NAME

ip - Internet Protocol

SYNOPSIS

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

int
socket(*AF_INET*, *SOCK_RAW*, *proto*);

DESCRIPTION

IP is the transport layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a "raw socket" when developing new protocols, or special-purpose applications.

There are several IP-level setsockopt(2) and getsockopt(2) options. IP_OPTIONS may be used to provide IP options to be transmitted in the IP header of each outgoing packet or to examine the header options on incoming packets. IP options may be used with any socket type in the Internet family. The format of IP options to be sent is that specified by the IP protocol specification (RFC-791), with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. To disable previously specified options, use a zero-length buffer:

```
setsockopt(s, IPPROTO_IP, IP_OPTIONS, NULL, 0);
```

IP_TOS may be used to set the differential service codepoint (DSCP) and the explicit congestion notfication (ECN) codepoint. Setting the ECN codepoint - the two least significant bits - on a socket using a transport protocol implementing ECN has no effect.

IP_TTL configures the time-to-live (TTL) field in the IP header for SOCK_STREAM, SOCK_DGRAM, and certain types of SOCK_RAW sockets. For example,

```
int tos = IPTOS_DSCP_EF;       /* see <netinet/ip.h> */
setsockopt(s, IPPROTO_IP, IP_TOS, &tos, sizeof(tos));

int ttl = 60;                  /* max = 255 */
setsockopt(s, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
```

IP_IPSEC_POLICY controls IPSec policy for sockets. For example,

```
const char *policy = "in ipsec ah/transport//require";
char *buf = ipsec_set_policy(policy, strlen(policy));
setsockopt(s, IPPROTO_IP, IP_IPSEC_POLICY, buf, ipsec_get_policylen(buf));
```

IP_MINTTL may be used to set the minimum acceptable TTL a packet must have when received on a socket. All packets with a lower TTL are silently dropped. This option is only really useful when set to 255, preventing packets from outside the directly connected networks reaching local listeners on sockets.

IP_DONTFRAG may be used to set the Don’t Fragment flag on IP packets. Currently this option is respected only on udp(4) and raw **ip** sockets, unless the IP_HDRINCL option has been set. On tcp(4) sockets, the Don’t Fragment flag is controlled by the Path MTU Discovery option. Sending a packet larger than the MTU size of the egress interface, determined by the destination address, returns an EMSGSIZE error.

If the IP_ORIGDSTADDR option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call will return the destination IP address and destination port for a UDP datagram. The *msg_control* field in the *msghdr* structure points to a buffer that contains a *cmsghdr* structure followed by the sockaddr_in structure. The *cmsghdr* fields have the following values:

```
cmsg_len = CMSG_LEN(sizeof(struct sockaddr_in))
cmsg_level = IPPROTO_IP
cmsg_type = IP_ORIGDSTADDR
```

If the IP_RECVDSTADDR option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call will return the destination IP address for a UDP datagram. The *msg_control* field in the *msghdr* structure points to a buffer that contains a *cmsghdr* structure followed by the IP address. The *cmsghdr* fields have the following values:

```
cmsg_len = CMSG_LEN(sizeof(struct in_addr))
cmsg_level = IPPROTO_IP
cmsg_type = IP_RECVDSTADDR
```

The source address to be used for outgoing UDP datagrams on a socket can be specified as ancillary data with a type code of IP_SENDSRCADDR. The msg_control field in the msghdr structure should point to a buffer that contains a *cmsghdr* structure followed by the IP address. The cmsghdr fields should have the following values:

```
cmsg_len = CMSG_LEN(sizeof(struct in_addr))
```

cmsg_level = IPPROTO_IP
cmsg_type = IP_SENDSRCADDR

The socket should be either bound to INADDR_ANY and a local port, and the address supplied with IP_SENDSRCADDR should't be INADDR_ANY, or the socket should be bound to a local address and the address supplied with IP_SENDSRCADDR should be INADDR_ANY. In the latter case bound address is overridden via generic source address selection logic, which would choose IP address of interface closest to destination.

For convenience, IP_SENDSRCADDR is defined to have the same value as IP_RECVDSTADDR, so the IP_RECVDSTADDR control message from recvmsg(2) can be used directly as a control message for sendmsg(2).

If the IP_ONESBCAST option is enabled on a SOCK_DGRAM or a SOCK_RAW socket, the destination address of outgoing broadcast datagrams on that socket will be forced to the undirected broadcast address, INADDR_BROADCAST, before transmission. This is in contrast to the default behavior of the system, which is to transmit undirected broadcasts via the first network interface with the IFF_BROADCAST flag set.

