Linux Kernel Networking – advanced topics (6)

Sockets in the kernel

Rami Rosen

ramirose@gmail.com

Haifux, August 2009

www.haifux.org

Linux Kernel Networking (6)- advanced topics

- Note:
- This lecture is a sequel to the following 5 lectures I gave in Haifux:

1) Linux Kernel Networking lecture

- http://www.haifux.org/lectures/172/
- **slides**:http://www.haifux.org/lectures/172/netLec.pdf

2) Advanced Linux Kernel Networking - Neighboring Subsystem and IPSec lecture

- http://www.haifux.org/lectures/180/
- **slides**:http://www.haifux.org/lectures/180/netLec2.pdf

Linux Kernel Networking (6)- advanced topics

3) Advanced Linux Kernel Networking - IPv6 in the Linux Kernel lecture

- http://www.haifux.org/lectures/187/
  - **Slides**: http://www.haifux.org/lectures/187/netLec3.pdf

4) Wireless in Linux

http://www.haifux.org/lectures/206/

  - **Slides**: http://www.haifux.org/lectures/206/wirelessLec.pdf

5) Sockets in the Linux Kernel

- http://www.haifux.org/lectures/217/
  - **Slides**: http://www.haifux.org/lectures/217/netLec5.pdf

Note

- Note: This is the second part of the “Sockets in the Linux Kernel” lecture which was given in Haifux in 27.7.09. You may find some background material for this lecture in its slides:
- http://www.haifux.org/lectures/217/netLec5.pdf

TOC

- TOC:
  - RAW Sockets
  - UNIX Domain Sockets
  - Netlink sockets
  - SCTP sockets.
  - Appendices
- Note: All code examples in this lecture refer to the recent **2.6.30** version of the Linux kernel.

RAW Sockets

- There are cases when there is no interface to create sockets of a certain protocol (ICMP protocol, NETLINK protocol) => use Raw sockets.
- raw socket creation is done thus, for example:
  - sd = socket(AF_INET, SOCK_RAW, 0);
  - sd = socket(AF_INET, SOCK_RAW,IPPROTO_UDP);
  - sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
    - ETH_P_IP tells to handle all IP packets.
  - When using AF_INET family, as in the first two cases, the socket is added to kernel RAW sockets hash table (the hash key is the protocol number). This is done by *raw_hash_sk()*, (net/ipv4/raw.c), which is invoked by *inet_create()*, when creating the socket.

  - When using AF_PACKET family, as in the third case, a socket is **not** added to the kernel RAW sockets hash table.
- See Appendix F for an example of using packet raw socket.

- Raw socket creation **MUST** be done as a super user.
  - In case an ordinary user try to create a raw socket, you will get:
  - “error: socket: Operation not permitted.” (**EPERM**).
  - You can set the CAP_NET_RAW capability to enable non root users to create raw sockets:
  - setcap cap_net_raw=+ep rawserver

Usage of RAW socket: ping

- You do not specify ports with RAW sockets; RAW sockets do not work with ports.
- When the kernel receives a raw packet, it delivers it to all raw sockets.
- Ping in fact is sending an ICMP packet.
  - The type of this ICMP packet is **ICMP ECHO REQUEST.**

Send a ping implementation(simplified)

```
#define BUFSIZE 1500

char sendbuf[BUFSIZE];

struct icmp *icmp;

int sockfd;

struct sockaddr_in target;

int datalen=56;


target.sin_family = AF_INET;

inet_aton("192.168.0.121",&target.sin_addr);

icmp = (struct icmp *)sendbuf;

icmp->icmp_type  = ICMP_ECHO;

icmp->icmp_code = 0;

icmp->icmp_id     = getpid();
```

```
memset(icmp->icmp_data, 0xa5, datalen);


icmp->icmp_cksum=0;

sockfd=socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

res = sendto(sockfd, sendbuf, len, 0, (struct sockaddr*)&target,   sizeof(struct
    sockaddr_in));
```

- Missing here is sequence number, checksum computation.
- The default number of data bytes to be sent is 56; the ICMP header is 8 bytes. So we get 64 bytes (or 84 bytes, if we include the IP header of 20 bytes).

