Network Programming (Sockets) Guide

1 Overview

Sockets are an *interprocess communication* (IPC) mechanism that allows processes on different *networked machines* to exchange information. They can also be used among processes on the same machine. Sockets provide that basis for communication in most *client-server* applications. Key reasons for this are that sockets support *bidirectional* data exchange and can provide separate handles for each client.

The sockets mechanism support different *communication domains* and different *protocols* within each domain. Among networked machines, the communication *domain* corresponds to the *Internet Protocol* (TCP/IP), either version 4 (IPv4) or version 6 (IPv6). Socket *protocols* for IP correspond to the IP *transport protocols* such as TCP and UDP. *Socket addresses* for network connections involve both *IP addresses* and *ports.* Older socket-related calls support only IPv4, while newer calls also support IPv6.

Socket-based communication within a machine uses *UNIX domain sockets.* UNIX domain sockets have addresses that are *filesystem pathnames.* Because sockets are *bidirectional* and have filesystem addresses, this makes them preferable to *pipes* even within a single machine. UNIX domain sockets are used within Linux desktop frameworks to allow component apps to communicate.

Sockets use *file descriptor* handles just like regular (disk) files, so some calls that work for regular files also work with sockets. Key examples are `read()` and `write()`. However, there are additional I/O calls that provide additional functionality in conjunction with sockets.

2 Socket Address (sockaddr) Structs

Several socket-related calls have a *socket address* as one parameter. When a socket is created (with `socket()`), a *file descriptor handle* is returned, but the socket has no *"name,"* so data cannot be sent to it. Associating an address with a socket is traditionally known as "assigning a name to a socket." Once a socket has a name (address), that name can be used to connect to the socket from another socket, and exchange data.

One interesting aspect of socket addresses is that they differ in structure depending on the *communication domain.* E.g., IP domain addresses consist of an IP address and port, while UNIX domain addresses are file pathnames. Obviously, socket addresses for different communication domains will have quite different structure, so require different C types. Since C does not support *polymorphic* functions, how can one have socket-related calls that can take multiple address types as argument?

The way this is done is by specifying socket function address parameters as being of a common, generic type, `struct sockaddr`, which is large enough to accomodate any domain-specific address type. When these functions are called, the argument addresses will be of the type appropriate for the particular dommunication domain the socket is to be using. Since C does not support type hierarchies, howevewer, passing *"subtypes"* of a *"supertype"* is somewhat messy. The specific addresses must be *cast* to the generic address type, and the size of the specific address must also be given.

2.1 Generic: sockaddr

Generic type for socket addresses:

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

This is the socket address type that is specified in socket and network calls when a socket address is required, such as in `bind()`, `connect()`, etc. Effectively this type serves as the *supertype* of all other socket address types. Since C does not support type hierarchies, one generally has to *cast* a specific socket address structs to this type in the various system calls that include addresses as parameters.

2.2 IPv4

2.2.1 sockaddr_in

Socket address type for IPv4:

```
struct sockaddr_in {
  sa_family_t    sin_family; /* address family: AF_INET */
  uint16_t       sin_port;   /* port in network byte order */
  struct in_addr sin_addr;   /* IPv4 address (a struct) */
}
```

2.2.2 in_addr

The actual IPv4 address is stored in:

```
struct in_addr {
  uint32_t       s_addr;     /* IPv4 address in network byte order */
}
```

2.3 IPv6

2.3.1 sockaddr_in6

IPv6 has its own version of the `sockadd` struct:

```
struct sockaddr_in6 {
  u_char          sin6_len;      /* length of this structure */
  u_char          sin6_family;   /* AF_INET6 */
  u_int16m_t      sin6_port;     /* Transport layer port # */
  u_int32m_t      sin6_flowinfo; /* IPv6 flow information */
  struct in6_addr sin6_addr;     /* IPv6 address (a struct) */
}
```

2.3.2 in6_addr

Socket address type for IPv6:

```
struct in6_addr {
  u_int8_t        s6_addr[16];   /* IPv6 address */
}
```

This struct contains an array of sixteen 8-bit elements, that together make up a single 128-bit IPv6 address, in network byte order.