This option allows applications to choose which interface is used to transmit an undirected broadcast datagram. For example, the following code would force an undirected broadcast to be transmitted via the interface configured with the broadcast address 192.168.2.255:

```
char msg[512];
struct sockaddr_in sin;
int onesbcast = 1;    /* 0 = disable (default), 1 = enable */

setsockopt(s, IPPROTO_IP, IP_ONESBCAST, &onesbcast, sizeof(onesbcast));
sin.sin_addr.s_addr = inet_addr("192.168.2.255");
sin.sin_port = htons(1234);
sendto(s, msg, sizeof(msg), 0, &sin, sizeof(sin));
```

It is the application's responsibility to set the IP_TTL option to an appropriate value in order to prevent broadcast storms. The application must have sufficient credentials to set the SO_BROADCAST socket level option, otherwise the IP_ONESBCAST option has no effect.

If the IP_BINDANY option is enabled on a SOCK_STREAM, SOCK_DGRAM or a SOCK_RAW socket, one can bind(2) to any address, even one not bound to any available network interface in the system. This functionality (in conjunction with special firewall rules) can be used for implementing a transparent proxy. The PRIV_NETINET_BINDANY privilege is needed to set this option.

If the IP_RECVTTL option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call will return the IP TTL (time to live) field for a UDP datagram. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by the TTL. The cmsghdr fields have the following values:

cmsg_len = CMSG_LEN(sizeof(u_char))
cmsg_level = IPPROTO_IP
cmsg_type = IP_RECVTTL

If the IP_RECVTOS option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call will return the IP TOS (type of service) field for a UDP datagram. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by the TOS. The cmsghdr fields have the following values:

cmsg_len = CMSG_LEN(sizeof(u_char))
cmsg_level = IPPROTO_IP
cmsg_type = IP_RECVTOS

If the IP_RECVIF option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call returns a *struct sockaddr_dl* corresponding to the interface on which the packet was received. The *msg_control* field in the *msghdr* structure points to a buffer that contains a *cmsghdr* structure followed by the *struct sockaddr_dl*. The *cmsghdr* fields have the following values:

cmsg_len = CMSG_LEN(sizeof(struct sockaddr_dl))
cmsg_level = IPPROTO_IP
cmsg_type = IP_RECVIF

IP_PORTRANGE may be used to set the port range used for selecting a local port number on a socket with an unspecified (zero) port number. It has the following possible values:

IP_PORTRANGE_DEFAULT use the default range of values, normally IPPORT_HIFIRSTAUTO through IPPORT_HILASTAUTO. This is adjustable through the sysctl setting: *net.inet.ip.portrange.first* and *net.inet.ip.portrange.last*.

IP_PORTRANGE_HIGH use a high range of values, normally IPPORT_HIFIRSTAUTO and IPPORT_HILASTAUTO. This is adjustable through the sysctl setting: *net.inet.ip.portrange.hifirst* and *net.inet.ip.portrange.hilast*.

IP_PORTRANGE_LOW use a low range of ports, which are normally restricted to privileged processes on UNIX systems. The range is normally from

IPPORT_RESERVED - 1 down to IPPORT_RESERVEDSTART in descending order. This is adjustable through the sysctl setting: *net.inet.ip.portrange.lowfirst* and *net.inet.ip.portrange.lowlast*.

The range of privileged ports which only may be opened by root-owned processes may be modified by the *net.inet.ip.portrange.reservedlow* and *net.inet.ip.portrange.reservedhigh* sysctl settings. The values default to the traditional range, 0 through IPPORT_RESERVED - 1 (0 through 1023), respectively. Note that these settings do not affect and are not accounted for in the use or calculation of the other *net.inet.ip.portrange* values above. Changing these values departs from UNIX tradition and has security consequences that the administrator should carefully evaluate before modifying these settings.

Ports are allocated at random within the specified port range in order to increase the difficulty of random spoofing attacks. In scenarios such as benchmarking, this behavior may be undesirable. In these cases, *net.inet.ip.portrange.randomized* can be used to toggle randomization off.

Multicast Options

IP multicasting is supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on networks where the interface driver supports multicasting.

