereços (AF_INET, AF_INET6) é especificada pelo parâmetro **family**
- `char* inet_ntop(int family, const void *src, char *dst, size_t cnt);`
 - Converte um endereço especificado no formato de rede para uma string no formato ASCII-dotted-decimal
 - Função *inversa* da função **inet_pton**

Sobre endereços IPv6... (1)

- Um endereço IPv6
 - tem 128 bits
 - É escrito com 8 blocos, cada bloco tem 16 bits
 - Cada bloco é representado em HEX com valor entre 0 e 0xFFFF (os zeros à esquerda podem ser omitidos)
 - É permitido que os 4 bytes menos significativos sejam um endereço IPv4 em formato doted-decimal
 - Exemplo 1:2:3:4:5:6:192.193.194.195


Continua >>

- Endereço IPv6
 - Permitido o uso da representação `::` (*double colon*) para representar um ou mais blocos com zeros
 - Exemplo o `1:2::7:8` → `1:2:0:0:0:0:7:8`
 - Apenas é permitido a existência de um `::`, caso contrário o endereço IPv6 não é considerado válido
 - Exemplo: `::1::` → errado
 - A representação `::` não pode aparecer no endereço IPv4 (4 bytes menos significativos)
 - Endereço de *loopback* (domínio *localhost*)
 - IPv4=127.0.0.1, IPv6=::1

Funções da API Socket: bind(4)

- Determinação do par (endereço, porto) na chamada do **bind**
- **INADDR_ANY** (ou **in6addr_any** no IPv6)
 - Constante que permite que o socket seja registado no porto local para todas as interfaces IP que a máquina possa ter
 - um computador pode ter várias interfaces IP (e.g., várias placas de rede, etc.)

INADDR_ANY



Process specifies		Result
IP address	port	
wildcard	0	kernel chooses IP address and port
wildcard	nonzero	kernel chooses IP address, process specifies port
local IP address	0	process specifies IP address, kernel chooses port
local IP address	nonzero	process specifies IP address and port

Figure 4.5 Result when specifying IP address and/or port number to bind.

- Preenchimento da estrutura de endereço para bind

```
#define PORTO 6080
struct sockaddr_in server_addr;
int serverfd;
...
/* inicia a estrutura com zeros */
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(PORTO);

if (bind(serverfd,
        (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1){
    ERROR(1, "Bind ao porto");
}
```

Funções da API Socket: connect(1)

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



- Chamada no cliente TCP
- Estabelece uma ligação com o servidor TCP especificado em *servaddr*
 - Parâmetros:
 - sockfd
 - Descritor do socket do cliente
 - servaddr
 - Endereço do servidor (endereço IP + porto)
 - addrlen
 - Tamanho do endereço
 - Retorno:
 - 0 OK; -1 Erro

■ Exemplo

```
#define SERVIDOR_STR "192.168.234.244"

#define PORTO 6080

struct sockaddr_in server_addr;

int clientfd;

...

/* inicia com zeros */
memset(&server_addr, 0, sizeof(server_addr));

server_addr.sin_family = AF_INET;

if (inet_pton(AF_INET, SERVIDOR_STR, &server_addr.sin_addr) <= 0){
    ERROR(1, "Invalid address");
}

server_addr.sin_port = htons(PORTO);

if (connect(clientfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1){
    ERROR(2, "Cannot connect to the server");
}
```

Funções da API Socket: listen (#1)

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

- Função empregue por um servidor TCP para assinalar que está pronto a aceitar ligações
- Permite também definir o tamanho da fila de espera (*backlog*)

- **sockfd** – identificador do socket
- **backlog** - tamanho máximo da fila de espera de pedido de ligações PENDENTES
 - Caso a fila de espera de pedidos de ligações esteja cheia, o servidor não envia nada para o cliente
 - Cliente assume que o pacote se perdeu, e volta a reenviar pedido de ligação
 - O máximo de ligações pendentes é definido pela constante **SOMAXCONN** (<sys/socket.h>)

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

- Retorno:

- 0 OK; -1 Erro

```
if (listen(serverfd, 5) == -1){  
    ERROR(1, "Nao foi possivel efectuar o listen");  
}
```

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

- O que sucede se a função **listen** é chamada com um socket que não foi registado? (i.e., não foi efetuado o **bind** ao socket)
 - O sistema efetua um *bind* automático
 - Interface **INADDR_ANY** (IPv4) ou **in6addr_any** (IPv6)
 - Porto é selecionado automaticamente
- Como determinar o porto que foi atribuído?
 - ```
int getsockname(int sockfd,
 struct sockaddr *addr,
 socklen_t *addrlen);
```