Receive a ping-implementation(simplified)

```
__u8 *buf;
char addrbuf[128];
struct iovec iov;
struct iphdr *iphdr;
int sockfd;
struct icmphdr *icmphdr;
char recvbuf[BUFSIZE];
char controlbuf[BUFSIZE];
struct msghdr msg;
sockfd=socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

```
iov.iov_base = recvbuf;
iov.iov_len  = sizeof(recvbuf);
memset(&msg, 0, sizeof(msg));
msg.msg_name = addrbuf;
msg.msg_namelen = sizeof(addrbuf);

msg.msg_iov = &iov;
msg.msg_iovlen = 1;
msg.msg_control    = controlbuf;
msg.msg_controllen = sizeof(controlbuf);
n = recvmsg(sockfd, &msg, 0);
```

```
buf   = msg.msg_iov->iov_base;
iphdr = (struct iphdr*)buf;
 icmphdr = (struct icmphdr*)(buf+(iphdr->ihl*4));
if (icmphdr->type == ICMP_ECHOREPLY)
      printf("ICMP_ECHOREPLY\n");
if (icmphdr->type == ICMP_DEST_UNREACH)
      printf("ICMP_DEST_UNREACH\n");
```

- The only SOL_RAW option a Raw socket can get is ICMP_FILTER.
- This can be done thus:

```
#define ICMP_FILTER 1

struct icmp_filter {

__u32 data;

};

filt.data = 1<<ICMP_DEST_UNREACH;

res = setsockopt(sockfd, SOL_RAW, ICMP_FILTER,
    (char*)&filt, sizeof(filt));
```

- Adding this code in the receive Ping application above will prevent **Destination Unreachable** ICMP messages from received in user space by recvmsg.
- There are quite a lot more ICMP options; by default, we do NOT filter any ICMP messages.
- Among the other options you can set by setsockopt are:
- ICMP_ECHO (echo request)
- ICMP_ECHOREPLY (echo reply)

- ICMP_TIME_EXCEEDED
- And more (see Appendix D for a full list).
- **Traceroute** also uses raw sockets.
  - Traceroute changes the TTL field in the ip header.
  - This is done by IP_TTL and control messages in current Linux traceroute implementation (Dmitry Butskoy).
  - In the original traceroute (by Van Jacobson) it was done with the IP_HDRINCL socket option:
    - (setsockopt(sndsock, IPPROTO_IP, IP_HDRINCL,...)

- The IP_HDRINCL tells the IP layer **not** to prepare an IP header when sending a packet.
  - IP_HDRINCL is also applicable in IPV6.
- When **receiving** a packet, the IP header is always included in the packet.
- When **sending** a packet, by specifying the the *IP_HDRINCL option you tell the kernel that the IP header is already included in the packet, so no need to prepare it in the kernel.*
  - *raw_send_hdrinc() in net/ipv4/raw.c*
  - *The IP_HDRINCL option is applied only to the SOCK_RAW type of protocol.*
- See Lawrence Berkeley National Laboratory traceroute:
- ftp://ftp.ee.lbl.gov/traceroute.tar.gz

- If a raw socket was created with protocol type of IPPROTO_RAW , this implies enabling IP_HDRINCL:

  Thus, this call from user space:

  socket(AF_INET,SOCK_RAW,IPPROTO_RAW)

  invokes this code in the kernel:

```
if (SOCK_RAW == sock->type) {
        inet->num = protocol;
        if (IPPROTO_RAW == protocol)
        inet->hdrincl = 1;
        ...
        ...
```

  (From inet_create(), net/ipv4//af_inet.c)

- **Spoofing attack:** setting the IP address of packets to be different than the real ones.
- UDP spoofing is easier since UDP is connectionless.
- Following is an example of UDP spoofing with raw sockets and IP_HDRINCL option:
  - We build an IP header.
    - We set the protocol field in this ip header to IP_PROTOUDP.
  - We build a UDP header.
  - Note : when behind a NAT, this probably will not work

```
unsigned short in_cksum(unsigned short *addr, int len);
int main(int argc, char **argv)
{
struct iphdr ip;
struct udphdr udp;
int sd;
const int on = 1;
struct sockaddr_in sin;
int res;
u_char *packet;
packet = (u_char *)malloc(60);
```