The IP_MULTICAST_TTL option changes the time-to-live (TTL) for outgoing multicast datagrams in order to control the scope of the multicasts:

```
u_char ttl;             /* range: 0 to 255, default = 1 */
setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Datagrams with a TTL of 1 are not forwarded beyond the local network. Multicast datagrams with a TTL of 0 will not be transmitted on any network, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with TTL greater than 1 may be forwarded to other networks if a multicast router is attached to the local network.

For hosts with multiple interfaces, where an interface has not been specified for a multicast group membership, each multicast transmission is sent from the primary network interface. The IP_MULTICAST_IF option overrides the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

where "addr" is the local IP address of the desired interface or INADDR_ANY to specify the default interface.

To specify an interface by index, an instance of *ip_mreqn* may be passed instead. The *imr_ifindex* member should be set to the index of the desired interface, or 0 to specify the default interface. The kernel differentiates between these two structures by their size.

The use of *IP_MULTICAST_IF* is *not recommended*, as multicast memberships are scoped to each individual interface. It is supported for legacy use only by applications, such as routing daemons, which expect to be able to transmit link-local IPv4 multicast datagrams (224.0.0.0/24) on multiple interfaces, without requesting an individual membership for each interface.

An interface's local IP address and multicast capability can be obtained via the SIOCGIFCONF and SIOCGIFFLAGS ioctls. Normal applications should not need to use this option.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. The IP_MULTICAST_LOOP option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;            /* 0 = disable, 1 = enable (default) */
setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

This option improves performance for applications that may have no more than one instance on a single host (such as a routing daemon), by eliminating the overhead of receiving their own transmissions. It should generally not be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

The sysctl setting *net.inet.ip.mcast.loop* controls the default setting of the IP_MULTICAST_LOOP socket option for new sockets.

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

A host must become a member of a multicast group before it can receive datagrams sent to the group. To join a multicast group, use the IP_ADD_MEMBERSHIP option:

```
struct ip_mreqn mreqn;
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreqn, sizeof(mreqn));
```

where *mreqn* is the following structure:

```
struct ip_mreqn {
   struct in_addr imr_multiaddr; /* IP multicast address of group */
   struct in_addr imr_interface; /* local IP address of interface */
   int            imr_ifindex;   /* interface index */
}
```

imr_ifindex should be set to the index of a particular multicast-capable interface if the host is multihomed. If *imr_ifindex* is non-zero, value of *imr_interface* is ignored. Otherwise, if *imr_ifindex* is 0, kernel will use IP address from *imr_interface* to lookup the interface. Value of *imr_interface* may be set to *INADDR_ANY* to choose the default interface, although this is not recommended; this is considered to be the first interface corresponding to the default route. Otherwise, the first multicast-capable interface configured in the system will be used.

Legacy *struct ip_mreq*, that lacks *imr_ifindex* field is also supported by IP_ADD_MEMBERSHIP setsockopt. In this case kernel would behave as if *imr_ifindex* was set to zero: *imr_interface* will be used to lookup interface.

Prior to FreeBSD 7.0, if the *imr_interface* member is within the network range 0.0.0.0/8, it is treated as an interface index in the system interface MIB, as per the RIP Version 2 MIB Extension (RFC-1724). In versions of FreeBSD since 7.0, this behavior is no longer supported. Developers should instead use the RFC 3678 multicast source filter APIs; in particular, MCAST_JOIN_GROUP.

Up to IP_MAX_MEMBERSHIPS memberships may be added on a single socket. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface.

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

where *mreq* contains the same values as used to add the membership. Memberships are dropped when the socket is closed or the process exits.

The IGMP protocol uses the primary IP address of the interface as its identifier for group membership. This is the first IP address configured on the interface. If this address is removed or changed, the results are undefined, as the IGMP membership state will then be inconsistent. If multiple IP aliases are configured on the same interface, they will be ignored.

This shortcoming was addressed in IPv6; MLDv2 requires that the unique link-local address for an

interface is used to identify an MLDv2 listener.

Source-Specific Multicast Options

Since FreeBSD 8.0, the use of Source-Specific Multicast (SSM) is supported. These extensions require an IGMPv3 multicast router in order to make best use of them. If a legacy multicast router is present on the link, FreeBSD will simply downgrade to the version of IGMP spoken by the router, and the benefits of source filtering on the upstream link will not be present, although the kernel will continue to squelch transmissions from blocked sources.

Each group membership on a socket now has a filter mode:

MCAST_EXCLUDE Datagrams sent to this group are accepted, unless the source is in a list of blocked source addresses.

MCAST_INCLUDE Datagrams sent to this group are accepted only if the source is in a list of accepted source addresses.

Groups joined using the legacy IP_ADD_MEMBERSHIP option are placed in exclusive-mode, and are able to request that certain sources are blocked or allowed. This is known as the *delta-based API*.

To block a multicast source on an existing group membership:

```
struct ip_mreq_source mreqs;
setsockopt(s, IPPROTO_IP, IP_BLOCK_SOURCE, &mreqs, sizeof(mreqs));
```

where *mreqs* is the following structure:

```
struct ip_mreq_source {
   struct in_addr imr_multiaddr; /* IP multicast address of group */
   struct in_addr imr_sourceaddr; /* IP address of source */
   struct in_addr imr_interface; /* local IP address of interface */
}
```

imr_sourceaddr should be set to the address of the source to be blocked.

To unblock a multicast source on an existing group:

```
struct ip_mreq_source mreqs;
setsockopt(s, IPPROTO_IP, IP_UNBLOCK_SOURCE, &mreqs, sizeof(mreqs));
```

The IP_BLOCK_SOURCE and IP_UNBLOCK_SOURCE options are *not permitted* for inclusive-mode

group memberships.

To join a multicast group in MCAST_INCLUDE mode with a single source, or add another source to an existing inclusive-mode membership:

```
struct ip_mreq_source mreqs;
setsockopt(s, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP, &mreqs, sizeof(mreqs));
```

To leave a single source from an existing group in inclusive mode:

```
struct ip_mreq_source mreqs;
setsockopt(s, IPPROTO_IP, IP_DROP_SOURCE_MEMBERSHIP, &mreqs, sizeof(mreqs));
```

If this is the last accepted source for the group, the membership will be dropped.

The IP_ADD_SOURCE_MEMBERSHIP and IP_DROP_SOURCE_MEMBERSHIP options are *not accepted* for exclusive-mode group memberships. However, both exclusive and inclusive mode memberships support the use of the *full-state API* documented in RFC 3678. For management of source filter lists using this API, please refer to sourcefilter(3).

The sysctl settings *net.inet.ip.mcast.maxsocksrc* and *net.inet.ip.mcast.maxgrpsrc* are used to specify an upper limit on the number of per-socket and per-group source filter entries which the kernel may allocate.

Raw IP Sockets

Raw IP sockets are connectionless, and are normally used with the sendto(2) and recvfrom(2) calls, though the connect(2) call may also be used to fix the destination for future packets (in which case the read(2) or recv(2) and write(2) or send(2) system calls may be used).

If *proto* is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If *proto* is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with), unless the IP_HDRINCL option has been set. Unlike in previous BSD releases, incoming packets are received with IP header and options intact, leaving all fields in network byte order.

IP_HDRINCL indicates the complete IP header is included with the data and may be used only with the SOCK_RAW type.

```
#include <netinet/in_systm.h>
#include <netinet/ip.h>

int hincl = 1;              /* 1 = on, 0 = off */
setsockopt(s, IPPROTO_IP, IP_HDRINCL, &hincl, sizeof(hincl));
```

Unlike previous BSD releases, the program must set all the fields of the IP header, including the following:

```
ip->ip_v = IPVERSION;
ip->ip_hl = hlen >> 2;
ip->ip_id = 0;  /* 0 means kernel set appropriate value */
ip->ip_off = htons(offset);
ip->ip_len = htons(len);
```

The packet should be provided as is to be sent over wire. This implies all fields, including *ip_len* and *ip_off* to be in network byte order. See byteorder(3) for more information on network byte order. If the *ip_id* field is set to 0 then the kernel will choose an appropriate value. If the header source address is set to INADDR_ANY, the kernel will choose an appropriate address.

ERRORS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket has not been connected;

[ENOBUFS] when the system runs out of memory for an internal data structure;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

[EACCES] when an attempt is made to create a raw IP socket by a non-privileged process.

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL] An unknown socket option name was given.

[EINVAL] The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

The following errors may occur when attempting to send IP datagrams via a "raw socket" with the IP_HDRINCL option set:

[EINVAL] The user-supplied *ip_len* field was not equal to the length of the datagram written to the socket.

SEE ALSO

getsockopt(2), recv(2), send(2), byteorder(3), CMSG_DATA(3), sourcefilter(3), icmp(4), igmp(4), inet(4), intro(4), multicast(4)

D. Thaler, B. Fenner, and B. Quinn, *Socket Interface Extensions for Multicast Source Filters*, RFC 3678, Jan 2004.

HISTORY

The **ip** protocol appeared in 4.2BSD. The *ip_mreqn* structure appeared in Linux 2.4.

BUGS

Before FreeBSD 10.0 packets received on raw IP sockets had the *ip_hl* subtracted from the *ip_len* field.

Before FreeBSD 11.0 packets received on raw IP sockets had the *ip_len* and *ip_off* fields converted to host byte order. Packets written to raw IP sockets were expected to have *ip_len* and *ip_off* in host byte order.