# Funções da API Socket: accept(#1)

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

- Função utilizada por um servidor TCP para aguardar um pedido de ligação
  - Fica bloqueada até que haja um pedido de ligação
  - Cria um novo socket quando ocorre um pedido de ligação
- Parâmetros:
  - sockfd
    - Descritor do socket do servidor
  - cliaddr
    - Endereço do cliente com o qual acabou de estabelecer uma ligação (endereço IP + porto)
  - addrlen
    - Tamanho do endereço. É necessário passar um ponteiro (passagem por referência)
- Retorno:
  - Descritor do socket do cliente (> 0) OK;
  - -1 Erro



- Exemplo

```
struct sockaddr_in client_addr;
int serversock, clientsock;
int len = sizeof(client_addr);
...
clientsock = accept(serversock, (struct sockaddr *)&client_addr, &len);
if(clientsock == -1){
 ERROR(1, "Não conseguiu aceitar a ligação");
}
```



socket  
ligado

socket de  
escuta

## ■ Exemplo

```
struct sockaddr_in client_addr;

int serversock, clientsock;

int len = sizeof(client_addr);

...

clientsock = accept(serversock, (struct sockaddr *)&client_addr, &len);

if(clientsock == -1){
 ERROR(1, "Não conseguiu aceitar a ligação");
}
```

- NOTA: o novo *socket* ***clientsock*** (socket ligado) usa o mesmo porto TCP local que está a ser empregue como porto de escuta pelo socket ***serversock*** (socket de escuta)

# Funções da API Socket: accept(3)

```
struct sockaddr_in client_addr;
int serverfd, clientfd;
int len = sizeof(client_addr);
/* Para ser mais correcto, a lista de
endereços proibidos deveria ser
construída dinamicamente a partir de
um ficheiro */
```

```
char * proibidos = {"192.168.5.", "192.168.234.22", 0};
```

```
...
```

```
if ((clientfd = accept(serverfd, (struct sockaddr *)&client_addr, &len)) == -1)
 ERROR(1, "Não conseguiu aceitar a ligação");
```

```
if (eProibido (proibidos, inet_ntoa(client_addr.sin_addr)) {
 printf("Ligação recusada ao IP: %\n", inet_ntoa(client_addr.sin_addr));
 close(clientfd);
} ...
```

```
int eProibido(char * lista[], char *ip_cliente) {
 int i = 0;
 while (lista[i])
 if (strstr(lista[i++], ip_cliente) != NULL)
 return 1;
 return 0;
}
```

**O mais correto será utilizar a função  
inet\_ntop (mais genérica, suporta IPV6)**



## ■ Funções orientadas à ligação

```
ssize_t read(int fd, void* buffer, size_t nbytes);
```

- Lê do descritor *fd*, *nbytes* para a zona de memória apontada por *buffer*
- Retorno:
  - OK: número de bytes lidos
  - Erro: -1

```
ssize_t write(int fd, void* buffer, size_t nbytes);
```

- Escreve *nbytes* do conteúdo da zona de memória apontada por *buffer* para o descritor *fd*
- Retorno:
  - OK: número de bytes escritos
  - Erro: -1

## ■ Funções orientadas à ligação

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

- Lê do socket *s* *len* bytes para a zona de memória apontada por *buf*
- Retorno:
  - OK: número de bytes recebidos;
  - 0: ligação fechada do outro lado
  - Erro: *-1*

```
ssize_t send(int s, const void* msg, size_t len, int flags);
```

- Escreve *len* bytes do conteúdo da zona de memória apontada por *msg* para o socket *s*
- Retorno:
  - OK: número de bytes enviados;
  - Erro: *-1*

## ■ Funções não orientadas à ligação

```
ssize_t recvfrom(int s, const void* buf, size_t len,
int flags, struct sockaddr* from, socklen_t* fromlen);
```

- Lê do socket *s* *len* bytes para a zona de memória apontada por *buf*
- Se *from*  $\neq$  NULL e *s* for um socket não orientado à ligação então:
  - *from* é preenchido com o endereço origem da mensagem
  - *fromlen* contém o tamanho do endereço
- Retorno
  - OK: número de bytes recebidos
  - Erro: -1

## ■ Funções não orientadas à ligação

```
ssize_t sendto (int s, const void* msg, size_t len,
int flags, const struct sockaddr* to, socklen_t tolen);
```

- Escreve *len* bytes do conteúdo da zona de memória apontada por *msg* para o socket *s*
- Se *to*  $\neq$  **NULL** e *s* for um socket não orientado à ligação então a mensagem é enviada para o destinatário com o endereço *to*
- Retorno
  - OK: número de bytes enviados
  - Erro: **-1**



```
ssize_t recv(int s, const void* buf, size_t len, int flags);
ssize_t recvfrom(int s, const void* buf, size_t len,
 int flags, struct sockaddr* from, socklen_t* fromlen);
```

