Linux Kernel Networking – advanced topics (5)

Sockets in the kernel

Rami Rosen

ramirose@gmail.com

Haifux, August 2009

www.haifux.org

Linux Kernel Networking (5)- advanced topics

- Note:
- This lecture is a sequel to the following 4 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 (5)- advanced topics

3) Advanced Linux Kernel Networking -

IPv6 in the Linux Kernel lecture

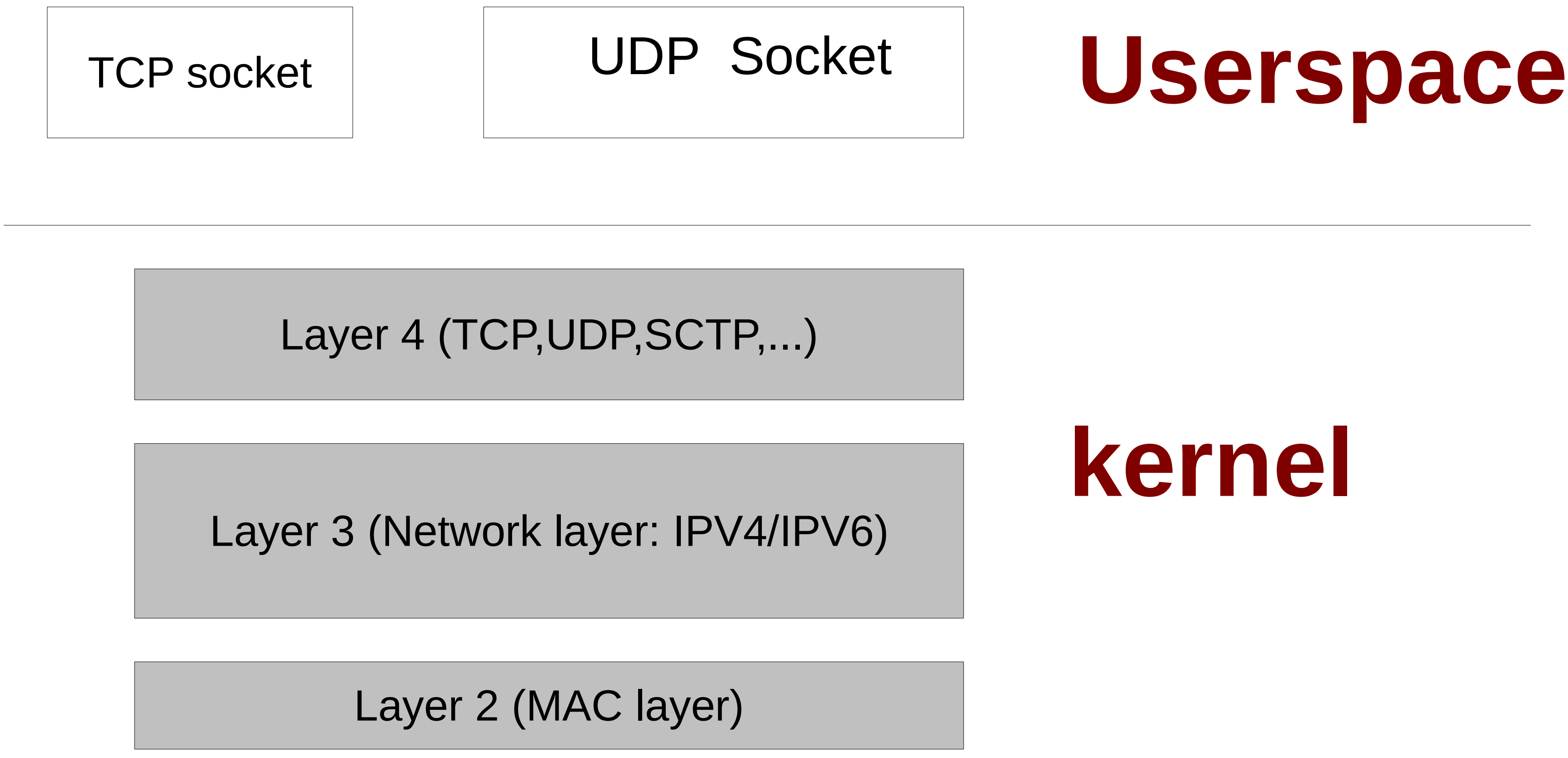
- http://www.haifux.org/lectures/187/
  - **Slides**: http://www.haifux.org/lectures/187/netLec3.pdf

4) Wireless in Linux

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

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

- Table of contents:
  - The *socket()* system call.
  - UDP protocol.
  - Control Messages.
  - Appendixes.
- Note: All code examples in this lecture refer to the recent **2.6.30** version of the Linux kernel.


TCP socket
UDP Socket
Userspace
Layer 4 (TCP,UDP,SCTP,...)
kernel
Layer 3 (Network layer: IPV4/IPV6)
Layer 2 (MAC layer)

- In user space, we have application, session and presentation layers(tcp/ip refers to all 3 as application layer)
- creating a socket **from user space** is done by the *socket()* system call:
  - int socket (int family, int type, int protocol);
  - From man 2 socket:
  - RETURN VALUE
  - On success, a file descriptor for the new socket is returned.
  - For open() system call (for files), we also get a file descriptor as the return value.
  - "Everything is a file" Unix paradigm.
- The first parameter, family, is also sometimes referred to as "domain".