```
ip.ihl = 0x5;
ip.version = 0x4;
ip.tos = 0x0;
ip.tot_len = 60;
ip.id = htons(12830);
ip.frag_off = 0x0;
ip.ttl = 64;
ip.protocol = IPPROTO_UDP;
ip.check = 0x0;
ip.saddr = inet_addr("192.168.0.199");
ip.daddr = inet_addr("76.125.43.103")
```

```
memcpy(packet, &ip, sizeof(ip));
udp.source = htons(45512);
udp.dest = htons(999);
udp.len = htons(10);
udp.check = 0;
memcpy(packet + 20, &udp, sizeof(udp));
memcpy(packet + 28,"ab",2);
if ((sd = socket(AF_INET, SOCK_RAW, 0)) < 0) {
        perror("raw socket");
exit(1);
}
```

```
if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
        perror("setsockopt");
        exit(1);
}
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = ip.daddr;
res=sendto(sd, packet, 60, 0, (struct sockaddr *)&sin, sizeof(struct sockaddr) );
if (res<0)
        perror("sendto");
else
        printf("ok %d bytes sent\n",res);
}
```

- Note: what will happen if we specify an illegal source address, like “255.255.255.255”?
  - The packet will be sent.
  - If we want to log such packets on the receiver side, (to detect spoofing attempts), we must set the **log_martians** kernel tunable thus:
    - echo "1" > /proc/sys/net/ipv4/conf/all/log_martians
  - Then we will see in the kernel syslog messages like this:
    - martian source 82.80.80.193 from 255.255.255.255, on dev eth0
    - Following will be the **ethernet header**:
    - ll header:

Raw sockets and sniffers

- When you activate tshark (formerly tethereal) or wireshark or tcpdump, you call the *pcap_open_live()* method of the pcap library. This method creates a raw socket thus:
  - socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))
  - *pcap_open_live() is implemented in* libpcap-0.9.8/pcap-linux.c.
- PF_PACKET sockets work with the network interface card.

- Note:
  - When you open tshark thus:
    - tshark -i any
    - Then the socket is opened thus:
    - socket(PF_PACKET, **SOCK_DGRAM**, htons(ETH_P_ALL))
    - This is called “cooked mode”
    - SLL. (Socket Link Layer).
    - With **SOCK_DGRAM,** the kernel is responsible for adding ethernet header (when sending a packet) or removing ethernet header (when receiving a packet).

- With SOCK_RAW, the application is responsible for adding an ethernet header when sending.
- Also you will get this message:
  - “Capturing on Pseudo-device that captures on all interfaces”
  - tshark: Promiscuous mode not supported on the "any" device

Unix Domain Sockets

- AF_UNIX / PF_UNIX / AF_LOCAL / PF_LOCAL.
- A way for interprocess communication. (IPC)
  - the client and server are on the same host.
- AF_UNIX sockets can be either SOCK_STREAM or SOCK_DGRAM.
  - And, since kernel 2.6.4, also SOCK_SEQPACKET.
- Usage: in rsyslogd(AF_UNIX/SOCK_DGRAM) and udev daemons (AF_LOCAL/ SOCK_DGRAM), hald, crond, and a lot more.

- Unix domain sockets do not support the transmission of out-of-band data.
- MSG_OOB is not supported at all in Unix domain sockets
  - This applies For all 3 types, SOCK_STREAM,SOCK_DGRAM and SOCK_SEQPACKET.

- Usually uses files in the local filesystem.
  - Abstract namespaces.
- Why not extend it to use between domains in virtualization which have access to shared filesystem ?
- With rsyslogd, the path is under /dev:
  - ls -al /dev/log
    - srw-rw-rw- 1 root root 0 01-07-09 13:17 /dev/log
    - Notice the 's' in the beginning => for socket.
  - ls -F /dev/log
    - /dev/log=
    - (with ls, -F is for appending indicator to entries)

Unix Domain Socket server Example