## ■ Flags

- 0 – Nenhuma opção
- MSG\_OOB – mensagem urgentes (mensagens “out of band” apenas para SOCK\_STREAM)
- MSG\_PEEK – acesso aos dados sem que sejam retirados do socket
- MSG\_NOSIGNAL – desactiva “sigpipe” quando a outra extremidade comunicante desaparece
- MSG\_WAITALL – bloqueia até a mensagem ser totalmente recebida

...





```
ssize_t send(int s, const void* msg, size_t len, int flags);
ssize_t sendto (int s, const void* msg, size_t len, int flags,
 const struct sockaddr* to, socklen_t tolen);
```

## ■ Flags

- 0 – Nenhuma opção
- MSG\_OOB – mensagem urgentes (mensagens “out of band” apenas para SOCK\_STREAM)
- MSG\_DONTWAIT – escrita não-bloqueante
- MSG\_NOSIGNAL – desactiva “sigpipe” quando a outra extremidade comunicante desaparece

....

```
int close(int fd);
```



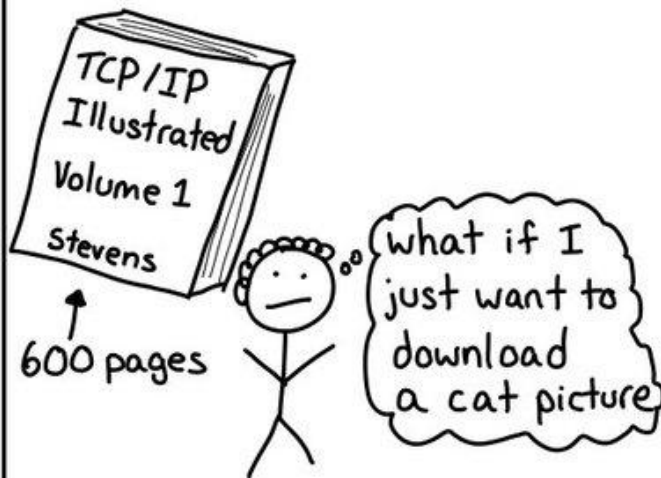
- Fecha o descritor *fd*
  - Liberta os recursos ocupados pelo descritor *fd*
- O socket deve ser fechado quando deixa de ser preciso!
  - No que respeita ao S.O., um socket é um descritor que ocupa uma posição na tabela de descritores abertos do processo
  - Cada processo tem um limite para o número de descritores abertos
    - Por exemplo, 1024
- NOTA
  - Quando não é explicitamente fechado, no término do processo o descritor é libertado

JULIA EVANS  
@b0rk

# sockets

drawings.jvns.ca

networking protocols  
are complicated



Unix systems have  
an API called the  
"socket API" that  
makes it easier to make  
network connections  
(Windows too! ☺)



Unix

you don't need to  
know how TCP works,  
I'll take care of it!

here's what getting  
a cat picture with the  
socket API looks like:

① Create a socket

```
fd = socket(AF_INET, SOCK_STREAM ...
```

② Connect to an IP/port  
`connect(fd, 12.13.14.15:80)`

③ Make a request

```
write(fd, "GET /cat.png HTTP/1.1 ...
```

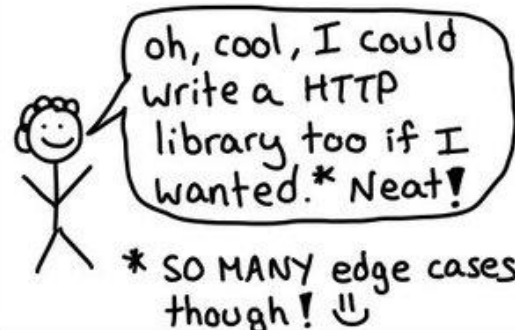
④ Read the response

```
cat-picture = read(fd ...
```

Every HTTP library uses  
sockets under the hood

\$ curl awesome.com  
Python: requests.get("yay.us")

SOCKETS



AF\_INET?  
What's that?

AF\_INET means basically  
"internet socket": it lets you  
connect to other computers  
on the internet using their  
IP address.

