Linux Unleashed

This blog gives the Linux specific information like networking, debugging and Linux internal.

Raw socket, Packet socket and Zero copy networking in Linux

Introduction If you are a Linux enthusiast and just curious to know how the Ethernet frame is processed, how to sniff the packets even if...

Debugging core using gdb

Introduction Many times applications fails in certain scenario or crash in regression testing , This kind of problems are difficult to repr...

Raw socket in Linux

Get thread Id in Linux

Pthread library provides the call pthread_self to get task id but this id is not the same as linux provided thread id, Linux view all the th...

SUNDAY, NOVEMBER 14, 2010

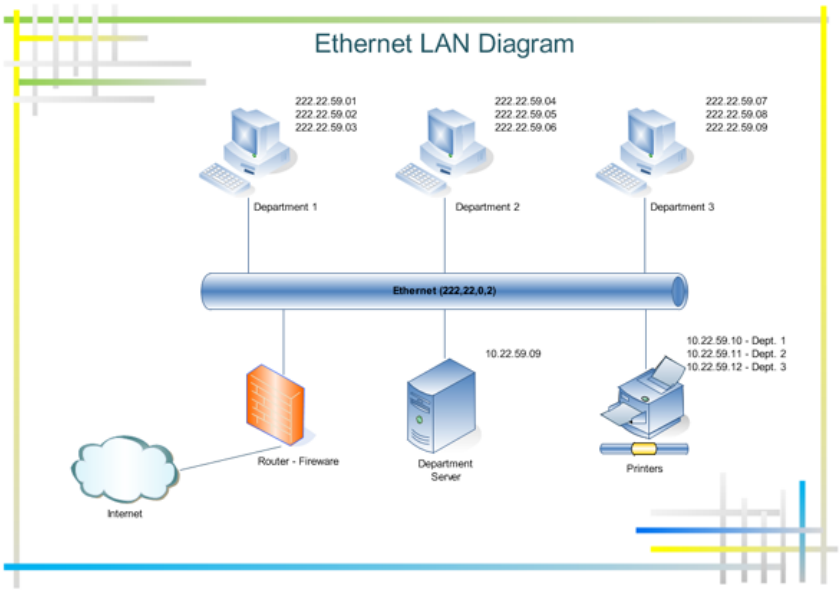
Raw socket, Packet socket and Zero copy networking in Linux

Introduction

If you are a Linux enthusiast and just curious to know how the Ethernet frame is processed, how to sniff the packets even if it is not destined for your computer then you are at the right place, You need to have basics of C and networking that's it.

Linux provides Packet sockets to sniff the link layer packet at the application, generally also known as raw sockets, but i would like to make a distinction here that packet socket are use to send and receive the packets at data link layer(layer 2) and where as raw sockets are use to send the raw packet till layer 3 and can only receive specific protocols like icmp at application layer, please refer the following blog for more detail on raw sockets rawSocket.

Lets have a brief introduction about Ethernet LAN and then we can move to the Linux specific support to sniff the packets, Ethernet segment is shared by all the host connected to same hub and the packet sent by one host is sent to all the host on the segment, The host on the LAN is identified by unique mac address, which is 6 bytes long.



As multiple host share the same Ethernet segment this might result into collision, thus to detect the collision and to re-transmit the packet Ethernet defines the minimum frame size of 64 bytes. There are three types of Ethernet

ABOUT ME

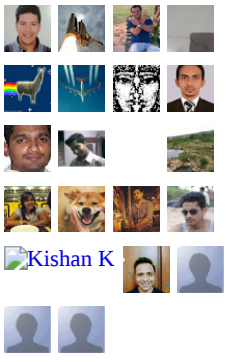
yusuf@linux

Hi All, I am a Linux consultant and a Linux community follower, Now the best of Linux features are just a click away, So enjoy reading on my blog and don't forget to provide your valuable comments

View my complete profile

FOLLOWERS

Followers (27) Next



Follow

frames possible

i)Unicast Frame

This is received by all the host on the hub but processed only by the host whose mac address match with destination mac address in the frame, other host just drops it.