- The **family** is PF_INET for IPV4 or PF_INET6 for IPV6.
  - The family is PF_PACKET for Packet sockets, which operate at the device driver layer. (Layer 2).
- pcap library for Linux uses PF_PACKET sockets:
  - pcap library is in use by sniffers such as tcpdump.
- Also hostapd uses PF_PACKET sockets:
- (hostapd is a wireless access point management project)
- From hostapd:
  - drv->monitor_sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

- Type:
  - SOCK_STREAM and SOCK_DGRAM are the mostly used types.
    - SOCK_STREAM for TCP, SCTP, BLUETOOTH.
    - SOCK_DGRAM for UDP.
    - SOCK_RAW for RAW sockets.
    - There are cases where protocol can be either SOCK_STREAM **or** SOCK_DGRAM; for example, Unix domain socket (AF_UNIX).
  - Protocol:usually 0 (IPPROTO_IP is 0, see: include/linux/in.h).
  - For SCTP, the protocol is **IPPROTO_SCTP**:
    - sockfd=socket(AF_INET, SOCK_STREAM,**IPPROTO_SCTP**);

- For bluetooth/RFCOMM:
- socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

- SCTP: Stream Control Transmission Protocol.
- For every socket which is created by a userspace application, there is a corresponding **socket** struct and **sock** struct in the kernel.
- This system call eventually invokes the *sock_create() method in the kernel.*
  - *An instance of struct socket is created* *(include/linux/net.h)*
  - *struct **socket** has only 8 members; struct **sock** has more than 20, and is one of the biggest structures in the networking stack. You can easily be confused between them. So the convention is this:*
  - ***sock** always refers to struct socket.*
  - ***sk** always refers to struct sock.*

struct sock: (include/net/sock.h)

```
struct sock {
...
        struct socket        *ssocket;
}
```

struct socket (include/linux/net.h)

```
struct socket {
socket_state              state;
short                             type;
unsigned long                  flags;
struct fasync_struct       *fasync_list;
wait_queue_head_t       wait;
struct file                      *file;
struct sock                     *sk;
const struct proto_ops     *ops;
```

- The state can be
  - SS_FREE
  - SS_UNCONNECTED
  - SS_CONNECTING
  - SS_CONNECTED
  - SS_DISCONNECTING
- These states are not layer 4 states (like TCP_ESTABLISHED or TCP_CLOSE).
- The sk_protocol member of struct sock equals to the third parameter (protocol) of the *socket()* system call.

- struct proto_ops (interface of struct socket)

	inet_stream_ops (i.e., TCP sockets)	**inet_dgram_ops** (i.e., UDP sockets)	**inet_sockraw_ops** (i.e., RAW sockets)
.family	PF_INET	PF_INET	PF_INET
.owner	THIS_MODULE	THIS_MODULE	THIS_MODULE
.release	inet_release	inet_release	inet_release
.bind	inet_bind	inet_bind	inet_bind
.connect	inet_stream_connect	inet_dgram_connect	inet_dgram_connect
.socketpair	sock_no_socketpair	sock_no_socketpair	sock_no_socketpair
.accept	inet_accept	sock_no_accept	sock_no_accept
.getname	inet_getname	inet_getname	inet_getname
.poll	tcp_poll	udp_poll	datagram_poll
.ioctl	inet_ioctl	inet_ioctl	inet_ioctl
.listen	inet_listen	sock_no_listen	sock_no_listen
.shutdown	inet_shutdown	inet_shutdown	inet_shutdown
.setsockopt	sock_common_setsockopt	sock_common_setsockopt	sock_common_setsockopt
.getsockopt	sock_common_getsockopt	sock_common_getsockopt	sock_common_getsockopt
.sendmsg	tcp_sendmsg	inet_sendmsg	inet_sendmsg
.recvmsg	sock_common_recvmsg	sock_common_recvmsg	sock_common_recvmsg
.mmap	sock_no_mmap	sock_no_mmap	sock_no_mmap
.sendpage	tcp_sendpage	inet_sendpage	inet_sendpage
.splice_read	tcp_splice_read	-	-

- Note: The inet_dgram_ops and inet_sockraw_ops differ only in the .poll member:
  - in inet_dgram_ops it is *udp_poll()*.
  - in inet_sockraw_ops, it is *datagram_poll().*

struct inet_sock

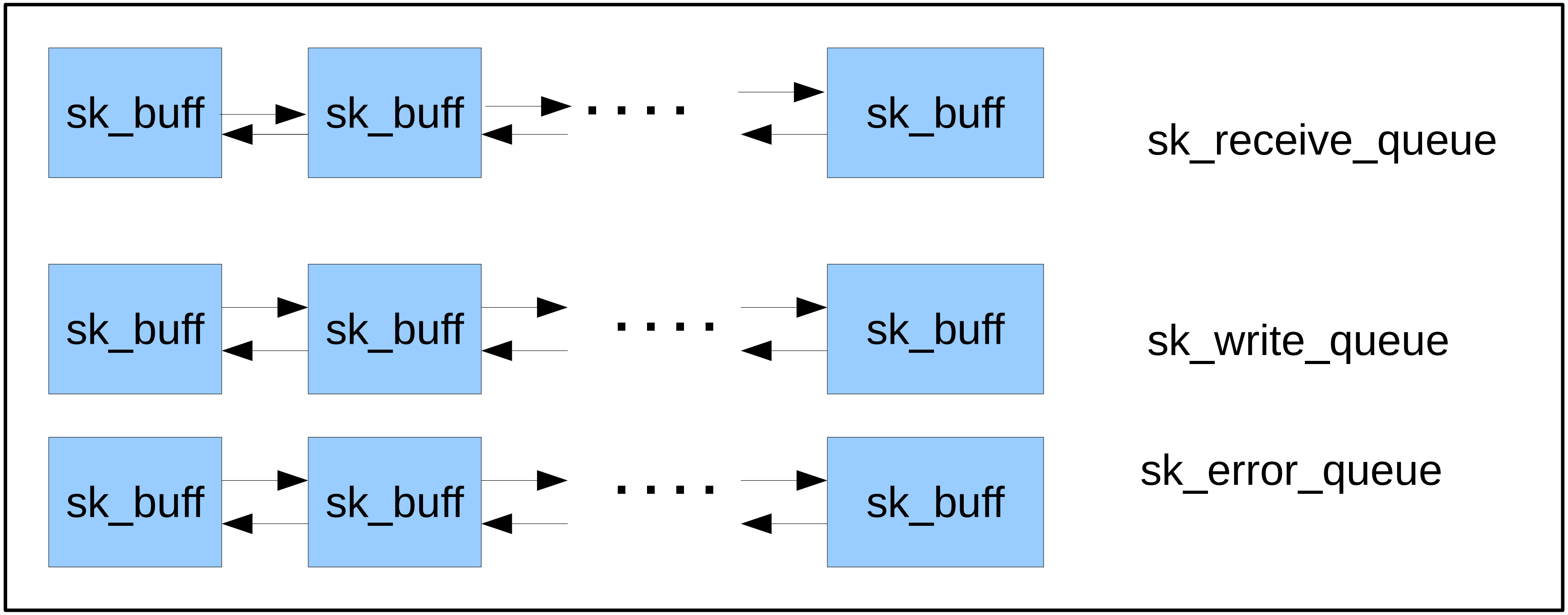
- Diagram:

struct sock (sk)

struct ip_options *opt;
__u8 tos;
__u8 recverr:1;
__u8 hdrincl:1;
.....

inet_sk(sock *sk) => returns the inet_sock which contains sk

- struct sock has three queues: rx , tx and err.