```
int s;
int res;
struct sockaddr_un name;
memset(&name,0,sizeof (name));
name.sun_family = AF_LOCAL;
strcpy(name.sun_path,"/work/test_unix");
s = socket(AF_UNIX, SOCK_STREAM,0);
if (s<0)
      perror("socket");
res = bind(s, (struct sockaddr*)&name, SUN_LEN(&name));
```

- Calling *bind()* in the example above will create a file named /work/test_unix
- ls -al /work/test_unix
  - srwxr-xr-x
  - Notice the “s” for socket.
- Notice that with DGRAM Unix domain sockets, calling sednto() without calling bind() before, will **not call autobind() as opposed to what happens in udp under the same scenario.**
- **In this case, the receiver cannot reply (because it does not know to who).**

- lsof -U : shows Unix domain sockets
- Also: netstat --unix –all
  - Tip: use netstat -ax for short.
    - [ACC] in the third column means that the socket is unconnected and waiting for connection. (SO_ACCECPTON).
- And also:
  - cat /proc/net/protocols | grep UNIX
  - cat /proc/net/unix
- struct sockaddr_un (*/usr/include/linux/un.h*)

- The pathname for a Unix domain socket should be an absolute pathname.
- For abstract namespaces:
  - address.sun_path[0] = 0
- The last column of netstat --unix --all is the path.
  - In case of abstract namespace, it will begin with @:
  - netstat --unix --all | grep udevd
  - unix 2 [ ] DGRAM 602 @/org/kernel/udev/udevd

- Control messages in Unix domain sockets:
  - SCM_RIGHTS - You can pass an open file descriptor from one process to another using Unix domain socket and control messages (ancillary data).
  - SCM_CREDENTIALS- for passing process credentials (uid and gid).
    - You need to set the **SO_PASSCRED** socket option with *setsockopt()* on the receiving side.

- **SCM** stands for : Socket Control Message ,and not Software configuration management :-)

- These credentials are passed via a cred struct in a control message:

kernel: in include/linux/socket.h:

```
struct ucred {
__u32   pid;     /* process ID of the sending process  */
__u32   uid;    /* user ID of the sending process        */
__u32 gid;    /* group ID of the sending process      */
};
```

For user space apps, it is in /usr/include/bits/socket.h

Unix domain client example

```
const char* const socket_name = "/tmp/server";
int socket_fd;
int res;
struct sockaddr_un remote;
socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);
memset(&remote, 0, sizeof(remote));
remote.sun_family = AF_LOCAL;
strcpy(remote.sun_path, socket_name);
res = connect(socket_fd, (struct sockaddr*)&remote, SUN_LEN(&remote));
if (res<0)
        perror("connect");
res = sendto(socket_fd,"aaa", 3, 0, (struct sockaddr*)&remote, sizeof(remote));
```

- If we will try to call *send()* in a stream-oriented socket after the stream-oriented server was closed, we will get EPIPE error:
  - send: Broken pipe
- The kernel also sends the user space a SIGPIPE signal in this case.
- In case the flags parameter in *send()* is **MSG_NOSIGNAL**, the kernel does NOT send a SIGPIPE signal.
- In BSD, you can avoid signals by setsockopt() with SO_NOSIGPIPE (SOL_SOCKET option).

- In IPV4, the only signal used is SIGURG for OOB in tcp.
- In case of datagram-oriented sockets, SIGPIPE is not sent; we just get connection refused error.

- If, in the above example, we tried to create a dgram client instead of stream client, thus;
  - socket_fd = socket(PF_LOCAL, SOCK_DGRAM, 0);
  - We would get:
  - connect: Protocol wrong type for socket (EPROTOTYPE)
    - see: *unix_find_other()*
- The *socketpair()* system call:
- Creates a pair of connected sockets.
- On Linux, the only supported domain for this call is **AF_UNIX** (or synonymously, **AF_LOCAL**).

Netlink sockets

- **Netlink sockets:** a message mechanism from user-space to kernel and also between kernel ingredients.
- Used widely in the kernel; mostly in networking, but also in other subsystems.
  - There are other mechanism for communication from user space to kernel:
    - ioctls (drivers)
    - /proc or /sys entries (VFS)
  - And there are of course signals from kernel to user space (like SIGIO, and more).

- Creating netlink sockets is done (in the kernel) by *netlink_kernel_create().*

- For example, in net/core/rtnetlink.c:

```
static int rtnetlink_net_init(struct net *net)
{
struct sock *sk;
sk = netlink_kernel_create(net, NETLINK_ROUTE,
   RTNLGRP_MAX,  rtnetlink_rcv, &rtnl_mutex,
   THIS_MODULE);
```

- With generic netlink sockets, this is done using the NETLINK_GENERIC protocol thus:
- *netlink_kernel_create(&init_net, NETLINK_GENERIC, 0, genl_rcv, &genl_mutex, THIS_MODULE);*
  - *See net/netlink/genetlink.c*

- *The second parameter, NETLINK_ROUTE, is the protocol. (kernel 2.6.30).*
- There are currently 19 netlink protocols in the kernel:

NETLINK_ROUTE NETLINK_UNUSED NETLINK_USERSOCK

NETLINK_FIREWALL NETLINK_INET_DIAG NETLINK_NFLOG

NETLINK_XFRM NETLINK_SELINUX NETLINK_ISCSI

NETLINK_AUDIT NETLINK_FIB_LOOKUP NETLINK_CONNECTOR

NETLINK_NETFILTER NETLINK_IP6_FW NETLINK_DNRTMSG

NETLINK_KOBJECT_UEVENT NETLINK_GENERIC NETLINK_SCSITRANSPORT

NETLINK_ECRYPTFS

(see include/linux/netlink.h).

- The fourth parameter, *rtnetlink_rcv, is the handler for netlink packets.*
- *rtnetlink_rcv() gets a packet (sk_buff) as its parameter.*
- *NETLINK_ROUTE messages are not confined to the routing subsystem; they include also other types of messages (for example, neighboring)*
- *NETLINK_ROUTE messages can be divided into families. Most of these families has three types of messages. (New, Del and Get).*

- *For example:*
- *RTM_NEWROUTE – create a new route.*
  - *Handled by inet_rtm_newroute().*
- *RTM_DELROUTE - delete a route.*
  - *Handled by inet_rtm_delroute().*
- *RTM_GETROUTE – retrieve information about a route.*
  - *Handled by inet_dump_fib().*
- *All three methods are in net/ipv4/fib_frontend.c.*

- Another family of METLINK_ROUTE is the NEIGH family:
  - RTM_NEWNEIGH
  - RTM_DELNEIGH
  - RTM_GETNEIGH

- How do these messages reach these handlers?
- Registration is done by calling *rtnl_register()*
- in *ip_fib_init():*
  - rtnl_register(PF_INET, RTM_NEWROUTE, inet_rtm_newroute, NULL);
  - rtnl_register(PF_INET, RTM_DELROUTE, inet_rtm_delroute, NULL);
  - rtnl_register(PF_INET, RTM_GETROUTE, NULL, inet_dump_fib);

- IPROUTE2 package is based on rtnetlink.
- (IPROUTE2 is “ip” with subcommands, for example: *ip route show to show the routing tables*)
- IPROUTE2 uses the libnetlink library.
- See libnetlink.h (in the IPROUTE2 library)
- *rtnl_open()* to open a socket in user space.
- *rtnl_send()* to send a message to the kernel.

- *rtnl_open()* calls the *socket()* system call to create an rtnetlink socket:
  - *socket(AF_NETLINK, SOCK_RAW, protocol);*
- *rtnl_listen() starts receiving messages by calling the recvmsg() system call.*
- *The AF_NETLINK protocol is implemented in net/netlink/af_netlink.c.*
  - *AF_ROUTE is a synonym of AF_NETLINK (due to BSD)*
  - *#define AF_ROUTE AF_NETLINK (include/linux/socket.h)*
  - *The kernel holds an array called nl_table; it has up to 32 elements. (MAX_LINKS).*
  - *Each element in this table corresponds to a protocol (in fact, the protocol is the index)*

Example