ii)Multicast Frame

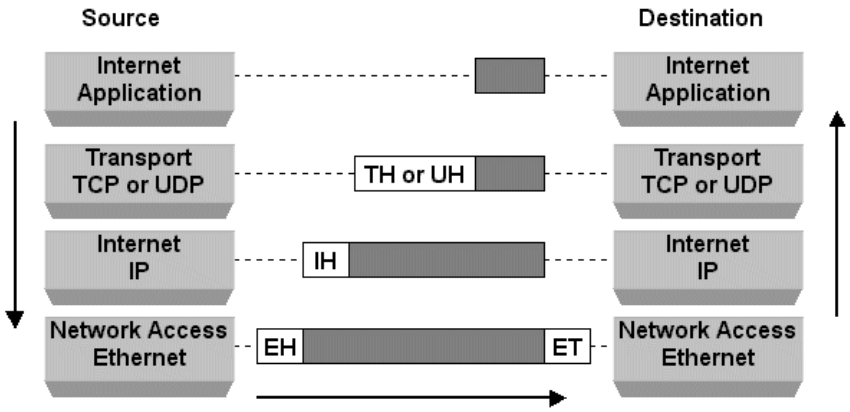
This received by all the host on the hub but processed by the host whose Ethernet controller is configured for the destination mac address in the frame, other host just drops it.

iii)Broadcast Frame

This received by all the host on the hub and processed by all the host.

Linux networking subsystem

Linux has well defined device model, Everything in Linux is a file, device is a special kind of file known as device file and each device has the major and the minor number associated with it but network devices are the exceptions and the Linux network interface are not based on device files, Network devices are identified by name like eth0...ethx and each device inside the kernel is identified by interface index, Mostly all the devices has the driver associated with it, which performs the basic I/O on the device, Network device also has the network driver, which receives and transmits the packets over the network using Ethernet controller(Mac and Phy), For high Ethernet rate like 1000 mbps link , DMA (Direct Memory Access) is use to transfer the data between Ethernet controller and the kernel buffer.Once the packet is in the kernel buffer the network driver is notified either by interrupt or may be by poller thread, Network driver fills the data in sk_buff, sk_buff is a structure in Linux networking stack,which represent the data along with lots of meta information, hence forth for further discussion we will refer it as skb. skb contains lots of meta information , which helps kernel to manage the network packet efficiently.



The networking stack inside the kernel is consist of layers and each layer does the well defined protocol processing, The driver receives the packet and pass it to the layer 3, depending on the layer 3 protocol type the corresponding handler is called and once the processing is done the layer 3 pass it to the layer 4 protocol handler, it could be udp or tcp, this is the very high level understanding of protocol processing in kernel ,there are lot more things involved in the processing like Net filter hooks and layer 2 ebtables hook, we are not going to talk about the filters and detailed processing because they deserve altogether a separate article,So lets look at the important information filed by the driver for further processing, Mostly driver will fill the below information in skb

i) Network data including Ethernet header
ii) Aligning of IP header to 16 byte boundary
iii) Setting of protocol,device info and pkt_type, here the protocol field contains the layer 2 protocol and pkt_type is the macro which determines whether the packet is layer 2 broadcast/multicast/unicast or destined for other host(promisc mode)

Once the above information is filled, driver makes a call to the netif_rx to enque the packet for soft irq, finally the packet lands up in netif_receive_skb softirq handler, if the packet socket has registered with protocol ETH_P_ALL then all packets are delivered to the socket, this is only true with ETH_P_ALL but when application registers with specific protocol like ETH_P_IP then linux only deliver the packets to specific sockets, Code can be found at following path http://lxr.linux.no/linux+v2.6.36/net/core/dev.c

Packet Socket

Traditionally the protocol specific processing is done inside the kernel and only the data is send/receive by the application, In this case application has no access to any of the protocol header because it is added/removed by the kernel, The IP packet contains Ethernet header, IP header and data, the final packet is constructed by appending the header for each layer, User only needs to put the data in socket, construction of the header is done inside the kernel.