- **Each queue has a lock (spinlock)**

- *skb_queue_tail() : Adding to the queue*
- *skb_dequeue() : removing from the queue*
  - *With MSG_PEEK, this is done in two stages:*
  - *skb_peek()*
  - *__skb_unlink(). (to remove the sk_buff from the queue).*

- For the error queue: *sock_queue_err_skb()* adds to its tail (include/net/sock.h). Eventually, it also calls *skb_queue_tail().*

- Errors can be ICMP errors or EMSGSIZE errors.
- For more about errors,see APPENDIX F: UDP errors.

UDP and TCP

- No explicit connection setup is done with UDP.
  - In TCP there is a preliminary connection setup.
- Packets can be lost in UDP (there is no retransmission mechanism in the kernel). TCP on the other hand is reliable (there is a retransmission mechanism).
- Most of the Internet traffic is TCP (like http, ssh).
  - UDP is for audio/video (RTP)/streaming.
    - Note: streaming with VLC is by UDP (RTP).
    - Streaming via YouTube is tcp (http).

The udp header

- There are a very few UDP-based servers like DNS, NTP, DHCP, TFTP and more.
- For DHCP, it is quite natural to be UDP (Since many times with DHCP, you don't have a source address, which is a must for TCP).
- TCP implementation is much more complex
  - The TCP header is much bigger than UDP header.

The udp header: *include/linux/udp.h*

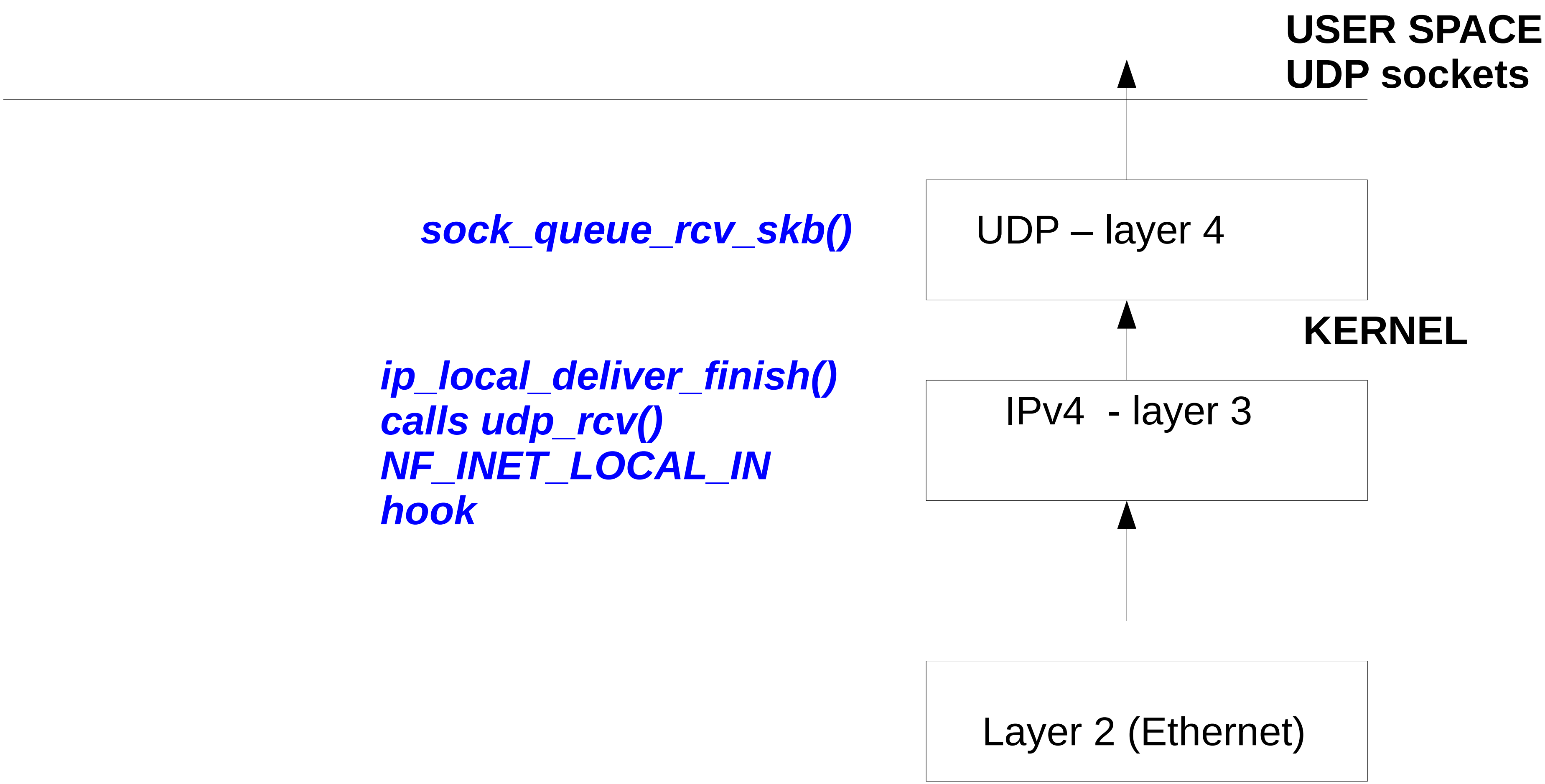
```
struct udphdr {
__be16	source;
__be16	dest;
__be16	len;
__sum16	check;
};
```

- UDP packet = UDP header + payload
- All members are 2 bytes (16 bits)

source port	dest port
len	checksum
Payload	

Receiving packets in UDP from kernel

- UDP kernel sockets can get traffic either from userspace or from kernel.