2.4 UNIX

Socket address type for UNIX sockets:

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
  sa_family_t    sun_family;               /* AF_UNIX */
  char           sun_path[UNIX_PATH_MAX];  /* pathname */
}
```

3 DNS Address Information Structs

When connecting to remote, networked machines, it is common to want to use the *Domain Name System* (**DNS**) to retrieve numeric addresses. There are a set of calls to support DNS queries, and several types are involved.

3.1 addrinfo

Type for use with `getaddrinfo()`. Can represent a *linked list* of results (`ai_next` points to the next element).

```
struct addrinfo {
  int              ai_flags;
  int              ai_family;
  int              ai_socktype;
  int              ai_protocol;
  size_t           ai_addrlen;
  struct sockaddr *ai_addr;
  char            *ai_canonname;
  struct addrinfo *ai_next;
}
```

So if `results` is a `struct addrinfo *`, one could retrieve the following:

- `struct sockaddr_in *` — `results->ai_addr`
- `struct sockaddr_in` — `*results->ai_addr`
- `struct in_addr` — `results->ai_addr->sin_addr`

3.2 hostent

Type for use with `gethostbyname()` and other calls.

```
struct hostent {
  char  *h_name;          /* name of host */
  char **h_aliases;       /* array of alternate host names */
  int    h_addrtype;      /* host address type */
  int    h_length;        /* length of address in bytes*/
  char **h_addr_list;     /* array of network addresses */
}

#define h_addr h_addr_list[0] /* for backward compatibility */
```

`h_addr_list` is an array of addresses. the type is confusing: `char **`. The rightmost `*` is due to it being an array, but this implies we have an array of "`char *`" (i.e., an array of strings). This is not the case, however, we have an array of network addresses. The "`char *`" is really a "`struct in_addr *`".

4 Socket-Related Symbolic Constants

There are a great many constants that are used in the socket calls. The most important classes and most common values are described here.

4.1 Address Family

- `AF_INET` – IPv4;
- `AF_INET6` – IPv6;
- `AF_UNIX` – UNIX sockets;
- `AF_LOCAL` – UNIX sockets;
- `AF_PACKET` – raw packets;

4.2 Protocol Family

Same meanings and values as the address family constants above. Some people insist the "PF" symbols are what should be used, but SUSv3 refers to only the "AF" symbols.

- `PF_INET` – IPv4;
- `PF_INET6` – IPv6;
- `PF_UNIX` – UNIX sockets;
- `PF_LOCAL` – UNIX sockets;
- `PF_PACKET` – raw packets;

4.3 Socket Type

- `SOCK_STREAM` – byte stream connection, e.g., Transmission Control Protocol (TCP);
- `SOCK_DGRAM` – datagram/message connection, e.g., User Datagram Protocol (UDP);
- `SOCK_RAW` – direct access to packet structure, e.g., raw IP packets;
- `SOCK_RDM` – reliable datagrams (nut does not guarantee ordering), e.g., Reliable Datagram Sockets (RDS).
- `SOCK_SEQPACKET` – reliable, ordered, but in messages, e.g., Stream Control Transmission Protocol (SCTP);
- `SOCK_DCCP` – message oriented w/congestion control, e.g., Datagram Congestion Control Protocol (DCCP);

4.4 Protocols

The `socket()` call takes *domain*, *socket type*, and *protocol* arguments. Normally only a single protocol exists to support a particular socket type within a given domain, so it is typical to use 0 (zero) as the protocol argument. However, it is possible that multiple protocols might be defined, and there are constants define to denote the intended one.

- `IPPROTO_IP` – dummy protocol (0);
- `IPPROTO_TCP` – TCP;
- `IPPROTO_UDP` – UDP;
- `IPPROTO_SCTP` – SCTP;
- `IPPROTO_DCCP` – DCCP;
- `IPPROTO_RAW` – raw IP sockets;
- `IPPROTO_ICMP` – ICMP (IP control protocol);

4.5 Miscellaneous

- `SOMAXCONN` – maximum queue length specifiable in `listen()`;
- `INADDR_ANY` – any local address (0x00000000);
- `INADDR_LOOPBACK` – any local address (0x7f000001);
- `INADDR_BROADCAST` – any host (0xffffffff);
- `INADDR_NONE` – error return address (0xffffffff);

5 IP Address Format Conversion

Often one is going to have an "IP address" as a string, in dotted quad (IPv4) or colon (IPv6) format, such as when an IP address is passed in to a program as a command-line argument. In order to be able to use these addresses in system calls like `bind` and `connect`, the string format will have to be converted to a true IP address (32-bit unsigned int).

5.1 Text/String to Numeric

5.1.1 inet_aton

This is the call to convert a dotted quad string address format to a true IPv4 address (in network byte order). Note that this call stores the IP address in a `struct in_addr` that the user must define, and pass a pointer to. Returns nonzero if the address is valid, zero if not. `inet_aton` should be used in preference to the older `inet_addr`.

The call syntax is:

```
  int inet_aton(const char *cp, struct in_addr *inp)
```

An example call to `inet_aton` is:

```
  struct sockaddr_in servaddr;
  inet_aton("131.230.133.20", &servaddr.sin_addr);
```

5.1.2 inet_addr

This is the older and now deprecated call to convert a dotted quad string address format to a true IPv4 address. The call is now deprecated, as it handles only IPv4 and does not properly handle the broadcast address 255.255.255.255 (since this is equivalent to -1 in two's complement notation, which is the error return value). You should use `inet_aton` instead.

The call syntax is:

```
  in_addr_t inet_addr(const char *cp)
```

An example call to `inet_addr` is:

```
  struct sockaddr_in servaddr;
  servaddr.sin_addr.s_addr = inet_addr("131.230.133.20");
```

5.1.3 inet_pton

This call extends `inet_aton()` to support multiple address families, but principally ipv6.

The call syntax is:
`int inet_pton(int af, const char *src, void *dst)`

The following address families are supported:

- AF_INET – `src` points to a string containing an IPv4 network address in the dotted-quad format, "ddd.ddd.ddd.ddd". The address is converted to a `struct in_addr` and copied to `dst`, which must be `sizeof(struct in_addr)` bytes long.
- AF_INET6 – `src` points to a string containing an IPv6 network address in any allowed IPv6 address format. The address is converted to a `struct in6_addr` and copied to `dst`, which must be `sizeof(struct in6_addr)` bytes long.

An example call to `inet_pton` with an IPv4 address is:

```
struct sockaddr_in servaddr;
inet_pton(AF_INET, "131.230.133.20", &servaddr.sin_addr);
```

An example call to `inet_pton` with an IPv6 address is:

```
struct sockaddr_in6 servaddr;
inet_pton(AF_INET6, "::ffff:131.230.133.20", &servaddr.sin6_addr);
```

5.2 Numeric to Text/String

5.2.1 inet_ntoa

5.2.2 inet_ntop

6 Basic Network Programming Syscalls

6.1 socket

`socket()` creates a socket structure and returns a *file descriptor handle* for the socket. The FD handle must be stored for use in other network syscalls plus `read()`, `write()`, etc.

The syntax is:

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

`domain` stands for "communication domain," and defines the *protocol/address family* for the socket. This determines the `type`'s of sockets/connections that are available. See Section **??** for a list of possible symbol values for `domain`.

`type` defines the type of communication semantics for the socket. The valid values must be valid for the specified `domain`. See Section **??** for a list of possible symbol values for `type`.

`protocol` is usually specified as 0 (zero), meaning use the default protocol for the specified socket `type`. However, it may be necessary to explicitly specify `protocol`, and possible symbol values are given in Section **??**.

For IP sockets, `protocol` is the IP protocol in the IP header to be received or sent. The only valid values for `protocol` are 0 and `IPPROTO_TCP` for `SOCK_STREAM` sockets, 0 and `IPPROTO_UDP` for sockets, etc. For IP sockets of `type` `SOCK_RAW`, however, you may specify any valid IANA IP protocol.

Common `socket()` calls:

- IPv4 socket: `socket(AF_INET, socket_type, protocol)`
- IPv4 TCP socket: `socket(AF_INET, SOCK_STREAM, 0))`
- IPv4 UDP socket: `socket(AF_INET, SOCK_DGRAM, 0))`
- IPv6 socket: `socket(AF_INET6, socket_type, protocol)`

6.2 bind

`bind()` is used to associate (bind) an address (name) with an open socket. Every *server* uses it since a socket must have an address in order for another process to connect to it. *Client* programs can use `bind()` if they need to be associated with a particular port, for example, but generally do not (so the OS assigns a port automatically).

The syntax is:

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

Standard setup for an IPv4 TCP socket for a *server*:

```
  struct sockaddr_in servaddr;

  //Set up server address struct
  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(1234);
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);  //Means accept from all local interf

  //Give socket a name/address by binding to a port:
  bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
```

One could also set up the address struct directly using the declaration:

```
  struct sockaddr_in servaddr = {AF_INET, htons(1234), htonl(INADDR_ANY)};
```

6.3 listen

`listen()` tell the kernel to allow incoming connection requests through the socket. This marks the socket as a *passive socket*. `accept()` can then be used to accept connection requests. A socket must have an address before `listen()` can be called.

The syntax is:

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

The standard call is:

```
  listen(sfd, 10);
```

The `sockfd` argument is a file descriptor that refers to a socket that must be of type `SOCK_STREAM` or `SOCK_SEQPACKET`.

6.4 accept

Accepts a connection on a (listening/passive) socket that is of a connection-oriented socket type (`SOCK_STREAM` `SOCK_SEQPACKET`). If there are pending connection(s) on the socket's connection queue, the first/earliest is immediately accepted, else the call *blocks* waiting for a connection request to complete.

`accept()` creates a *new socket* for the particular connection, and returns a *file descriptor* handle for this new connection socket. The original socket remains listening (the new socket is not in the listening state).

The syntax is:

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

The standard call if one is not interested in the caller's address is:

```
  accept(sockfd, (struct sockaddr *) NULL, NULL);
```

If one is interested in getting information about the caller:

```
  struct sockaddr_in clientaddr;
  sock_len clientlen;
  accept(sockfd, (struct sockaddr *) &clientaddr, &clientlen);
```

6.5 connect

`connect()` is used in a *client* program, to establish a connection to the server. It *must* be used with TCP connections, but can also be used to create a "connected UDP socket." (though this is less common).

The syntax is:

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

The standard setup for an IPv4 TCP socket:
(assuming a socket has already been created)

```
  struct sockaddr_in servaddr;

  //Set up server address struct:
  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(1234);
  inet_aton("131.230.133.20",&servaddr.sin_addr);

  connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))
```

6.6 write, send, sendto, sendmsg

While `write()` can be used with *connected* sockets, there are other calls that provide additional functionality or must be used with sockets that are *unconnected.* TCP (`SOCK_STREAM`) sockets are necessarily connected. UDP (`SOCK_DGREAM`) sockets are unconnected by default, but can become connected by using `connect()`.

6.6.1 send

`send()` must be used with connected sockets. It differs from using `write()` only in the addition of the `flags` parameter.

The syntax is:

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

6.6.2 sendto

`sendto()` is the primary call used with unconnected UDP sockets. It includes the address to send the message to.

The syntax is:
```
  ssize_t sendto(int s, const void *buf, size_t len, int flags,
                 const struct sockaddr *to, socklen_t tolen)
```

6.6.3 sendmsg

The syntax is:
```
    ssize_t sendmsg(int s, const struct msghdr *msg, int flags)
```

6.7 read, recv, recvfrom, recvmsg

While `read()` can be used with *connected* sockets, there are other calls that provide additional functionality or must be used with *unconnected* sockets.

6.7.1 recv

`recv()` must be used with connected sockets. It differs from using `read()` only in the addition of the `flags` parameter.

The syntax is:
```
  ssize_t recv(int s, void *buf, size_t len, int flags)
```

6.7.2 recvfrom

`recvfrom()` is the primary call used with unconnected UDP sockets. It includes the address that sent the message.

The syntax is:
```
  ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                   struct sockaddr *from, socklen_t *fromlen)
```

6.7.3 recvmsg

The syntax is:
```
  ssize_t recvmsg(int s, struct msghdr *msg, int flags)
```

6.8 getsockopt, setsockopt

There are various options that can be set on sockets, and two calls allow one to retrieve and set these options:

```
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen)
```

For example, to allow a port to be immediately reused one can do:

```
int i=1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
```

7 Address Translation (DNS Lookup) Calls

7.1 gethostbyname

This is the older call to lookup hostnames. It is much simpler than the newer call, `getaddrinfo()`, bu can handle only IPv4.

The call syntax is:
```
  struct hostent *gethostbyname(const char *name)
```

Here is example code showing how a client could lookup a server's IP address and prepare the socket address struct for use with `connect()`:

```
  struct sockaddr_in servaddr;
  struct hostent *hostinfo;

  //Lookup server's IP address info using gethostbyname:
  hostinfo = gethostbyname("pc00.cs.siu.edu");

  //Set up server address:
  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(1234);
  servaddr.sin_addr = *(struct in_addr *)hostinfo->h_addr;  //cast is required
```

There are actually a number of alternative methods/syntax that can be used for setting up the `sin_addr` field of the `sockaddr_in` struct:

```
  servaddr.sin_addr = *(struct in_addr *)hostinfo->h_addr_list[0];

  servaddr.sin_addr = *(struct in_addr *)*hostinfo->h_addr_list;

  memcpy(&servaddr.sin_addr, hostinfo->h_addr, hostinfo->h_length);

  bcopy(hostinfo->h_addr, &servaddr.sin_addr, hostinfo->h_length);
```

7.2 getaddrinfo

The syntax for is:
```
  int getaddrinfo(const char *node, const char *service,
                  const struct addrinfo *hints, struct addrinfo **results)
```
The arguments:

- node – typically the target hostname, as a C string;
- service – target service name or port, as a C string;

- hints – addrinfo struct with values to limit addresses returned;
- results – linked list of addrinfo structs containing the result addresses.

By default, `service` is to be the service name for the server, as can be found in `/etc/services`. The port number (as a string) can be supplied as well, but then the `ai_flags` field of `hints` must include the value: `AI_NUMERICSERV`.

The `hints` struct must generally have at least the following two fields set:

- `ai_family` – `AF_INET` or `AF_INET6`
- `ai_socktype` – `SOCK_STREAM` or `SOCK_DGRAM`

Here is example code showing how a client can get an IP address(es) to connect to:
(very limited error checking)

```
struct addrinfo hints, *results, *r;
int sfd, cfd;

//Set up hints struct for TCP connection:
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;     //Allow IPv4 or IPv6 address
hints.ai_socktype = SOCK_STREAM; //TCP connection
hints.ai_flags = 0;              //No flags
hints.ai_protocol = 0;           //Any protocol

//Get address using supplied hostname and service, checking if it worked:
getaddrinfo("www.cs.siu.edu", "http", &hints, &results)) != 0);

//Loop through the linked list of addresses, trying each until successful:
for (r = results; r != NULL; r = r->ai_next) {
  //Create socket for specific address, using info from result:
  //(here could potentiall have both IPv4 and IPv6 results)
  sfd = socket(r->ai_family, r->ai_socktype, r->ai_protocol);

  //Try connecting to the current address:
  if ((cfd = connect(sfd, r->ai_addr, r->ai_addrlen)) != -1)
    //Succeeded, so stop looping and use connection:
    break;


  //Did not succeed, so close socket and continue trying:
  close(sfd);
}
```

Another example, that simply uses the first address:
(code does not include error checking)

```
struct addrinfo hints;
```

```
  struct addrinfo *servinfo;

  //Setup hints and lookup server's IP address:
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_NUMERICSERV;  //to allow numeric port rather than service name
  getaddrinfo("www.cs.siu.edu", "1234", &hints, &servinfo);

  //Create socket:
  sockfd = socket(AF_INET, SOCK_STREAM, 0));

  //Connect to server using first address found:
  connect(sock_fd, (struct sockaddr *)servinfo->ai_addr, servinfo->ai_addrlen);

  freeaddrinfo(servinfo);  //Free the address linked list from getaddrinfo
```

`getaddrinfo()` has two associated functions:

```
  void freeaddrinfo(struct addrinfo *res)

  const char *gai_strerror(int errcode)
```

`freeaddrinfo()` is to be called to free the memory for the linked list of addresses. It should be called when one is done with the addresses.

`gai_strerror()` is needed to produce an informative error message if `getaddrinfo()` fails. It would typically be used like this:

```
  int status;
  if ((status = getaddrinfo(hostname, service, &hints, &results)) != 0); {
    fprintf(stderr, "getaddrinfo failed: %s\n", gai_strerror(status)); }
    exit(EXIT_FAILURE);
```

8 Raw Sockets and Packet Sockets

Typically when sending/receiving data over sockets, one wants to have the kernel's network stack automatically deal with the IP packet headers. This is what will happen with the socket setups shown earlier: data will automatically be properly encapsulated/extracted according to the chosen transport-layer protocol (e.g. TCP, UDP).

"Raw sockets" provide user programs with more control over IP packet headers. In particular, raw sockets can allow/require user programs to write entire datagrams, including the header(s). Raw sockets are necessary for programs like `nmap` that do port/network scanning and for many malicious programs that spoof IP addresses, send invalid TCP flag combinations, etc. Raw sockets are also necessary for programs like packet sniffers, and they can be used to implement new transport-layer protocols in userspace.

Packet sockets are similar but allow even more control over the datagrams (or alternatively, require even more work to setup the datagrams). It is common to use the term "raw sockets" to refer to both true raw sockets and packet sockets. More info on raw sockets can be found by doing "`man 7 raw`" and on packet sockets by doing "`man 7 packet`".

To understand material on raw/packet sockets, it is important to be clear about several of the layers in the OSI networking model:

- layer 2: data link – node-to-node data transfer, e.g., Ethernet with MAC addresses
- layer 3: network – host-to-host data transfer, e.g., IP with IP addresses
- layer 4: transport – app-to-app data transfer, e.g., TCP and UDP with IP addresses and ports

Remember that the syntax for `socket()` is as follows:
```
  int socket(int domain, int type, int protocol)
```
"Raw scokets" (IPv4) are created with this call:
```
  socket(AF_INET, SOCK_RAW, int protocol)
```
"Packet sockets" are created with this call:
```
  socket(AF_PACKET, int socket_type, int protocol)
```
Note that only processes with effective UID 0 (root) or the capability `CAP_NET_RAW` may open raw or packet sockets!

8.1 Raw Sockets (IPv4)

"Raw scokets" (IPv4) are created with this call:
```
  socket(AF_INET, SOCK_RAW, int protocol)
```

8.1.1 Writing/Sending

The kernel IPv4 layer generates an IP header when sending a packet unless the socket option `IP_HDRINCL` is enabled on the socket. When `IP_HDRINCL` is enabled, the packet must contain an IP header.

`int` values for `protocol` are listed in the file `/etc/protocols`. E.g., IP is 0, TCP is 6, and UDP is 17. There are symbolic constants for the protocols with names `IPPROTO_*`: `IPPROTO_IP`, `IPPROTO_TCP`, etc. See Section **??**.

The `protocol` symbol `IPPROTO_RAW`

A protocol of `IPPROTO_RAW` implies enabled `IP_HDRINCL`

8.1.2 Reading/Receiving

When a packet is received, it is passed to any raw sockets that have been bound to its protocol before it is passed to other protocol handlers (e.g., kernel protocol modules).

For receiving the IP header is always included in the packet.

All packets or errors matching the protocol number specified for the raw socket are passed to this socket. For a list of the allowed protocols see RFC 1700 assigned numbers and getprotobyname(3).

Note that an `IPPROTO_RAW` socket is *send only*. If you want to receive all IP packets, use a packet socket with the `ETH_P_IP` protocol.

The packet parameters that can be modified are determined by which combination of Domain, Type, and Protocol arguments are used.

```
socket (AF_INET, SOCK_RAW, IPPROTO_RAW);
```

The kernel fills out layer 2 (data link) information (source and next-hop MAC addresses). Tell the kernel the IP header is included (by us) by using `setsockopt()` and the `IP_HDRINCL` flag. We can modify all values within the packet.

```
socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL));
```

We fill out all values, including the layer 2 (data link) information (source and next-hop MAC addresses), the info that will go into the Ethernet frame headers. To do this, we must know the MAC address of the router/host the frames will be routed to next, as well as the MAC address of the network interface (”network card”) we’re sending the packet from.