On the other hand Packet socket is very powerful feature of Linux ,It allows to implement the protocols completely in application, including the link layer processing. Application can open the packet socket and can read the packets from the kernel, No special API's are required, normal socket API's works well with packet socket.

Packet socket completely bypass the kernel networking stack, it directly receives and sends the packet to link layer, the above figure give the brief overview of Linux packet socket sub-system, Packet socket is created using call **socket(PF_PACKET,SOCK_RAW,protocol)** , to open a socket you need to specify the domain, socket type and protocol, in this case the domain(family) is PF_PACKET, the socket type can be SOCK_RAW or SOCK_DGRAM, depending on the application requirement, if SOCK_DGRAM is used in packet socket then application receives the packet without ethernet header, If SOCK_RAW is used then application receives the complete frame include link layer header, socket types are defined in Protocol is use to filter only specific types of packet, like if protocol is ETH_P_IP then only IP packets are received, to receive all protocol packets, application can register with ETH_P_ALL protocol, Protocol Id's are defined in http://lxr.linux.no/linux+v2.6.36/include/linux/if_ether.h, If you have multiple interfaces then you can also bind the socket to particular interface to receive and send the packet. To receive the packets which are not destined to the local host ,we need to set the Ethernet in promisc mode, Since packet socket may have serious security implications, only root user can create this kind of sockets.

We have so far covered the basics of packet sockets and I think its also worth mentioning about the**BPF (Berkeley packet filters)**, These are the filters used by the kernel to filter the packets on user based criteria, depending on the protocol used by application, it might receive all the packets, filters can be used to restrict the packet reception to achieve higher performance, To generate the BPF code there is a easy way out, thanks to tcpdump, which provide the BPF code in C format for the applied filter for example **tcpdump -dd ether proto 0x8100 -i eth0** will display the filter in C code as shown below

```
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00008100 },
{ 0x6, 0, 0, 0x00000060 },
{ 0x6, 0, 0, 0x00000000 },
```

This could be use in our code for filtering.

Socket options

Once packet socket created we need to set some socket options in order to get the desired behaviour, settings like putting the Ethernet interface in promisc mode and binding the socket to specific interface, Ethernet in promisc mode receives all kinds of packets ,even if not destined to local host (destination mac is different from local mac), Normally Ethernet will accept all the broadcast packets and unicast packets intended for it, accepts multicast packet only if enabled. If you want to see all the packets in Hub then you need to set the Ethernet in promisc mode, to do so the user need to have root permissions. We can set the socket in promisc mode using sockopt.
To bind or to set the interface in promisc mode we need to find out the interface number use in kernel

```
struct ifreq ifr;
strncpy ((char *) ifr.ifr_name, interface.c_str (), IFNAMSIZ);
ioctl (sockId, SIOCGIFINDEX, &ifr)
```

We use ifreq structure to fill the interface name and then ioctl to get the interface id, ifreq structure is use to configure the network devices in Linux, Once the interface index is found we can go ahead and attached socket to the specific interface

```
struct sockaddr_ll sll;
```

```
sll.sll_family = AF_PACKET;
sll.sll_ifindex = ifr.ifr_ifindex;
sll.sll_protocol = htons (protocol);
bind ( sockId, (struct sockaddr *) &sll, sizeof (sll) )
```

We fill up the the information like socket domain, protocol and the interface number in sockaddr_ll and use bind call to bind the socket.Then finally we set the interface to promisc mode, first we fill up the packet_mreq structure with interface index and interface flag(PACKET_MR_PROMISC) and finally pass it to the sockopt call.

```
struct packet_mreq      mr;
memset (&mr, 0, sizeof (mr));
mr.mr_ifindex = ifr.ifr_ifindex;
mr.mr_type = PACKET_MR_PROMISC;
setsockopt (sockId, SOL_PACKET,PACKET_ADD_MEMBERSHIP, &mr, sizeof (mr))
```

Once the above setting is done , we are ready to use the packet socket for sending and receiving the raw data.