- From **user space**, you can receive udp traffic in three system calls:
  - *recv() (when the socket is connected)*
  - *recvfrom()*
  - *recvmsg()*
    - All three are handled by *udp_recvmsg()* in the kernel.
- *Note that fourth parameter of these 3 methods is flags; however, this parameter is NOT changed upon return. If you are interested in returned flags , you must use **only** recvmsg(), and to retrieve the msg.msg_flags member.*

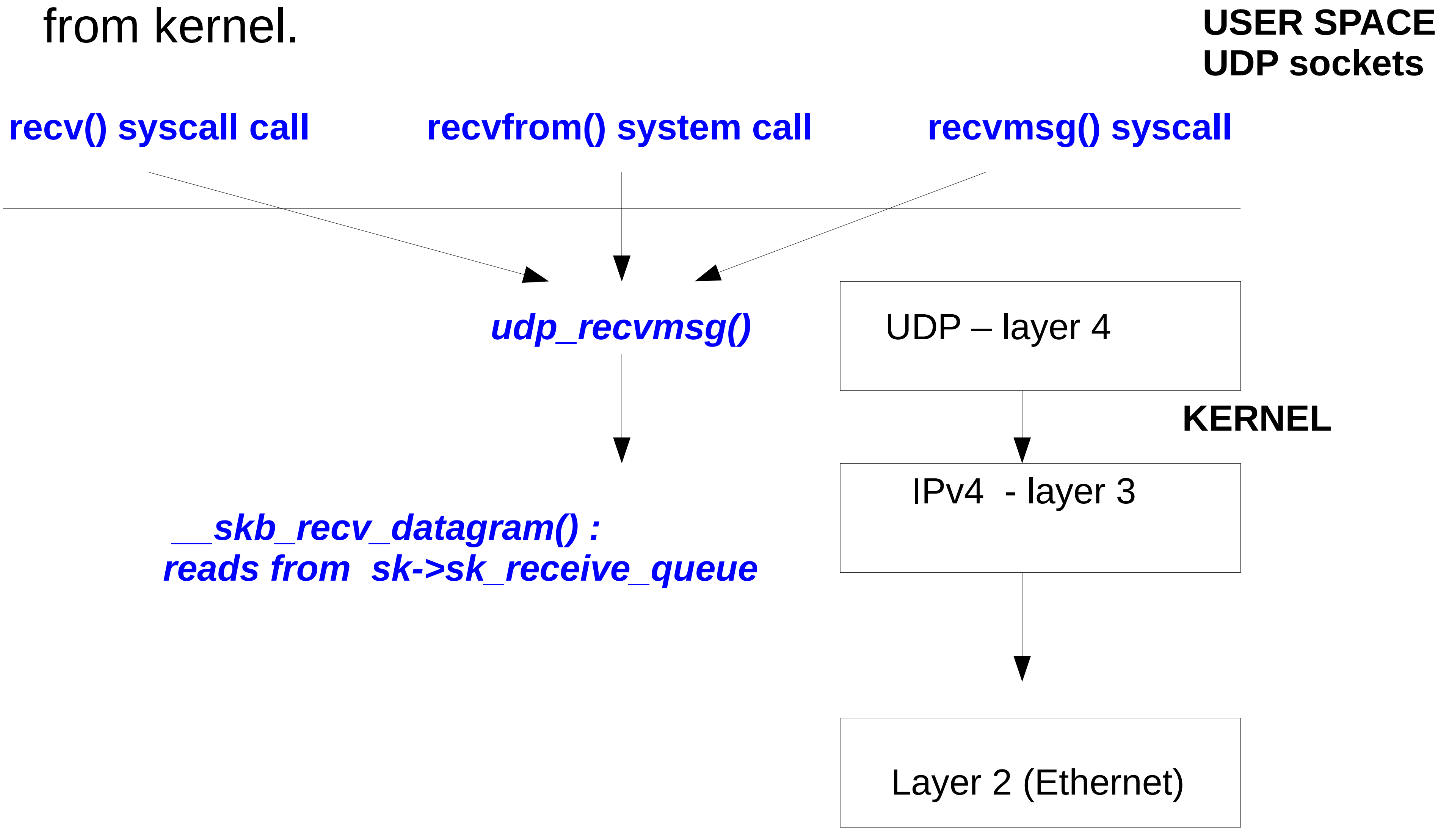
- For example, suppose you have a client-server udp applications, and the sender sends a packets which is longer then what the client had allocated for input buffer. The kernel than truncates the packet, and send **MSG_TRUNC** flag. In order to retrieve it, you should use something like:

```
recvmsg(udpSocket, &msg, flags);
if (msg.msg_flags & MSG_TRUNC)
    printf("MSG_TRUNC\n");
```

- There was a new suggestion recently for *recvmmsg() system call for receiving multiple messages (By Arnaldo Carvalho de Melo)*
- *The recvmmsg() will reduce the overhead caused by multiple system calls of recvmsg() in the usual case.*

Receiving packets in UDP from user space

- UDP kernel sockets can get traffic either from userspace or from kernel.



Receiving packets - udp_rcv()

- *udp_rcv()* is the handler for all UDP packets from the IP layer. It handles all incoming packets in which the protocol field in the ip header is IPPROTO_UDP (17) after ip layer finished with them.

See the udp_protocol definition: (net/ipv4/af_inet.c)

```
struct net_protocol udp_protocol = {
.handler =    udp_rcv,
.err_handler =   udp_err,
...
};
```

- In the same way we have :
  - *raw_rcv()* as a handler for raw packets.
  - *tcp_v4_rcv()* as a handler for TCP packets.
  - *icmp_rcv()* as a handler for ICMP packets.
- Kernel implementation: the *proto_register()* method registers a protocol handler.
  (net/core/sock.c)

udp_rcv() implementation:

- For broadcasts and multicast – there is a special treatment:

```
if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
return __udp4_lib_mcast_deliver(net, skb, uh,
                saddr, daddr, udptable);
```

- Then perform a lookup in a hashtable of struct sock.
  - Hash key is created from destination port in the udp header.
  - If there is no entry in the hashtable, then there is no sock listening on this UDP destination port => so send ICMP back: (of **port unreachable**).
  - icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);

udp_rcv()

- In this case, a corresponding SNMP MIB counter is incremented (UDP_MIB_NOPORTS).
- UDP_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
- You can see it by:

netstat -s

.....