```
#include "libnetlink.h"
int accept_msg(const struct sockaddr_nl *who, struct nlmsghdr *n, void *arg) {
        if (n->nlmsg_type == RTM_NEWROUTE)
        printf("got RTM_NEWROUTE message \n");
}
int main() {
        int res;
        struct rtnl_handle rth;
        unsigned int groups = ~RTMGRP_TC | RTNLGRP_IPV4_ROUTE;
        if (rtnl_open(&rth,groups) < 0) {
                printf("rtnl_open() failed in %s %s\n",__FUNCTION__,__FILE__);
                return -1;
}
```

```
if (rtnl_listen(&rth,accept_msg, stdout)<0) {
        printf("failed in rtnl_listen()\n");
        return -1;
        }
}
```

Adding a route will be logged to stdout:

ip route add 10.0.0.10 via 10.10.10.11

will print:

        got RTM_NEWROUTE message

- In this case, the rtnl_open() invokes

        socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

- The example can be expanded also for RTM_DELROUTE, etc.

Generic Netlink

- The iw tools (wireless user space management) use the Generic Netlink API.
- This API is based on Netlink sockets.
- You register handlers in *nl80211_init()*
- *net/wireless/nl80211.c*

- For example, for wireless interfaces we have these messages and handlers:
- NL80211_CMD_GET_INTERFACE
- Handled by nl80211_dump_interface()
- NL80211_CMD_SET_INTERFACE
  - Handled by nl80211_set_interface()
- NL80211_CMD_NEW_INTERFACE
  - Handled by nl80211_new_interface()
- NL80211_CMD_DEL_INTERFACE
- Handled by nl80211_del_interface()

- In the wireless subsystem there are currently 35 messages, each with its own handler.
  - See appendix A.

- You can use the NETLINK_FIREWALL protocol for a netlink socket to catch packets in user space with the help of an iptables kernel module named ip_queue.ko.
- iptables -A OUTPUT -p UDP --dport 9999 -j NFQUEUE --queue-num 0
- The user space application uses libnetfilter_queue-0.0.17 API (which replaced the libipq lib).
- Netlink sockets usage: xorp, (routing daemons: http://www.xorp.org/) , iproute2, iw.

SCTP

- General:
  - Combines features of TCP and UDP.
  - Reliable (like TCP).
  - RFC 4960 (obsoletes RFC 2960).
  - Target: VoIP, telecommunications.
- People:
  - Randall Stewart (Cisco): co inventor, FreeBSD.
  - Peter Lei (Cisco)
  - Michael Tuxen (MacOS).

- Linux Kernel SCTP Maintainers:
    - Vlad Yasevich (HP)
    - Sridhar Samudrala (IBM).
- SCTP support in the Linux kernel tree is from versions 2.5.36 and following.
- Location in the kernel tree: net/sctp.

SCTP

- There are two types of SCTP sockets:
  - **One to one socket**
    - **socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP).**
    - Much like TCP connection.
  - **One to many socket**
    - **socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP).**