Frame construction

The Ethernet header consist of 6 bytes destination address, 6 bytes source address and 2 bytes Ethernet protocol type, The other bytes in frame like start delimiter and FCS is added by the Ethernet H/w, Mac address is the unique H/W address to identify the host, here the destination mac address is the receivers mac address and source address is the senders address, packet type identify the layer 3 protocols like Ip, Arp contained in the Ethernet frame, Ip has value 0x0800 and Arp has 0x0806, they are defined in **http://lxr.linux.no/linux+v2.6.36/include/linux/if_ether.h,** Note the 4 byte is added extra after source mac addr if the packet is Vlan tagged, out of four bytes two bytes will be 0x8100 denoting Vlan packet and another 2 bytes will have Vlan id and pcp bits, immediately after Vlan tag the two bytes will denote the layer three protocol. Lets see how to construct layer 2 header in C code.

```
char buf[1522];
struct ethhdr  *eth;
eth = (struct ethhdr*) buf;
memcpy(eth->h_dest,dest_mac,ETH_ALEN);
memcpy(eth->h_source,src_mac,ETH_ALEN);
eth->h_proto = ETH_P_IP;
```

The logic above is very straight forward, we take the buffer of 1522 bytes, 1522 bytes because Ethernet packet including Vlan cannot cross beyond that, Then we type cast the start of the packet to Ethernet header i.e. struct ethhdr (declare in include/linux/if_ether.h) and then fill up the destination ,source mac address, source mac address can also be retrieved from kernel interface, please find complete code here, destination mac address normally is learned through the arp protcol but just to try something with packet sockets we can send the layer 2 boradcast packet. finally we fill the layer 3 protocol type as IP.

After 14 bytes of Ethernet header the layer 3 protocol header will start, depending on the layer 2 protocol type the layer 3 header is type casted, If its a IP protcol then Ethernet data will be type casted as struct iphdr.

```
struct iphdr *ip =  (struct iphdr*) (eth +1);
```

Further IP has many fields like protocol version(IPV4), the header length, protocol type within IP (it could be udp,tcp or icmp), tos(type of service for QOS), total length (including the IP header and IP data) and 2 bytes check sum. The most important field of iphdr is the source and destination IP address, IP address is used at layer 3 for routing information and also to identify the network domain, Please refer the complete code here to understand further. The IP packet has 2 bytes of checksum and it is defined by the standard, checksum needs to be correct otherwise the receiving IP stack will not accept the packet.

How to send packet

Once the socket is open and packet construction is done, sending of packet is just using the write command.

```
write(fd,buf,len);
```

write system call takes 3 parameters ,the socket descriptor, buffer pointer and the length of buffer.

How to receive packet

To receive we can use read system call, Once the packet is received the application protocol should process it further.

```
read(fd,buf,len);
```

read system call takes 3 parameters ,the socket descriptor, buffer pointer and the length of buffer.

Zero copy networking

So far we have read the normal packet socket details but packet socket provides one of the very powerful feature of Zero copy, Normal flow of packets involves the copying of packet from kernel space to user space and vice versa, switching of modes(kernel<>user) can be very expensive in real time designs, Unfortunately not much has been written about this feature of packet socket and through this article I will try to bridge this gap, This feature allows Kernel to share the buffer with application, Kernel and application both operate on the same buffer without any overhead of copying the data, synchronization is achieved through some status flags. To enable this feature, Kernel should be compiled with below configuration

```
CONFIG_PACKET=y
CONFIG_PACKET_MMAP=y
```

In Linux 2.4/2.6 if PACKET_MMAP is not enabled, the capture process is very inefficient. It uses very limited buffers and requires one system call to capture each packet, it requires two if you want to get packet's timestamp (like libpcap always does). In the other hand PACKET_MMAP is very efficient. PACKET_MMAP provides a size configurable circular buffer mapped in user space that can be used to either send or receive packets. This way reading packets just needs to wait for them, most of the time there is no need to issue a single system call. Concerning transmission, multiple packets can be sent through one system call to get the highest bandwidth. To use mmap , I would suggest to use libcap for portability reasons but this can also be done for understanding and learning low level implementation.