Udp:

...

35 packets to unknown port received.

udp_rcv() - contd

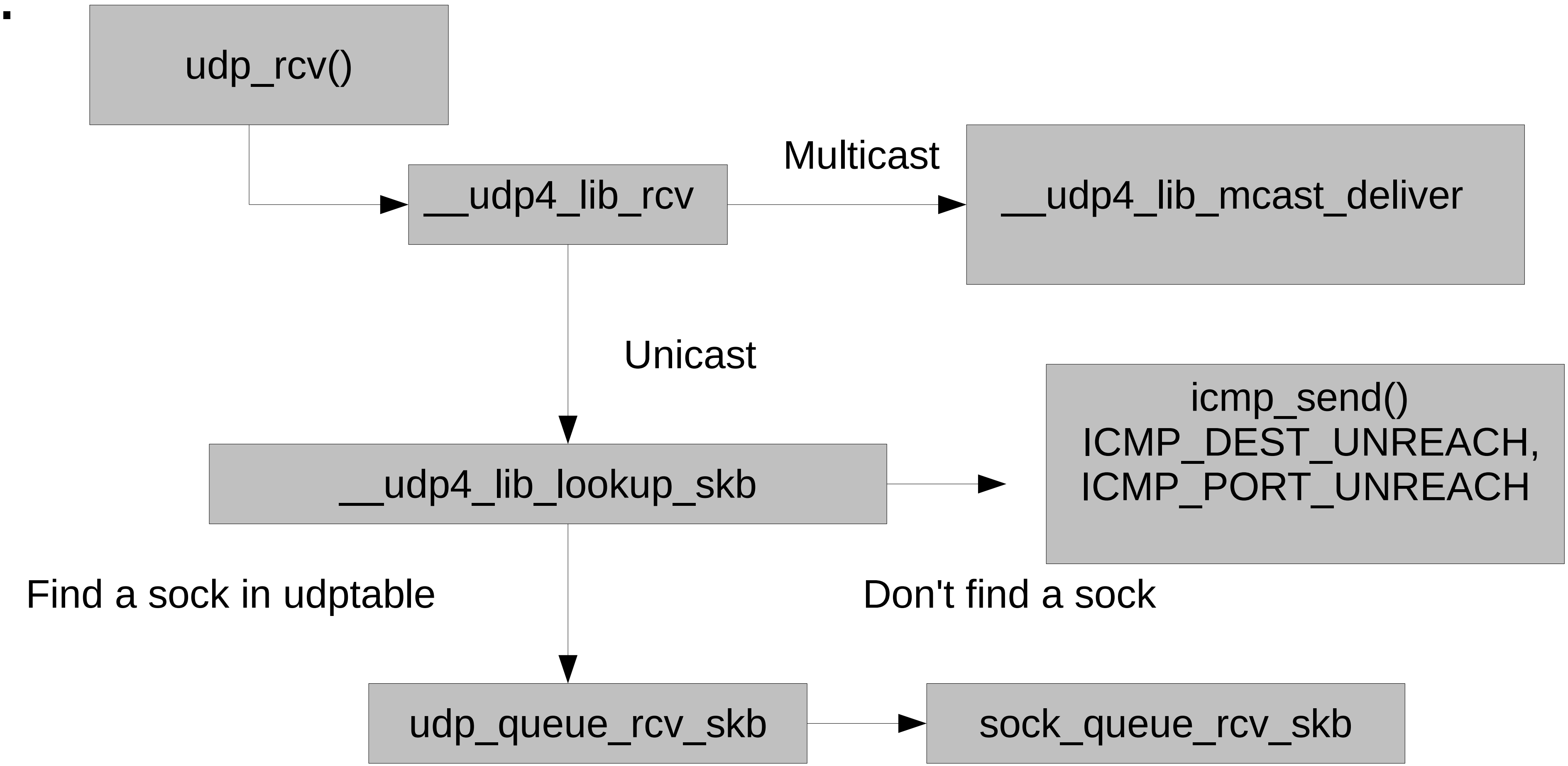
- Or, by:
- cat /proc/net/snmp | grep Udp:

Udp: InDatagrams NoPorts InErrors OutDatagrams RcvbufErrors SndbufErrors

Udp: 14 35 0 30 0 0

- If there is a sock listening on the destination port, call *udp_queue_rcv_skb().*
  - *Eventually calls sock_queue_rcv_skb().*
    - Which adds the packet to the **sk_receive_queue** by *skb_queue_tail()*

udp_rcv() diagram



- *udp_recvmsg():*
- Calls *__skb_recv_datagram()* , for receiving one sk_buff.
  - The *__skb_recv_datagram()* may block.
  - Eventually, what *__skb_recv_datagram()* does is read one sk_buff from the *sk_receive_queue* queue.
- *memcpy_toiovec()* performs the actual copy to user space by invoking *copy_to_user().*
- One of the parameters of *udp_recvmsg()* is a pointer to struct ***msghdr***. Let's take a look:

MSGHDR

From include/linux/socket.h:

```
struct msghdr {
    void     *msg_name;                /* Socket name          */
    int      msg_namelen;              /* Length of name       */
    struct iovec *msg_iov;             /* Data blocks          */
    __kernel_size_t  msg_iovlen;       /* Number of blocks     */
    void    *msg_control;
    __kernel_size_t  msg_controllen;  /* Length of cmsg list  */
    unsigned  msg_flags;
};
```

Control messages (ancillary messages)

- The msg_control member of msgdhr represent a control message.
  - Sometimes you need to perform some special things. For example, getting to know what was the destination address of a received packet.
    - Sometimes there is more than one address on a machine (and also you can have multiple addresses on the same nic).
  - How can we know the destination address of the ip header in the application?
  - struct **cmsghdr** (/usr/include/bits/socket.h) represents a control message.

