Linux Kernel Networking

by Rami Rosen

Rami Rosen website

This is a 189 pages document with a broad overview of Linux kernel networking. It reflects the most recent Linux kernel network git tree, net-next, and it is updated every week according to most recent changes in this tree.

- Going deep into design and implementation details as well as theory behind it
- Linux Kernel Networking info is scattered in too many places around the web; and sometimes this info is partial/not updated/missing. This document servers as a central document about this subject.

Note: when referring to ethtool or iproute2 in this doc, we are talking about the most recent git versions. In case you are suspecting that your ethtool or iproute utils do not support some functionality mentioned here, and you want nevertheless to try it, please try with the most recent git version of the package. Some features mentioned here are known not to work with non git versions of these packages.

Last update: March 2013

Though it is intended mostly for developers, I do hope and believe that administrators and researchers can get some advice here.

It is based on a my practical experience with Linux kernel networking and a series of lectures I gave in the Technion:

See:
Rami Rosen lectures

Please feel free send any feedback or questions or suggestions to Rami Rosen by sending email to: ramirose@gmail.com

I will try hard to answer each and every question (though sometimes it takes time).

Contents

Introduction

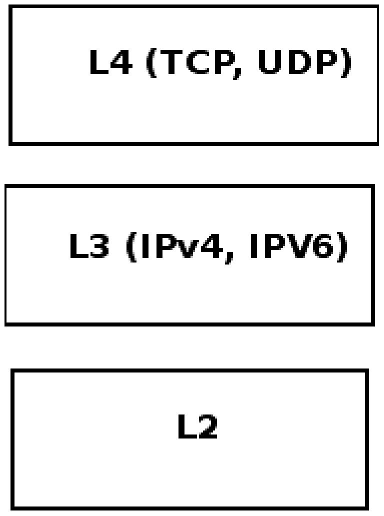
● Understanding a packet walkthrough in the kernel is a key to understanding kernel networking. Understanding it is a must if we want to understand Netfilter or IPSec internals, and more.

● We will deal with this walkthrough in this document (design and implementation details).

Hierarchy of networking layers:

• The layers that we will deal with (based on the 7 layers model) are:

- Link Layer (L2) (ethernet)
- Network Layer (L3) (ip4, ipv6)
- Transport Layer (L4) (udp,tcp...)



Networking Data Structures:

• The two most important structures of linux kernel network layer are:

– **sk_buff** struct (defined in *include/linux/skbuff.h*)

- **net_device** struct (defined in *include/linux/netdevice.h*)

It is better to know a bit about them before delving into the walkthrough code.

SK_BUFF structure

All network-related queues and buffers in the kernel use a common data structure, **struct sk_buff**. This is a large struct containing all the control information required for the packet (datagram, cell, whatever). The **sk_buff** elements are organized as a doubly linked list, in such a way that it is very efficient to move an sk_buff element from the beginning/end of a list to the beginning/end of another list. A queue is defined by **struct sk_buff_head**, which includes a head and a tail pointer to **sk_buff** elements.

All the queuing structures include an sk_buff_head representing the queue. For instance, **struct sock** includes a receive and send queue. Functions to manage the queues (*skb_queue_head(), skb_queue_tail(), skb_dequeue(), skb_dequeue_tail()*) operate on an sk_buff_head. In reality, however, the sk_buff_head is included in the doubly linked list of sk_buffs (so it actually forms a ring).

When a sk_buff is allocated, also its data space is allocated from kernel memory. sk_buff allocation is done with
alloc_skb() or *dev_alloc_skb()*; drivers use *dev_alloc_skb()*; (freeing the skb is done by *kfree_skb()* and dev_kfree_skb()). However, sk_buff provides an additional management layer. The data space is divided into a head area and a data area. This allows kernel functions to reserve space for the header, so that the data doesn't need to be copied around. Typically, therefore, after allocating an sk_buff, header space is reserved using ***skb_reserve()***. skb_pull(int len) – removes data from the start of a buffer (skipping over an existing header) by advancing data to data+len and by decreasing len.

We also handle alignment when allocating sk_buff:

```
 - when allocating an sk_buff, by netdev_alloc_skb(), we eventually
  call __alloc_skb() and in fact, we have two allocations here:
         - the sk_buff itself (struct sk_buff *skb)

         this is done by
         ...
         skb = kmem_cache_alloc_node(cache, gfp_mask &
~__GFP_DMA, node);
      ....
      see __alloc_skb() in net/core/skbuff.c

      the second is allocating data:
      ...
        size = SKB_DATA_ALIGN(size);
        data = kmalloc_node_track_caller(size + sizeof(struct
                    skb_shared_info),
                    gfp_mask, node);
            ...
```

see also __alloc_skb() in net/core/skbuff.c

the data is for packet headers (layer 2, layer 3 , layer 4) and packet data

Now, the data pointer is not fixed; we advance/decrease it as we move from layer to layer. The head pointer is fixed.

The allocation of data above forces alignment.

Now, when we call *netdev_alloc_skb()* from the network driver, the data points to the Ethernet header. The IP header follows immediately after the Ethernet header. Since the ethernet header is 14 bytes, this means that assuming data = kmalloc_node_track_caller() returned a 16-bytes aligned address, as mentioned above, the IP header will **not** be 16 bytes aligned. (it starts on data+14). In order to align it, we should advance data in 2 bytes before putting there the ethernet header. This is done by skb_reserve(skb, NET_IP_ALIGN);
NET_IP_ALIGN is 2, and what skb_reserve() does is increment data in 2 bytes. (let's ignore the increment of the tail, it is not important to this discussion)
So now the ip header is 16 bytes aligned.
see netdev_alloc_skb_ip_align() in include/linux/skbuff.h

The struct sk_buff objects themselves are private for every network layer. When a packet is passed from one layer to another, the struct sk_buff is cloned. However, the data itself is not copied in that case. Note that struct sk_buff is quite large, but most of its members are unused in most situations. The copy overhead when cloning is therefore limited.

- In most cases, sk_buff instances appear as “skb” in the kernel code.

struct sk_buff members:

Note: sk_buff members appear in the same order as in the header file, skbuff.h.

struct sk_buff *next;

struct sk_buff *prev;

ktime_t tstamp - time stamp of receiving the packet.

- ***net_enable_timestamp()*** must be called in order to get valid timestamp values.
  Helper method: static inline ktime_t skb_get_ktime(const struct sk_buff *skb) : returns tstamp of the skb.

struct sock *sk - The socket who owns the skb.

Helper method: static inline void skb_orphan(struct sk_buff *skb)

- If the skb has a destructor, call this destructor;
- set skb->sk and skb->destructor to null.

struct net_device *dev - The net_device on which the packet was received, or the net_device on which the packet will be transmitted.

char cb[48] __aligned(8) - control buffer for private variables.

Many network modules define a private skb cb of their own, and use the skb->cb for their own needs. For example, in include/net/bluetooth/bluetooth.h, we have:
#define bt_cb(skb) ((struct bt_skb_cb *)((skb)->cb))

unsigned long skb_refdst;

- helper method:
- static inline struct dst_entry *skb_dst(const struct sk_buff *skb)
- struct dst_entry *dst – the route for this sk_buff; this route is determined by the routing subsystem.
- It has 2 important function pointers:
  - ***int (*input)(struct sk_buff*);***
  - ***int (*output)(struct sk_buff*);***

- **input()** can be assigned to one of the following : ip_local_deliver, ip_forward, ip_mr_input, ip_error or dst_discard_in.
- **output()** can be assigned to one of the following :ip_output, ip_mc_output, ip_rt_bug, or dst_discard_out.
- We will deal more with dst when talking about routing.
- In the usual case, there is only one dst_entry for every skb.
- When using IPsec, there is a linked list of dst_entries and only the last one is for routing; all other dst_entries are for IPSec transformers ; these other dst_entries have the **DST_NOHASH** flag set. These entries , which has this **DST_NOHASH** flag set are not kept in the routing cache, but are kept instead on the flow cache.

struct sec_path *sp - used by IPSec (xfrm)

helper method:

static inline int secpath_exists(struct sk_buff *skb)

- returns 1 if sp is not NULL; defined in *include/net/xfrm.*

unsigned int len

unsigned int data_len;

Helper method: **static inline bool skb_is_nonlinear(const struct sk_buff *skb)**

returns data_len (when data_len is not 0, the skb is nonlinear).

__u16 mac_len: The length of the link layer (L2) header

hdr_len;

union {

__wsum csum;

struct {

__u16 csum_start;

__u16 csum_offset;

};

};

__u32 priority;

- Packet queueing priority
- by default the priority of the skb is 0.
- ***skb->priority***, in the TX path, is set from the socket priority (sk->sk_priority);

  See, for example, ***ip_queue_xmit()*** method in ip_output.c:

  skb->priority = sk->sk_priority;

  You can set sk_priority of sk by setsockopt; for example, thus:

  setsockopt(s, SOL_SOCKET, SO_PRIORITY, &prio, sizeof(prio))

  When we are forwarding the packet, there is no socket attached to the skb. Therefore, in ***ip_forward()***, we set skb->priority according to a special table, called ip_tos2prio; this table has 16 entries; see *include/net/route.h*

```
And we have
int ip_forward(struct sk_buff *skb)
{
 ...

 skb->priority = rt_tos2priority(iph->tos);
...
}
```

There are other cases when we set the priority of the skb.
For example, in ***vlan_do_receive()*** (net/8021q/vlan_core.c).

kmemcheck_bitfield_begin(flags1);

__u8 local_df:1,

cloned:1,

ip_summed:2,

nohdr:1,

nfctinfo:3;

__u8 pkt_type:3

- The packet type is determined in ***eth_type_trans()*** method.
- ***eth_type_trans()*** gets skb and net_device as parameters. (see net/ethernet/eth.c).
- The packet type depends on the destination mac address in the ethernet header.
- it is **PACKET_BROADCAST** for broadcast.
- it is **PACKET_MULTICAST** for multicast.
- it is **PACKET_HOST** if the destination mac address is mac address of the device which was passed as a parameter.
- It is **PACKET_OTHERHOST** if these conditions are not met.
- (there is another type for outgoing packets, **PACKET_OUTGOING, *dev_queue_xmit_nit()***)
- Notice that ***eth_type_trans()*** is unique to ethernet; for FDDI, for example, we have ***fddi_type_trans()*** (see net/802/fddi.c).

fclone:2,

ipvs_property:1,

peeked:1,

nf_trace:1 - netfilter packet trace flag

kmemcheck_bitfield_end(flags1);

__be16 protocol;

- skb->protocol is set in ethernet network drivers by assigning it to return value of ***eth_type_trans()***

void (*destructor)(struct sk_buff *skb);

Helper method:

static inline void skb_orphan(struct sk_buff *skb)

- If the skb has a destructor, call this destructor;
- set skb->sk and skb->destructor to null.

struct nf_conntrack *nfct;

struct sk_buff *nfct_reasm;

struct nf_bridge_info *nf_bridge

int skb_iif - The ifindex of device we arrived on. ***__netif_receive_skb()*** sets skb_iif to be the ifindex of the device on which we arrived, skb->dev.

__u32 rxhash;

- The rxhash of the skb is calculated in the receive path, in ***get_rps_cpu(),*** invoked from both from ***netif_receive_skb()*** and from ***netif_rx()***. The hash is calculate according to the source and dest address of the ip header, and the ports from the transport header.

__u16 vlan_tci - vlan tag control information; (2 bytes). Composed of ID and priority.

__u16 tc_index; /* traffic control index */

__u16 tc_verd; /* traffic control verdict */

__u16 queue_mapping;

kmemcheck_bitfield_begin(flags2);

__u8 ndisc_nodetype:2;

__u8 pfmemalloc:1;

__u8 ooo_okay:1;

__u8 l4_rxhash:1 - A flag which is set when we use 4-tuple hash over transport ports

__skb_get_rxhash() sets the rxhash.

__u8 wifi_acked_valid:1;

__u8 wifi_acked:1;

__u8 no_fcs:1;

__u8 head_frag:1;

__u8 encapsulation:1 - indicates that the skb contains encapsulated packet. This flag is set, for example, in ***vxlan_xmit()*** in the vxlan driver. (drivers/net/vxlan.c)

kmemcheck_bitfield_end(flags2);

dma_cookie_t dma_cookie;

__u32 secmark;

union {

__u32 mark;

__u32 dropcount;

__u32 avail_size;

};

sk_buff_data_t transport_header - the transport layer (L4) header (can be for example tcp header/udp header/icmp header, and more)

- Helper method: ***skb_transport_header().***

sk_buff_data_t network_header - network layer (L3) header

– (can be for example ip header/ipv6 header/arp header).

Helper method: ***skb_network_header(skb)***.

sk_buff_data_t mac_header; The Link layer (L2) header

- Helper method: ***skb_mac_header()***

sk_buff_data_t tail;

sk_buff_data_t end;

unsigned char *head,

unsigned char *data;

unsigned int truesize;

atomic_t users - a reference count. Initialized to 1. Increased in RX path for each protocol handler in ***deliver_skb()***. Decreased in ***kfree_skb().***

- Helper method: skb_shared() returns true if users > 1.
- Helper method: static inline struct sk_buff *skb_get(struct sk_buff *skb)

  Increments users.

Receive Packet Steering (rps)

There is a global table called **rps_sock_flow_table.**
Each call to ***recvmsg()*** or ***sendmsg()*** updates the rps_sock_flow_table by calling sock_rps_record_flow() which eventually calls ***rps_record_sock_flow()***.

struct rps_sock_flow_table has an array called "ents".
- The index to this array is a hash (sk_rxhash) of the socket (sock) from user space.
- The value of each element is the (desired) CPU.

Each call to send/receive from user space updates the CPU according to the CPU on which the call was done.

For example, in *net/ipv4/af_inet.c*:

```
int inet_recvmsg()
{
...
rps_record_sock_flow()
...
}
```

In *net/ipv4/tcp.c*:

```
ssize_t tcp_splice_read(struct socket *sock, loff_t *ppos,
struct pipe_inode_info *pipe, size_t len,
unsigned int flags)
{
...
sock_rps_record_flow(sk);
...
}
```

struct rps_flow_table is per rx queue. It is a member of
struct netdev_rx_queue, which represents an instance of an RX queue.

The number of entries in this table is **rps_flow_cnt.**

It can be set via:
echo *numEntries > /sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt*

RFS

RFS is Receive Flow Steering

Accelerated RFS

To work with Accelerated RFS (Accelerated Receive Flow Steering),

CONFIG_RFS_ACCEL should be set, and the driver should have support for it.

For example, look in ***drivers/net/ethernet/sfc/filter.c.*** Moreover, the driver should implement the ***ndo_rx_flow_steer*** callback of the ***net_device_ops***. The driver should also call ***rps_may_expire_flow()*** method.

XPS: Transmit Packet Steering

get_xps_queue() is called to determine which tx queue to use.
when CONFIG_XPS is not set, this method returns -1.
get_xps_queue() is called in the TX path,
from ***netdev_pick_tx()*** which is invoked by ***dev_queue_xmit().***

XPS code was written by Tom Herbert from google.

Under sysfs we have xps_cpus entry.
For example, for tx-0:
/sys/class/net/em1/queues/tx-0/xps_cpus

netdevice structure includes a member called **xps_dev_maps** which includes maps (xps_map) which are indexed by CPU.

net_device structure:

struct net_device members:

- char name[IFNAMSIZ];
  - The interface name, like eth0, eth1, p2p1, ...
  - Can be up to 16 characters (IFNAMSIZ).

struct hlist_node name_hlist - device name hash chain.

char *ifalias - snmp alias interface name; its length can be up to 256 (IFALIASZ)

- Helper method:
  - ***int dev_set_alias(struct net_device *dev, const char *alias, size_t len).***

unsigned long mem_end - shared mem end

unsigned long mem_start - shared mem start

unsigned long base_addr - device I/O address

unsigned int irq - device IRQ number (this is the irq number with which we call ***request_irq()***).

unsigned long state;

- A flag which can be one of these values:

```
__LINK_STATE_START
__LINK_STATE_PRESENT
__LINK_STATE_NOCARRIER
__LINK_STATE_LINKWATCH_PENDING
__LINK_STATE_DORMANT
```

struct list_head dev_list;

struct list_head napi_list;

struct list_head unreg_list;

netdev_features_t features - currently active device features.

Following are some examples for features:

NETIF_F_NETNS_LOCAL is set for devices that are not allowed to move between network namespaces; sometime these devices are named "local devices"; For example, for loopback device, ppp device, vxlan device and pimreg (multicast) device, we set **NETIF_F_NETNS_LOCAL**. If we try to move an interface whose **NETIF_F_NETNS_LOCAL** flag is set to a network namespace we created, we will get "RTNETLINK answers: Invalid argument" error message from dev_change_net_namespace() method (net/core/dev.c). See below in the Network namespaces section. Notice that for vxlan, we must set the NETIF_F_NETNS_LOCAL since vxlan works over UDP socket, and the UDP socket is part of the namespace it is created in. Moving the vxlan device does not move that state.

NETIF_F_VLAN_CHALLENGED is set for devices which can't handle with VLAN headers. (usually because of too large MTU due to vlan). For example, some types of Intel e100
(see e100_probe()in drivers/net/ethernet/intel/e100.c).

NETIF_F_VLAN_CHALLENGED is also set when creating a bonding, before enslaving first ethernet interface to it;

```
bond_setup()
{
....
bond_dev->features |= NETIF_F_VLAN_CHALLENGED;
...
```

in *drivers/net/bonding/bond_main.c*

This is done to avoid problems that occur when adding VLANs over an empty bond. See also later in the bonding section.

NETIF_F_LLTX is LockLess TX flag and is considered deprecated. When it is set, we don't use the generic TX lock (This is why it is called LockLess TX)

See the following macro (HARD_TX_LOCK) from in net/core/dev.c:

```
#define HARD_TX_LOCK(dev, txq, cpu) { \
if ((dev->features & NETIF_F_LLTX) == 0) { \
__netif_tx_lock(txq, cpu); \
} \
}
```

NETIF_F_LLTX is used in tunnel drivers like ipip, vxlan, veth, and in IPv4 over IPSec tunneling driver:

For example, in ipip tunnel (net/ipv4/ipip.c), we have:

```
ipip_tunnel_setup() {
...
dev->features |= NETIF_F_LLTX;
...
}
```

and in vxlan: (drivers/net/vxlan.c) we have:

```
static void vxlan_setup(struct net_device *dev) {
...
dev->features |= NETIF_F_LLTX;
..
}
```

in veth: (drivers/net/veth.c)

```
static void veth_setup(struct net_device *dev)
{
...
dev->features |= NETIF_F_LLTX;
...
}
```

and

also in the IPv4 over IPSec tunneling driver, net/ipv4/ip_vti.c, we have:

```
static void vti_tunnel_setup(struct net_device *dev) {
...
dev->features |= NETIF_F_LLTX;
...
}
```

NETIF_F_LLTX is also used in a few drivers which has their own Tx lock, like
drivers/net/ethernet/chelsio/cxgb:

in drivers/net/ethernet/chelsio/cxgb/cxgb2.c, we have:

```
static int __devinit init_one(struct pci_dev *pdev,
const struct pci_device_id *ent)
{
...
netdev->features |= NETIF_F_SG | NETIF_F_IP_CSUM |
NETIF_F_RXCSUM | NETIF_F_LLTX;
...
}
```

For the full list of net_device features, look in:
include/linux/netdev_features.h.

See more info in *Documentation/networking/netdev-features.txt* by Michal Miroslaw.

netdev_features_t hw_features - user-changeable features

hw_features should be set only in ndo_init callback and not changed later.

netdev_features_t wanted_features - user-requested features

netdev_features_t vlan_features - mask of features inheritable by VLAN devices.

int ifindex - Interface index. A unique device identifier.

- Helper method:
    - ***static int dev_new_index(struct net *net)***
- When creating a network device, ifindex is set.
- The ifindex is incremented by 1 each time we create a new network device.

This is done by the ***dev_new_index()*** method. (Since ifindex is an int, the method takes into account cyclic overflow of integer).

- The first network device we create, which is mostly always the loopback device, has ifindex of 1.
- You can see the ifindex of the loopback device by:
    - ***cat /sys/class/net/lo/ifindex***
- You can see the ifindex of any other network device, which is named netDeviceName, by:
    - ***cat /sys/class/net/netDeviceName/ifindex***

int iflink;

struct net_device_stats stats - device statistics, like number of rx_packets, number of tx_packets, and more.

atomic_long_t rx_dropped - dropped packets by core network

should not be used this in drivers. There are some cases when the stack increments the rx_dropped counter; for example, under certain conditions in ***__netif_receive_skb()***

const struct iw_handler_def * wireless_handlers;

struct iw_public_data * wireless_data;

const struct net_device_ops *netdev_ops;

net_device_ops includes pointers with several callback methods which we want to define in case we want to override the default behavior. net_device_ops object MUST be initialized (even to an empty struct) prior to calling ***register_netdevice()*** ! The reason is that in register_netdevice() we check if dev->netdev_ops->ndo_init exist *without* verifying before that dev->netdev_ops is not null. In case we won't initialize netdev_ops, we will have here a null pointer exception.

const struct ethtool_ops *ethtool_ops;

- Helper method: SET_ETHTOOL_OPS() macro – sets ethtool_ops for a net_device.

  This structure includes callbacks (including offloads). Management of ethtool is done in net/core/ethtool.c.

- Instead of forcing device drivers to provide empty ethtool_ops, there is a generic empty ethtool_ops named **default_ethtool_ops** (net/core/dev.c).

It was added in this patch, http://www.spinics.net/lists/netdev/msg210568.html, from Eric Dumazet.
You can get (ethtool user space tool) from:
http://www.kernel.org/pub/software/network/ethtool/

- The maintainer of ethtool is Ben Hutchings.
- Older versions are available in:
- http://sourceforge.net/projects/gkernel/
  or from this git repository:
  git://git.kernel.org/pub/scm/network/ethtool/ethtool.git

const struct header_ops *header_ops;

unsigned int flags - interface flags, you see or set from user space using ifconfig utility):

For example,
IFF_RUNNING, IFF_NOARP, IFF_POINTOPOINT, IFF_PROMISC, IFF_MASTER, IFF_SLAVE.

IFF_NOARP is set for tunnel devices for example. With tunnel devices, there is no need for sending ARP requests because you can connect only to the other device in the end of the tunnel. So we have, for example, in ipip_tunnel_setup() (*net/ipv4/ipip.c*),

```
static void ipip_tunnel_setup(struct net_device *dev)
{
...
dev->flags = IFF_NOARP;
...
}
```

IFF_POINTOPOINT is set for ppp devices.

For example, in *drivers/net/ppp/ppp_generic.c*, we have:

```
static void ppp_setup(struct net_device *dev)
{
...
dev->flags = IFF_POINTOPOINT | IFF_NOARP | IFF_MULTICAST;
...
}
```

IFF_MASTER is set for master devices (whereas IFF_SLAVE is set for slave devices).

For example, for bond devices, we have, in net/bonding/bond_main.c,

```
static void bond_setup(struct net_device *bond_dev)
{
bond_dev->flags |= IFF_MASTER|IFF_MULTICAST;
}
```

unsigned int priv_flags;

- These are flags you cannot see from user space with ifconfig or other utils.
- Some examples of priv_flags:
- **IFF_EBRIDGE** for a bridge interface.
- This flag is set in br_dev_setup() in net/bridge/br_device.c
- IFF_BONDING
- This flag is set in bond_setup() method.
- This flag is set also in bond_enslave() method.
- both methods are in drivers/net/bonding/bond_main.c.
- IFF_802_1Q_VLAN
- This flag is set in vlan_setup() in net/8021q/vlan_dev.c
- IFF_TX_SKB_SHARING
- In ieee80211_if_setup() , net/mac80211/iface.c we have:
- dev->priv_flags &= ~IFF_TX_SKB_SHARING;
- IFF_TEAM_PORT
  This flag is set in team_port_enter() method in drivers/net/team/team.c

- **IFF_UNICAST_FLT**
- Specifies that the driver handles unicast address filtering.
- In mv643xx_eth_probe(), drivers/net/ethernet/marvell/mv643xx_eth.c,
- ...
- dev->priv_flags |= IFF_UNICAST_FLT;
- ...