Below are the system calls involved with Zero copy configuration

```
socket() - Opening of packet socket
setsockopt() - Setting up the size of receive and transmit circular buffer
mmap() - map the kernel buffer in user space
poll() - wait for new incoming packets
send() - send the batch of packet
close() - close the socket
```

socket creation and the binding of the socket to interface is the same shown above, the important action is to allocation of RX ring buffer and TX ring buffer. To do this we need to use the setsockopt call as below

```
setsockopt(fd , SOL_PACKET , PACKET_RX_RING , (void*)&req , sizeof(req));
setsockopt(fd , SOL_PACKET , PACKET_TX_RING , (void*)&req , sizeof(req));
```

The most important argument in the above call is the request structure, which defines the ring buffer parameters

```
struct tpacket_req
{
unsigned int tp_block_size; /* Minimal size of contiguous block */
unsigned int tp_block_nr; /* Number of blocks */
unsigned int tp_frame_size; /* Size of frame */
unsigned int tp_frame_nr; /* Total number of frames */
};
```

This structure is defined in include/include/if_packet.h, The above call of setsockopt sets the circular buffer, which is unswapable memory in kernel, As this buffers are mapped to user space , the applications can directly read the packet and read the meta information like timestamp without using any system call, Frame are grouped in a block, block is a contigous memory and contains the frames. We need to specify the the number of blocks, sizeof block, number of frames spread across blocks and size of frame.

So if we need lets say a buffer of 16 frames of size 256 bytes each, then we can have below configuration

```
struct tpacket_req req;
req.tp_block_size =  2048
req.tp_block_nr   =  2
req.tp_frame_size =  256
req.tp_frame_nr   =  16
```

The idea above is to split the 16 frame into 2 blocks of 8 frames, So each block will contain 8 frames, there are 2 blocks of 2K bytes each, The number of blocks depends on the static array in kenrel for block pointers and total size of blocks depends on the memory available in the kernel , for better understanding please refer http://lxr.linux.no/linux+v2.6.36/Documentation/networking/packet_mmap.txt

Once the data-structure is initialized and ring buffers are allocated in the kernel, application can do the mmap to map the memory at user space, once the mmap is done user is ready to receive and the send the packet using Zero copy socket.

Note: To use the Zero copy, socket should be bound to an interface.

Conclusion

Packet socket can be very handy in designing real-time applications like Video streaming, Audio streaming, Ethernet Service OAM and SCTP protocol. Applications like video streaming and audio streaming requires real-time packet reception in order to provide noise-less service, these kind of real-time requirements can be achieved by the combination of BPF and Zero copy networking. New protocols like Ethernet OAM and SCTP can be developed at application layer without any kind of special support from kernel.

Posted by yusuf@linux at 8:19 AM

14 comments:

Jeff December 4, 2010 at 6:59 PM

Regarding zero-copy, it's not.

PF_PACKET is designed for more efficient bandwidth handling in network sniffers by removing the need for vigorous read/write system calls (as you noted). It works by installing an *additional* protocol handler on target network devices, meaning data packets will also be delivered to any other existing protocol handler (ETH_P_IP aka PF_INET) sockets in other applications (security issue). Furthermore, the NIC driver is unaware of the PF_PACKET framework and the kernel doesn't abstract skb memory allocation enough that the driver can DMA data directly into the PACKET_RX_RING. The NIC just allocates a random skb (and possibly DMAs data to it to avoid a copy), then passes the skb to the protocol handlers. If you review the PF_PACKET protocol handler implementation when a PACKET_RX_RING is installed, specifically net/packet/af_packet.c:tpacket_rcv(), you will notice that the protocol handler uses skb_copy_bits() to *copy* the data from the NIC allocated skb to the PACKET_RX_RING.