- cmsghdr members can mean different things based on the type of socket.
- There is a set of macros for handling cmsghdr like CMSG_FIRSTHDR(), CMSG_NXTHDR(), CMSG_DATA(), CMSG_LEN() and more.
- There are no control messages for TCP sockets.

Socket options:

In order to tell the socket to get the information about the packet destination, we should call setsockopt().

- *setsockopt()* and *getsockopt()* - set and get options on a socket.
  - Both methods return 0 on success and -1 on error.
- Prototype: int setsockopt(int sockfd, **int level**, int optname,...

There are two levels of socket options:

To manipulate options at the sockets API level: SOL_SOCKET

To manipulate options at a protocol level, that protocol number

should be used;

  - for example, for UDP it is IPPROTO_UDP or SOL_UDP (both are equal 17) ; see include/linux/in.h and include/linux/socket.h
    - SOL_IP is 0.

- There are currently 19 Linux socket options and one another on option for BSD compatibility.

See Appendix B for a full list of socket options.

- There is an option called IP_PKTINFO.

– We will set the IP_PKTINFO option on a socket in the following example.

```
// from /usr/include/bits/in.h
#define IP_PKTINFO        8  /* bool */

/* Structure used for IP_PKTINFO.  */
struct in_pktinfo
  {
    int ipi_ifindex;                         /* Interface index  */
    struct in_addr ipi_spec_dst;     /* Routing destination address  */
    struct in_addr ipi_addr;             /* Header destination address  */
  };
```

```
const int on = 1;

sockfd = socket(AF_INET, SOCK_DGRAM,0);

if (setsockopt(sockfd, SOL_IP, IP_PKTINFO, &on,
   sizeof(on))<0)

      perror("setsockopt");

...

...

...
```

When calling recvmsg(), we will parse the msghr like this:

```
for (cmptr=CMSG_FIRSTHDR(&msg); cmptr!=NULL;
  cmptr=CMSG_NXTHDR(&msg,cmptr))
  {
  if (cmptr->cmsg_level == SOL_IP && cmptr->cmsg_type ==
    IP_PKTINFO)
    {
    pktinfo = (struct in_pktinfo*)CMSG_DATA(cmptr);
      printf("destination=%s\n", inet_ntop(AF_INET, &pktinfo->ipi_addr,
                  str, sizeof(str)));
  }
}
```

- In the kernel, this calls *ip_cmsg_recv()* in net/ipv4/ip_sockglue.c. (which eventually calls *ip_cmsg_recv_pktinfo()*).
- You can in this way retrieve other fields of the ip header:
  - For getting the TTL:
    - setsockopt(sockfd, SOL_IP, IP_RECVTTL, &on, sizeof(on))<0).
    - But: cmsg_type == IP_TTL.
  - For getting ip_options:
    - setsockopt() with IP_OPTIONS.

- Note: you cannot get/set ip_options in Java app.

Sending packets in UDP

- From **user space**, you can send udp traffic with three system calls:
  - *send() (when the socket is connected).*
  - *sendto()*
  - *sendmsg()*
    - All three are handled by *udp_sendmsg()* in the kernel.
    - *udp_sendmsg()* is much simpler than the tcp parallel method , *tcp_sendmsg().*
    - *udp_sendpage()* is called when user space calls sendfile() (to copy a file into a udp socket).
      - sendfile() can be used also to copy data between one file descriptor and another.

- *udp_sendpage() invokes udp_sendmsg().*
  - *udp_sendpage() will work only if the nic supports Scatter/Gather (NETIF_F_SG feature is supported).*

Example – udp client

```
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <string.h>

int main()
 {
      int s;
      struct sockaddr_in target;
      int res;
      char buf[10];
```

```
target.sin_family = AF_INET;
target.sin_port=htons(999);
inet_aton("192.168.0.121",&target.sin_addr);
strcpy(buf,"message 1:");
s = socket(AF_INET, SOCK_DGRAM, 0);
if (s<0)
  perror("socket");
res = sendto(s, buf, sizeof(buf), 0,(struct sockaddr*)&target,
    sizeof(struct sockaddr_in));
if (res<0)
       perror("sendto");
else
       printf("%d bytes were sent\n",res);
}
```

- For comparison, there is a tcp client in appendix C
- The source port of the UDP packet here is chosen randomly in the kernel.
- If I want to send from a specified port ?

You can bind to a specific source port (888 in this example) by adding:

```
source.sin_family          = AF_INET;
source.sin_port            = htons(888);
source.sin_addr.s_addr  = htonl(INADDR_ANY);
if (bind(s, (struct sockaddr*)&source, sizeof(struct sockaddr_in)) == -1)
    perror("bind");
```

- You **cannot** bind to privileged ports (ports lower than 1024) **when you are not root !**
  - Trying to do this will give:
  - “Permission denied” (**EPERM**).
  - You can enable non root binding on privileged port by running as root: (You will need at least a 2.6.24 kernel)
  - setcap 'cap_net_bind_service=+ep' udpclient
  - This sets the **CAP_NET_BIND_SERVICE** capability.

- You cannot bind on a port which is already bound.
  - Trying to do this will give:
  - “Address already in use” (**EADDRINUSE**)
- You cannot bind **twice or more** with the same UDP socket (even if you change the port).
  - You will get “bind: Invalid argument” error in such case **(EINVAL)**

- If you try *connect()* on an unbound UDP socket and then *bind()* you will also get the EINVAL error. The reason is that connecting to an unbound socket will call *inet_autobind()* to automatically bind an unbound socket (on a random port). So after connect(), the socket is bounded. And the calling bind() again will fail with EINVAL (since the socket is already bonded).
- Binding in the kernel for UDP is implemented in *inet_bind() and inet_autobind()*
  - *(in IPV6: inet6_bind())*

Non local bind

- What happens if we try to bind on a non local address ? (a non local address can be for example, an address of interface which is temporarily down)
  - We get EADDRNOTAVAIL error:
  - “bind: Cannot assign requested address.”
  - However, if we set /proc/sys/net/ipv4/ip_nonlocal_bind to 1, by
  - echo "1" > /proc/sys/net/ipv4/ip_nonlocal_bind
  - Or adding in /etc/sysctl.conf: net.ipv4.ip_nonlocal_bind=1
  - The *bind()* will succeed, but it may sometimes break applications.