- The patch which added IFF_UNICAST_FLT: http://www.spinics.net/lists/netdev/msg172606.html
- **IFF_LIVE_ADDR_CHANGE**
- When this flag is set, we can change the mac address with eth_mac_addr() when the flag is set. Many drivers use eth_mac_addr() as the ndo_set_mac_address() callback of struct net_device_ops.
- see eth_mac_addr() in net/ethernet/eth.c.

unsigned short gflags;

unsigned short padded - How much padding added by ***alloc_netdev()***

unsigned char operstate - RFC2863 operstate

Can be one of the following:

IF_OPER_UNKNOWN
IF_OPER_NOTPRESENT
F_OPER_DOWN
IF_OPER_LOWERLAYERDOWN
IF_OPER_TESTING
IF_OPER_DORMANT
IF_OPER_UP

unsigned char link_mode - mapping policy to operstate.

unsigned char if_port - Selectable AUI, TP,...

unsigned char dma - DMA channel

unsigned int mtu - interface MTU value

Helper method: int eth_change_mtu(struct net_device *dev, int new_mtu)

Maximum Transmission Unit: the maximum size of frame the device can handle. RFC 791 sets 68 as a minimum for internet module MTU. The eth_change_mtu() method above does not permit setting mtu which are lower then 68. It should not be confused with path MTU, which is 576 (also according to RFC 791).

- int dev_set_mtu(struct net_device *dev, int new_mtu) - helper method to set new mtu. In case ndo_change_mtu is defined, we also call ndo_change_mtu of net_dev_ops. NETDEV_CHANGEMTU message is sent.

- Each protocol has mtu of its own; the default is 1500 for Ethernet.
- you can change the mtu from user space with ifconfig or with ip or via sysfs; for example,like this:

– ifconfig eth0 mtu 1400

– ip link set eth0 mtu 1400

- ***echo 1400 > /sys/class/net/eth0/mtu***

you can show the mtu of interface eth0 by:

ifconfig eth0

or by:

ip link show

or by:

cat /sys/class/net/eth0/mtu

- You cannot change it to values higher than 1500 on 10Mb/s network:
- *ifconfig eth0 mtu 1501*

will give: "SIOCSIFMTU: Invalid argument.

unsigned short type - interface hardware type.

type is the hw type of the device.

- For ethernet it is ARPHRD_ETHER
- In ethernet, the device type ARPHRD_ETHER is assigned in *ether_setup()*. see: net/ethernet/eth.c
- For ppp, the device type ARPHRD_PPP is assigned in ppp_setup(). see drivers/net/ppp/ppp_generic.c.
- For IPv4 tunnels, the type is ARPHRD_TUNNEL . For IPv6 tunnels, the type is ARPHRD_TUNNEL6 .

For example, for ip in ip tunnel in IPv4 (net/net/ipv4/ipip.c), we have:

```
static void ipip_tunnel_setup(struct net_device *dev) {
...
dev->type = ARPHRD_TUNNEL;
...
}
```

And for ip in ip tunnel in IPv6, we have:

```
static void ip6_tnl_dev_setup(struct net_device *dev)
{
...
dev->type = ARPHRD_TUNNEL6;
...
}
```

For example, in vti_tunnel_setup(), net/ipv4/ip_vti.c, we have

```
static void vti_tunnel_setup(struct net_device *dev) {
...
dev->type = ARPHRD_TUNNEL;
...
}
```

for tun devices, the type is ARPHRD_NONE. (see drivers/net/tun.c)

```
static void tun_net_init(struct net_device *dev) {
...
case TUN_TUN_DEV:
dev->type = ARPHRD_NONE;
...
}
```

unsigned short hard_header_len; This is the hardware header length. In case of ethernet, it is 14 (MAC SA + MAC DA + TYPE) . It is set to 14 (ETH_HLEN) in ether_setup():

```
void ether_setup(struct net_device *dev)
{
...
dev->hard_header_len = ETH_HLEN;
...
}
```

In case of tunnel devices, it is set to different values, according to the tunnel specifics. So in case of vxlan, we have, in drivers/net/vxlan.c

```
static void vxlan_setup(struct net_device *dev)
{
...
dev->hard_header_len = ETH_HLEN + VXLAN_HEADROOM;
...
}
```

where VXLAN_HEADROOM is size of IP header (20) + sizeof UDP header (20) + size of VXLAN header (8) + size of Ethernet header (14); so VXLAN_HEADROOM is 50 bytes in total.

With ipip tunnel we have in ipip_tunnel_setup(), (*net/ipv4/ipip.c*)

```
static void ipip_tunnel_setup(struct net_device *dev)
{
...
dev->hard_header_len = LL_MAX_HEADER + sizeof(struct iphdr);
...
}
```

/* extra head- and tailroom the hardware may need, but not in all cases

* can this be guaranteed, especially tailroom. Some cases also use

* LL_MAX_HEADER instead to allocate the skb.

*/

unsigned short needed_headroom;

unsigned short needed_tailroom;

/* Interface address info. */

unsigned char perm_addr[MAX_ADDR_LEN] - permanent hw address

unsigned char addr_assign_type - hw address assignment type.

By default, the mac address is permanent (NET_ADDR_PERM). In case the mac address was generated with a helper method called eth_hw_addr_random(), the type of the mac address is NET_ADD_RANDOM. There is also a type called NET_ADDR_STOLEN, which is not used. The type of the mac address is stored in addr_assign_type member of the net_device. Also when we change the mac address of the device, with eth_mac_addr(), we reset the addr_assign_type with ~NET_ADDR_RANDOM (in case it was marked as NET_ADDR_RANDOM before).

When we register a network device (in register_netdevice()), in case if the addr_assign_type is NET_ADDR_PERM, we set dev->perm_addr to be dev->dev_addr.

unsigned char addr_len - hardware address length

unsigned char neigh_priv_len;

unsigned short dev_id - for shared network cards.

spinlock_t addr_list_lock;

struct netdev_hw_addr_list uc - Unicast mac addresses.

Helper method: int dev_uc_add(struct net_device *dev, const unsigned char *addr)
Add a unicast address to the device; in case this address already exists,
increase the reference count.

Helper method: void dev_uc_flush(struct net_device *dev)

Flush unicast addresses of the device and zeroes the reference count.

struct netdev_hw_addr_list mc - Multicast mac addresses.

bool uc_promisc;

unsigned int promiscuity;

a counter of the times a NIC is told to set to work in promiscuous mode; used to enable more than one sniffing client; it is used also in the bridging subsystem, when adding a bridge interface; see the call to dev_set_promiscuity() in br_add_if(), net/bridge/br_if.c). ***dev_set_promiscuity()*** sets the **IFF_PROMISC** flag of the netdevice. Since promiscuity is an int, **dev_set_promiscuity()** takes into account cyclic overflow of integer.

unsigned int allmulti - a counter of allmulti.

Helper method: **dev_set_allmulti()** updates allmulti count on a device. Moreover, it sets (or removes) the **IFF_ALLMULTI** flag.

You set allmulti by:

ifconfig eth0 allmulti. (This invokes dev_set_allmulti() kernel method, with inc=1, in net/core/dev.c)

You remove allmulti by: ***ifconfig eth0 -allmulti***

(This invokes ***dev_set_allmulti()*** kernel method, with inc=-1, in *net/core/dev.c*)

"allmulti" counter in netdevice enables or disables all-multicast mode. When selected, all multicast packets on the network will be received by the interface.

Note that in case that the flags of the device did not include IFF_ALLMULTI (when enabling allmulti) or did not include ~IFF_ALLMULTI (when disabling allmulti) then we also call :

```
 dev_change_rx_flags(dev, IFF_ALLMULTI);
    dev_set_rx_mode(dev);
```

/* Protocol specific pointers */

struct vlan_info __rcu *vlan_info - VLAN info

struct dsa_switch_tree *dsa_ptr - dsa specific data

void *atalk_ptr - AppleTalk link

struct in_device __rcu *ip_ptr - IPv4 specific data

- This pointer is assigned to a pointer to struct in_device in inetdev_init() (*net/ipv4/devinet.c*)

struct dn_dev __rcu *dn_ptr - DECnet specific data

struct inet6_dev __rcu *ip6_ptr; /* IPv6 specific data

void *ax25_ptr - AX.25 specific data

struct wireless_dev *ieee80211_ptr - IEEE 802.11 specific data

should be assigned before registering.

unsigned long last_rx;

Time of last Rx; This should not be set in drivers, unless really needed, because network stack (bonding) use it if/when necessary, to avoid dirtying this cache line.

- The following patchset by Jiri Pirko suggested to remove master and netdev_set_master() from the network stack:
- *http://www.spinics.net/lists/netdev/msg220857.html*
  struct net_device *master and netdev_set_master() indeed were removed
- A list was added instead:

- **struct list_head upper_dev_list** /* List of upper devices */
  - struct netdev_upper also added.

unsigned char *dev_addr;

- The MAC address of the device (6 bytes).
- *dev_set_mac_address(struct net_device *dev, struct sockaddr *sa)* - helper method. Changes the mac address (dev_addr member) by invoking the ndo_set_mac_address() callback and sends **NETDEV_CHANGEADDR** notification. Many drivers use the ethernet generic eth_mac_addr() method (*net/ethernet/eth.c)* as the ndo_set_mac_address() callback.

struct netdev_hw_addr_list dev_addrs; - list of device hw addresses

unsigned char broadcast[MAX_ADDR_LEN] /* hw bcast add */

struct kset *queues_kset;

struct netdev_rx_queue *_rx;

unsigned int num_rx_queues

number of RX queues allocated at register_netdev() time

- **unsigned int real_num_rx_queues -**

  Number of RX queues currently active in device.

  netif_set_real_num_rx_queues() sets real_num_rx_queues and updates sysfs entries.

  (*/sys/class/net/deviceName/queues/**)

  Notice that ***alloc_netdev_mq()*** initializes num_rx_queues, real_num_rx_queues, num_tx_queues and real_num_tx_queues to the same value.

  You can set the number of tx queues and rx queues by "ip link" when adding a device.

  For example, if we want to create a vlan device with 6 tx queues and 7 rx
  queues, we can run:

  ip link add link p2p1 name p2p1.100 numtxqueues 6 numrxqueues 7 type vlan id 100

  Under corresponding sysfs p2p1.100 entry, we will see indeed
  7 rx queues (numbered from 0 to 6) and 6 tx queues (numbered from 0 to 5).

```
ls /sys/class/net/p2p1.100/queues
rx-0 rx-1 rx-2 rx-3 rx-4 rx-5 rx-6
tx-0 tx-1 tx-2 tx-3 tx-4 tx-5
```

```
/* CPU reverse-mapping for RX completion interrupts, indexed
* by RX queue number. Assigned by driver. This must only be
* set if the ndo_rx_flow_steer operation is defined. */
```

struct cpu_rmap *rx_cpu_rmap;

rx_handler_func_t __rcu *rx_handler;

Helper method:

```
netdev_rx_handler_register(struct net_device *dev,
                           rx_handler_func_t *rx_handler,
                           void *rx_handler_data)
```

rx_handler is set by ***netdev_rx_handler_register().***
It is used in bonding, team, openvswitch, macvlan, and bridge devices.

void __rcu *rx_handler_data;

rx_handler_data is also set by netdev_rx_handler_register(); see here above.

struct netdev_queue __rcu *ingress_queue;

/*

* Cache lines mostly used on transmit path

*/

struct netdev_queue *_tx ____cache line_aligned_in_smp;

unsigned int num_tx_queues; /* Number of TX queues allocated at ***alloc_netdev_mq()*** time */

/* Number of TX queues currently active in device */

unsigned int real_num_tx_queues;

Helper method:
netif_set_real_num_tx_queues() sets real_num_tx_queues and updates sysfs entries.

struct Qdisc *qdisc - root qdisc from userspace point of view.

dev_init_scheduler() method initializes qdisc
in **register_netdevice().**

unsigned long tx_queue_len;

- Max frames per queue allowed; can be set by ifconfig, for example:

ifconfig p2p1 txqueuelen 1600

spinlock_t tx_global_lock;

struct xps_dev_maps __rcu *xps_maps;

unsigned long trans_start; /* Time (in jiffies) of last Tx */

int watchdog_timeo; /* used by dev_watchdog() */

st**ruct timer_list watchdog_timer;**

int __percpu *pcpu_refcnt - Number of references to this device

helper methods:
static inline void dev_hold(struct net_device *dev)
increments reference count of the device.
static inline void dev_put(struct net_device *dev)
decrements reference count of the device.
int netdev_refcnt_read(const struct net_device *dev)
reads sum of all CPUs reference counts of this device.

/* delayed register/unregister */

struct list_head todo_list;

/* device index hash chain */

struct hlist_node index_hlist;

struct list_head link_watch_list;

/* register/unregister state machine */

enum { NETREG_UNINITIALIZED=0,

NETREG_REGISTERED, /* completed register_netdevice */

NETREG_UNREGISTERING, /* called unregister_netdevice */

NETREG_UNREGISTERED, /* completed unregister todo */

NETREG_RELEASED, /* called free_netdev */

NETREG_DUMMY, /* dummy device for NAPI poll */

} reg_state:8;

bool dismantle : device is going do be freed. This flag is set in ***rollback_registered_many()*** when unregistering a device. It is referenced for example in ***macvlan_stop().***

enum rtnl_link_state -

- rtnl_link_state can be RTNL_LINK_INITIALIZING or RTNL_LINK_INITIALIZED.
- When creating a new link, in ***rtnl_newlink()***, the rtnl_link_state is set to be RTNL_LINK_INITIALIZING (this is done by ***rtnl_create_link(),*** which is invoked from ***rtnl_newlink()***); later on, when calling ***rtnl_configure_link()***, the rtnl_link_state is set to be RTNL_LINK_INITIALIZED.

/* Called from unregister, can be used to call free_netdev */

void (*destructor)(struct net_device *dev);

struct netpoll_info *npinfo;

struct net *nd_net;

● nd_net: The network namespace this network device is inside.

dev_net_set(struct net_device *dev, struct net *net) - a helper method which sets the nd_net of net_device to the specified net namespace. (include/linux/netdevice.h)

- *dev_change_net_namespace()* - a helper method.
  move the network device to a different network namespace.
  If the **NETIF_F_NETNS_LOCAL** flag of the net device is set, the operation is not performed and an error is returned. Callers of this method must hold the rtnl semaphore. This method returns 0 upon success.

/* mid-layer private */

union {

void *ml_priv;

struct pcpu_lstats __percpu *lstats; /* loopback stats */

struct pcpu_tstats __percpu *tstats; /* tunnel stats */

struct pcpu_dstats __percpu *dstats; /* dummy stats */

};

struct garp_port __rcu *garp_port;

struct device dev - used for class/net/name entry.

/* space for optional device, statistics, and wireless sysfs groups */

const struct attribute_group *sysfs_groups[4];

const struct rtnl_link_ops *rtnl_link_ops;

rtnetlink link ops instance.

We can use rtnl_link_ops in a network driver or network software module, and declare methods which we want to call when for example we create a new link with "ip link" command.

In case we use rtnl_link_ops, we should register it with ***rtnl_link_register()*** in the driver in its init method, and unregister it in module exit by ***rtnl_link_unregister()***. See for example in the vxlan driver code, drivers/net/vxlan.c.

In case we do not use rtnl_link_ops, then we will use the generic rtnetlink callbacks which are called upon receiving certain messages.

For example, in ***register_netdevice()***, in case dev->rtnl_link_ops is NULL, we send an RTM_NEWLINK message. This message is handled by ***rtnl_newlink()*** callback in net/core/rtnetlink.c.

/* for setting kernel sock attribute on TCP connection setup */

#define GSO_MAX_SIZE 65536

unsigned int gso_max_size;

#define GSO_MAX_SEGS 65535

u16 gso_max_segs;

const struct dcbnl_rtnl_ops *dcbnl_ops - Data Center Bridging netlink ops

u8 num_tc - number of traffic classes in the net device.

Helper method: ***netdev_set_num_tc()***

sets num_tc of a device (max num_tc can be TC_MAX_QUEUE, which is 16).

struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE];

u8 prio_tc_map[TC_BITMASK + 1];

/* max exchange id for FCoE LRO by ddp */

unsigned int fcoe_ddp_xid;

struct netprio_map __rcu *priomap;

/* phy device may attach itself for hardware timestamping */

struct phy_device *phydev;

- The phy device associated with the network device.
  see phy_device struct definition in include/linux/phy.h.

struct lock_class_key *qdisc_tx_busylock;

int group;

- The group the device belongs to.
  - Helper method: void dev_set_group(struct net_device *dev, int new_group): a helper method to set a new group.

struct pm_qos_request pm_qos_req - for power management requests.

● macros starting with IN_DEV like: IN_DEV_FORWARD() or IN_DEV_RX_REDIRECTS() are related to net_device. struct in_device has a member named conf (instance of ipv4_devconf). Setting/proc/sys/net/ipv4/conf/all/forwarding eventually sets the forwarding member of in_device to 1. The same is true to accept_redirects and send_redirects; both are also members of cnf (ipv4_devconf).

- In most distros, */proc/sys/net/ipv4/conf/all/forwarding* is 0
- There are cases when we work with virtual devices.

– For example, bonding (setting the same IP for two or more NICs, for load balancing and for high availability.)

– Many times this is implemented using the private data of the device (the void *priv member of net_device);

- struct net_device_ops has methods for network device management:
- ***ndo_set_rx_mode()*** is used to initialize multicast addresses (It was done in the past by ***set_multicast_list()*** method, which is now deprecated).
- ndo_change_mtu() is for setting mtu.
- Recently, three methods were added to support bridge operations: (John Fastabend)

- ***ndo_fdb_add()***
- ***ndo_fdb_del()***
- ***ndo_fdb_dump()***
- Intel ixgbe driver, for example, uses these methods.
- See ***drivers/net/ethernet/intel/ixgbe/ixgbe_main.c.***
- Also, a new command which uses these methods is to be added to iproute2 package; this command is called "bridge'.
- see http://patchwork.ozlabs.org/patch/117664/
  There are some macros which operate on net_device struct and which are defined in include/linux/netdevice.h:

  SET_NETDEV_DEVTYPE(net, devtype):

  SET_NETDEV_DEVTYPE() is used, for example, in br_dev_setup(), in /net/bridge/br_device.c:

```
static struct device_type br_type = {
.name = "bridge",
};


br_dev_setup()
{
SET_NETDEV_DEVTYPE(dev, &br_type);
}
```

Calling thus SET_NETDEV_DEVTYPE() enables us to see **DEVTYPE=bridge** when running udevadm command on the bridge sysfs entry:

```
udevadm info -q all -p /sys/devices/virtual/net/mybr
P: /devices/virtual/net/mybr
E: DEVPATH=/devices/virtual/net/mybr
E: DEVTYPE=bridge
E: ID_MM_CANDIDATE=1
E: IFINDEX=7
E: INTERFACE=mybr
```

```
E: SUBSYSTEM=net
E: SYSTEMD_ALIAS=/sys/subsystem/net/devices/mybr
E: TAGS=:systemd:
E: USEC_INITIALIZED=4288173427
```

The following patch from Doug Goldstein (which was applied) adds sysfs type to vlan:

The patch itself:

http://www.spinics.net/lists/netdev/msg214013.html

The patch approval:

http://www.spinics.net/lists/netdev/msg216184.html

Also a recent patch, also from Doug Goldstein, enables seeing bonding type with udevadm:

http://www.spinics.net/lists/netdev/msg226618.html

udevadm info -q all -p /sys/devices/virtual/net/bond0

```
P: /devices/virtual/net/bond0
E: DEVPATH=/devices/virtual/net/bond0
E: DEVTYPE=bond
E: ID_MM_CANDIDATE=1
E: IFINDEX=3
E: INTERFACE=bond0
...
```

Another example of usage of SET_NETDEV_DEVTYPE() macro is in cfg80211_netdev_notifier_call() in net/wireless/core.c

```
static struct device_type wiphy_type = {
.name = "wlan",
};
cfg80211_netdev_notifier_call() {
...
SET_NETDEV_DEVTYPE(dev, &wiphy_type);
...
```

Another macro is SET_NETDEV_DEV(net, pdev)

- It sets the sysfs physical device reference for the network logical device.

struct skb_shared_info

This structure is at the end of the skb.

Helper macro:

skb_shinfo(SKB): returns a pointer to skb_shared_info* of the skb.

skb_shared_info members:

```
unsigned char  nr_frags - Number of elements in the frags array;
                          see note below.
__u8           tx_flags;
unsigned short gso_size;
unsigned short gso_segs;
```

```
	unsigned short  gso_type;
	struct sk_buff   *frag_list;
	struct skb_shared_hwtstamps hwtstamps;
	__be32          ip6_frag_id;

	/*
	 * Warning : all fields before dataref are cleared in __alloc_skb()
	 */
	atomic_t   dataref;

	/* Intermediate layers must ensure that destructor_arg
	 * remains valid until skb destructor */
	void *              destructor_arg;

	/* must be last field, see pskb_expand_head() */
	skb_frag_t       frags[MAX_SKB_FRAGS];
```

Helper method:

skb_frag_size(const skb_frag_t *frag) returns the size of skb_frag_t.

Note: when setting the interface in scatter-gather mode, and when setting tx checksum, gso and tso, and running a default iperf test (iperf client, which sends TCP traffic) then nr_frags

of skb_shared_info (which is the number of elements in the frags array of skb_shared_info) is set to be larger than 0.

This is done in ***tcp_sendmsg ()*** (net/ipv4/tcp.c) by calling ***skb_frag_size_add()*** , which eventually invokes **skb_fill_page_desc()**.

Broadcasts

Limited Broadcast - Sent to 255.255.255.255 - all NICs on the same network segment
as the source NIC.
Direct broadcast : Example: For network 192.168.0.0, the Direct broadcast
is 192.168.255.255.

helper method:
ipv4_is_lbcast(address): checks whether the address is 255.255.255.255 (which is INADDR_BROADCAST)

Reverse Path Filter(rp_filter)

When trying to send with a packet with a source IP which is not configure on any interfaces of the machine("spoofing"), the other side will discard the packet. Where is it done and how can we prevent it ?
The reason is that ***__fib_validate_source()*** returns -EXDEV ("Cross-device link") in such a case, when the RPF (Reverse Path Filter) is set, which is the default.
We can avoid this problem and set RPF to off:
by
echo 0 > /proc/sys/net/ipv4/conf/eth0/rp_filter
echo 0 > /proc/sys/net/ipv4/conf/all/rp_filter

We can view the number of packets rejected by Reverse Path Filter by:
netstat -s | grep IPReversePathFilter
IPReversePathFilter: 12
This displays the LINUX_MIB_IPRPFILTER MIB counter, which is incremented whenever
ip_rcv_finish() gets -EXDEV error from ip_route_input_noref().

We can also view it by cat /proc/net/netstat

IN_DEV_RPFILTER(idev) macro - used in fib_validate_source().

See myping.c as an example of spoofing in the following link:
http://www.tenouk.com/Module43a.html

Network interface drivers

- Most of the nics are PCI devices; There are cases, especially with SoC (System On Chip) vendors, where the network interfaces are not PCI devices. There are also some USB network devices.

- The drivers for network PCI devices use the generic PCI calls, like ***pci_register_driver()*** and ***pci_enable_device().***

● For more info on nic drives see the article “Writing Network Device Driver for Linux” (link no. 9 in links) and chap17 in ldd3.

● There are two modes in which a NIC can receive a packet.

– The traditional way was interrupt driven, and nowadays it is NAPI.

each received packet is an asynchronous event which causes an interrupt.

NAPI

● NAPI (New API).

– The NIC works in polling mode.

– This enables handling high traffic load.

– In order that the nic will work in polling mode it should be built with a proper flag.

– Most of the new drivers support this feature.

– When working with NAPI and when there is a very high load, packets are lost; but this occurs before they are fed into the network stack. (in the nonNAPI driver they pass into the stack)

The initial change to napi_struct is explained in: *http://lwn.net/Articles/244640/*

You register a napi struct by: ***netif_napi_add()***

The poll callback is a member of napi_struct.

The prototype of poll is:

int (*poll)(struct napi_struct *, int);

The second parameter is usually called the budget.

You register a napi struct by: ***netif_napi_del()***

napi_complete() is for turning off polling; it removes the napi struct from the NAPI poll list.

In the driver, when we process less than the number of packets of budget, this means that there are no more packets in the RX queue; then we should call napi_complete() to turn off polling and also the enable IRQs.

net_rx_action() is the handler for NET_RX_SOFTIRQ soft irqs.

User Space Tools:

● iputils (including ping, arping, tracepath, tracepath6, ifenslave and more)

● net tools (ifconfig, netstat, route, arp and more)

● IPROUTE2 (ip command with many options)

– Uses rtnetlink API.

– Has much wider functionalities the net tools; for example, you can create tunnels with "ip" command. Note: no need for "n" flag when using IPROUTE2 (because it does not work with DNS).

–

Routing Subsystem

● The routing table enable us to find the net device and the address of the host to which a packet will be sent.

● Reading entries in the routing table is done by calling fib_lookup

- In IPv4: ***int fib_lookup(struct net *net, struct flowi4 *flp, struct fib_result *res)***
- In IPv6 :struct dst_entry *fib6_rule_lookup(struct net *net, struct flowi6 *fl6, int flags, pol_lookup_t lookup)

● FIB is the "Forwarding Information Base".