The main alternative is  
AF\_UNIX ("unix domain socket")  
for connecting to programs  
on the same computer

3 kinds of internet  
(AF\_INET) sockets:

SOCK\_STREAM = TCP

curl uses this

SOCK\_DGRAM = UDP

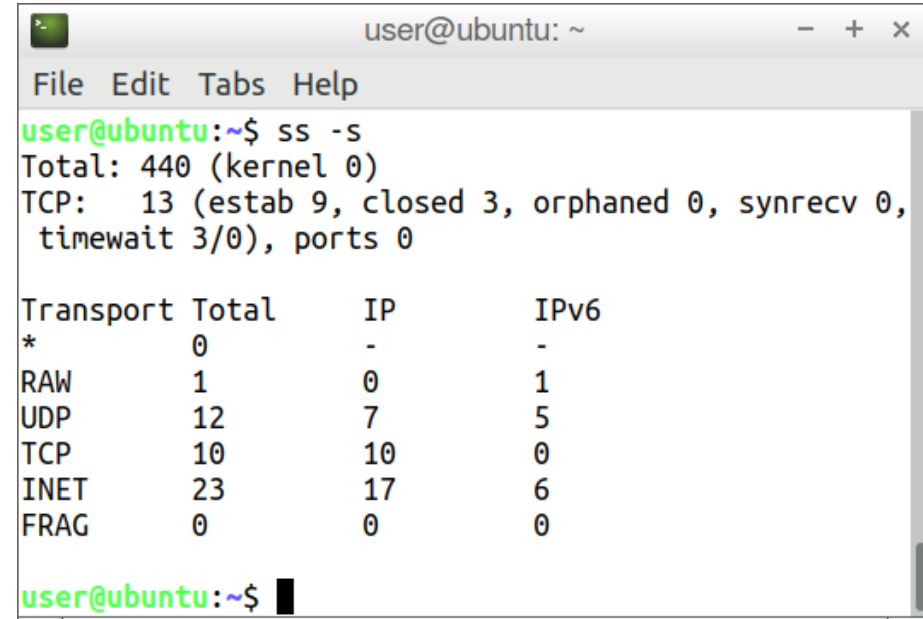
dig (DNS) uses this

SOCK\_RAW = just let me  
send IP packets  
I will implement  
my own protocol

ping uses  
this

# Utilitário ss (linux)

- ss: informação sobre sockets
- `SS -S`
  - Estatísticas
- `ss -l`
  - Lista portos à escuta
- `ss -t -a`
  - Todos os sockets TCP
- `ss -u -a`
  - Todos os sockets UDP

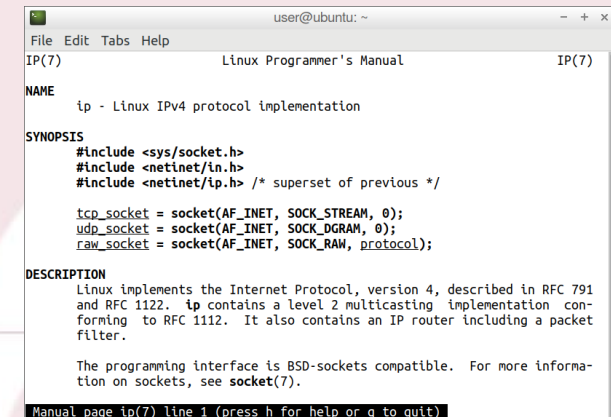
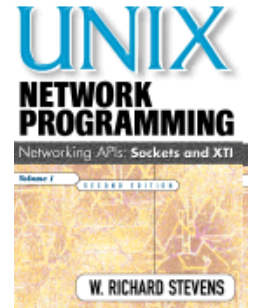


```
user@ubuntu: ~
File Edit Tabs Help
user@ubuntu:~$ ss -s
Total: 440 (kernel 0)
TCP: 13 (estab 9, closed 3, orphaned 0, synrecv 0,
timewait 3/0), ports 0

Transport Total IP IPv6
* 0 - -
RAW 1 0 1
UDP 12 7 5
TCP 10 10 0
INET 23 17 6
FRAG 0 0 0

user@ubuntu:~$
```

- “Beej's Guide to Network Programming - Using Internet Sockets”, Brian “Beej Jorgensen” Hall, 2016  
(<http://beej.us/guide/bgnet/>)
- “UNIX Network Programming”, Volume 1, Second Edition: Networking APIs: Sockets and XTI, Prentice Hall, 1998, ISBN 0-13-490012-X.  
(<http://www.kohala.com/start/unpv12e.html>)
- “Basic Socket Interface Extensions for IPv6”, RFC 3493, Fev. 2003 (<https://www.ietf.org/rfc/rfc3493.txt>)
- man 7 ip
- man 7 ipv6
- man 7 socket



```
user@ubuntu: ~
File Edit Tabs Help
IP(7) Linux Programmer's Manual IP(7)

NAME
 ip - Linux IPv4 protocol implementation

SYNOPSIS
 #include <sys/socket.h>
 #include <netinet/in.h>
 #include <netinet/ip.h> /* superset of previous */

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

DESCRIPTION
 Linux implements the Internet Protocol, version 4, described in RFC 791 and RFC 1122. ip contains a level 2 multicasting implementation conforming to RFC 1112. It also contains an IP router including a packet filter.

 The programming interface is BSD-sockets compatible. For more information on sockets, see socket(7).

Manual page ip(7) line 1 (press h for help or q to quit)
```