- What will happen if in the above udp client example, we will try setting a broadcast address as the destination (instead of 192.168.0.121), thus: inet_aton("255.255.255.255",&target.sin_addr);
- We will get EACCESS error (“Permission denied”) for *sendto().*
- *In order that UDP broadcast will work, we have to add:*

```
int flag = 1;

if (setsockopt (s, SOL_SOCKET, SO_BROADCAST,&flag,
   sizeof(flag)) < 0)

    perror("setsockopt");
```

UDP socket options

- For **IPPROTO_UDP/SOL_UDP** level, we have two socket options:
- UDP_CORK socket option.
  - Added in Linux kernel 2.5.44.

```
int state=1;
setsockopt(s, IPPROTO_UDP, UDP_CORK, &state,
    sizeof(state));
 for (j=1;j<1000;j++)
   sendto(s,buf1,...)
 state=0;
setsockopt(s, IPPROTO_UDP, UDP_CORK, &state,
    sizeof(state));
```

- The above code fragment will call *udp_sendmsg()* 1000 times **without** actually sending anything on the wire (in the usual case, when without *setsockopt()* with UDP_CORK, 1000 packets will be send).
- Only after the second *setsockopt()* is called, with UDP_CORK and state=0, one packet is sent on the wire.
- Kernel implementation: when using UDP_CORK, *udp_sendmsg()* passes MSG_MORE to *ip_append_data()*.

- Implementation detail: UDP_CORK is not in glibc-header (/usr/include/netinet/udp.h); you need to add in your program:
- #define UDP_CORK 1

- UDP_ENCAP socket option.
  - For usage with IPSEC.
    - Used, for example, in ipsec-tools.
    - Note: UDP_ENCAP does not appear yet in the man page of udp (UDP_CORK does appear).
- Note that there are other socket options at the SOL_SOCKET level which you can get/set on UDP sockets: for example, SO_NO_CHECK (to disable checksum on UDP receive). (see Appendix E).

- SO_DONTROUTE (equivalent to MSG_DONTROUTE in send().
- The SO_DONTROUTE option tells “don't send via a gateway, only send to directly connected hosts.”
- Adding:
  - setsockopt(s, SOL_SOCKET, SO_DONTROUTE, val, sizeof(one)) < 0)
  - And sending the packet to a host on a different network will cause “Network is unreachable” error to be received. (ENETUNREACH)
  - The same will happen when MSG_DONTROUTE flag is set in *sendto()*.
- SO_SNDBUF.
- getsockopt(s, SOL_SOCKET, SO_SNDBUF, (void *) &sndbuf).

- Suppose we want to receive ICMP errors with the UDP client example (like ICMP destination unreachable/port unreachable).
- How can we achieve this ?
- First, we should set this socket option:
  - int val=1;
  - setsockopt(s, SOL_IP, **IP_RECVERR**,(char*)&val, sizeof(val));

- Then, we should add a call to a method like this for receiving error messages:

```
int recv_err(int s)

{

int res;

char cbuf[512];

struct iovec iov;

struct msghdr msg;

struct cmsghdr *cmsg;

struct sock_extended_err *e;

struct icmphdr icmph;

struct sockaddr_in target;
```

```
for (;;)
{
iov.iov_base = &icmph;
iov.iov_len  = sizeof(icmph);
msg.msg_name = (void*)&target;
msg.msg_namelen = sizeof(target);
msg.msg_iov     = &iov;
msg.msg_iovlen  = 1;
msg.msg_flags   = 0;
msg.msg_control = cbuf;
msg.msg_controllen = sizeof(cbuf);
res = recvmsg(s, &msg, MSG_ERRQUEUE | MSG_WAITALL);
```

```
  if (res<0)
continue;
for (cmsg = CMSG_FIRSTHDR(&msg);cmsg; cmsg =CMSG_NXTHDR(&msg, cmsg))
{
if (cmsg->cmsg_level == SOL_IP)
if (cmsg->cmsg_type == IP_RECVERR)
  {
  printf("got IP_RECVERR message\n");
  e = (struct sock_extended_err*)CMSG_DATA(cmsg);
  if (e)
    if (e->ee_origin == SO_EE_ORIGIN_ICMP) {
    struct sockaddr_in *sin = (struct sockaddr_in *)(e+1);
```

```
if ( (e->ee_type == ICMP_DEST_UNREACH) && (e->ee_code ==
ICMP_PORT_UNREACH) )

 printf("Destination port unreachable\n");

   }

  }

 }

}

}
```

udp_sendmsg()

- *udp_sendmsg*(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t len)
- Sanity checks in *udp_sendmsg():*

- The destination UDP port must not be 0.
- If we try destination port of 0 we get EINVAL error as a return value of *udp_sendmsg()*
  - The destination UDP is embedded inside the msghdr parameter (In fact, msg->msg_name represents a sockaddr_in; **sin_port** is sockaddr_in is the destination port number).
- MSG_OOB is the only illegal flag for UDP. Returns EOPNOTSUPP error if such a flag is passed. (only permitted to SOCK_STREAM)
- MSG_OOB is also illegal in AF_UNIX.

- OOB stands for “Out Of Band data”.
- The MSG_OOB flag is permitted in TCP.
  - It enables sending one byte of data in urgent mode.
  - (telnet , “ctrl/c” for example).
- The destination must be either:
  - specified in the msghdr (the **name** field in msghdr).
  - Or the socket is connected.
    - sk->sk_state == TCP_ESTABLISHED
      - Notice that though this is UDP, we use TCP semantics here.

Sending packets in UDP (contd)

- In case the socket is not connected, we should find a route to it; this is done by calling *ip_route_output_flow().*
- In case it is connected, we use the route from the sock (*sk_dst_cache* member of sk, which is an instance of *dst_entry*).
  - When the *connect()* system call was invoked, *ip4_datagram_connect()* find the route by *ip_route_connect()* and set *sk->sk_dst_cache* in *sk_dst_set()*
- Moving the packet to Layer 3 (IP layer) is done by *ip_append_data().*

- *In TCP, moving the packet to Layer 3 is done with ip_queue_xmit().*
  - *What's the difference ?*
- *UDP does not handle fragmentation; ip_append_data() does handle fragmentation.*
  - *TCP handles fragmentation in layer 4. So no need for ip_append_data().*

- *ip_queue_xmit() is (naturally) a simpler method.*
- Basically what the *udp_sendmsg()* method does is:
- Finds the route for the packet by *ip_route_output_flow()*
- Sends the packet with

  ip_local_out(skb)

Asynchronous I/O

- There is support for Asynchronous I/O in UDP sockets.
- This means that instead of polling to know if there is data (by *select()*, for example), the kernel sends a SIGIO signal in such a case.