● There are two routing tables by default: (non Policy Routing case)

– local FIB table (ip_fib_local_table ; ID 255).

– main FIB table (ip_fib_main_table ; ID 254) – See : include/net/ip_fib.h.

- Routes can be added into the main routing table in one of 3 ways:
  - By sys admin command (route add/ip route).
  - By routing daemons.
  - As a result of ICMP (REDIRECT).
- A routing table is implemented by struct fib_table.

Routing Tables

- ***fib_lookup()*** first searches the local FIB table (ip_fib_local_table).
- In case it does not find an entry, it looks in the main FIB table (ip_fib_main_table).
- Why is it in this order ?
- There was in the past a routing cache; there was a single routing cache, regardless of how many routing tables there were. The routing cache was removed in July 2012;
see http://www.spinics.net/lists/netdev/msg205372.html

One of the reason for removal of routing cache was that it was easy to launch denial of service attacks against it.

- You can see the routing cache by running ”route -C”.
- Alternatively, you can see it by : “cat /proc/net/rt_cache”.
  - con: this way, the addresses are in hex format.

You should distinguish between two cases: when CONFIG_IP_MULTIPLE_TABLES is set (which is the default in some distros, like FEDORA) and between when it is not set.

Some fib methods have two implementations, for each of these cases.

For example, fib_get_table() when CONFIG_IP_MULTIPLE_TABLES is set is implemented in net/ipv4/fib_frontend.c, whereas when CONFIG_IP_MULTIPLE_TABLES is not defined, it is implemented in ***include/net/ip_fib.***

Routing Cache

Note: In recent kernels, routing cache is removed.

• The routing cache is built of rtable elements:

• struct rtable (see: ***include/net/route.h***)

The first member of rtable is dst (struct dst_entry dst).

• The dst_entry is the protocol independent part.

– Thus, for example, we have a first member called dst also in rt6_info in IPv6; rt6_info is the parallel of rtable for IPv6 (*include/net/ip6_fib.h*).

• rtable is created in __mkroute_input() and in __mkroute_output(). (*net/ipv4/route.c*)

•There is a member in rtable called rt_is_input, specifying whether it is input route or output route.

• There are also two helper methods, rt_is_input_route() and rt_is_output_route(), which return whether the route is input route or output route.

• The key for a lookup operation in the routing cache is an IP address (whereas in the routing table the key is a subnet).

• the lookup is done by fib_trie (net/ipv4/fib_trie.c)

• It is based on extending the lookup key.

• By Robert Olsson et al (see links).

– TRASH (trie + hash)

– Active Garbage Collection

• You can view fib tries stats by:

cat /proc/net/fib_triestat

• You can flush the routing cache by: ip route flush cache

caveat: it sometimes takes 2-3 seconds or more; it depends on your machine.

• You can show the routing cache by:

ip route show cache

Creating a Routing Cache Entry

• Allocation of rtable instance (rth) is done by: dst_alloc().

– dst_alloc() in fact creates and returns a pointer to dst_entry and we cast it to rtable (net/core/dst.c).

• Setting input and output methods of dst:

rth->u.dst.input and rth->u.dst.output

• Setting the flowi member of dst (rth->fl)

– Next time there is a lookup in the cache,for example , ip_route_input(), we will compare against rth->fl.

• A garbage collection call which delete eligible entries from the routing cache.

• Which entries are not eligible ?

Policy Routing (multiple tables)

• Generic routing uses destination address based decisions.

• There are cases when the destination address is not the sole parameter to decide which route to give; Policy Routing comes to enable this.

● Adding a routing table : by adding a line to: /etc/iproute2/rt_tables. – For example: add the line “252 my_rt_table”. There can be up to 255 routing tables.

● Policy routing should be enabled when building the kernel (CONFIG_IP_MULTIPLE_TABLES should be set.)

● Example of adding a route in this table:

● ***ip route add default via 192.168.0.1 table my_rt_table***

● Show the table by:

ip route show table my_rt_table

● You can add a rule to the routing policy database (RPDB) by “ip rule add ...”

In ***fib4_rules_init()***, we set net->ipv4.fib_has_custom_rules to false.

This is because when working with default tables and not adding any other tables, there are no custom rules.

Each time that we add a new rule (by "ip rule add"), we set net->ipv4.fib_has_custom_rules to true;

See: fib4_rule_configure() in net/ipv4/fib_rules.c

Also when we delete a rule in fib4_rule_delete(), we set net->ipv4.fib_has_custom_rules to true;

ip rule add

– The rule can be based on input interface, TOS, fwmark (from netfilter).

● ip rule list – show all rules.

struct fib_rule represents the rules created by policy routing.

Policy Routing: add/delete a rule example

● ip rule add tos 0x04 table 252 – This will cause packets with tos=0x08 (in the iphdr) to be routed by looking into the table we added (252) – So the default gw for these type of packets will be

192.168.0.1 – ip rule show will give: – 32765: from all tos reliability lookup my_rt_table – ... Policy Routing: add/delete a rule example

• Delete a rule : ip rule del tos 0x04 table 252

• Breaking the fib_table into multiple data structures gives flexibility and enables fine grained and high level of sharing. – Suppose that we 10 routes to 10 different networks have the same next hop gw. – We can have one fib_info which will be shared by 10 fib_aliases. – fz_divisor is the number of buckets

• Each fib_ node element represents a unique subnet. – The fn_key member of fib_ node is the subnet (32 bit)

• In the usual case there is one fib_nh (Next Hop). – If the route was configured by using a multipath route, there can be more than one fib_nh.

• Suppose that a device goes down or enabled.

• We need to disable/enable all routes which use this device.

• But how can we know which routes use this device ?

• In order to know it efficiently, there is the fib_info_devhash table.

• This table is indexed by the device identifier.

• See fib_sync_down() and fib_sync_up() in net/ipv4/fib_semantics.c

Routing Table lookup algorithm

• LPM (Longest Prefix Match) is the lookup algorithm.

• The route with the longest netmask is the one chosen.

• Netmask 0, which is the shortest netmask, is for the default gateway. – What happens when there are multiple entries with netmask=0? – fib_lookup() returns the first entry it finds in the fib table where netmask length is 0.

• It may be that this is not the best choice default gateway.

• So in case that netmask is 0 (prefixlen of the fib_result returned from fib_look is 0) we call fib_select_default().

• ***fib_select_default()*** will select the route with the lowest priority (metric) (by comparing to fib_priority values of all default gateways).

Receiving a packet

• When working in interrupt driven model, the nic registers an interrupt handler with the IRQ with which the device works by calling ***request_irq().***

• This interrupt handler will be called when a frame is received

• The same interrupt handler will be called when transmission of a frame is finished and under other conditions. (depends on the NIC; sometimes, the interrupt handler will be called when there is some error).

● Typically in the handler, we allocate sk_buff by calling dev_alloc_skb() ; also eth_type_trans() is called; among other things it advances the data pointer of the sk_buff to point to the IP header ; this is done by calling skb_pull(skb, ETH_HLEN).

• See : net/ethernet/eth.c – ETH_HLEN is 14, the size of ethernet header.

• The handler for receiving an IPV4 packet is ip_rcv(). (net/ipv4/ip_input.c)

• The handler for receiving an IPV6 packet is ipv6_rcv() (net/ipv6/ip6_input.c)

• Handler for the protocols are registered at init phase.

– Likewise, arp_rcv() is the handler for ARP packets.

• First, ip_rcv() performs some sanity checks. For example: if (iph->ihl < 5 || iph->version != 4) goto inhdr_error; – iph is the ip header ; iph->ihl is the ip header length (4 bits). – The ip header must be at least 20 bytes. – It can be up to 60 bytes (when we use ip options)

• Then it calls ip_rcv_finish(), by: NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish);

• This division of methods into two stages (where the second has the same name with the suffix finish or slow, is typical for networking kernel code.)

• In many cases the second method has a “slow” suffix instead of “finish”; this usually happens when the first method looks in some cache and the second method performs a lookup in a table, which is slower.

• ip_rcv_finish() implementation: if (skb->dst == NULL) { int err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos,skb->dev); ... } ... return dst_input(skb);

• ip_route_input(): First performs a lookup in the routing cache to see if there is a match. If there is no match (cache miss), calls ip_route_input_slow() to perform a lookup in the routing table. (This lookup is done by calling fib_lookup()).

• fib_lookup(const struct flowi *flp, struct fib_result *res) The results are kept in fib_result.

• ip_route_input() returns 0 upon successful lookup. (also when there is a cache miss but a successful lookup in the routing table.)

According to the results of fib_lookup(), we know if the frame is for local delivery or for forwarding or to be dropped.

• If the frame is for local delivery , we will set the input() function pointer of the route to ip_local_deliver(): rth->u.dst.input= ip_local_deliver;

• If the frame is to be forwarded, we will set the input() function pointer to ip_forward(): rth->u.dst.input = ip_forward; Local Delivery

• Prototype: ip_local_deliver(struct sk_buff *skb) (net/ipv4/ip_input.c). calls NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev,NULL,ip_local_deliver_finish);

• Delivers the packet to the higher protocol layers according to its type.

Forwarding

- prototype: ***int ip_forward(struct sk_buff *skb)*** (net/ipv4/ip_forward.c)

– decreases the ttl in the ip header; If the ttl is <=1 , the methods send ICMP message (ICMP_TIME_EXCEEDED) with ICMP_EXC_TTL ("TTL count exceeded"), and drops the packet.

– Calls NF_HOOK(PF_INET,NF_IP_FORWARD, skb, skb->dev,rt->u.dst.dev,   ip_forward_finish);

- ***ip_forward_finish()***: sends the packet out by calling dst_output(skb).

- dst_output(skb) is just a wrapper, which calls skb->dst->output(skb). (see include/net/dst.h)

You can see the number of forwarded packets by "netstat -s | grep forwarded".

or by cat /proc/net/snmp (IPv4) and cat /proc/net/snmp6 (IPV6), and look in ForwDatagrams column (IPv4)/Ip6OutForwDatagrams (IPv6).

Sending a Packet

- We need to perform routing lookup also in the case of transmission.
- There are cases when we perform two lookups, like in ipip tunnels.
- Handling of sending a packet is done by ip_route_output_key().
- In case of a cache miss, we calls ip_route_output_slow(), which looks in the routing table (by calling fib_lookup(), as also is done in *ip_route_input_slow().*)
- If the packet is for a remote host, we set dst->output to ip_output()
- ip_output() will call ip_finish_output() – This is the NF_IP_POST_ROUTING point.

• ***ip_finish_output()*** will eventually send the packet from a neighbor by: – dst->neighbour->output(skb) – arp_bind_neighbour() sees to it that the L2 address of the next hop will be known. (net/ipv4/arp.c)

• If the packet is for the local machine: – dst->output = ip_output – dst->input = ip_local_deliver – ip_output() will send the packet on the loopback device, – Then we will go into ip_rcv() and ip_rcv_finish(), but this time dst is NOT null; so we will end in ***ip_local_deliver()***.

• See: net/ipv4/route.c

GRO:

GRO stands for Generic Receive Offload.
In order to work with GRO (Generic Receive Offload):
- you must set NETIF_F_GRO in device features.
- you should call napi_gro_receive() from the RX path of the driver.

When **NETIF_F_GRO** is not set, napi_gro_receive() continues to the usual RX
path, namely it calls netif_receive_skb().
This is done by returning GRO_NORMAL from dev_gro_receive(), and then
calling netif_receive_skb() from napi_skb_finish() (see net/core/dev.c).

GRO replaces LRO (Large Receive Offload), as LRO was only for TCP in IPv4.
LRO was removed from the network stack.

GRO works in conjunction with GSO (Generic Segmentation Offload).

Multipath routing

• This feature enables the administrator to set multiple next hops for a destination.

• To enable multipath routing, **CONFIG_IP_ROUTE_MULTIPATH** should be set when building the kernel.

- There was also an option for multipath caching: (by setting CONFIG_IP_ROUTE_MULTIPATH_CACHED).
- It was experimental and removed in 2.6.23 See links (6).

Multicast and Multicast routing

Internet Group Management Protocol, Version 2 (IGMPv2)
RFC 2236

The Internet Group Management Protocol (IGMP) is used by IP hosts to report their multicast group memberships to any immediately-neighboring multicast routers.

In linux kernel, IGMP for IPv4 is implemented in net/ipv4/igmp.c

There are three types of IGMP messages:

Membership Query (Type: 0x11)
Membership Report (Version 2) (Type: 0x16)
Leave Group (Type: 0x17)

And one Legacy (for IGMPv1 backward compatibility) message:
Membership Report (Version 1) (0x12)

To add a multicast address at MAC level, you can use "ip maddr add".

Note that "ip maddr add" expects a MAC address, not an IP address!
So this is ok:
ip maddr add 01:00:5e:01:01:25 dev eth0
but this is wrong: (pay attention, you will not get any error message!)

ip maddr add 226.1.2.3

You can join a multicast group also by setsockopt with ***IP_ADD_MEMBERSHIP***; see for example: https://github.com/troglobit/toolbox/blob/master/mcjoin.c

All Mulitcast addresses in mac presentations start with 01:00:5E according to IANA requirements.
Multicast addresses are translated from IP notation to mac address by a formula; see *ip_eth_mc_map()* in *include/net/ip.h.*
This is needed for example in arp translation, *arp_mc_map()* in *net/ipv4/arp.c.*

The handler for multicast RX is ip_mr_input() in ***net/ipv4/ipmr.c.***

The code which handles multicast routing is net/ipv4/ipmr.c for IPv4, and net/ipv6/ip6mr.c for IPv6.

In order to work with Multicast routing, the kernel should be build with

IP_MROUTE=y.

You should also need to work with multicast routing user space daemons, like **pimd** or **xorp**. (In the past there was a daemon called mrouted). Notice that

/proc/sys/net/ipv4/conf/all/mc_forwarding entry is a read only entry;

ls -al /proc/sys/net/ipv4/conf/all/mc_forwarding

shows:
-r--r--r-- 1 root root

However, starting a daemon like pimd changes its value to 1.

(stopping the daemon changes it again to 0).

PIM stands for Protocol Independent Multicast

see: http://en.wikipedia.org/wiki/Protocol_Independent_Multicast

pimd open source project:
https://github.com/downloads/troglobit/pimd/pimd-2.1.8.tar.bz2

When pimd starts, the following happens:
It sends IGMP V2 membership query.
The membership query has a TTL of 1.

The membership query is sent each 125 seconds.
(IGMP_QUERY_INTERVAL).
This is done in query_groups() method, pimd-2.1.8/igmp_proto.c.

pimd joins these two multicast groups:
224.0.0.2 : The All Routers multicast group addresses all routers on the same network segment.
224.0.0.13 : All PIM Routers.

This is done in *k_join()* method of pimd-2.1.8/kern.c.

Two membership reports are sent as a result.

- These membership reports also has a TTL of 1.

see IPv4 Multicast Address Space Registry:
http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xml

pimd creates an IGMP socket.

pimd adds entries to the multicast cache (MFC). This is done by setsockopt with MRT_ADD_MFC which
invokes **ipmr_mfc_add()** method in *net/ipv4/ipmr.c*

You can see entries and statistics of the multicast cache (MFC) by:
cat /proc/net/ip_mr_cache

This patch (4.12.12) from Nicolas Dichtel enables to advertise mfc stats
via rtnetlink. This is done by adding a struct named rta_mfc_stats
in *include/uapi/linux/rtnetlink.h.*

see: ipmr/ip6mr: advertise mfc stats via rtnetlink:
http://permalink.gmane.org/gmane.linux.network/251481%20 %20 " target="_blank">http://permalink.gmane.org/gmane.linux.network/251481

Secondary addresses:
An address is considered "secondary" if it is included in the subnet of another address on the same interface.
Example:

```
ip address add 192.168.0.1/24 dev p2p1
ip address add 192.168.0.2/24 dev p2p1
```

```
ip addr list dev p2p1
3: p2p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP qlen 1000
link/ether 00:a1:b0:69:74:00 brd ff:ff:ff:ff:ff:ff
inet 192.168.0.1/24 scope global p2p1
inet 192.168.0.2/24 scope global secondary p2p1
inet6 fe80::2a1:b0ff:fe69:7400/64 scope link
valid_lft forever preferred_lft forever
```

IGMP snooping

IGMP snooping can be controlled through sysfs interface.
For brN, the settings can be found
under /sys/devices/virtual/net/brN/bridge.

For example,for:
brctl addbr br0
cat /sys/devices/virtual/net/br0/bridge/multicast_snooping
This multicast_disabled of net_bridge struct represents
multicast_snooping.

rtnl_register()

The *rtnl_register()* method gets 3 callbacks as parameters:
doit, dumpit, and calcit callbacks.
We have two *rtnl_register()* calls in the routing subsystem with
RTM_GETROUTE
rtnl_register(PF_INET, RTM_GETROUTE, inet_rtm_getroute, NULL,
NULL)
in *net/ipv4/route.c*
and
rtnl_register(PF_INET, RTM_GETROUTE, NULL, inet_dump_fib, NULL);
in *net/ipv4/fib_frontend.c*

They are called according to the type of userspace call:
ip route get 192.168.1.1 is implemented via ***inet_rtm_getroute()***
ip route show is implemented via ***inet_dump_fib()***

We also have callbacks for adding/deleting a route:
rtnl_register(PF_INET, RTM_NEWROUTE, inet_rtm_newroute, NULL, NULL);
rtnl_register(PF_INET, RTM_DELROUTE, inet_rtm_delroute, NULL, NULL);

in *net/ipv4/fib_frontend.c*

Note:

In ***rtnetlink_net_init()***, which is called e have:
sk = netlink_kernel_create(net, NETLINK_ROUTE, &cfg);

rtnetlink_net_init() is called
from netlink_proto_init(), net/netlink/af_netlink.c.

We also have, in net/ipv4/fib_frontend.c:

netlink_kernel_create(net, NETLINK_FIB_LOOKUP, &cfg);

NETLINK_FIB_LOOKUP is not used by iproute2; it *is* used by libnl, in a util called util named nl-fib-lookup, and also in other libnl code.

NETLINK_FIB_LOOKUP has one callback, named nl_fib_input(struct sk_buff *skb), which in fact performs eventually a fib lookup; you might wonder for what is NETLINK_FIB_LOOKUP socket needed if we have "ip route get", which uses NETLINK_ROUTE socket
and RTM_GETROUTE message; the answer is
that NETLINK_FIB_LOOKUP was added when adding the trie code, and it stayed probably as a legacy. see:**http://lists.openwall.net/netdev/2009/05/25/33**

VRRP

VRRP stands for Virtual Router Redundancy Protocol
http://en.wikipedia.org/wiki/Virtual_Router_Redundancy_Protocol

You can find a GPL licensed implementation of VRRP designed
for Linux operating systems here:
http://sourceforge.net/projects/vrrpd/

what is VRRPd daemon is is an implementation of VRRPv2 as specified in rfc2338.
It runs in userspace on linux.

xorp project:

http://www.xorp.org/

- fea is the Forwarding Engine Abstraction
- mfea is the Multicast Forwarding Engine Abstraction.

XORP git tree https://github.com/greearb/xorp.ct.git

In case you download the xorp tar.gz and you had build problem, you might consider git cloning the XORP git tree and building by scons && scons install.

Netfilter

● Netfilter is the kernel layer to support applying iptables rules.

– It enables:

- ● Filtering
- ● Changing packets (masquerading)
- ● Connection Tracking
- ● see: http://www.netfilter.org/

Xtables modules are prefixed with xt, for example, *net/netfilter/xt_REDIRECT.c*.

Xtables matches are always lowercase.

Xtables targets are always uppercase (for example, xt_REDIRECT.c)

struct xt_target : defined in *include/linux/netfilter/x_tables.h*

Registering xt_target is done by xt_register_target().

Registering an array of xt_target is done by xt_register_targets(). see *net/netfilter/xt_TPROXY.c*

● "Writing Netfilter modules" (67 pages pdf) by Jan Engelhardt, Nicolas Bouliane:

http://jengelh.medozas.de/documents/Netfilter_Modules.pdf

Netfilter tables:

You register/unregister a netfilter table
by ***ipt_register_table()/ipt_unregister_table().***

we have the following 5 netfilter tables in IPv4:

nat table - has 4 chains:

NF_INET_PRE_ROUTING
NF_INET_POST_ROUTING
NF_INET_LOCAL_OUT
NF_INET_LOCAL_IN

- see *net/ipv4/netfilter/iptable_nat.c*
- REDIRECT is a NAT table target; implemented in *net/netfilter/xt_REDIRECT.c*

mangle table - has 5 chains:

NF_INET_PRE_ROUTING
NF_INET_LOCAL_IN
NF_INET_FORWARD
NF_INET_LOCAL_OUT
NF_INET_POST_ROUTING

see:net/ipv4/netfilter/iptable_mangle.c

TPROXY is a mangle table target; implemented
in net/netfilter/xt_TPROXY.c

raw table - has 2 chains:

NF_INET_PRE_ROUTING
NF_INET_LOCAL_OUT

see:net/ipv4/netfilter/iptable_raw.c

filter table - has 3 chains:

NF_INET_LOCAL_IN
NF_INET_FORWARD
NF_INET_LOCAL_OUT

REJECT is example of a filter table target. It is implemented in net/ipv4/netfilter/ipt_REJECT.c.

DROP is also a filter table target.

Both in DROP and in REJECT we drop the packet. The difference is that with

REJECT target we send ICMP packet (port-unreachable is the default)

- You can set the ICMP type with --reject-with type: it can be icmp-net-unreachable, icmp-host-unreach-able, icmp-port-unreachable, icmp-proto-unreachable, icmp-net-prohibited, icmp-host-prohibited or icmp-admin-prohibited.

see net/ipv4/netfilter/iptable_filter.c

security table - has 3 chains:

NF_INET_LOCAL_IN
NF_INET_FORWARD
NF_INET_LOCAL_OUT

see: net/ipv4/netfilter/iptable_security.c

Xtables2 vs. nftables

http://lwn.net/Articles/531752/

Formal submission of Xtables2

http://lwn.net/Articles/531877/

Jan Engelhardt lecture about Xtables2 **:**

http://inai.de/documents/Love_for_blobs.pdf

Connection Tracking

A connection entry is represented by struct nf_conn.

- see include/net/netfilter/nf_conntrack.h

Each connection tracking entry is kept until a certain timeout elapse. This timeout period is different for TCP, UDP and ICMP.

You can see the connection tracking entries by:
cat /proc/net/nf_conntrack

SNAT and DNAT is implemented in net/netfilter/xt_nat.c

MASQUERADE is implemented
in net/ipv4/netfilter/ipt_MASQUERADE.c

Traffic Control

Tc utility (from iproute package) is used to configure Traffic Control in the Linux kernel.

There are three areas which Traffic Control handles:

tc qdisc - Queuing discipline.

Implementation: in net/sched/sch_* files (for example, net/sched/sch_fifo.c).

tc class

Implementation: Also in net/sched/sch_* files.

tc filter

Implementation: in net/sched/cls_* files.

important structures:

struct Qdisc : declared in include/net/sch_generic.h

- net_device has a Qdisc member (named qdisc).

struct Qdisc_ops : declared in include/net/sch_generic.h

The noqueue_qdisc is an example of Qdisc which is used in virtual devices.

The noqueue_qdisc_ops is an example of Qdisc_ops (member of noqueue_qdisc).

Both are defined in source/net/sched/sch_generic.

pfifo_fast is the default qdisc on all network interfaces.

- Enqueing/Dequeing is done by ***pfifo_fast_enqueue()*** and ***pfifo_fast_dequeue().***

pfifo_fast is a classless queueing discipline, as opposed, for example, to CBQ or HTB, which are class-based queuing disciplines. We can easily determine from looking at the qdisc declaration whether it is classless or class based, by inspecting if there is a class_ops member:

- cbq_qdisc_ops has cbq_class_ops; see net/sched/sch_cbq.c; it is a class-based qdisc
- htb_qdisc_ops has htb_class_ops; see net/sched/sch_htb.c ; it is a class-based qdisc
- pfifo_fast_ops doesn't have class_ops; see net/sched/sch_generic.c; it is a classles qdisc.

Note: Sometimes you will encounter classful terminology for class-based qdiscs.

CBQ is Class Based Queuing.

There is a pfifo qdisc and bfifo qdisc: net/sched/schfifo.c

The difference between pfifo and bfifo is that pfifo is for packets, bfifo is for bytes.

The difference between pfifo_fast and pfifo/bfifo is that pfifo_fast has 3 bands, while

pfifo/bfifo has 1 band. The number of bands is hard coded and cannot be changed

(PFIFO_FAST_BANDS is defined as 3). When having bands, we consider the TOS of the packet.

The three queues in pfifo_fast_priv struct (a member named "q") represent these three bands.

Packets are put into bands according to their TOS, where band 0 has the highest priority.