In order to achieve true zero copy using PF_PACKET, the NIC driver would need to be able to allocate skb's where the data segment was part of the PACKET_RX_RING naturally. It's probably not worth the effort to use DMA to copy an *already* allocated skb to the ring because it would inject latency into a framework designed for efficient sniffing, doing more harm than good.

Reply

yusuf@linux December 6, 2010 at 8:18 AM

HI Jeff,

Thanks for pointing this out, Yes you are right inside the kernel its not zero copy and even if the packet can be dma directly to skb, still it requires 1 copy to pkt_rx_ring, but i wrote this article from user point of view ,for users the most time consuming copy is from the kernel to user memory and it is the major bottleneck for real-time protocols at application layer, With this feature we avoid this most expensive copy and that's the reason i call it zero copy.

Reply

Steven December 23, 2010 at 5:47 PM

Great Article!

In reference to Zero Copy, is this what is used by PF_RING?

Reply

yusuf@linux February 2, 2011 at 2:30 AM

Steven sorry for very late reply, Yes PF_RING used for so called Zero Copy :-)

Reply

arjun February 18, 2011 at 7:02 AM

I m trying to implement a very light weight protocol for transferring files between nodes in a cluster system.
I m using Zero copy concept as described above. The problem is that i m facing packet losses. I just want to add reliability to it without using any retransmission scheme. Is it Possible??

I have tried to map a larger buffer on receiver side to prevent packet loss..
Please HELP!!!!

Thanks

Reply

Ricardo Tubío-Pardavila June 4, 2013 at 12:46 AM

Great article! Thanks for posting it...

However, I have a big issue when trying to implement a zero-copy socket following the instructions that you provide. The problem comes when I try to call twice to "setsockopt()" using the same socket, in order to get, with the first call, access to the TX_RING and, with the second one, access to the RX_RING (or viceversa). The first call to setsockopt() always works fine (either for accessing the TX_RING or the RX_RING), but the second call to setsockopt() always returns an error (-16 in accordance with perror(), i.e., EBUSY kernel signal). I could access both RINGS using TWO DIFFERENT SOCKETS, one for TX_RING and a different one for RX_RING.

Is this the normal approach or am I doing anything wrong?

Reply

Anand July 4, 2013 at 12:45 AM

Excellent article. Thanks.

Reply

jovidsilva July 21, 2013 at 9:51 AM

hi this is a nice article...i am in need of help in designing a zero copy ftp client and server using C in linux

Reply

westtrd August 15, 2013 at 1:14 PM

What's about joins to specific multicast groups?
I have tried this (setsockopt() succeeds) but no joins established.

Can you share your experience?

Thanks in advance.
Regards

Reply

John Papadopolulos March 7, 2014 at 11:17 AM

Nice post and really informative. Is there any way to filter pf_packets? Perhaps ebtables or is it a fact that packet socket cant be filtered at all?

Reply

Mayur Parmar March 8, 2016 at 1:40 AM

I want zero copy code any one have it ??

Reply

Athif Abdul Aziz March 22, 2016 at 11:53 AM

I have worked on PF_RING for quite sometime as part of network monitoring solutions. At the time, was not aware that AF_PACKET existed for the same purpose that almost seems to provide the same functionality as PF_RING.

There are a couple of doubts however:
1. Are there any test results to prove which one is better in performance ? Assume PACKET_MMAP is enabled
2. To use PACKET_MMAP, kernel has to be compiled with this flag. PF_RING does not require a kernel compile, just a kernel module compile. That makes it's more easy to use than AF_PACKET. Any drawback though ?
3. Setting BPF filters in PF_RING was simple, just enter the string containing the BPF expression and call the API. In AF_PACKET seems more complex to do

Reply

geek_ji November 8, 2016 at 3:53 AM

Hey Yusuf, Thanks for this amazing article. Helped me understand a lot of stuff.

Reply

AKASH MISHRA February 2, 2018 at 10:46 AM

Thanks alot Yusuf, really the expalnation are very crystal clear

Reply

Enter your comment...

Comment as: Google Accoun

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

BLOG ARCHIVE

November (4)