- Using Asynchronous I/O UDP in a user space application is done in three stages:
  - 1) Adding a SIGIO signal handler by calling *sigaction()* system call
  - 2) Calling *fcntl()* with F_SETOWN and the pid of our process to tell the process that it is the owner of the socket (so that SIGIO signals will be delivered to it). Several processes can access a socket. If we will not call *fcntl()* with F_SETOWN, there can be ambiguity as to which process will get the SIGIO signal. For example, if we call fork() the owner of the SIGIO is the parent; but we can call, in the son, fcntl(s,F_SETOWN, getpid()).
  - 3) Setting flags: calling fcntl() with F_SETFL and O_NONBLOCK | FASYNC.

- In the SIGIO handler, we call *recvfrom()*.
- *Example:*

```
struct sockaddr_in source;
struct sigaction handler;
source.sin_family = AF_INET;
source.sin_port   = htons(888);
source.sin_addr.s_addr = htonl(INADDR_ANY);
servSocket = socket(AF_INET, SOCK_DGRAM, 0);
bind(servSocket,(struct sockaddr*)&source,sizeof(struct
    sockaddr_in));
```

```
handler.sa_handler = SIGIOHandler;
sigfillset(&handler.sa_mask);
handler.sa_flags = 0;
sigaction(SIGIO, &handler, 0);
fcntl(servSocket,F_SETOWN, getpid());
fcntl(servSocket,F_SETFL, O_NONBLOCK | FASYNC);
```

- *The fcntl() which sets the O_NONBLOCK | FASYNC flags invokes sock_fasync() in net/socket.c to add the socket.*
  - *The **SIGIOHandler()** method will be called when there is data (since a SIGIO signal was generated) ; it should call recvmsg().*

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_MORE 0x8000 - 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: UDP errors

Running :

cat /proc/net/snmp | grep Udp:

will give something like:

Udp: InDatagrams NoPorts InErrors OutDatagrams RcvbufErrors SndbufErrors

Udp: 2625 1 0 2100 0 0

InErrors - (UDP_MIB_INERRORS)

RcvbufErrors – UDP_MIB_RCVBUFERRORS:

- Incremented in *__udp_queue_rcv_skb()* (net/ipv4/udp.c).

SndbufErrors – (UDP_MIB_SNDBUFERRORS)

- Incremented in *udp_sendmsg()*

- Another metric:
  - cat /proc/net/udp
  - The last column in: drops
    - Represents sk->sk_drops.
    - Incremented in __udp_queue_rcv_skb()
      - net/ipv4/udp.c
- When do RcvbufErrors occur ?
  - The total number of bytes queued in sk_receive_queue queue of a socket is sk->sk_rmem_alloc.
  - The total allowed memory of a socket is sk->sk_rcvbuf.
    - It can be retrieved with getsockopt() using SO_RCVBUF.

- Each time a packet is received, the sk->sk_rmem_alloc is incremented by skb->truesize:
  - skb->truesize it the size (in bytes) allocated for the data of the skb plus the size of sk_buff structure itself.
  - This incrementation is done in skb_set_owner_r()

```
...
atomic_add(skb->truesize, &sk->sk_rmem_alloc);
...
```

    - see: include/net/sock.h

- When the packet is freed by kfree_skb(), we decrement ***sk->sk_rmem_alloc*** by ***skb->truesize***; this is done in *sock_rfree*():
- sock_rfree()

  ...

  atomic_sub(skb->truesize, &sk->sk_rmem_alloc);

  ...

  Immediately in the beginning of sock_queue_rcv_skb(), we have this check:

  if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=

  (unsigned)sk->sk_rcvbuf) {

  err = -ENOMEM;

- When returning -ENOMEM, this notifies the caller to drop the packet.
- This is done in *__udp_queue_rcv_skb()* method:

```
static int __udp_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
{
...
if ((rc = sock_queue_rcv_skb(sk, skb)) < 0) {
/* Note that an ENOMEM error is charged twice */
if (rc == -ENOMEM) {
UDP_INC_STATS_BH(sock_net(sk), UDP_MIB_RCVBUFERRORS,
        is_udplite);
atomic_inc(&sk->sk_drops);
```

- The default size of sk->sk_rcvbuf is SK_RMEM_MAX (sysctl_rmem_max).
- It equals to
- (sizeof(struct sk_buff) + 256) * 256
- See: SK_RMEM_MAX definition in net/core/sock.c
- This can be viewed and modified by:
  - */proc/sys/net/core/rmem_default* entry.
  - getsockopt()/setsockopt() with **SO_RCVBUF**.

- For the send queue *(sk_write_queue), we have in ip_append_data() a call to sock_alloc_send_skb(), which eventually invokes sock_alloc_send_pskb().*
- *In sock_alloc_send_pskb(), we peform this check:*

```
...
if (atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf)
...
```

- *If it is true, everything is fine.*
- *If not, we end with setting* ***SOCK_ASYNC_NOSPACE*** *and* **SOCK_NOSPACE** *flags of the socket:*

```
set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
```

- *In udp_sendmsg(), we check the SOCK_NOSPACE flag. If it is set, we increment the UDP_MIB_SNDBUFERRORS counter.*
- *sock_alloc_send_pskb() calls skb_set_owner_w().*
- *In skb_set_owner_w(), we have:*

```
...
atomic_add(skb->truesize, &sk->sk_wmem_alloc);
...
```

When the packet is freed by kfree_skb(), we decrement sk_wmem_alloc, in sock_wfree() method:

sock_wfree()

...

atomic_sub(skb->truesize, &sk->sk_wmem_alloc);

...

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