Example: using HTB (Hierarchical Token Bucket)

```
tc qdisc add dev p2p1 root handle 10: htb
```

- This triggers invocation of tc_modify_qdisc() in net/sched/sch_api.c (handler of RTM_NEWQDISC message, sent from user space)

```
tc qdisc show dev p2p1
qdisc htb 10: root refcnt 2 r2q 10 default 0 direct_packets_stat 0
```

show statistics:

```
tc -s qdisc show dev p2p1
qdisc htb 10: root refcnt 2 r2q 10 default 0 direct_packets_stat 0
Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
```

```
tc class add dev p2p1 parent 10:0 classid 10:10 htb rate 5Mbit
```

- This triggers invocation of tc_ctl_tclass() in net/sched/cls_api.c (handler of RTM_NEWTFILTER message, sent from user space)
  A class can be a parent class or a child class.

```
tc class show dev p2p1
class htb 10:10 root prio 0 rate 5000Kbit ceil 5000Kbit burst 1600b
cburst 1600b
```

```
tc -s class show dev p2p1
class htb 10:10 root prio 0 rate 5000Kbit ceil 5000Kbit burst 1600b
cburst 1600b
Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
rate 0bit 0pps backlog 0b 0p requeues 0
lended: 0 borrowed: 0 giants: 0
tokens: 40000 ctokens: 40000
```

TC filter

The main function of filters is to assign the incoming packets to classes for a qdisc.
The classification of packets can be based on the IP address, port numbers, etc.

Two structures are important for filter: struct tcf_proto_ops and struct tcf_proto. Both are declared in include/net/sch_generic.h.

You register/unregister tcf_proto_ops with register_tcf_proto_ops()/unregister_tcf_proto_ops().

- tc filter add will trigger invocation of tc_ctl_tfilter() in net/sched/cls_api.c
- u32 filter is implemented in net/sched/cls_u32.c
- route is implemented in net/sched/cls_route.c

see also:

Linux Advanced Routing & Traffic Control HOWTO: http://www.lartc.org/lartc.html

Transparent proxy:

NETFILTER_XT_TARGET_TPROXY kernel config item should be set for Transparent proxy (TPROXY) target support.

TPROXY target is somewhat similar to REDIRECT. It can only be used in the mangle table and is useful to redirect traffic to a transparent proxy.

As opposed to REDIRECT, it does not depend on Netfilter connection tracking and NAT.

xt_TPROXY.c

Port 3128 is the default port of squid; in /etc/squid/squid.conf, you can define a tproxy port; for example,

http_port 3128 tproxy

Adding tproxy will trigger calling setsockopt() with IP_TRANSPARENT, when starting the squid daemon. This in turn will set the transparent member of struct inet_sock.

An iptables rule to work with TPROXY can be for example:

iptables -t mangle -A PREROUTING -p tcp --dport 80 -j TPROXY --tproxy-mark 0x1/0x1 --on-port 3128

--tproxy-mark 0x1/0x1 is for setting **skb->mark** in the TPROXY module.

Remember that inet_sock is in fact a casting of the socket:

struct inet_sock *inet = inet_sk(sk);

...

```
static inline struct inet_sock *inet_sk(const struct sock *sk)
{
return (struct inet_sock *)sk;
}
```

Netfilter hooks

struct nf_hook_ops - represents a netfilter hook.

Registration of netfilter hook is done by nf_register_hook(). nf_register_hook() is implemented in net/netfilter/core.c

Netfilter rule example

- Short example:
- Applying the following iptables rule:

– iptables A INPUT p udp dport 9999 j DROP

- This is NF_IP_LOCAL_IN rule;
- The packet will go to:
- ***ip_rcv()***
- and then: ***ip_rcv_finish()***

- And then ***ip_local_deliver()***
- but it will NOT proceed to ip_local_deliver_finish() as in the usual case, without this rule.
- As a result of applying this rule it reaches nf_hook_slow() with verdict == NF_DROP (calls skb_free() to free the packet)
- See net/netfilter/core.c.
- iptables -t mangle A PREROUTING -p udp -dport 9999 -j MARK -setmark 5

– Applying this rule will set skb->mark to 0x05 in ip_rcv_finish().

ICMP redirect message

- ICMP protocol is used to notify about problems.
- A REDIRECT message is sent in case the route is suboptimal (inefficient).
- There are in fact 4 types of REDIRECT
- Only one is used :

– Redirect Host (ICMP_REDIR_HOST)

- See RFC 1812 (Requirements for IP Version 4 Routers).
- To support sending ICMP redirects, the machine should be configured to send redirect messages.

– /proc/sys/net/ipv4/conf/all/send_redirects should be 1.

- In order that the other side will receive redirects, we should set /proc/sys/net/ipv4/conf/all/accept_redirects to 1.
- Example:
- Add a suboptimal route on 192.168.0.31:
- route add net 192.168.0.10 netmask 255.255.255.255 gw 192.168.0.121

• Running now “route” on 192.168.0.31 will show a new entry:
Destination Gateway Genmask Flags Metric Ref Use Iface
192.168.0.10 192.168.0.121 255.255.255.255 UGH 0 0 0 eth0

• Send packets from 192.168.0.31 to 192.168.0.10 :

• ping 192.168.0.10 (from 192.168.0.31)

• We will see (on 192.168.0.31): – From 192.168.0.121: icmp_seq=2 Redirect Host(New nexthop: 192.168.0.10)

• now, running on 192.168.0.121: – route Cn | grep .10 ● shows that there is a new entry in the routing cache:

• 192.168.0.31 192.168.0.10 192.168.0.10 ri 0 0 34 eth0

• The “r” in the flags column means: RTCF_DOREDIRECT.

• The 192.168.0.121 machine had sent a redirect by calling ip_rt_send_redirect() from ip_forward(). (net/ipv4/ip_forward.c)

• And on 192.168.0.31, running “route -c" | grep .10” shows now a new entry in the routing cache: (in case accept_redirects=1)

• 192.168.0.31 192.168.0.10 192.168.0.10 0 0 1 eth0

• In case accept_redirects=0 (on 192.168.0.31), we will see:

• 192.168.0.31 192.168.0.10 192.168.0.121 0 0 0 eth0

• which means that the gw is still 192.168.0.121 (which is the route that we added in the beginning).

• Adding an entry to the routing cache as a result of getting ICMP REDIRECT is done in ip_rt_redirect(),net/ipv4/route.c.

• The entry in the routing table is not deleted.

Neighboring Subsystem

• Most known protocol: ARP (in IPV6: ND, neighbour discovery)

• ARP table.

• Ethernet header is 14 bytes long: – Source mac address (6 bytes). – Destination mac address (6 bytes). – Type (2 bytes).

• 0x0800 is the type for IP packet (ETH_P_IP)

● 0x0806 is the type for ARP packet (ETH_P_ARP)

● 0x8100 is the type for VLAN packet (ETH_P_8021Q)

● see: include/linux/if_ether.h

● When there is no entry in the ARP cache for the destination IP address of a packet, a broadcast is sent (ARP request, ARPOP_REQUEST: who has IP address x.y.z...). This is done by a method called arp_solicit(). (net/ipv4/arp.c)

● You can see the contents of the arp table by running: "cat /proc/net/arp" or by running the "arp" from a command line .

● You can delete and add entries to the arp table; see man arp.

Bridging Subsystem

● Bridging implementation in Linux conforms to IEEE 802.1d standard (which describes Bridging and Spanning tree).
See **http://en.wikipedia.org/wiki/IEEE_802.1D**

● You can define a bridge and add NICs to it ("enslaving ports") using brctl (from bridge-utils).

- bridge-utils is maintained by Stephen Hemminger.
- you can get the sources by:
- git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/bridge-utils.git
- Building is simple: first run: autoconf (in order to create "configure" file)
- Then run make.

There are two important structures in the bridging subsystem:
struct net_bridge represents a bridge.
struct net_bridge_port represents a bridge port.
(Both are defined in net/bridge/br_private.h)
net_bridge has a hash table inside called "hash".
It has 256 entries (BR_HASH_SIZE).

• You can have up to 1024 ports for every bridge device (BR_MAX_PORTS) .

• Example:

• brctl addbr mybr (Create a bridge named "mybr")

• brctl addif mybr eth0 (add a port to a bridge).

• brctl show

• brctl delbr mybr (Delete the bridge named "mybr")

Note:

You can see the fdb by

./bridge/bridge fdb show

For example, after

brctl addbr mybr

brctl addif mybr p2p1

Output can be, for example,

00:a1:b0:69:74:00 dev p2p1 permanent

Note:

The "bridge" util is part of iproute2 package. In case you don't have the "bridge" util,

you can git clone it by:

git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git

You cannot add a wireless device to a bridge.

The following series will fail:

brctl addbr mybr

brctl addif mybr wlan0

can't add wlan0 to bridge mybr: Operation not supported.

You cannot add a loopback device to a bridge:
brctl addbr mybr

```
brctl addif mybr lo
can't add lo to bridge mybr: Invalid argument
```

The reason:

In ***br_add_if()***, we check the priv_flags of the device, and in case IFF_DONT_BRIDGE

is set, we return -EOPNOTSUPP (Operation not supported).

In case of wireless device, cfg80211_netdev_notifier_call() method sets the IFF_DONT_BRIDGE (see *net/wireless/core.c*)

TBD:

Under In which circumstances do we remove the IFF_DONT_BRIDGE flag in cfg80211_change_iface() in net/wireless/util.c?

● When a NIC is configured as a bridge port, the br_port member of net_device is initialized. – (br_port is an instance of struct net_bridge_port).

When a bridge is created, we call netdev_rx_handler_register() to register a method

for handling a bridge method to handle packets. This method is called br_handle_frame().

Each packet which is received by the bridge is handled by ***br_handle_frame()***.

See ***br_add_if()*** method is ***net/bridge/br_if.c.***

(Besides the bridging interface, also the macvlan interface and the bonding interface invokes *netdev_rx_handler_register()*; In fact what this method does is assign a method

to the net_device rx_handler member, and assign rx_handler_data to net_device

rx_handler_data member. You cannot call
twice *netdev_rx_handler_register()* on the same network device; this will return an error ("Device or resource busy", EBUSY).

see drivers/net/macvlan.c and net/bonding/bond_main.c.

● In the past, when we received a frame, netif_receive_skb() called handle_bridge().

Now we call br_handle_frame(), via
invoking ***rx_handler()*** (see __netif_receive_skb() in

net/core/dev.c)

● The bridging forwarding database is searched for the destination MAC address.

● In case of a hit, the frame is sent to the bridge port
with ***br_forward()*** (net/bridge/br_forward.c).

● If there is a miss, the frame is flooded on all bridge ports
using ***br_flood()*** (net/bridge/br_forward.c).

● Note: this is not a broadcast !

● The ebtables mechanism is the L2 parallel of L3 Netfilter.

● Ebtables enable us to filter and mangle packets at the link layer (L2).

- The ebtables are implemented under net/bridge/netfilter.

● There are five points in the Linux bridging layer where we have the bridge hooks:

- NF_BR_PRE_ROUTING (br_handle_frame()).
- NF_BR_LOCAL_IN (br_pass_frame_up()/br_handle_frame())
- NF_BR_FORWARD (__br_forward())
- NF_BR_LOCAL_OUT(__br_deliver())
- NF_BR_POST_ROUTING (br_forward_finish())
-

Open vSwitch

Open vSwitch is an open source project implementing virtual switch.

http://openvswitch.org/

The code is under net/openvswitch/

The maintainer is Jesse Gross.

See also *Documentation/networking/openvswitch.txt.*

Network namespaces

A network namespace is logically another copy of the network stack, with it's own routes, firewall rules, and network devices.

- A network device belongs to exactly one network namespace.
- A socket belongs to exactly one network namespace.

A network namespace provides an isolated view of the networking stack
- network device interfaces

- IPv4 and IPv6 protocol stacks,

- IP routing tables

- firewall rules

- /proc/net and /sys/class/net directory trees

- sockets

- more

Network namespace is implemented by struct net, *include/net/net_namespace.h*

By running:

ip netns add netns_one

we create a file under /var/run/netns/ called netns_one.

See: man ip netns

In order to show all of the named network namespaces, we run:

ip/ip netns list

Next you run:

ip link add name if_one type veth peer name if_one_peer

ip link set dev if_one_peer netns netns_one

Example for network namespaces usage:

Create two namespaces, called "myns1" and "myns2":
ip netns add myns1
ip netns add myns2

Assigning p2p1 interface to myns1 network namespaces:
ip link set p2p1 netns myns1

Now:
Running:
ip netns exec myns1 bash
will transfer me to myns1 network namespaces; so if I will run there:
ifconfig -a
I will see p2p1;

On the other hand,
running
ip netns exec myns2 bash
will transfer me to myns2 network namespaces; but if I will run there:
ifconfig -a
I will not see p2p1.

You move back p2p1 to the initial network namespace by
ip link set p2p1 netns 1

The 'netns' argument can be either a netns name or a process ID (pid).
Providing a pid, you'll be moving the interface to the netns of the given process, which is the initial network namespace for pid = 1 (the init process).

There are some devices whose **NETIF_F_NETNS_LOCAL** flag is set, and they are considered local devices; we do not permit moving them to any other namespace.

Among these devices are the loopback interface (lo), the bridge interface, the ppp interface, the GRE tunnel interface,
VXLAN interface, and more.

Trying to move an interface whose **NETIF_F_NETNS_LOCAL** flag is set to a different network namespace, we result with "RTNETLINK answers: Invalid argument" error message
from dev_change_net_namespace() method (net/core/dev.c). Behind the scenes, dev_change_net_namespace() checks
the **NETIF_F_NETNS_LOCAL** flag of the net device. If it is set, we will not permit changing of network namespace, and we will
return EINVAL.

Under the hood, when calling ip netns exec , we have here invocation of two system calls from user space:
setns system call with CLONE_NEWNET (kernel/nsproxy.c)
unshare system call with CLONE_NEWNS in (kernel/fork.c)

see netns_exec() in *ip/ipnetns.c* (iproute package)

Note:

Currently there is an issue ("Device or resource busy" error) when trying to delete a namespace. The following series gives an error:

```
ip netns add netns_one
ip netns add netns_two
ip link add name if_one type veth peer name if_one_peer
ip link add name if_two type veth peer name if_two_peer
ip link set dev if_one_peer netns netns_one
ip link set dev if_two_peer netns netns_two
```

```
ip netns exec netns_one bash
```

```
# in other terminal:
```

```
ip netns delete netns_two
```

# => Cannot remove /var/run/netns/netns_two: Device or resource busy

See:

http://permalink.gmane.org/gmane.linux.network/240875

In the future, there is intention to add these commands to iproute2:

"ip netns pids" and "ip netns identify".

see: http://www.spinics.net/lists/netdev/msg217958.html

There is **CLONE_NEWNET** for fork (since Linux 2.6.24)

- If **CLONE_NEWNET** is set, then create the process in a new network namespace. If this flag is not set, then the
process is created in the same network namespace as the calling process. This flag is intended for the implementation of containers.

Three lwn articles about namespaces:

"network namespaces"

http://lwn.net/Articles/219794/

"PID namespaces in the 2.6.24 kernel"

http://lwn.net/Articles/259217/

"Notes from a container"

http://lwn.net/Articles/256389/

A new approach to user namespaces: Jonathan Corbet, April 2012

http://lwn.net/Articles/491310/

Checkpoint/restore mostly in the userspace:

http://lwn.net/Articles/451916/

Checkpoint and Restore: are we there yet? lecture by Pavel Emelyanov

http://linux.conf.au/schedule/30116/view_talk?day=thursday

TCP

TCP: RFC 793: http://www.ietf.org/rfc/rfc793.txt
TCP - provides connected-orienetd service.
MSS = Maximum segment size

tcp_sendmsg() is the main handler in the TX path.
sk_state is the state of the TCP socket.
In case it is not in TCPF_ESTABLISHED or TCPF_CLOSE_WAIT we cannot send data.
Allocation of a new segment is done via ***sk_stream_alloc_skb()***.

helper: ***tcp_current_mss()***: compute the current effective MSS.

Important structures:
struct tcp_sock:

- u32 snd_cwnd - the congestion sending window size.
- u8   ecn_flags - ECN status bits.

- ECN stands for Explicit Congestion Notification.
- can be one of the following:
- TCP_ECN_OK
- TCP_ECN_QUEUE_CWR
- TCP_ECN_DEMAND_CWR
- TCP_ECN_SEEN

There is a configurable procfs tcp_ecn entry:
/proc/sys/net/ipv4/tcp_ecn
Possible values are:
0 Disable ECN. Neither initiate nor accept ECN.
1 Always request ECN on outgoing connection attempts.
2 Enable ECN when requested by incoming connections
but do not request ECN on outgoing connections.
Default: 2

see more in *Documentation/networking/ip-sysctl.txt*

You can change the TCP initcwnd thus:

ip route change 192.168.1.101 via 192.168.1.10 dev em1 initcwnd 11
Then:
ip route

will show that the action was performed:

```
...
192.168.1.101 via 192.168.1.10 dev em1 initcwnd 11
```

tcp_v4_init_sock(): initialization of the TCP socket is done in net/ipv4/tcp_ipv4.c;
invokes tcp_init(). The the congestion sending window size is initialized to 10 (TCP_INIT_CWND).

tcp_v4_connect(): create a TCP connection. (net/ipv4/tcp_ipv4.c)

Each socket (struct sock instance) has a transmit queue named sk_write_queue.

from include/uapi/linux/tcp.h:

```
struct tcphdr {
__be16 source;
__be16 dest;
__be32 seq;
__be32 ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
__u16 res1:4,
doff:4,
fin:1,
syn:1,
rst:1,
psh:1,
ack:1,
urg:1,
ece:1,
cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
__u16 doff:4,
res1:4,
cwr:1,
ece:1,
urg:1,
ack:1,
psh:1,
rst:1,
syn:1,
fin:1;
#else
#error "Adjust your <asm/byteorder.h> defines"
#endif
__be16 window;
__sum16 check;
__be16 urg_ptr;
};
```

TCP packet loss can be detected by two events:

- a timeout
- receiving duplicate ACKs.
- When and why do we get "duplicate ACKs"?
- According to RFC 2581, "TCP Congestion Control"
- http://www.ietf.org/rfc/rfc2581.txt:
-
-

A TCP receiver SHOULD send an immediate duplicate ACK when
an out-
of-order segment arrives. The purpose of this ACK is to inform the
sender that a segment was received out-of-order and which sequence
number is expected.

see:
Congestion Avoidance and Control
Van Jacobson
Lawrence Berkeley Laboratory

Michael J. Karels
University of California at Berkeley

ee.lbl.gov/papers/congavoid.pdf

TCP timers:

Keep Alive timer - implemented in tcp_keepalive_timer() in net/ipv4/tcp_timer
TCP retransmit timer - implemented in tcp_retransmit_timer() in net/ipv4/tcp_timer

RTO - retransmission timeout.
RTT   - round trip time.

IPSEC

- Works at network IP layer (L3).
- Used in many forms of secured networks like VPNs.
- Mandatory in IPv6. (not in IPv4)
- Implemented in many operating systems: Linux, Solaris, Windows, and more.
- RFC2401
- In 2.6 kernel : implemented by Dave Miller and Alexey Kuznetsov.
- IPSec subsystem Maintainers:

  Herbert Xu and David Miller.

  Steffen Klassert was added as a maintainer in October 2012.

see:

http://marc.info/?t=135032283000003&r=1&w=2

IPSec git kernel repositories:

There are two git trees at kernel.org, an 'ipsec' tree that tracks the net tree and an 'ipsec-next' tree that tracks the net-next tree.

They are located at

git://git.kernel.org/pub/scm/linux/kernel/git/klassert/ipsec.git
git://git.kernel.org/pub/scm/linux/kernel/git/klassert/ipsec-next.git

Two data structures are important for IPSec configuration:

struct xfrm_state and struct xfrm_policy.

Both defined in ***include/net/xfrm.h***

We handle IPSec rules management (add/del/update actions, etc) from user space by accessing methods in net/xfrm/xfrm_user.c.

For example, adding a policy is done by ***xfrm_add_policy().***

This is done in response to getting XFRM_MSG_NEWPOLICY message from userspace.

Deleting a policy is done by ***xfrm_get_policy()*** when receiving XFRM_MSG_DELPOLICY.

xfrm_get_policy() also handles XFRM_MSG_GETPOLICY messages (which perform a lookup).

• Transformation bundles.

• Chain of dst entries; only the last one is for routing.

• User space tools: http://ipsectools.sf.net

• Openswan: http://www.openswan.org/ (Open Source project).

Also strongSwan: http://www.strongswan.org/

• There are also non IPSec solutions for VPN

– example: pptp

• struct xfrm_policy has the following member:

– struct dst_entry *bundles.

– ***__xfrm4_bundle_create()*** creates dst_entries (with the DST_NOHASH flag) see: *net/ipv4/xfrm4_policy.c*

• Transport Mode and Tunnel Mode.

● Show the security policies:

ip xfrm policy show

● Show xfrm states

ip xfrm state show

● Create RSA keys:

– ipsec rsasigkey verbose 2048 > keys.txt

– ipsec showhostkey left > left.publickey – ipsec showhostkey right > right.publickey

Some IPSec links:

USAGI IPv6 IPsec Development for Linux
http://hiroshi1.hongo.wide.ad.jp/hiroshi/papers/SAINT2004_kanda-ipsec.pdf

Design and Implementation to Support Multiple Key Exchange Protocols for IPsec
http://ols.fedoraproject.org/OLS/Reprints-2006/miyazawa-reprint.pdf

Linux IPv6 Networking" there is a section about IPSec
http://www.kernel.org/doc/ols/2003/ols2003-pages-507-523.pdf

Linux IPv6 Stack Implementation Based on Serialized Data State Processing
http://hiroshi1.hongo.wide.ad.jp/hiroshi/papers/yoshifuji_Mar2004.pdf

Example: Host to Host VPN (using openswan)

in /etc/ipsec.conf:

conn linuxtolinux left=192.168.0.189 leftnexthop=%direct leftrsasigkey=0sAQPPQ... right=192.168.0.45 rightnexthop=%direct rightrsasigkey=0sAQNwb... type=tunnel auto=start

● ***service ipsec start*** (to start the service)

• ***ipsec verify*** : Check your system to see if IPsec got installed and started correctly.

• ***ipsec auto –status*** – If you see “IPsec SA established” , this implies success.

• Look for errors in /var/log/secure (fedora core) or in kernel syslog Tips for hacking

• Documentation/networking/ipsysctl.txt: networking kernel tunabels

• Example of reading a hex address:

• iph->daddr == 0x0A00A8C0 or means checking if the address is 192.168.0.10 (C0=192,A8=168,00=0,0A=10).

Disable ping reply:

• echo 1 >/proc/sys/net/ipv4/icmp_echo_ignore_all

• Disable arp: ip link set eth0 arp off (the NOARP flag will be set)

• Also ifconfig eth0 arp has the same effect.

• How can you get the Path MTU to a destination (PMTU)? – Use tracepath (see man tracepath). – Tracepath is from iputils.

• Keep iphdr struct handy (printout): (from linux/ip.h)

struct iphdr { __u8 ihl:4, version:4; __u8 tos; __be16 tot_len; __be16 id; __be16 frag_off; __u8 ttl; __u8 protocol; __sum16 check; __be32 saddr; __be32 daddr; /*The options start here. */ };

• NIPQUAD() : macro for printing hex addresses

• CONFIG_NET_DMA is for TCP/IP offload.

• When you encounter: xfrm / CONFIG_XFRM this has to to do with IPSEC. (transformers). New and future trends

• IO/AT.

• NetChannels (Van Jacobson and Evgeniy Polyakov).

• TCP Offloading.

• RDMA - Remote Direct Memory Access.

- iWARP - stands for: Internet Wide Area RDMA Protocol

- Currently there are only two drivers in the kernel tree for NICS with RDMA support: (rnics) 1) drivers/infiniband/hw/amso1100

2) drivers/infiniband/hw/cxgb3. - driver for the Chelsio T3 1GbE and 10GbE adapters.

The kernel maintainer of the INFINIBAND SUBSYSTEM is Roland Dreier.

● Mulitqueus : some new nics, like e1000 and IPW2200, allow two or more hardware Tx queues. Also with virtio, patches which support multiqueue were recently sent.

In case you want to override the kernel selection of tx queue, you should implement

ndo_select_queue() member of the net_device_ops struct in your driver.

For example, this is done in ieee80211_dataif_ops struct in net/mac80211/iface.c

...

ndo_select_queue = ieee80211_netdev_select_queue

...

see Documentation/networking/multiqueue.txt

and also

Documentation/networking/scaling.txt

Managing multiple queues: affinity and other issues

Ben Hutchings - netconf 2011

vger.kernel.org/netconf2011_slides/bwh_netconf2011.pdf

In some drivers, the number of queues is passed as a module parameter: see, for example, drivers/net/ethernet/broadcom/bnx2x/bnx2x_main.c

num_queues is a module parameter (number of queues) in this driver.