    - Much like UDP server with many clients.

- You need to have lksctp-tools to use SCTP in userspace applications.
- http://lksctp.sourceforge.net
  - In fedora,
    - lksctp-tools rpm.
    - lksctp-tools-devel rpm. (for /usr/include/netinet/sctp.h)

Future lectures

- Netfilter kernel implementation:
  - NAT and connection tracking; dnat, snat.
  - MASQUERADING.
  - Filter and mangle tables.
  - Netfilter verdicts.
  - The new generation: nftables
- Network namespaces (Containers / OpenVZ).
- DCCP
- Virtio

- IPVS/LVS (Linux Virtual Server).
- Bluetooth, RFCOMM.
- Multiqueues.
- LRO (Large Receive Offload)
- Multicasting.
- TCP protocol.

Appendix A : wireless messages

NL80211_CMD_GET_WIPHY, NL80211_CMD_SET_WIPHY,

NL80211_CMD_GET_INTERFACE,NL80211_CMD_SET_INTERFACE,
NL80211_CMD_NEW_INTERFACE,NL80211_CMD_DEL_INTERFACE,

NL80211_CMD_GET_KEY,NL80211_CMD_SET_KEY,NL80211_CMD_NEW_KEY,NL80211_CMD_DEL_KEY,

NL80211_CMD_SET_BEACON, NL80211_CMD_NEW_BEACON, NL80211_CMD_DEL_BEACON,

NL80211_CMD_GET_STATION, NL80211_CMD_SET_STATION, NL80211_CMD_NEW_STATION, NL80211_CMD_DEL_STATION,

NL80211_CMD_GET_MPATH, NL80211_CMD_SET_MPATH, NL80211_CMD_NEW_MPATH, NL80211_CMD_DEL_MPATH,

NL80211_CMD_SET_BSS, NL80211_CMD_GET_REG,

NL80211_CMD_SET_REG, NL80211_CMD_REQ_SET_REG,

NL80211_CMD_GET_MESH_PARAMS,NL80211_CMD_SET_MESH_PARAMS,

NL80211_CMD_TRIGGER_SCAN, NL80211_CMD_GET_SCAN,

NL80211_CMD_AUTHENTICATE,NL80211_CMD_ASSOCIATE, NL80211_CMD_DEAUTHENTICATE,
NL80211_CMD_DISASSOCIATE,

NL80211_CMD_JOIN_IBSS,NL80211_CMD_LEAVE_IBSS,

Appendix B : **Socket options**

- **Socket options by protocol:**

IP protocol (SOL_IP) 19 socket options:

IP_TOS	IP_TTL
IP_HDRINCL	IP_OPTIONS
IP_ROUTER_ALERT	IP_RECVOPTS
IP_RETOPTS	IP_PKTINFO
IP_PKTOPTIONS	IP_MTU_DISCOVER
IP_RECVERR	IP_RECVTTL
IP_RECVTOS	IP_MTU
IP_FREEBIND	IP_IPSEC_POLICY
IP_XFRM_POLICY	IP_PASSSEC
IP_TRANSPARENT	

Note: For BSD compatibility there is IP_RECVRETOPTS (which is identical to IP_RETOPTS).

- AF_UNIX:
  - SO_PASSCRED for AF_UNIX sockets.
  - Note:For historical reasons these socket options are specified with a SOL_SOCKET type even though they are PF_UNIX specific.
- UDP:
  - UDP_CORK (IPPROTO_UDP level).
- RAW:
  - ICMP_FILTER
- TCP:
  - TCP_CORK
  - TCP_DEFER_ACCEPT
  - TCP_INFO
  - TCP_KEEPCNT

  - TCP_KEEPIDLE
  - TCP_KEEPINTVL
  - TCP_LINGER2
  - TCP_MAXSEG
  - TCP_NODELAY
  - TCP_QUICKACK
  - TCP_SYNCNT
  - TCP_WINDOW_CLAMP
- AF_PACKET
  - PACKET_ADD_MEMBERSHIP
  - PACKET_DROP_MEMBERSHIP

Socket options for socket level:

SO_DEBUG

SO_REUSEADDR

SO_TYPE

SO_ERROR

SO_DONTROUTE

SO_BROADCAST

SO_SNDBUF

SO_RCVBUF

SO_SNDBUFFORCE

SO_RCVBUFFORCE

SO_KEEPALIVE

SO_OOBINLINE

SO_NO_CHECK

SO_PRIORITY

SO_LINGER

SO_BSDCOMPAT

Appendix C: tcp client