You should also use ***alloc_etherdev_mq()*** in your network driver instead of ***alloc_etherdev()***

● See: “Enabling Linux Network Support of Hardware Multiqueue Devices”, OLS 2007.

● Some more info in: Documentation/networking/multiqueue.txt in recent Linux kernels.

See also Dave Miller multiqueue networking presentation he gave at the 5th Netfilter Workshop,September 11th-14th, 2007, Karlsruhe, Germany:

http://vger.kernel.org/~davem/multiqueue.odp

and also:

"Multiqueue networking", article by Corbet: *http://lwn.net/Articles/289137/*

● Devices with multiple TX/RX queues will have the NETIF_F_MULTI_QUEUE feature (include/linux/netdevice.h)

● MultiQueue nic drivers will call alloc_etherdev_mq() or alloc_netdev_mq() instead of alloc_etherdev() or alloc_netdev().

● We pass the setup method as a parameter to these methods; So , for example, with ethernet devices we pass *ether_setup()*; with wifi devices, we pass ***ieee80211_if_setup()***. (see ***ieee80211_if_add()*** in net/mac80211/iface.c)

Instat tool

Instat tool is a powerful tool, part of iproute 2 package.

Examples of usage:

lnstat -f rt_cache -k entries
shows number of routing cache entries

lnstat -f rt_cache -k in_hit
shows number of routing cache hits

Misc:

In this section there are some topics on which I intend to add more info during time.

Fragmentation:

Fragmentation of outgoing packets:

When the length of the skb is larger then the MTU of the device from which the packet is transmitted, we preform fragmentation; this is done in ***ip_fragment()*** method (net/ipv4/ip_output.c); in IPv6, it is done in ***ip6_fragment()*** in net/ipv6/ip6_output.c

Fragmentation can be done in two ways:

  - via a page array (called skb_shinfo(skb)->frags[]) (There can be up to MAX_SKB_FRAGS; MAX_SKB_FRAGS is 16 when page size is 4K).

  - via a list of SKBs (called skb_shinfo(skb)->frag_list)

      - Then method ***skb_has_frag_list()*** tests the second case (This method was called **skb_has_frags()** in the past).

When creating a socket in user space, we can tell it not to support fragmentation. This is done for example in tracepath util (part of iputils), with calling ***setsockopt().*** (tracepath util finds the path MTU)

```
...
in on = IP_PMTUDISC_DO;
setsockopt(fd, SOL_IP, IP_MTU_DISCOVER, &on, sizeof(on));
...
```

In the kernel, ***ip_dont_fragment()*** checks the value of pmtudisc field of the socket (struct inet_sock, which is embedded
the sock structure). In case pmtudisc equals IP_PMTUDISC_DO, we set the IP_DF (Don't fragment) flag in the ip header by

iph->frag_off = htons(IP_DF). See for example,
ip_build_and_send_pkt() in net/ipv4/ip_output.c

raw_sendmsg() and ***udp_sendmsg()*** use ***ip_appand_data()***, which uses the generic ip fragmentation method, ip_generic_getfrag().

Exception to this is udplite sockets, which uses udplite_getfrag() for fragmentation.

Extracting the fragment offset from the ip header and the fragmen flags:
The "frag_off" field (which is 16 bit in length) in the ip header represents the offset and the flags of the fragment.
- 13 leftmost bits are the offset. (the offset units is 8-bytes)
- 3 rightmost bits are the flags.

So in order getting the offset and the flag from the ip header can be done thus:

IP_OFFSET is 0x1FFF: a mask for getting 13 leftmost bits.

(see #define IP_OFFSET   0x1FFF in ip.h)

```
int offset, flags;
offset = ntohs(ip_hdr(skb)->frag_off);
flags = offset & ~IP_OFFSET;
offset &= IP_OFFSET;
offset <<= 3;        /* offset is in 8-byte chunks */
```

- see for example, ip_frag_queue() in net/ipv4/ip_fragment.c

Each fragment has the IP_MF flag ("More fragments") set, except for the last fragment.

The id field of the ip header is the same for all fragments.

- If a fragment is not received at the second side after a predetermined time, an ICMP is sent back; this is an ICMP_TIME_EXCEEDED with

  "Fragment Reassembly Timeout exceeded" message (ICMP_EXC_FRAGTIME).

  Notice that ICMP_TIME_EXCEEDED also is sent when ttl is set to 0, in ip_forward().

But, in that case, it is ICMP_EXC_TTL ("TTL count exceeded").

Setting the ip header id field ("identification") is very important for performing fragmentation; all fragments must have the same id so that the other side will

be able to reassemble. Assigning id to ip header is done by ***__ip_select_ident()***; see net/ipv4/route.c.

Neighboring Subsystem

● Why do we need the neighboring subsystem ?

● “The world is a jungle in general, and the networking game contributes many animals.” (from RFC 826, ARP, 1982)

● Most known protocol: ARP (in IPv6: ND, neighbour discovery)

● Ethernet header is 14 bytes long:

● Source Mac address and destination Mac address are 6 bytes each.

– Type (2 bytes). For example, (include/linux/if_ether.h)

● 0x0800 is the type for IP packet (ETH_P_IP)

● 0x0806 is the type for ARP packet (ETH_P_ARP)

● 0X8035 is the type for RARP packet (ETH_P_RARP)

Neighboring Subsystem – struct neighbour

● neighbour (instance of struct neighbour) is embedded in dst, which is in turn is embedded in sk_buff:

● Implementation: important data structures

● struct neighbour (*include/net/neighbour.h*)

– ha is the hardware address (MAC address when dealing with Ethernet) of the neighbour. This field is filled when an ARP response arrives.

- primary_key – The IP address (L3) of the neighbour.

● lookup in the arp table is done with the primary_key.

- nud_state represents the Network Unreachability Detection state of the neighbor. (for example, NUD_REACHABLE).

• int (*output)(struct sk_buff *skb);

– output() can be assigned to different methods according to the state of the neighbour. For example, neigh_resolve_output() and neigh_connected_output().

Initially, it is neigh_blackhole().

– When a state changes, than also the output function may be assigned to a different function.

• refcnt incremented by neigh_hold(); decremented by neigh_release().

We don't free a neighbour when the refcnt is higher than 1;instead, we set dead(a member of neighbour) to 1.

• timer (The callback method is neigh_timer_handler()).

• struct hh_cache *hh (defined in include/linux/netdevice.h)

• confirmed – confirmation timestamp.

– Confirmation can be also done from L4 (transport layer). – For example, dst_confirm() calls neigh_confirm(). – dst_confirm() is called from tcp_ack()

(*net/ipv4/tcp_input.c)* – and by udp_sendmsg() (net/ipv4/udp.c) and more. –

neigh_confirm() does NOT change the state

– it is the job of neigh_timer_handler().

• dev (net_device)

• arp_queue – every neighbour has a small arp queue of itself. – There can be only 3 elements by default in an arp_queue.

– This is configurable:/proc/sys/net/ipv4/neigh/default/unres_qlen

struct neigh_table

• struct neigh_table represents a neighboring table – (/include/net/neighbour.h)

– The arp table (arp_tbl) is a neigh_table. (include/net/arp.h)

– In IPv6, nd_tbl (Neighbor Discovery table) is a neigh_table also (include/net/ndisc.h) – There is also dn_neigh_table (DEcnet) (linux/net/decnet/dn_neigh.c) and clip_tbl (for ATM) (net/atm/clip.c) –

gc_timer: neigh_periodic_timer() is the callback for garbage collection. – neigh_periodic_timer() deletes FAILED entries from the ARP table. Neighboring Subsystem arp

● When there is no entry in the ARP cache for the destination IP address of a packet, a broadcast is sent (ARP request,ARPOP_REQUEST: who has IP address x.y.z...). This is done by a method called arp_solicit().(net/ipv4/arp.c) – In IPv6, the parallel mechanism is called ND (Neighbor discovery) and is implemented as part of ICMPv6. – A multicast is sent in IPv6 (and not a broadcast).

● If there is no answer in time to this arp request, then we will end up with sending back an ICMP error (Destination Host Unreachable).

● This is done by arp_error_report() , which indirectly calls ipv4_link_failure() ; see net/ipv4/route.c.

● You can see the contents of the arp table by running: “cat /proc/net/arp” or by running the “arp” from a command line.

- You can view statistics of arp cache (IPV4) by: cat /proc/net/stat/arp_cache
- You can view statistics of ndisc cache (IPV6) by: cat /proc/net/stat/ndisc_cache

● "ip neigh show" is the new method to show arp (from IPROUTE2)

- In IPv6 it is "ip -6 neigh show".

● You can delete and add entries to the arp table; see man arp/man ip.

● When using “ip neigh add” you can specify the state of the entry which you are adding (like permanent, stale, reachable, etc).

● arp command does not show reachability states except the incomplete state and permanent state: Permanent entries are marked with M in Flags:

example : arp output

Address HWtype HWaddress Flags Mask Iface 10.0.0.2 (incomplete) eth0 10.0.0.3 ether 00:01:02:03:04:05 CM eth0 10.0.0.138 ether 00:20:8F:0C:68:03 C eth0

Neighboring Subsystem – ip neigh show.

● We can see the current neighbour states:

● Example :

● ip neigh show

192.168.0.254 dev eth0 lladdr 00:03:27:f1:a1:31 REACHABLE
192.168.0.152 dev eth0 lladdr 00:00:00:cc:bb:aa STALE
192.168.0.121 dev eth0 lladdr 00:10:18:1b:1c:14 PERMANENT
192.168.0.54 dev eth0 lladdr aa:ab:ac:ad:ae:af STALE

● arp_process() handles both ARP requests and ARP responses.
– net/ipv4/arp.c

– If the target ip (tip) address in the arp header is the loopback then arp_process() drops it since loopback does not need ARP

. ... if (LOOPBACK(tip) || MULTICAST(tip))

          goto out;

out:

... kfree_skb(skb); return 0;

(see: #define LOOPBACK(x) (((x) & htonl(0xff000000)) == htonl(0x7f000000)) in linux/in.h

● If it is an ARP request (ARPOP_REQUEST) we call ip_route_input().

● Why ?

● In case it is for us, (RTN_LOCAL) we send and ARP reply. – arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha ,dev> dev_addr,sha); – We also update our arp table with the sender entry (ip/mac).

● Special case: ARP proxy server.

● In case we receive an ARP reply – (ARPOP_REPLY) –

We perform a lookup in the arp table. (by calling __neigh_lookup()) – If we find an entry, we update the arp table by neigh_update().

● If there is no entry and there is NO support for unsolicited ARP we don't create an entry in the arp table. – Support for unsolicited ARP by setting /proc/sys/net/ipv4/conf/all/arp_accept to 1. – The corresponding macro is: IPV4_DEVCONF_ALL(ARP_ACCEPT)) – In older kernels, support for unsolicited ARP was done by: – CONFIG_IP_ACCEPT_UNSOLICITED_ARP Neighboring Subsystem – lookup

● Lookup in the neighboring subsystem is done via: neigh_lookup() parameters: – neigh_table (arp_tbl) – pkey (ip address, the primary_key of neighbour struct) – dev (net_device) – There are 2 wrappers: – __neigh_lookup()

● just one more parameter: creat (a flag: to create a neighbor by ***neigh_create()*** or not))

● and ***__neigh_lookup_errno()***

Neighboring Subsystem – static entries

● Adding a static entry is done by:

arp -s ipAddress MacAddress

● Alternatively, this can be done by:

ip neigh add ipAddress dev eth0 lladdr MacAddress nud permanent

● The state (nud_state) of this entry will be NUD_PERMANENT

- ip neigh show will show it as PERMANENT.

● Why do we need PERMANENT entries ?

arp_bind_neighbour() method

● Suppose we are sending a packet to a host for the first time.

● a dst_entry is added to the routing cache by rt_intern_hash().

● We should know the L2 address of that host. – so rt_intern_hash() calls arp_bind_neighbour().

● only for RTN_UNICAST (not for multicast/broadcast). – arp_bind_neighbour(): net/ipv4/arp.c – dst-> neighbour=NULL, so it calls__neigh_lookup_errno(). – There is no such entry in the arp table. – So we will create a neighbour with neigh_create() and add it to the arp table.

• neigh_create() creates a neighbour with NUD_NONE state

– setting nud_state to NUD_NONE is done in neigh_alloc()

The IFF_NOARP flag

• Disabling and enabling arp

• ifconfig eth1 -arp

– You will see the NOARP flag now in ifconfig a

• ifconfig eth1 arp (to enable arp of the device)

• In fact, this sets the IFF_NOARP flag of net_device.

• There are cases where the interface by default is with the IFF_NOARP flag (for example, ppp interface, see *ppp_setup() (drivers/net/ppp_generic.c)*

Changing IP address

• Suppose we try to set eth1 to an IP address of a different machine on the LAN:

• First, we will set an ip for eth1 in (in Fedora Core 8,for example)

• /etc/sysconfig/networkscripts/ifcfg-eth1

• ... IPADDR=192.168.0.122 ...

and than run:

•ifup eth1

• we will get:

• Error, some other host already uses address 192.168.0.122.

• But:

• ifconfig eth0 192.168.0.122

• works ok !

• Why is it so ?

Duplicate Address Detection (DAD)

• Duplicate Address Detection mode (DAD)

• arping I eth0 D 192.168.0.10

– sends a broadcast packet whose source address is 0.0.0.0.

0.0.0.0 is not a valid IP address (for example, you cannot set an ip address to 0.0.0.0 with ifconfig)

• The mac address of the sender is the real one.

• -D flag is for Duplicate Address Detection mode.

Code: (from arp_process() ; see /net/ipv4/arp.c) /* Special case: IPv4 duplicate address detection packet (RFC2131)*/ if (sip == 0) { if (arp> ar_op == htons(ARPOP_REQUEST) &&

inet_addr_type(tip) == RTN_LOCAL && !arp_ignore(in_dev,dev,sip,tip)) arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,tip,sha,dev->dev_addr,dev> dev_addr);

goto out;

}

Neighboring Subsystem – Garbage Collection

• Garbage Collection – neigh_periodic_timer() – neigh_timer_handler() – neigh_periodic_timer() removes entries which are in NUD_FAILED state. This is done by setting dead to 1, and calling neigh_release(). The refcnt must be 1 to ensure no one else uses this neighbour. Also expired entries are removed.

• NUD_FAILED entries don't have MAC address ; see ip neigh show) Neighboring Subsystem – Asynchronous Garbage Collection

• neigh_forced_gc() performs asynchronous Garbage Collection.

• It is called from neigh_alloc() when the number of the entries in the arp table exceeds a (configurable) limit.

• This limit is configurable (gc_thresh2,gc_thresh3) /proc/sys/net/ipv4/neigh/default/gc_thresh2

/proc/sys/net/ipv4/neigh/default/gc_thresh3

– The default for gc_thresh3 is 1024.

Candidates for cleanup: Entries which their reference count is 1, or which their state is NOT permanent.

• Changing the neighbour state is done only in neigh_timer_handler().

LVS (Linux Virtual Sever)

● *http://www.linuxvirtualserver.org/*

● Integrated into the Linux kernel (in 2.4 kernel it was a patch).

● Located in: net/netfilter/ipvs in the kernel tree.

● LVS has eight scheduling algorithms.

● LVS/DR is LVS with direct routing (a load balancing solution).

● ipvsadm is the user space management tools (available in most distros).

● Direct Routing is the packet forwarding method.

● -g, gatewaying => Use gatewaying (direct routing)

● see man ipvsadm.

LVS/DR

● Example: 3 Real Servers and the Director all have the same VirtualIP (VIP).

● There is an ARP problem in this configuration.

● When you send an ARP broadcast, and the receiving machine has two or more NICs, each of them responds to this ARP request. Example: a machine with two NICs ;

● eth0 is 192.168.0.151 and eth1 is 192.168.0.152.

LVS and ARP

● Solutions

1) Set ARP_IGNORE to 1:

- echo “1” > /proc/sys/net/ipv4/conf/eth0/arp_ignore
- echo “1” > /proc/sys/net/ipv4/conf/eth1/arp_ignore

2) Use arptables. – There are 3 points in the arp walkthrough: (include/linux/netfilter_arp.h) – NF_ARP_IN (in arp_rcv() , net/ipv4/arp.c). – NF_ARP_OUT (in arp_xmit()),net/ipv4/arp.c) – NF_ARP_FORWARD (in br_nf_forward_arp(), net/bridge/br_netfilter.c)

● http://ebtables.sourceforge.net/download.html

– Ebtables is in fact the parallel of netfilter but in L2.

LVS example (ipvsadm)

- An example for setting LVS/DR on TCP port 80 with three real servers:
- ipvsadm C // clear the LVS table
- ipvsadm A t DirectorIPAddress:80
- ipvsadm -a t DirectorIPAddress:80 r RealServer1 g
- ipvsadm -a t DirectorIPAddress:80 r RealServer2 g
- ipvsadm -a t DirectorIPAddress:80 r RealServer3 g
- This example deals with tcp connections (for udp connection we should use u instead of t in the last 3 lines).

LVS example:

- ipvsadm -Ln // list the LVS table
- /proc/sys/net/ipv4/ip_forward should be set to 1
- In this example, packets sent to VIP will be sent to the load balancer; it will delegate them to the real server according to its scheduler. The dest MAC address in L2 header will be the MAC address of the real server to which the packet will be sent. The dest IP header will be VIP.
- This is done with NF_IP_LOCAL_IN.

ARPD – arp user space daemon

- ARPD is a user space daemon; it can be used if we want to remove some work from the kernel.
- The user space daemon is part of iproute2 (/misc/arpd.c)
- ARPD has support for negative entries and for dead hosts.

– The kernel arp code does NOT support these type of entries!

- The kernel by default is not compiled with ARPD support; we should set CONFIG_ARPD for using it:
- Networking Support-> Networking Options-> IP: ARP daemon support.
- see: /usr/share/doc/iproute2.6.22/arpd.ps (Alexey Kuznetsov).

• We should also set app_probes to a value greater than 0 by setting – /proc/sys/net/ipv4/neigh/eth0/app_solicit – This can be done also by the a (active_probes) parameter. – The value of this parameter tells how many ARP requests to send before that neighbour is considered dead.

• The k parameter tells the kernel not to send ARP broadcast; in such case, the arpd daemon is not only listening to ARP requests, but also send ARP broadcasts.

• Activation:

• arpd a 1 k eth0 &

• On some distros, you will get the error db_open: No such file or directory unless you simply run mkdir /var/lib/arpd/ before (for the arpd.db file).

• Pay attention: you can start arpd daemon when there is no support in the kernel(CONFIG_ARPD is not set).

• In this case you, arp packets are still caught by arpd daemon get_arp_pkt()

```
(misc/arpd.c)
```

• But you don't get messages from the kernel.

• get_arp_pkt() is not called.(misc/arpd.c)

• Tip: to check if CONFIG_ARPD is set, simply see if there are any results from

– cat /proc/kallsyms | grep neigh_app

Mac addresses

• MAC address (Media Access Control)

• According to specs, MAC address should be unique.

• The 3 first bytes specify a hw manufacturer of the card.

• Allocated by IANA.

There are exceptions to this rule.

– Ethernet HWaddr 00:16:3E:3F:6E:5D

ARPwatch (detect ARP cache poisoning)

• Changing MAC address can be as a result of some security attack (ARP cache poisoning).

• Arpwatch can help detect such an attack.

• Activation: arpwatch d i eth0 (output to stderr)

• Arpwatch keeps a table of ip/mac addresses and senses when there is a change.

• d is for redirecting the log to stderr (no syslog, no mail).

• In case someone changed MAC address on the same network, you will get a message like this: ARPwatch Example

From: root (Arpwatch) To: root Subject: changed ethernet address (jupiter) hostname: jupiter ip address: 192.168.0.54 ethernet address: aa:bb:cc:dd:ee:ff ethernet vendor: <unknown> old ethernet address: 0:20:18:61:e5:e0 old ethernet vendor: ...

Neighbour states

• neighbour states

neigh_alloc() Reachable Incomplete None Stale Delay Probe Neighboring Subsystem

•– NUD_NONE

– NUD_REACHABLE

– NUD_STALE

– NUD_DELAY

– NUD_PROBE

– NUD_FAILED

– NUD_INCOMPLETE

• Special states:

• NUD_NOARP

• NUD_PERMANENT

• No state transitions are allowed from these states to another state.

Neighboring Subsystem – states

- NUD state combinations:
- NUD_IN_TIMER (NUD_INCOMPLETE|NUD_REACHABLE| NUD_DELAY| NUD_PROBE)
- NUD_VALID (NUD_PERMANENT|NUD_NOARP| NUD_REACHABLE| NUD_PROBE|NUD_STALE|NUD_DELAY)
- NUD_CONNECTED (NUD_PERMANENT|NUD_NOARP| NUD_REACHABLE)
- When a neighbour is in a STALE state it will remain in this state until one of the two will occur – a packet is sent to this neighbour. – Its state changes to FAILED.
- neigh_resolve_output() and neigh_connected_output().
- net/core/neighbour.c
- A neighbour in INCOMPLETE state does not have MAC address set yet (ha member of neighbour)
- So when neigh_resolve_output() is called, the neighbour state is changed to INCOMPLETE.
- When neigh_connected_output() is called, the MAC address of the neighbour is known; so we end up with calling ***dev_queue_xmit()***, which calls the ndo_start_xmit() callback method of the NIC device driver.
- The ***ndo_start_xmit()*** method actually puts the frame on the wire.

Change of IP address/Mac address

- Change of IP address does not trigger notifying its neighbours.
- Change of MAC address , NETDEV_CHANGEADDR ,also does not trigger notifying its neighbours.
- It does update the local arp table by *neigh_changeaddr().*

– Exception to this is irlan eth: irlan_eth_send_gratuitous_arp() – (net/irda/irlan/irlan_eth.c) – Some nics don't permit changing of MAC address – you get: SIOCSIFHWADDR: Device or resource busy.

Flushing the arp table

- Flushing the arp table:
- ip statistics neigh flush dev eth0
-

- Round 1, deleting 7 entries ***

-

- Flush is complete after 1 round ***

- Specifying twice statistics will also show which entries were deleted, their mac addresses, etc...
- ip statistics statistics neigh flush dev eth0
- 192.168.0.254 lladdr 00:04:27:fd:ad:30 ref 17 used 0/0/0 REACHABLE
-
- *** Round 1, deleting 1 entries ***
- *** Flush is complete after 1 round ***
- calls neigh_delete() in net/core/neighbour.c
- Changes the state to NUD_FAILED

Virtual network devices

The tx_queue_len of virtual devices is usually 0 as they do not hold a queue of their own; so, for example, if you will create a vlan with vconfig or a bridge with brctl, ifconfig will show that tx_queue_len is 0.

see:

```
br_dev_setup() in net/bridge/br_device.c
    ...
    dev->tx_queue_len = 0;
    ...
and
vlan_setup() in
```

```
    net/8021q/vlan_dev.c
    ...
        dev->tx_queue_len    = 0;
    ...
```

and
bond_setup()
in drivers/net/bonding/bond_main.c:

```
...
bond_dev->tx_queue_len = 0;
...
```

and macvlan_setup()
in drivers/net/macvlan.c:

```
    ...
    dev->tx_queue_len    = 0;
    ...
```

and
vxlan_setup() in drivers/net/vxlan.c

```
   ...
   dev->tx_queue_len = 0;
   ...
```

With pimreg (multicast) device, tx_queue_len is not initialized at all; so when running ifconfig on pimreg device, you get:

```
txqueuelen 0  (UNSPEC)
```

Notice that for virtual devices, like loopback and vlan, the qdisc is the noqueue qdisc.