```
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <arpa/inet.h>

int main()
{
```

tcp client - contd.

```
struct sockaddr_in sa;
int sd = socket(PF_INET, SOCK_STREAM, 0);
if (sd<0)
        printf("error");
memset(&sa, 0, sizeof(struct sockaddr_in));
sa.sin_family = AF_INET;
sa.sin_port   = htons(853);
inet_aton("192.168.0.121",&sa.sin_addr);
if (connect(sd, (struct sockaddr*)&sa, sizeof(sa))<0) {
        perror("connect");
        exit(0);
        }
close(sd);
}
```

tcp client - contd.

- If on the other side (192.168.0.121 in this example) there is no TCP server listening on this port (853) you will get this error for the socket() system call:
  - connect: Connection refused.
- You can send data on this socket by adding, for example:

```
const char *message = "mymessage";

int length;

length = strlen(message)+1;

res = write(sd, message, length);
```

- write() is the same as send(), but with no flags.

Appendix D : ICMP options

- These are ICMP options you can set with setsockopt on RAW ICMP socket: (see /usr/include/netinet/ip_icmp.h)

ICMP_ECHOREPLY

ICMP_DEST_UNREACH

ICMP_SOURCE_QUENCH

ICMP_REDIRECT

ICMP_ECHO

ICMP_TIME_EXCEEDED

ICMP_PARAMETERPROB

ICMP_TIMESTAMP

ICMP_TIMESTAMPREPLY

ICMP_INFO_REQUEST

ICMP_INFO_REPLY

ICMP_ADDRESS

ICMP_ADDRESSREPLY

APPENDIX E: flags for send/receive

MSG_OOB

MSG_PEEK

MSG_DONTROUTE

MSG_TRYHARD - Synonym for MSG_DONTROUTE for DECnet

MSG_CTRUNC

MSG_PROBE - Do not send. Only probe path f.e. for MTU

MSG_TRUNC

MSG_DONTWAIT - Nonblocking io

MSG_EOR - End of record

MSG_WAITALL - Wait for a full request

MSG_FIN

MSG_SYN

MSG_CONFIRM - Confirm path validity

MSG_RST

MSG_ERRQUEUE - Fetch message from error queue

MSG_NOSIGNAL - Do not generate SIGPIPE

MSG_MORE0x8000 - Sender will send more.

Example: set and get an option

- This simple example demonstrates how to set and get an IP layer option:

```
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
int main()
 {
int s;
int opt;
int res;
int one = 1;
int size = sizeof(opt);
```

```
s = socket(AF_INET, SOCK_DGRAM, 0);
if (s<0)
        perror("socket");
res = setsockopt(s, SOL_IP, IP_RECVERR, &one, sizeof(one));
if (res==-1)
        perror("setsockopt");
res = getsockopt(s, SOL_IP, IP_RECVERR,&opt,&size);
if (res==-1)
        perror("getsockopt");
printf("opt = %d\n",opt);
close(s);
 }
```

Example: record route option

- This example shows how to send a record route option.

```
#define NROUTES 9


int main()
{
int s;
int optlen=0;
struct sockaddr_in target;
int res;
```

```
char rspace[3+4*NROUTES+1];
char buf[10];
target.sin_family = AF_INET;
target.sin_port=htons(999);
inet_aton("194.90.1.5",&target.sin_addr);
strcpy(buf,"message 1:");
s = socket(AF_INET, SOCK_DGRAM, 0);
if (s<0)
        perror("socket");
memset(rspace, 0, sizeof(rspace));
rspace[0] = IPOPT_NOP;
rspace[1+IPOPT_OPTVAL] = IPOPT_RR;
rspace[1+IPOPT_OLEN] = sizeof(rspace)-1;
```

```
rspace[1+IPOPT_OFFSET] = IPOPT_MINOFF;
optlen=40;
if (setsockopt(s, IPPROTO_IP, IP_OPTIONS, rspace,
 sizeof(rspace))<0)
{
        perror("record route\n");
        exit(2);
}
```

Appendix F: using packet raw socket

```
int main()
{
int s;
int n;
char buffer[2048];
unsigned char *iphdr;
unsigned char *ethhdr;
s = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP));
while (1)
{
printf("*******\n");
n = recvfrom(s, buffer, 2048, 0, NULL, NULL);
printf("n bytes read\n")
```

```
ethhdr = buffer;
printf("source MAC address = %02x:%02x:%02x:%02x:%02x:%02x\n",
ethhdr[0],ethhdr[1],ethhdr[2],
ethhdr[3],ethhdr[4],ethhdr[5]);


}
}
```

Tips

- To find out socket used by a process:
- ls -l /proc/[pid]/fd|grep socket|cut -d: -f3|sed 's/\[//;s/\]//'
- The number returned is the inode number of the socket.
- Information about these sockets can be obtained from
  - netstat -ae
- After starting a process which creates a socket, you can see that the inode cache was incremented by one by:
- more /proc/slabinfo | grep sock
- sock_inode_cache 476 485 768 5 1 : tunables 0 0 0 : slabdata 97 97 0
- The first number, 476, is the number of active objects.

END

-
- Thank you!
-
  - ramirose@gmail.com