So for example, when running "ip addr show" you will see for the loopback device:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
```

and for the a vlan device (eth0.6 in this example):
eth0.6@eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state

where as in other, non virtual devices, you will have pfifo_fast qdisc.

Some more implementation details about achieving it:
in attach_one_default_qdisc() (net/sched/sch_generic.c) we have this code:

```
static void attach_one_default_qdisc(struct net_device *dev,
                                     struct netdev_queue *dev_queue,
                                     void *_unused)
 {
 struct Qdisc *qdisc = &noqueue_qdisc;

 if (dev->tx_queue_len) {
   qdisc = qdisc_create_dflt(dev_queue, &pfifo_fast_ops, TC_H_ROOT);
   if (!qdisc) {
      netdev_info(dev, "activation failed\n");
      return;
   }
 }
 dev_queue->qdisc_sleeping = qdisc;
}
```

So when dev->tx_queue_len is 0, as in the case with virtual devices, we use the noqueue_qdisc and do not call qdisc_create_dflt().

Another feature of virtual devices is that they appear under **/sys/devices/virtual/net**. So for example, after boot, we have **/sys/devices/virtual/net/lo/** entry for the loopback device. The entries which are created under **/sys/devices/virtual/net** for virtual network device are created not because tx_queue_len of virtual devices is 0, and not because of the noqueue_qdisc of virtual devices. The reason they are created is because with virtual devices, we do **not** call the SET_NETDEV_DEV() macro. In case you'll look at this simple macro, which should always be called before **register_netdev()**, you'll see that all it does is assign the parent member in net_device. How does this has to do with the virtual entry under sysfs? The answer is that devices which have no parent are considered "virtual" class-devices. And if you will look for the implementation details, you see that ***register_netdev()*** calls ***device_add()*** in **netdev_register_kobject().** And ***device_add()*** (in drivers/base/core) creates an entry under /sys/devices/virtual/net for a device whose parent is null (see get_device_parent() method, which is invoked from device_add().

So, for example, in the case of creating a tun device (which is a virtual device) by:

ip tuntap add tun0 mode tun

You will have an entry under:

/sys/devices/virtual/net/tun0/

And when creating a tap device (which is also a virtual device) by:

ip tuntap add tap0 mode tap

You will have an entry under:

/sys/devices/virtual/net/tap0/

We remove the tuntap devices by:
ip tuntap del tap0 mode tap
ip tuntap del tun0 mode tun

Tunnels

What is the difference between ipip tunnel and gre tunnel?

gre tunnel supports multicasting whereas ipip tunnel does support only unicast.

MTU

MTU stands for Maximum Transfer Unit (or sometimes also Maximum Transfer Unit).

MTU is symmetrical and applies both to receive and transmit.

Layer 3 should not pass pass an skb which has payload bigger than an MTU.

GSO and TSO are exceptions; in such cases, the device will separate the packet into smaller packets, which are smaller than the MTU.

Multicasting

struct net_device holds two lists of addresses (instances of struct netdev_hw_addr_list):

- uc is the unicast mac addresses list
- mc is the multicast mac addresses list

You add multicast addresses to the multicast mac addresses list (mc) both in IPv4 and IPv6 by:

dev_mc_add() (in net/core/dev_addr_lists.c).

In ipv4, a device adds the 224.0.0.1 multicast address (IGMP_ALL_HOSTS , see include/linux/igmp.h), in ip_mc_up() (see net/ipv4/igmp.c).

GSO

For implementing GSO, a method called gso_segment was added to net_protocol
struct in ipv4 (see include/net/protocol.h)
For tcp, this method is tcp_tso_segment() (see tcp_protocol in net/ipv4/af_inet.c).
There are drivers who implement TSO; for example, e1000e of Intel.

A member called gso_size was added to skb_shared_info
Also a helper method called ***skb_is_gso()*** was added; this method checks whether gso_size of skb_shared_info is 0 or not (returns true when gso_size is not 0).

Grouping net devices

An interesting patch from Vlad Dogaru (January 2011) added support for network device groups. This was done by adding a member called "group" to struct net_device, and an API to set this group from kernel (dev_set_group()) and from user space. By default, all network devices are assigned to the default group, group 0.
(INIT_NETDEV_GROUP); see alloc_netdev_mqs() in net/core/dev.c

ethtool

struct ethtool_ops had recently been added EEE support (Energy Efficient Ethernet) in the form of a new struct
called ethtool_eee (added in include/linux/ethtool.h) and two methods ***get_eee()*** and ***set_eee()***

IP address

In IPv4, when you set and IP address, you in fact assign it to ifa->ifa_local.

(ifa is a pointer to struct in_ifaddr)

When running "ifconfig" or "ip addr show", you in fact issue an SIOCGIFADDR ioctl,

for getting interface address, which is handled by

struct in_device from include/linux/inetdevice.h has a list : ifa_list, which is the IP ifaddr chain.

ifa_local is a member of struct in_ifaddr which represents ipv4 address.

IPV6

In IPV6, the neighboring subsystem uses ICMPV6 for Neighboring messages (instead of ARP messages in IPV4).

- There are 5 types of ICMP codes for neighbour discovery messages:

NEIGHBOUR SOLICITATION (135) parallel to ARP request in IPV4
NEIGHBOUR ADVERTISEMENT (136) parallel to ARP reply in IPV4

ROUTER SOLICITATION (133)

ROUTER ADVERTISEMENT (134)
REDIRECT (137)

Special Addresses:

All nodes (or : All hosts) address: FF02::1
– ipv6_addr_all_nodes() sets address to FF02::1
– All Routers address: FF02::2

– ipv6_addr_all_routers() sets address to FF02::2
Both in include/net/addrconf.h

- In IPV6: All addresses starting with FF are multicast address.

• IPV4: Addresses in the range 224.0.0.0 – 239.255.255.255
are multicast addresses (class D).

Privacy Extensions
● Since the address is build using a prefix and MAC address,
the identity of the machine can be found.
● To avoid this, you can use Privacy Extensions.
– This adds randomness to the IPV6 address creation process.
(calling get_random_bytes() for example).
● RFC 3041 Privacy Extensions for Stateless Address
Autoconfiguration in IPv6.
● You need CONFIG_IPV6_PRIVACY to be set when building the kernel.

Hosts can disable receiving Router Advertisements by
setting Autoconfiguration
● When a host boots, (and its cable is connected) it first
creates a Link Local Address.
– A Link Local address starts with FE80.
– This address is tentative (only works with ND messages).
● The host sends a Neighbour Solicitation message.
– The target is its tentative address, the source is all zeros.
– This is DAD (Double Address Detection).
● If there is no answer in due time, the state is changed to
permanent. (IFA_F_PERMANENT)

• Then the host send Router Solicitation.
– The target address of the Router Solicitation
message is the All Routers multicast address
FF02::2
– All the routers reply with a Router Advertisement message.
– The host sets address/addresses according to the prefix/prefixes
received and starts the DAD process as before.

• At the end of the process, the host will have two (or more) IPv6
addresses:

– Link Local IPV6 address.
– The IPV6 address/addresses which was built using the prefix (in case that there is one or more routers sending RAs).

• There are three trials by default for sending Router Solicitation.
– It can be configured by:
● /proc/sys/net/ipv6/conf/eth0/router_solicitations

VLAN (802.1Q)

VLAN (Virtual LAN) enables us to partition a physical network. Thus, different broadcast domains are created. This is achieved by inserting VLAN tag into the packet.

The VLAN tag is 4 bytes: 2 bytes are Tag Protocol Identifier (TPID), which has a value of 0x8100; 2 bytes are the Tag Control Identifier (TCI). (In linux documentation, TCI is termed "tag control information", see vlan_tci in sk_buff struct, *include/linux/sk_buff*)

The VLAN tag is inserted between the source mac address and ethertype of the eth header. The ***vlan_insert_tag()*** method implements this tag insertion (*include/linux/if_vlan.h*).

struct **vlan_ethhdr** represents vlan ethernet header (ethhdr + vlan_hdr).

h_vlan_proto in this struct will get always 0x8100 value.

h_vlan_TCI in this struct is the TCI, composed from priority and VLAN ID.

vlan_insert_tag() is invoked from the vlan rx handler, ***vlan_do_receive().***

(see *include/linux/if_vlan.h*).

VLAN support in linux is under net/8021q.

There is also the macvlan driver (drivers/net/macvlan.c).

The header file for vlan is include/linux/if_vlan.h
The header file for macvlan is *include/linux/if_macvlan.h*

The maintainer of vlan is Patrick McHardy.

VLAN supports almost everything a regular ethernet interface does, including firewalling, bridging, and of course IP traffic.

You will need the 'vconfig' tool from the VLAN project in order to effectively use VLANs.

In fedora, there is a package ("rpm") called **vconfig**; you install it by "yum install vconfig".

In Ubuntu, vconfig belongs to a package named "vlan"; you install it by "apt-get install vlan"

You can also set vlan/macvlan with "vconfig" utility thus:

vconfig add p2p1 vlanID

Notice that you can add up to 4094 VLANs per ethernet interface.

In case you try to add more than 4094, you will get this error:

ERROR: trying to add VLAN #vlanID to IF -:p2p1:- error: Numerical result out of range

According to **http://en.wikipedia.org/wiki/IEEE_802.1Q**:

"The hexadecimal values of 0x000 and 0xFFF are reserved.".

You can also set vlan/macvlan with "ip" utility:

ip link add link p2p1 name p2p1.100 type vlan id 5
ip link add link p2p1 name p2p1#101 address 00:aa:bb:cc:dd:ee type macvlan

You can get some info about vlan devices in procfs under:

- **/proc/net/vlan**
- **/proc/net/vlan/config** (this includes info about vlan id).

See More info in VLAN web page:

http://www.candelatech.com/~greear/vlan.html

VLAN traffic has 0x8100 type (ETH_P_8021Q).

For network devices which do not support VLAN TX HW acceleration (the NETIF_F_HW_VLAN_TX flag is not set), we insert the VLAN tag by calling ***__vlan_put_tag()*** in ***dev_hard_start_xmit().***

__vlan_put_tag() is a wrapper which calls ***vlan_insert_tag()*** (both are in ***include/linux/if_vlan.h)***.
In ***vlan_insert_tag()*** the mac_header pointer (skb->mac_header) is decremented by 4 (VLAN_HLEN) and we insert the vlan tag where needed.
Also skb->protocol is set to be 8021q (ETH_P_8021Q)

Example for such driver without VLAN TX HW acceleration support is RealTek 8139too driver: ***drivers/net/ethernet/realtek/8139too.c.***

Example of a driver with VLAN TX HW acceleration support is:

drivers/net/ethernet/realtek/r8169.c.

VLAN interface is a virtual device (you set the netdevice tx_queue_len to be 0). In case VLAN is compiled as a kernel module, its name is 8021q.ko.

Adding/Deleting vlans is done via ioctls which are sent from user space; for example, adding vlan is triggered by receiving **ADD_VLAN_CMD** ioctl from user space. This triggers the ***register_vlan_device()*** method. As said above, you cannot add more than 4094 vlans to a single ethernet device. In the beginning of ***register_vlan_device()*** we have:

```
if (vlan_id >= VLAN_VID_MASK)
   return -ERANGE;
```

And VLAN_VID_MASK is 0x0fff (4095).

When returning -ERANGE, we get the error mentioned above:

error: Numerical result out of range

Deleting vlan is done by receiving DEL_VLAN_CMD ioctl from user space. This triggers the ***unregister_vlan_dev()*** method.

These ioctls are defined in include/uapi/linux/if_vlan.h
(Once they were defined in include/linux/if_vlan.h).

The handler for this ioctls is ***vlan_ioctl_handler()*** in net/8021q/vlan.c

By default, ethernet header reorders are turned off. (The VLAN_FLAG_REORDER_HDR flag is not set). When ethernet header reorders are set, dumping the device will appear as a common ethernet device without vlans.

VLAN private device data is represented by
struct **vlan_dev_priv** (net/8021q/vlan.h)

It has two arrays in it: egress_priority_map and ingress_priority_map.

We add entries to egress_priority_map array
by ***vlan_dev_set_egress_priority().***
This is triggered by sending SET_VLAN_EGRESS_PRIORITY_CMD ioctl from user space (vconfig set_egress_map)

We add entries to ingress_priority_map array
by ***vlan_dev_set_ingress_priority()***.
This is triggered by sending SET_VLAN_INGRESS_PRIORITY_CMD ioctl from user space (vconfig set_ingress_map)

You can enable vlan reordering with vconfig thus:

vconfig set_flag eth0.100 1 1

And you can view the reordering flag thus:
cat /proc/net/vlan/eth0.100

TBD: Should it be done with eth0.100 0 0 ?
You can disable vlan reordering with vconfig thus:
vconfig set_flag eth0.100 1 1

Note that there are chances that the man page/help of some distros is not accurate about this.

It says
set_flag [vlan-device] 0 | 1
And it should be:
set_flag [vlan-device] [flag-num] 0 | 1

See for example: https://bugzilla.redhat.com/show_bug.cgi?id=468813

Helper methods:

int is_vlan_dev(struct net_device *dev) : checks whether the device is a vlan device, by checking the priv_flags of net_device. Defined in *include/linux/if_vlan.h*

bool vlan_uses_dev(const struct net_device *dev) : checks whether is device is used by vlan (by checking whether vlan_info member of the device is null or not).

vlan_tx_tag_present(skb) : checks whether the VLAN_TAG_PRESENT flag is set. (Defined in ***include/linux/if_vlan.h***).

When we encounter in the RX path packets with vlan tag, the VLAN packets are handled by ***vlan_do_receive()*** which is invoked from ***__netif_receive_skb().***

vlan_do_receive() is implemented in *net/8021q/vlan_core.c.*

There are some adapters which support VLAN hardware acceleration offloading. You can get info about VLAN hardware acceleration offloading with ethtool:

ethtool -k p2p1

```
...
rx-vlan-offload: on
tx-vlan-offload: on
...
```

When **NETIF_F_HW_VLAN_FILTER** is set, adding/deleting a vlan interface triggers calling **ndo_vlan_rx_add_vid/ndo_vlan_rx_kill_vid**, respectively. There are drivers who implement the ndo_vlan_rx_add_vid/ndo_vlan_rx_kill_vid callbacks (of net_device_ops); for example, e1000 and e1000e of Intel, starfire of adaptec, and more.

Bonding Driver (Link aggregation)

The bonding network driver is for putting multiple physical ethernet devices into one logical one, what is often termed link aggregation/trunking/Link bundling/Ethernet/network/NIC bonding. (these terms can be considered as synonyms).

The new generation of the bonding driver is called teaming. It has also a user space part called libteam.

see also _Teaming driver section._

ifenslave is an iputils package.

You can set link aggregation with ifenslave like in the following example:

```
modprobe bonding mode=balance-alb miimon=100
ifconfig bond0 192.168.1.1
ifenslave bond0 eth0
ifenslave bond0 eth1
```

You can set vlan device over a bonding interface;

For example, on the bond0 you created, you configure a vlan thus:

vconfig add bond0 100

If you will try to configure a vlan on an empty bonding device (before enslaving at least one interface to it) you will get an error:

```
#> vconfig add bond0 100
ERROR: trying to add VLAN #100 to IF -:bond0:- error: Operation not supported.
```

How is this implemented ?

An empty bonding device has NETIF_F_VLAN_CHALLENGED set.

In vlan_check_real_dev(), which is invoked
from register_vlan_device() when
configuring VLAN over a device, we check the
NETIF_F_VLAN_CHALLENGED flag
of the device on which we are setting the VLAN.
If this flag is set, we return -EOPNOTSUPP:

```
int vlan_check_real_dev(struct net_device *real_dev, u16 vlan_id)
{
...
...
if (real_dev->features & NETIF_F_VLAN_CHALLENGED) {
   pr_info("VLANs not supported on %s\n", name);
   return -EOPNOTSUPP;
  }
...
...
}
```

The Maintainers of the bonding driver are Jay Vosburgh and Andy Gospodarek.

In the kernel, the bonding code is in drivers/net/bonding.

-

Teaming network device

location: *drivers/net/team*

Teaming network device is in fact the new bonding driver.
Teaming network device is for putting multiple physical ethernet devices
into one logical one, what is often termed link aggregation/trunking/Link bundling/Ethernet/network/NIC bonding.
(these terms can be considered as synonyms).
Team has also a user-space util, libteam.

The team driver registers an RX handler
by netdev_rx_handler_register().
The handler is ***team_handle_frame().***
This is common in a virtual driver; also the bonding driver registers an RX handler
named bond_handle_frame() and also the bridge driver registers a handler named ***br_handle_frame()***. These handlers are invoked in ***__netif_receive_skb()*** (net/core/dev.c)

Adding/Deleting a team device is done by:
ip link add name team0 type team
ip link del team0

ip link add name team0 type team triggers a call to team_newlink(), which is one of the rtnl_link_ops callbacks.

When you add a team device thus, the hw address is random, generated by
eth_hw_addr_random(). In case you want to specify an hw address when creating
the team device, you can do it thus, for example:
ip link add name team0 address 00:11:22:33:44:55 type team

Notice that the "type team" should be in the end.
Trying:
ip link add name team0 type team address 00:11:22:33:44:55
will fail with this error:
Garbage instead of arguments "address ...". Try "ip link help"

You can notice that team rtnl_link_ops does has a newlink callback (team_newlink)
but does not have dellink callback. So how is unregistering of the team0 done
in this case ? The answer is simple, and apply also to other devices which do not set the dellink callback in rtnl_link_ops: When registering a device, in case we
did not define dellink in the rtnl_link_ops, then we assign
the generic unregister_netdevice_queue() method to the dellink callback of rtnl_link_ops. And when running "ip link del team0", we arrive at rtnl_dellink() , which
eventually calls unregister_netdevice_queue() and unregisters the net_device.

see, in net/core/rtnetlink.c

```
int __rtnl_link_register(struct rtnl_link_ops *ops)
{
if (!ops->dellink)
   ops->dellink = unregister_netdevice_queue;
```

...
return 0;
}

Adding p2p1:
ip link set p2p1 master team0

ip link set p2p1 master team0 triggers a call to team_port_add()

Removing p2p1:
ip link set p2p1 nomaster

ip link set eth1 nomaster triggers a call to team_port_del(). (In fact, this is done via invoking the ndo_del_slave()member of rtnl_link_ops in do_set_master() of rtnetlink (net/core/rtnetlink.c)

Notice that p2p1 must be down for this operation to succeed; in case it is up, you
will get "RTNETLINK answers: Device or resource busy" error.

Trying to add a loopback device to a team device will fail.

For example,

ip link set lo master team0

emits this error in the kernel log:

team0: Device lo is loopback device. Loopback devices can't be added as a team port

There are four modules (or for "modes", which is the word the team code uses) in the team driver:

team_mode_broadcast.c
The broadcast mode is a basic mode in which all packets are sent via all available ports.

team_mode_roundrobin.c

The roundrobin mode is a basic mode with very simple transmit port-selecting algorithm based on looping around the port list.
This is the only mode able to run on its own without userspace interactions.

team_mode_activebackup.c
The activebackup mode, in which only one port is active at a time and able to perform transmit and receive of skb.
The rest of the ports are backup ports.
This Mode exposes activeport option through which userspace application can specify the active port.

team_mode_loadbalance.c
The loadbalance mode is a more complex mode used for example for LACP (Link Aggregation Control Protocol) and userspace controlled transmit and receive load balancing.
LACP protocol is part of the 802.3ad standard
and is very common for smart switches.

team_mode_register()/team_mode_unregister() is the API for registering/unregsitering a mode.

A mode can register options via ***team_options_register().***
Only two modes uses the options mechanism. One is **team_mode_activebackup,** and the second is **team_mode_loadbalance.**

The teaming network driver uses the Generic Netlink API; it calls ***genl_register_family()***
and ***genl_register_mc_group()*** and other methods of the Generic Netlink API.

In fedora 16/17 there is an rpm for the user-space util (libteam).

Team Infrastructure Specification:

https://fedorahosted.org/libteam/wiki/InfrastructureSpecification

see: https://fedorahosted.org/libteam/

https://github.com/jpirko/libteam

Jiri Pirko presentation: http://www.pirko.cz/teamdev.pp.pdf

The maintainer of the teaming driver is Jiri Pirko.

PPP

The most commonly used user space daemon for ppp is pppd.

It can be downlowded from here:
ftp://ftp.samba.org/pub/ppp/
pppd website is:
http://ppp.samba.org/
In case you need to use pppoe in conjunction with ppp, you should install rp-pppoe:
http://www.roaringpenguin.com/products/pppoe

ppp setting are configurable via /etc/ppp.

The generic ppp layer is implemented
in ppp_generic.c (***drivers/net/ppp/ppp_generic.c***).

PPPoE and PPPL2TP uses the generic ppp layer.

You register a ppp generic channel by calling
the **ppp_register_net_channel()** method of the ppp_generic module.

This is done, for example,
in ***pppoe_connect()*** (drivers/net/ppp/pppoe.c)
and in ***pppol2tp_connect()*** (net/l2tp/l2tp_ppp.c).

These two modules also call ***ppp_input()*** for handling
receiving of PPP packets over the ppp channel.

Unregistering is done by the ***ppp_unregister_channel()*** method of the ppp_generic module.

pppox_unbind_sock() calls ***ppp_unregister_channel()*** in drivers/net/ppp/pppox.c.
For pppoe, ***pppox_unbind_sock()*** is invoked when a PPPoE socket is closed. (***pppoe_release()*** in http://lxr.free-electrons.com/source/drivers/net/ppp/pppoe.c).

For l2tp_ppp, ***pppox_unbind_sock()*** is invoked
by ***pppol2tp_session_close()*** and ***pppol2tp_release().***

PPPoE

PPPoE stand for Point-to-Point Protocol over Ethernet.
defined in RFC 2516:
http://www.ietf.org/rfc/rfc2516.txt

PPPoE is implemented in pppoe.c. (*drivers/net/ppp/pppoe.c*)

For establishing PPPoE connection, there are two stages: the Discovery stage and the Session stage.

The Discovery stage consists of four steps between the client computer
and the PPPoE server (access concentrator) at the ISP.

1) PADI (Initiation)
2) PADO (Offer)
3) PADR (Request)
4) PADS (Session confirmation).

The Discovery stage is managed the pppd daemon.
You end a session by sending a PADT packet (termination packet).

The Discovery stage packets has an ehtertype of 0x8863 (ETH_P_PPP_DISC, defined in include/uapi/linux/if_ether.h).

The session stage packets has an ehtertype of 0x8864 (ETH_P_PPP_SES, also
defined in include/uapi/linux/if_ether.h).

SKB RECYCLE

skb_recycle was a Linux kernel network stack feature which was removed. When we don't need anymore an skb, we free its memory by calling (for example) ***__kfree_skb()***. The skb_recycle patch is based mainly on adding code in ***__kfree_skb(),*** so that this skb will not be freed. Instead we will initialized members of skb so the result will be as of a new skb which was just created.

See: "generic skb recycling" - a patch by Lennert Buytenhek
http://lwn.net/Articles/332037/

On 5.10.12, a patch was sent to netdev by Eric Dumazet titled "net: remove skb recycling"; this patch was applied.
see
http://marc.info/?l=linux-netdev&m=134945424730580&w=2
http://marc.info/?l=linux-netdev&m=134958489526234&w=2

According to this patch, since the skb recycling feature got little interest and many bugs, it was suggested to remove it.

Usage of skb_recycle was only in 5 ethernet drivers:

calxeda/xgmac.c ,freescale/gianfar.c ,freescale/ucc_geth.c, marvell/mv643xx_eth.c and stmicro/stmmac/stmmac_main.c

TUN/TAP

TUN/TAP provides packet reception for transmission for user space programs. It can be seen as a simple Point-to-Point or Ethernet device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program.

TUN/TAP is a driver which enables us to receive packets from user space and send packets to user space. TUN/TAP is different from other virtual devices in that it does not relay on real devices for its work; it is a purely sw driver which work with user space sockets.

The implementation is in drivers/net/tun.c.

The tun driver has two net_device_ops instances:

- tap_netdev_ops for tap devices.
- tun_netdev_ops for tun devices .

The tun device is /dev/net/tun; it is a character device, created with ***misc_register().***

To insert tuntap module you should run: modprobe tun.
With recent iproute2, you can create tun/tap devices with ip tuntap

command.
see: ip tuntap help

For example:

ip tuntap add tap0 mode tap

or
ip tuntap add tun0 mode tun

Notice that if you try to delete a nonexistent tun or tap device, you will not get an error message or any warning.

Calling ***register_netdevice()*** creates a folder under sysfs for this device. So if the device name is "deviceName",
then ***/sys/class/net/deviceName*** will
be generated. This is also the case with regular ethernet devices like eth0, eth1,.... However, with tun/tap, three additional entries are created (via a call to ***device_create_file()*** in ***tun_set_iff()***). These are "tun_flags", "owner" and "group".

Notice that you will fail with "rmmod tun" if you did not remove the tuntap devices before, with "Module tun is in use" error.

Tun devices do not have mac addresses, but tap devices have a random hw address which was created by
calling ***eth_hw_addr_random().***

Trying to set a mac address on a tun2 device will give an error; for example,

```
ifconfig tun2 hw ether 00:01:02:03:04:05
SIOCSIFHWADDR: Operation not supported
```

On a tap devices, changing the mac address in this way is possible.

With tun device, the ***tun_net_open()*** and ***tun_net_close()*** methods are called when you run "ifconfig tun0 up" and "ifconfig tun0 down", respectively.

The same is true also with tap device; ***tun_net_open()*** is invoked when calling "ifconfig tap0 up" and ***tun_net_close()*** is invoked when calling "ifconfig tap0 down"

calling fd = open("/dev/net/tun")
triggers ***tun_chr_open() ,****and* calling close(fd) triggers
tun_chr_close().

Following is a simple user space app which create a tun device.

Notice that calling TUNSETPERSIST is mandatory in this program. In case we will not call this method, then when exiting the program the fd (of "/dev/net/tun") will be closed and tun_chr_close() will be invoked, as described above. In case TUNSETPERSIST is not set, ***unregister_netdevice(dev)*** will be called (by ***__tun_detach()***).

In case we set the TUN_NO_PI flag (note set in the example below) this means that packet information (PI) will not be provided. Packet Information is 4 bytes which are added when the flag is not set. These 4 bytes are 2 bytes of flags, and 2 bytes of protocol. Wireshark sniffer does not show these 4 bytes.

see: include/uapi/linux/if_tun.h:

```
struct tun_pi {
__u16 flags;
__be16 proto;
};
```

```
// tun.c

#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/if_tun.h>
#include <linux/socket.h>
#include <stdio.h>

int main()
{
  int fd,err;
  struct ifreq ifr;
```

```
 fd = open("/dev/net/tun",O_RDWR);
 if (fd < 0) {
   printf("fd < 0 in open\n");
  return -1;
  }
 memset(&ifr, 0, sizeof(ifr));
  ifr.ifr_flags = IFF_TUN;
 strncpy(ifr.ifr_name, "tun1", IFNAMSIZ);

  err = ioctl(fd, TUNSETIFF, (void*)&ifr);

  if (err < 0) {
    printf("err<0 after TUNSETIFF, ioctl\n");
    close(fd);
    return -1;
  }

  err = ioctl(fd, TUNSETPERSIST, 1);
  if (err < 0) {
    printf("err<0 after TUNSETPERSIST ioctl\n");
    close(fd);
    return -1;
  }
}
```

The only method that the tun driver export is tun_get_socket(), and it is
used in the vhost driver (drivers/vhost/net.c).

tunctl is an older tool for creating tun/tap devices http://tunctl.sourceforge.net/

You can also use a util from openvpn to create a tun/tap device:

openvpn --mktun --dev tun2
openvpn --rmtun --dev tun2

By default, when the device name starts with "tun", "openvpn --mktun" creates a TUN device. When the device name starts with "tap", "openvpn --mktun" creates a TAP device. However, if you need for some reason to create a tap device which its name starts with tun, you still can do it thus:

```
 openvpn --mktun --dev tun11 --dev-type tap
```

Notice that also here when you try to delete a nonexisting tun/tap device, you don't get any warning.

TUN/TAP devices are widely used, in virtualization and in other fields. For example, with virt-manager, libvirt and KVM, when we start a guest, a TAP
device named vnet0 is created on the host. It is added to a bridge interface
on the host, called virbr0, with 192.168.122.1 ip address. In the guest, you can add the host bridge interface as a default gateway in order to be connected to the outside WAN.

For implementation details of creating the tap device in libvirt, look in virNetDevTapCreate() method in src/util/virnetdevtap.c of the libvirt package.

See more info about tuntap in Documentation/networking/tuntap.txt

See also this good link about tuntap:

http://backreference.org/2010/03/26/tuntap-interface-tutorial/

The maintainer is Maxim Krasnyansky.
web site: http://vtun.sourceforge.net/tun.

In interesting patch series, adding multiqueue support for tuntap, was sent by Jason Wang in October 2012:
http://www.spinics.net/lists/netdev/msg214869.html

Also an ioctl called TUNSETQUEUE was added ; this ioctl, this IFF_ATTACH_QUEUE/IFF_DETACH_QUEUE flags, enables attaching/detaching a queue from user space.

see:

http://www.spinics.net/lists/kernel/msg1429560.html

Following is an example of using tun multiqueues. Please notice that we set IFF_MULTI_QUEUE when calling TUNSETIFF; later on, we call TUNSETPERSIST on the same fd , and then open a new fd and call TUNSETQUEUE with IFF_ATTACH_QUEUE flag set, and a third fd, on which we again call TUNSETQUEUE with IFF_ATTACH_QUEUE flag set. The reason for the pause() at the end is that without it all the file descriptors will be closed. Closing the fd invokes tun_chr_close(), which subsequently call tun_detach(), removes the sys queue entries and unregisters the device.

calling twice TUNSETQUEUE as in this example will result with having 3 queues in the end ; we can view these queues also under sys queue entry:

```
ls /sys/class/net/tun1/queues/
rx-0 rx-1 rx-2 tx-0 tx-1 tx-2
```

```
// tuntap/tunMultiQueue.c

#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/if_tun.h>
#include <linux/socket.h>
#include <stdio.h>

int main()
{
int fd, fd1, fd2, err;
struct ifreq ifr;

fd = open("/dev/net/tun",O_RDWR);

if (fd < 0) {
printf("fd < 0 in open\n");
return -1;
}
```

```
memset(&ifr, 0, sizeof(ifr));
ifr.ifr_flags = IFF_TUN | IFF_MULTI_QUEUE;
strncpy(ifr.ifr_name, "tun1", IFNAMSIZ);

err = ioctl(fd, TUNSETIFF, (void*)&ifr);

if (err < 0) {
printf("err<0 after TUNSETIFF, ioctl\n");
close(fd);
return -1;
}

err = ioctl(fd, TUNSETPERSIST, 1);
if (err < 0) {
printf("err<0 after TUNSETPERSIST ioctl\n");
close(fd);
return -1;
}
fd1 = open("/dev/net/tun",O_RDWR);
if (fd1 < 0) {
printf("fd1 < 0 in open\n");
return -1;
}

memset(&ifr, 0, sizeof(ifr));
ifr.ifr_flags = IFF_TUN | IFF_ATTACH_QUEUE;
strncpy(ifr.ifr_name, "tun1", IFNAMSIZ);
err = ioctl(fd1, TUNSETQUEUE, (void*)&ifr);

if (err < 0) {
perror("TUNSETQUEUE (second)\n");
close(fd1);
return -1;
}
```

```
printf("calling TUNSETQUEUE again with a third fd\n");
fd2 = open("/dev/net/tun",O_RDWR);
if (fd2 < 0) {
printf("fd2 < 0 in open\n");
return -1;
}

memset(&ifr, 0, sizeof(ifr));
ifr.ifr_flags = IFF_TUN | IFF_ATTACH_QUEUE;
strncpy(ifr.ifr_name, "tun1", IFNAMSIZ);
err = ioctl(fd2, TUNSETQUEUE, (void*)&ifr);

if (err < 0) {
perror("TUNSETQUEUE (second)\n");
close(fd2);
return -1;
} else
printf("Third call to TUNSETIFF on fd1 is OK\n");

pause();
}
```

When we issue ***open()*** system call on tun/tap device file (/dev/tun), we create a socket by ***sk_alloc()*** and assign it to to a pointer to tfile. This is done in ***tun_chr_open().***

When we issue close() system call on a tun/tap device file (/dev/tun), we call ***tun_detach()*** in order to release the socket (by ***sk_release_kernel()***).

In case you create a tun or tap device, and you want later to know the type of the device, you can do it by:

eththool -i unknownTypeDeviceName| grep bus-info

virtio

virtio was developed by Rusty Russell for his lguest project.

virtio has a common API for different types of devices (block devices, net devices, pci devices, and more).
The virtio network driver is implemented in ***drivers/net/virtio_net.c.***

BLUETOOTH

Bluetooth is a wireless technology standard for exchanging data over short distances.

Bluetooth implementation in the Linux kernel is found in two locations:

Bluetooth core:

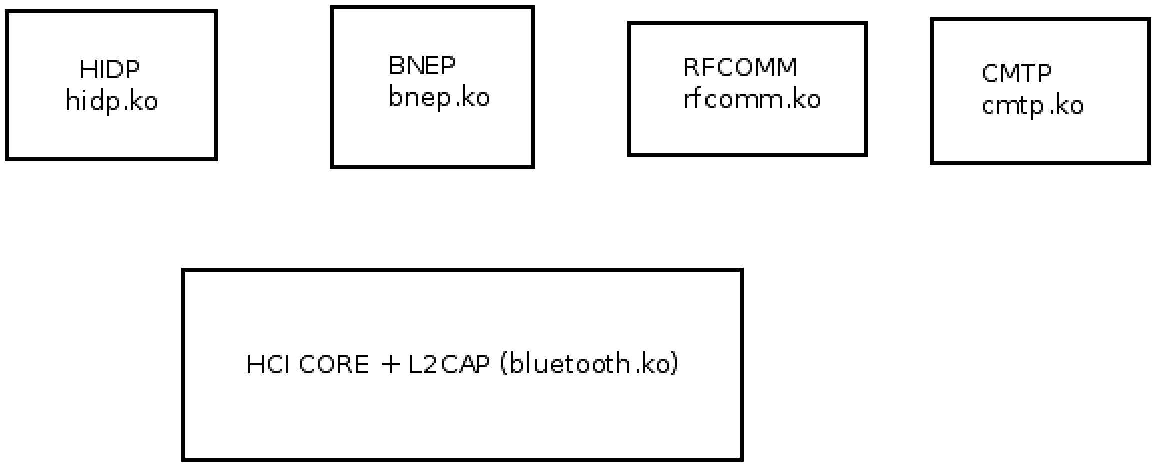
net/bluetooth

Bluetooth drivers:

drivers/bluetooth/

Bluetooth kernel diagram:

(Note: cmtp is a module for ISDN, not commonly used)



Note that there are very few drivers for bluetooth, as many devices use the generic drivers. We have, for example, the Generic Bluetooth USB driver (*drivers/bluetooth/btusb.c*) which is used for many USB BT devices.

For example, the ASUS USB BT21 dongle, which has a Broadcom chip, uses this driver.

BlueZ

BlueZ is the user space package for bluetooth.

- From version 4.0 of BlueZ, the main daemon is called ***bluetoothd*** (instead of hcid in earlier versions).
- The main configuration file is /etc/bluetooth/main.conf.
- This daemon also creates an sdp server, calling ***start_sdp_server()***.
- ***start_sdp_server()*** is implemented in bluez-4.99/src/sdpd-server.c.
- This sdp server opens two sockets:
    - UNIX local domain socket, for getting requests sent from the local machine, such as adding a service (sdptool add). The socket is opened on /var/run/sdp.
    - L2CAP socket for getting requests from outside (for example, when a remote machine runs "sdptool browse" with the machine address.

There is also a Bluetooth virtual HCI driver, *drivers/bluetooth/hci_vhci.c*; it works with a misc character device, /dev/vhci. The hciemu, HCI emulator, from bluez package, uses this driver.

Example:

```
modprobe hci_vhci
```

then:

```
./hciemu -n 10
```

This will create a virtual BT hci device. hciconfig will show it with "Bus: VIRTUAL." By default, it will be BREDR device (Type: BR/EDR). In case we need AMP device, we should first run "modprobe hci_vhci amp=1". Then hciconfig will show AMP device (Type: AMP).

notice that you should NOT run

```
mknod /dev/vhci c 10 250
```

As appears in some deprecated docs (for example, http://www.hanscees.com/bluezhowto.html)

Following here is the list of Bluetooth kernel sockets; we will discuss them in the following text.

- BTPROTO_L2CAP
- BTPROTO_HCI
- BTPROTO_SCO
- BTPROTO_RFCOMM
- dund uses an RFCOMM socket (bluez-4.99/compat/dund.c)
- BTPROTO_BNEP
- BTPROTO_CMTP (for ISDN).
- BTPROTO_HIDP
- BTPROTO_AVDTP
- Audio / Video Distribution Transport Protocol.
- There is no AVDTP in the kernel, this is reserved for future use.

Bluetooth userspace utils:

Getting info about hci devices is done by:

hciconfig

hciconfig shows PSCAN and ISCAN flags. In case ISCAN is not set, most likely the device will not be discovered. So
In case ISCAN is not set, you can run "hciconfig hci0 piscan" to set it. PSCAN is Page scan and ISCAN is Inquiry scan.

hciconfig shows also the type of the device, whether it is USB or UART. USB dongles are naturally USB, whereas in mobile phones (Like Smasung Mini Galaxy for example) it is usually UART.

Getting detailed info about hci devices is done by:

hciconfig -a

Bringing down/up an hci interface is done by:

hciconfig hci0 down

hciconfig hci0 up

These two commands send HCIDEVDOWN/HCIDEVUP ioctls from user space.

These ioctls are handled by hci_dev_open() and hci_dev_close(), respectively

- see *net/bluetooth/hci_sock.c*

Resetting a bluetooth device can be done by:

hcionfig hci0 reset

Scanning for bluetooth devices is done by:

hcitool scan

hcitool scan triggers a call to ***hci_inquiry()*** in user space (bluez-4.99/lib/hci.c).This method creates a BTPROTO_HCI socket and send HCIINQUIRY to the kernel. This IOCTL is handled int ***hci_inquiry()*** in the kernel (*net/bluetooth/hci_core.c*).

hcitool con

show active connections.

Bluetooth sniffing can be done by:

hcidump

(you can add flags, like hcidump -Xt).

hcidump in Fedora is part of the bluez-hcidump package.

hciattach is used to attach a serial UART to the Bluetooth stack.

You can change the address of the HCI adapter by using the bdaddr util from bluez. The bdaddr util is not installed as part of the bluez package binaries in most distros (like Fedora, for example).

For building bdaddr from source you should first run
./configure --enable-test
and then run make.

Get more info by ./test/bdaddr -help

You should use the -t flag for temporary change or permanent. The -r flag is for soft reset. Without it, you should perform hard reset by removing and replugging the device.

Using Bluetooth input devices, like a mouse/keyboard:

Bluetooth Input devices are handled by the hidp kernel module (net/bluetooth/hidp/hidp.ko).

You can connect to the Bluetooth mouse by:
hidd --server --search
then push the connect or reset button on the mouse and it will find it and pair. Connecting to the Bluetooth mouse in this way is in fact sending HIDPCONNADD ioctl to the kernel via hidp socket (socket which protocol is BTPROTO_HIDP).This ioctl is handled by ***hidp_add_connection()***. It creates a kernel thread named ***khidpd_vid_pid*** (vid is vendor id, pid is product id).

This kernel thread runs the ***hidp_session()*** method.

The BTPROTO_HIDP ioctls are handled by hidp_sock_ioctl() net/bluetooth/hidp/sock.c

You can show the connections by:
hidd --show

- This command sends HIDPCONNLIST ioctl to the BTPROTO_HIDP kernel socket.
- You will get something like:
- 00:C0:DF:04:89:A9 BT Mouse [0458:00a7] connected

You can terminate the connection by:
hidd --unplug 00:C0:DF:04:89:A9

- This command sends HIDPCONNDEL ioctl to the BTPROTO_HIDP kernel socket.

Reconnecting can be done by:
hidd --connect 00:C0:DF:04:89:A9

hidd is from bluez-compat package.

It connects to the kernel hci device by ***hci_open_dev()*** (user space API) and by opening a Raw HCI socket, with HCI protocol.

socket(AF_BLUETOOTH, SOCK_RAW, BTPROTO_HCI);

l2ping - L2CAP ping util.

sdptool browse XX:XX:XX:XX:XX:XX

- shows opened services on the specified device.
- sdptool browse btAddr does the following:
- creates a L2CAP socket (by ***l2cap_sock_create()***, in **net/bluetooth/l2cap_sock.c.**
- connect to this socket (by ***l2cap_sock_connect()*** in *net/bluetooth/l2cap_sock.c.*
- calls ***hci_connect()*** with ACL_LINK, which eventually calls ***hci_connect_acl()***, in ***net/bluetooth/hci_conn.c.***
-

sdptool browse local

- shows opened services on the local device.

sdptool add serviceName - adds a service to the local sdpd.

Example: sdptool add --channel=15 SP

This adds the Serial Port service on channel 15.

- The service name can be one from the following list: "DID","SP","DUN","LAN","FAX","OPUSH","FTP","PRINT", "HS","HSAG","HF","HFAG","SAP","PBAP","NAP","GN","PANU","HCRP","HID","KEYB","WIIMOTE","CIP","CTP","A2SRC","A2SNK","AVRCT","AVRTG", "UDIUE","UDITE","SEMCHLA","SR1","SYNCML","SYNCMLSERV","ACTIVESYNC","HOTSYNC","PALMOS","NOKID","PCSUITE","NFTP","NSYNCML","NGAGE","APPLE","ISYNC", "GATT".
  When we run:
- ***pand --listen --role NAP***
- Then "sdptool browse" on that device will show, among other SDP services, the "Network Access Point" service, which might be something like this:

- Service Name: Network service
  Service Description: Network service
  Service RecHandle: 0x10004
  Service Class ID List:
  "Network Access Point" (0x1116)
  Protocol Descriptor List:
  "L2CAP" (0x0100)
  PSM: 15
  "BNEP" (0x000f)
  Version: 0x0100
  SEQ16: 800 806
  Language Base Attr List:
  code_ISO639: 0x656e
  encoding: 0x6a
  base_offset: 0x100
  Profile Descriptor List:
  "Network Access Point" (0x1116)
  Version: 0x0100
- And when we run:
- pand --listen --role GP
- Then "sdptool browse" on that device will show, among other SDP services, the "PAN Group Network" service, which might be something like this:
- Service Name: Group Network Service
  Service Description: BlueZ PAN Service
  Service Provider: BlueZ PAN
  Service RecHandle: 0x10007
  Service Class ID List:
  "PAN Group Network" (0x1117)
  Protocol Descriptor List:
  "L2CAP" (0x0100)
  PSM: 15
  "BNEP" (0x000f)
  Version: 0x0100
  SEQ16: 800 806
  Language Base Attr List:
  code_ISO639: 0x656e
  encoding: 0x6a

base_offset: 0x100
Profile Descriptor List:
"PAN Group Network" (0x1117)
Version: 0x0100

sdptool search --bdaddr XX:XX:XX:XX:XX:XX FTP

- shows opened OBEX FTP service on the specified device and the respective channel.
- openobex site: http://dev.zuckschwerdt.org/openobex/

RFCOMM

- Acronym for: Radio Frequency Communications protocol.

Following is a practical example of establishing PC to PC connection with RFCOMM over serial:

Run set a Serial Port service (SP) on both sides:

sdptool add --channel=1 SP
(you can choose a different channel than 1, but it should be the same on the client and server)

Now, run on the listener side the following:
rfcomm listen rfcomm0 1
This command triggers creating an BTPROTO_RFCOMM kernel socket and calling
rfcomm_sock_listen() method and afterwards rfcomm_sock_accept().
Only after the socket is created, a device named /dev/rfcomm0 is created by sending an ioctl (RFCOMMCREATEDEV) to this socket.

struct rfcomm_dev represents the rfcomm device. A sysfs entry is generated for this device, /sys/class/tty/rfcomm0. This is done by device_create_file(). This folder contains values such as the address and the channel of this device. The address is the dst member of struct rfcomm_dev and the channel is the channel member of struct rfcomm_dev.

On the sender side, run
rfcomm connect rfcomm0 00:11:22:33:44:55 1
This command triggers creating an BTPROTO_RFCOMM kernel socket and then calling rfcomm_sock_bind() and rfcomm_sock_connect() and

creating a device named /dev/rfcomm0 by sending an ioctl (RFCOMMCREATEDEV) to
this socket.

You should get on the sender this message:

```
Connected /dev/rfcomm0 to 00:11:22:33:44:55 on channel 15
Press CTRL-C for hangup
```

Now you can send text from the sender to the listener thus:
first, run on the listener, on a different console:

```
cat /dev/rfcomm0
```

then, on the sender, run:

```
echo "foo" >> /dev/rfcomm0
```

You should see "foo" on the listener terminal.

The RFCOMM tty module (net/bluetooth/rfcomm/tty.c) implements serial emulation of
Bluetooth using tty driver API,
calling tty_register_driver() and tty_port_register_device().

We can establish TCP/IP connection over Bluetooth devices in this way,

for example:

on the server side:

```
pand --listen --role=NAP
```

And on the client-side

```
pand --connect btAddressOfServer
```

An interface called bnep0 will be created on both sides.
We can assign IP addresses on these two interfaces and have TCP/IP traffic.

In case you encounter problems, like "Connect to btAddr failed. Invalid exchange(52)" , or "connection refused", used hcidump to try to debug the problem. Make sure the the ISCAN and PSCAN flags are set on both sides.

pand --connect btAddressOfServer triggers the following sequence:
First, create a L2CAP socket and connect to it, by invoking socket()

system call
with BTPROTO_L2CAP protocol and then calling connect(), from user space. This is handled by ***l2cap_sock_connect()*** in the kernel (net/bluetooth/l2cap_sock.c)

l2cap_sock_connect() also creates a new connection. In this process,an entry is added under sysfs. This is done
by hci_conn_init_sysfs() and hci_conn_add_sysfs() in net/bluetooth/hci_sysfs.c.

When a new connection is removed, this entry is removed from sysfs, with hci_conn_del_sysfs(). This entry has only 3 attributes (besides the generic device attributes): type, address and features.

Then send BNEPCONNADD ioctl to the bnep socket; this is handled by bnep_sock_ioctl() in net/bluetooth/bnep/sock.c, and
invokes bnep_add_connection()
in the bnep module (net/bluetooth/bnep/core.c).

The bnep_add_connection() creates a kernel thread name kbnepd and creates a network device named bnep0.

The kbnepd kernel thread handles both Tx and Rx
by ***bnep_tx_frame()*** and ***bnep_rx_frame()***, respectively.

Bluetooth sysfs entries are under: /sys/class/bluetooth/

Using bluez dbus API:
You can use dbus-send to access a dbus device.
For example,
dbus-send --system --dest=org.bluez --print-reply /
org.bluez.Manager.DefaultAdapter
will give you a path to the BT adapter.

You can get detailed info about BlueZ DBUS api here:

http://bluez.cvs.sourceforge.net/viewvc/bluez/utils/hcid/dbus-api.txt

See more about pand here: *http://wiki.openmoko.org/wiki/Manually_using_Bluetooth*

L2CAP

L2CAP header is 4 bytes:

- 2 bytes for length of the entire L2CAP PDU in bytes(without the header).

- The maximum length can be 65529 or 65531 bytes (according to 3.3.1 in the spec).

- 2 bytes for cid (Channel Identifier)

- each L2CAP channel endpoint on any device has a different CID.

L2CAP socket is created by l2cap_sock_create().

When allocating a new socket (l2cap_sock_alloc()), we also create a channel which is associated with this socket (l2cap_chan_create()).

Channel Security level:

There are four levels of channel security: BT_SECURITY_SDP, BT_SECURITY_LOW, BT_SECURITY_MEDIUM, and BT_SECURITY_HIGH.

The security level of the channel is BT_SECURITY_LOW by default; it is set in *l2cap_chan_set_defaults()*method, net/bluetooth/l2cap_core.c.

The L2CAP header is represented by l2cap_hdr struct in include/net/bluetooth/l2cap.h.

Two types of controllers are defined in Bluetooth version 3 by the core specification:

- a Basic Rate / Enhanced Data Rate controller (HCI_BREDR)
- an Alternate MAC/PHY (AMP) (HCI_AMP)

You can find the type of your bluetooth device by hciconfig.

The first line shows the type.

For example, for Basic Rate / Enhanced Data Rate controller (HCI_BREDR) we have:

```
hciconfig
hci0: Type: BR/EDR...
```

You can also get the type by reading the bluetooth sysfs entry:

```
cat /sys/class/bluetooth/hci0/type
BR/EDR
```

The type can be BR/EDR or AMP or UNKNOWN.

Info about establishing PAN can be found here:

http://bluez.sourceforge.net/contrib/HOWTO-PAN

Notice that some of the info about the pand daemon is not updated to recent pand releases.

For example, running the following command, which is mentioned in this howto:

```
pand --listen --role NAP --sdp
```

will give the following error with pand of bluez-compat-4.99-2.fc17.x86_64:

```
            pand: unrecognized option '--sdp'
```

(The --nodsp option does exist)

BNEP layer is for the transmission of IP packets in the Personal Area Networking Profile and is implemented innet/bluetooth/bnep.

Site for Linux Bluetooth:
http://www.bluez.org/

The BlueZ Project started in 2001 by Qualcomm.

Obexd is the Object Exchange Protocol(OBEX) and is part of BlueZ.
The Linux BLUETOOTH subsystem and drivers are maintained by Marcel Holtmann, Gustavo Padovan and Johan Hedberg

BD (bluetooth device) address is 48 bits, and it looks like this:

XX:XX:XX:XX:XX:XX

Lower Address Part (LAP): 24bits

Upper Address Part (UAP): 8 bits

Nonsignificant Address Part (NAP): 16 bits

Helper methods:
static inline int bacmp(bdaddr_t *ba1, bdaddr_t *ba2)
    compares two bt addresses; return 0 if equal.
static inline void bacpy(bdaddr_t *dst, bdaddr_t *src)
    copy src address to dst address.

int ba2str(const bdaddr_t *ba, char *str)
    converts from bdaddr_t to a zero-terminated string.
int str2ba(const char *str, bdaddr_t *ba)
    converts from zero-terminated string to bdaddr_t.

Read more:
http://wiki.answers.com/Q/Whats_a_bd_address_as_it_asks_for_it_on_my_phone_for_bluetooth#ixzz28Jwz51ca

Blueman is a GTK+ bluetooth management utility for GNOME using bluez dbus backend.

Linux bluetooth mailing list archive:

http://www.spinics.net/lists/linux-bluetooth/

This mailing list is for patches:

- Patches starting with "Bluetooth" are for kernel.
- Patches starting with "BlueZ" are for user space.

Some Bluetooth acronyms:

BNEP: The Bluetooth Network Encapsulation Protocol.

BD: Bluetooth device.

L2CAP: The Logical Link Control and Adaption protocol

RFCOMM: The Radio Frequency Communications protocol.

DUND: Dial-Up Networking Daemon.

- The DUND service allows ppp connections via bluetooth.

ACL: The Asynchronous Connection-oriented Logical transport protocol.

SCO: Synchronous Connection-Oriented logical transport.

SSP: Secure Simple Pairing (SSP).

- The headline feature of Bluetooth 2.1
- hciconfig hci0 sspmode 0 - this command disable sspmode.
- hciconfig hci0 sspmode 1 - this command enables sspmode.
- hciconfig hci0 sspmode   - this command shows sspmode.

BlueDroid

With Android 4.2 release, BlueZ-based Bluetooth stack was replaced with a new stack , named "Bluedroid", which is a collaboration between Google and Broadcom.

See:

https://developer.android.com/about/versions/jelly-bean.html

http://lwn.net/Articles/525816/

http://lwn.net/Articles/525636/

Links:

Bluetooth git tree for developers (for submitting patches): git://git.kernel.org/pub/scm/linux/kernel/git/bluetooth/bluetooth-next.git

http://www.bluez.org/release-of-bluez-5-0/

The 5.0 BlueZ, By Nathan Willis, January 3, 2013:

http://lwn.net/Articles/531133/

BlueZ 5 API introduction and porting guide:
http://www.bluez.org/bluez-5-api-introduction-and-porting-guide/

Using Bluetooth article by Ben DuPont on DrDobbs (January 31, 2012)

http://www.drdobbs.com/mobile/using-bluetooth/232500828

OLS: Audio Streaming over Bluetooth - article by Ian Ward

http://lwn.net/Articles/293692/

Bluetooth profiles book:

http://www.amazon.com/Bluetooth-Profiles-Dean-Anthony-Gratton/dp/0130092215/ref=sr_1_1?s=books&ie=UTF8&qid=1355583216&sr=1-1&keywords=bluetooth+profiles

Bluetooth Security (Artech House Computer Security Series)

http://www.amazon.com/Bluetooth-Security-Artech-House-Computer/dp/1580535046

Desktop integration of Bluetooth. Marcel Holtmann, OLS 2007:

kernel.org/doc/ols/2007/ols2007v1-pages-201-204.pdf

NETILTER

Linux 3.7 kernel includes support for IPv6 NAT

See:

http://lwn.net/Articles/514087/

Most patches are from Patrick McHardy.

VXLAN

VXLAN stand for Virtual eXtensible Local Area Network.

VXLAN is a standard protocol to transfer layer 2 Ethernet packets over UDP. VXLAN code for Linux kernel is developed by Stephen Hemminger. It integrates a Virtual Tunnel Endpoint (VTEP) functionality that learns MAC to IP address mapping.

Why do we need VXLAN and not use instead ipip or gre tunnel?

There are firewalls which block tunnels and allow, for example, only TCP/UDP traffic.

iproute has support for managing VXLAN tunnels ***(ip/iplink_vxlan.c)***
This patch:
http://www.spinics.net/lists/netdev/msg212202.html
is for adding support for managing vxlan tunnels in iproute2.

The basic way to add vxlan virtual interface is by:
ip link add myvxlan type vxlan id 1
This sets the vni member of vxlan_dev struct to 1
(via vxlan_newlink() method).

vni is the virtual network id; the vni can be in the range 0-16777215 (whereas in vlans the id is restricted to 0-4094).

You can add vxlan with group address and ttl thus:

ip link add myvxlan type vxlan id 1 group 239.0.0.42 ttl 10
This sets also the ttl and the gaddr (multicast group address) of the vxlan device (vxlan_dev)

Removing vxlan virtual interface is done thus:
ip link del myvxlan
(This triggers the vxlan_dellink() method)

You can view the fdb of the vxlan interface by:

./bridge/bridge fdb show

The VXLAN module creates a kernel UDP socket
by ***sock_create_kern()*** (in ***vxlan_init_net()***).

This is a UDP encapsulation socket (this is set by udp_encap_enable()).

This means that the kernel inserts UDP header into the packet. UDP encapsulation is done also for NAT traversal. For example, with l2tp; when we want to use L2TP UDP encapsulation (L2TP_ENCAPTYPE_UDP), we also call udp_encap_enable()) when creating the l2tp tunnel (l2tp_tunnel_create()), net/l2tp/l2tp_core.c).

VXLAN module currently uses UDP destination port 8472, which is assigned for Overlay Transport Virtualization (OTV), untill IANA will assign a special VXLAN port.

see: http://www.speedguide.net/port.php?port=8472

However, the UDP destination port is a module parameter. First patches were sent on September 2012:

See: http://www.spinics.net/lists/netdev/msg211564.html

Setting up VXLAN:

http://vincent.bernat.im/en/blog/2012-multicast-vxlan.html#setting-up-vxlan

http://blogs.cisco.com/datacenter/digging-deeper-into-vxlan/

Stephan hemminger blog about vxlan:

http://linux-network-plumber.blogspot.co.il/2012/09/just-published-linux-kernel.html

A First Look At VXLAN over Infiniband Network On Linux 3.7-rc7: by Naoto MATSUMOTO on Nov 29, 2012

VXLAN draft:

http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-02

This draft does not support IPv6, but probably IPv6 will be supported in the future.

See also Documentation/networking/vxlan.txt

VXLAN tools/userspace

Two userspace apps ,"vxland" and "vxlanctl".

vxland, is a vxlan daemon, forwards packet to VXLAN Overlay Network.

vxlanctl is command for controlling vxlan.
You can create/destroy vxlan tunnel interface using vxlanctl.

git clone git://github.com/upa/vxlan.git
requires uthash package late 1.9 (for the hash table usage).
you can fetch uthash from http://uthash.sourceforge.net/. Then put the header file, uthash.h, under /usr/include, and run "Make" for the vxlan project from github.

VXLAN includes support for Distributed Overlay Virtual Ethernet (DOVE) networks by David L Stevens from IBM.

vti (IPv4 over IPSec tunneling driver)

VTI stands for Virtual Tunnel Interface.

The linux implementation is in net/ipv4/ip_vti.c

insmoding the kernel module (net/ipv4/ip_vti.ko) creates an ip_vti0 interface.

NFC

NFC stands for: Near Field Communication.

AF_NFC sockets are implemented under net/nfc.

neard, The Near Field Communication manager, is available in:
http://git.kernel.org/?p=network/nfc/neard.git;a=summary

linux-nfc web site:
https://www.01.org/linux-nfc

linux-nfc mailing list
https://lists.01.org/mailman/listinfo/linux-nfc

Near Field Communication with Linux slides, elc2012, Barcelona:
http://elinux.org/images/d/d1/Near_Field_Communication_with_Linux.pdf

GRE over IPv6

Dmitry Kozlov added support for GRE over IPv6.

These patches were applied in August 2012

See:

http://lwn.net/Articles/508786/

http://comments.gmane.org/gmane.linux.network/239706

Linux virtual server:

http://www.linuxvirtualserver.org/

Implemented in net/netfilter/ipvs

IP Virtual Server lets you build a high-performance
virtual server based on cluster of two or more real servers.

Tracing with kernel events:

Enabling tracing via kernel events:

Example:
In ***dev_hard_start_xmit()*** there are two identical calls to ***trace_net_dev_xmit()***.

In case we want a log with these traces, we should do the following;

– /sys/kernel/debug should be mounted with debugfs.

echo 1 > /sys/kernel/debug/tracing/tracing_enabled
echo 1 > events/net/net_dev_xmit/enable
The traced calls will appear in /sys/kernel/debug/tracing
We can stop the tracing of ***net_dev_xmit()*** by
echo 0 > events/net/net_dev_xmit/enable

Note: you can add a stacktrace to the trace log by
echo stacktrace > /sys/kernel/debug/tracing/trace_options

More info in http://lxr.free-electrons.com/source/Documentation/trace/events.txt

Sockets:

There are two types of socket in the kernel; most of them are sockets created from user space. There are also kernel sockets; they are created by ***sock_create_kern()***.

For example, in bluetooth kernel stack (net/bluetooth/rfcomm/core.c):

```
rfcomm_l2sock_create(struct socket **sock)
{
...
err = sock_create_kern(PF_BLUETOOTH, SOCK_SEQPACKET,
BTPROTO_L2CAP, sock);
...
}
```

and in vxlan , Virtual eXtensible Local Area Network (drivers/net/vxlan.c):

```
static __net_init int vxlan_init_net(struct net *net)
{
...
rc = sock_create_kern(AF_INET, SOCK_DGRAM, IPPROTO_UDP, &vn-
>sock);
...
}
```

Creating a socket from user space is done by the socket() system call.

On success, a file descriptor for the new socket is returned.

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).

PF_PACKET sockets are used, for example, in pcap library for Linux.

pcap library is in use by sniffers such as tcpdump or wireshark.

Also hostapd uses PF_PACKET sockets (hostapd is a wireless access point management project).

From hostapd source code:

...

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.

The sk_protocol member of struct sock equals to the third parameter (protocol) of the socket() system call.

struct sock has three queues:

- sk_receive_queue for rx
- sk_write_queue for tx
- sk_error_queue for errors.

- skb_queue_tail() : Adding to the queue.
- skb_dequeue() : removing 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.

UDP and TCP sockets:

- 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 {
  __be16source;
  __be16dest;
  __be16len;
  __sum16 check;
};
```

UDP packet = UDP header + payload.

From user space, you can receive udp traffic by 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 suggestion recently for recvmmsg() system call for receiving multiple
messages (By Arnaldo Carvalho de Melo).

The recvmmsg() meant to reduce the overhead caused by multiple system calls of recvmsg() in the usual case.

udp_rcv() is the handler for all UDP packets. It handles all incoming packets in which the protocol field in the ip header
is IPPROTO_UDP (17).
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);

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 view it by:
netstat -s

```
.....
Udp:
...
35 packets to unknown port received.
```

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_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:

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.

• There is an option of SO_BINDTODEVICE (assigning socket to a specified device).

- This patch added also an option to get SO_BINDTODEVICEvia getsockopt: http://www.spinics.net/lists/netdev/msg214004.html

● 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

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).

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).

• 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));

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.

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() finds 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().

RDMA (Infiniband)

See this sites by Dotan Barak about userspace for Infiniband:

http://www.rdmamojo.com/

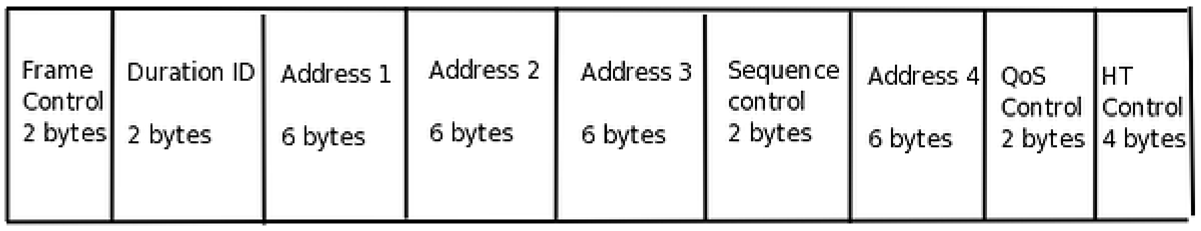
http://www.rdmamojo.com/links/

Infiniband core support is under ***drivers/infiniband.***

Infiniband maintainer is Roland Dreier.

Linux Wireless Subsystem (802.11).

Each MAC frame consists of a MAC header, a frame body of variable length and an
FCS (Frame Check Sequence) of 32 bit CRC. Next figure shows the 802.11 header.



Frame Control 2 bytes	Duration ID 2 bytes	Address 1 6 bytes	Address 2 6 bytes	Address 3 6 bytes	Sequence control 2 bytes	Address 4 6 bytes	QoS Control 2 bytes	HT Control 4 bytes

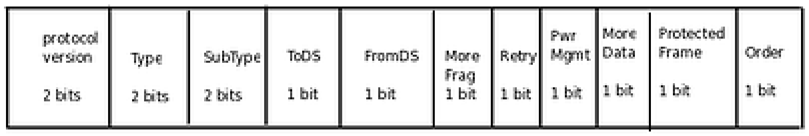
The 802.11 header is represented in mac80211 by a structure called ieee80211_hdr

(include/linux/ieee80211.h).

As opposed to an Ethernet header (struct ethhdr), which contains only three fields
(source MAC address, destination MAC address, and type), the 802.11 header contains
four addresses and not two, and some other fields.

frame control:

The first field in the 802.11 header is called the frame control: it is a an important
field and in many cases, its contents determine the meaning of other fields of the 802.11
header (especially addresses). The frame control length is 16 bits;
following here is a discussion of its fields.



protocol version	Type	SubType	ToDS	FromDS	More Frag	Retry	Pwr Mgmt	More Data	Protected Frame	Order
2 bits	2 bits	2 bits	1 bit	1 bit	1 bit	1 bit	1 bit	1 bit	1 bit	1 bit

Protocol version:

The version of the MAC 802.11 we use. Currently there is only
one version of MAC, so this field is always 0.

Type:

There are three types of packets in 802.11:management, control and data.

• Management packets (IEEE80211_FTYPE_MGMT) are for management
actions like association, authentication, scanning and more. We will deal more
with management packets in the following sections.

• Control packets (IEEE80211_FTYPE_CTL) usually have some relevance to data
packets; for example, a PS Poll packet is for retrieving packets from an Access
Point buffer. Another example: a station that wants to transmit first sends a control
packet called RTS (request to send); if the medium is free, the

destination station
will send a control packet called CTS (clear to send).

- Data packets (IEEE80211_FTYPE_DATA) are the raw data packets. Null
packets are a special case of raw packets.

There is also IEEE80211_FTYPE_EXT type - we will not discuss it.

These types are defined in /include/linux/ieee80211.h

Subtype:
For all the aforementioned three types of packets (management, control and data),
there is a subtype field which identify the character of the packet we use. For
example, a value of 0100 for the subtype field in a management frame denotes that
the packet is a Probe Request (IEEE80211_STYPE_PROBE_REQ) management
packet, which is used in a scan operation. Notice that the action management frame
(IEEE80211_STYPE_ACTION) was introduced with 802.11h amendment, which
dealt with spectrum and transmit power management; however, since there is a lack
of space for management packets subtypes, action management frames are used also
in 802.11n management packets. A value of 1011 for the subtype field in a
control packet denotes that this is a request to send (IEEE80211_STYPE_RTS)
control packet. A value of 0100 for the subtype field of a data packet denotes that
this a a null data (IEEE80211_STYPE_NULLFUNC) packet, which is used
for power management control. A value of 1000 (IEEE80211_STYPE_QOS_DATA)
for a subtype of a data packet means that this is a QoS data packet; this subtype was

added by the IEEE802.11e amendment, which dealt with QoS enhancements.

ToDS:
When this bit is set, this means that the packet is for the distribution system.

FromDS:
When this bit is set, this means that the packet is from the distribution system.

More Frag:
When we use fragmentation, this bit is set to 1.

Retry:
When a packet is retransmitted, this packet is set to 1. A common case of
retransmission is when a packet that was sent did not receive an acknowledgment in
time. The acknowledgements are sent by the firmware of the wireless driver.

Pwr Mgmt:
When the power management bit is set, this means that the station will enter
power save mode.

More Data:
When an Access Points sends packets that it buffered for a sleeping station, it
sets the more data bit to 1 when the buffer is not empty. Thus the station knows
that there are more packets it should retrieve. When the buffer has been
emptied, this bit is set to 0.

Protected Frame:
This bit is set to 1 when the frame body in encrypted; only data frames and
authentication frames can be encrypted.

Order:
There is a MAC service called “strict ordering”. With this service, the order of
frames is important. When this service is in use, the order bit is set to 1. It is
rarely used.

Duration/ID:
The duration holds values for the Network Allocation Vector (NAV) in microseconds, and it consists of 15 bits of the duration field. The sixteenth field is
0. When working in power save mode it is the AID (Association ID) of a station.
The Network Allocation Vector (NAV) is a virtual carrier sensing mechanism.

Sequence control:
This is a 2 byte field specifying the sequence control. In 802.11, it is possible that a packet will be received more than once. The most common cause for such a case is when an acknowledgement is not received for some reason. The sequence control field consists of a fragment number (4 bits) and a sequence number (12 bits). The sequence number is generated by the transmitting station, in ieee80211_tx_h_sequence(). In case of a duplicate frame in a retransmission, it is
dropped, and a counter of the dropped duplicate frames (dot11FrameDuplicateCount)
is incremented by 1; this is done in ieee80211_rx_h_check(). Sequence Control field is not present in control packets.

Address Fields:

There are four addresses, but we don't always use all of them. Address 1 is the Receive
Address (RA), and is used in all packets. Address 2 is the Transmit Address (TA), and it
exists in all packets except ACK and CTS packets. Address 3 is used only for
management and data packets. Address 4 is used when ToDS and

FromDS bits of the
frame control are set; this happens when operating in a Wireless Distribution System
(WDS).

OoS Control:
The QoS Control field was added by 802.11e amendment and it is only present in QoS
data packets. Since it is not part of the original 802.11 spec, it is not part of the original
mac80211 implementation, so it is not a member of the ieee80211_hdr struct. In fact, it
was added at the end of 802.11 header and it can be accessed by
ieee80211_get_qos_ctl() method. The QoS Control field includes the
tid (Traffic Identification), the Ack Policy, and a field called A-MSDU present, which tells whether
an A-MSDU is present.

HT Control Field:

HT Control Field was added by 802.11n amendment. HT stands for High Throughput.
One of the most important features of 802.11n amendment is increasing the rate to up
to 600 Mbps.

- All stations must authenticate and associate and with the Access Point prior to

  communicating.

  Stations usually perform scanning prior to authentication and association in order to get details about the Access Point (like mac address, essid, and more).

  Scanning is done thus:

```
ifconfig  wlan0 up
iwlist  wlan0 scan
```

Scanning is triggered by issuing SIOCSIWSCAN ioctl (include/linux/wireless.h)

iwlist (and iwconfig) is from wireless-tools package.

Please note: wireless-tools is regarded deprecated. We should use "iw", which is more

modern and which is based on **nl80211**.

You can download iw from http://linuxwireless.org/download/iw/.

iw git repositories are at: http://git.sipsolutions.net/iw.git

Eventually, scanning starts by calling __ieee80211_start_scan()

(net/mac80211/scan.c)

Active Scanning is performed by sending Probe

Requests on all the channels which are supported by the station

Open-system authentication (WLAN_AUTH_OPEN) is the only mandatory
authentication method required by 802.11.

(WLAN_AUTH_OPEN is defined in include/linux/ieee80211.h)

• At a given moment, a station may be associated with no more than one AP.
● A Station ("STA") can select a BSS and authenticate and associate to it.

• In Ad-Hoc : authentication is not defined.

• An Access Point will not receive any data frames from a station before it it is associated with the AP.

• An Access Point which receive an association request will check whether the mobile station parameters match the Access point parameters.

– These parameters are SSID, Supported Rates and capability information.

● When a station associates to an Access Point, it gets an ASSOCIATION ID (AID) in the range 1-2007.

• Trying unsuccessfully to associate more than 3 times results with this message in the kernel log:

"association with AP apMacAddress timed out"

(IEEE80211_ASSOC_MAX_TRIES is the number of max tries to associate, see

net/mac80211/mlme.c)

Hostapd

hostapd is a user space daemon implementing access point functionality (and authentication servers). It supports Linux and FreeBSD.

• http://hostap.epitest.fi/hostapd/

• Developed by Jouni Malinen

• hostapd.conf is the configuration file.

• Certain devices, which support Master Mode,
can be operated as Access Points by running
the hostapd daemon.
● Hostapd implements part of the MLME AP code which is not in the kernel and probably will not be in the near future.
● For example: handling association requests which are received from wireless clients.

Hostapd manages:
● Association/Disassociation requests.
● Authentication/deauthentication requests.

wpa_supplicant is part of hostapd project

You can clone hostap by:

git clone git://w1.fi/srv/git/hostap.git

Power save mode

Hardware can handle power save by itself; when this is done, it should set the IEEE80211_HW_SUPPORTS_PS flag.

There are three types of IEEE80211 packets: Management, control and data.

(These correspond to IEEE80211_FTYPE_MGMT,

IEEE80211_FTYPE_CTL and IEEE80211_FTYPE_DATA In the mac80211 stack).

- Control packets include RTS (Request to Send), CTS (Clear to Send) and ACK packets.
- Management packets are used for Authentication and Association.

- Mobile devices are usually battery powered most of the time.
- A station may be in one of two different modes:
  - Awake (fully powered)
  - Asleep (also termed “dozed” in the specs)
- Access points never enters power save mode and does not transmit Null packets.
- In power save mode, the station is not able to transmit or receive and consumes very low power.

In order to sniff wireless traffic in Linux with wireshark, you can do this:

```
iwconfig  wlan0 mode monitor
ifconfig  wlan0 up
```

And then start wireshark and select the wlan0 interface.

You can know the channel number while sniffing by
looking at the radiotap header in the sniffer output;
channel frequency translates to a channel number
(1 to 1 correspondence.) Moreover, the channel number appears in square
brackets. Like:
– channel frequency 2437 [BG 6]

The radiotap header is added in certain cases under monitor mode.

It precedes the 802.11 header.

It is done in ieee80211_add_rx_radiotap_header() in net/mac80211/rx.c

ieee80211_add_rx_radiotap_header() is invoked from:

- ieee80211_rx_monitor().
- ieee80211_rx_cooked_monitor().

You can know the mac address of your wireless nic by:

cat /sys/class/ieee80211/phy*/macaddress

● A station send a null packet by calling ieee80211_send_nullfunc()

(net/mac80211/mlme.c)

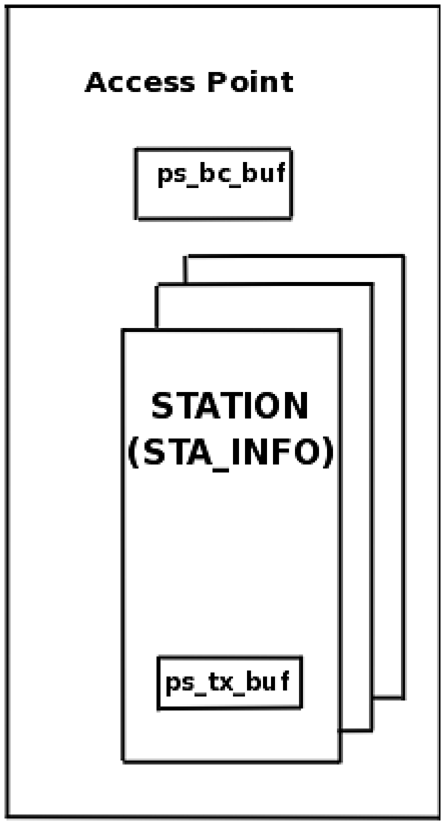
The PM bit in the frame control of this packet is set. (IEEE80211_FCTL_PM bit)

● Each access point has an array of skbs for buffering unicast packets from the stations which enter power save mode.

● It is called ps_tx_buf (in struct sta_info; see net/mac80211/sta_info.h)

An access point also has a ps_bc_buf queue for for multicast and broadcast packets.

ps_tx_buf can buffer up to 64 skbs. (STA_MAX_TX_BUFFER=64, in net/mac80211/sta_info.h)



In case the buffer is filled, old skbs will be dropped.

• When a station enters PS mode it turns off its RF. From time to time it turns the RF on, but only for receiving beacons.

• An Access Point sends beacon frames periodically (usually about 10 beacons per second).

• Each beacon has a TIM (Traffic Indication Map) field.

ieee80211_rx_mgmt_beacon() handles receiving beacons. (net/mac80211/mlme.c).

A beacon is a management, represented by struct beacon, which is one

of the members in a union (named "u") in ieee80211_mgmt struct.

the "variable" member of struct beacon represents the "Information Elements"

this beacon can contain; "Information Elements" can be SSID, Supported rates, FH Params, DS Params, CF Params, IBSS Params, TIM, and more.

struct ieee802_11_elems represent "Information Element". It conatins a structure

called tim (ieee80211_tim_ie) , representing the tim (Traffic Indication Map).

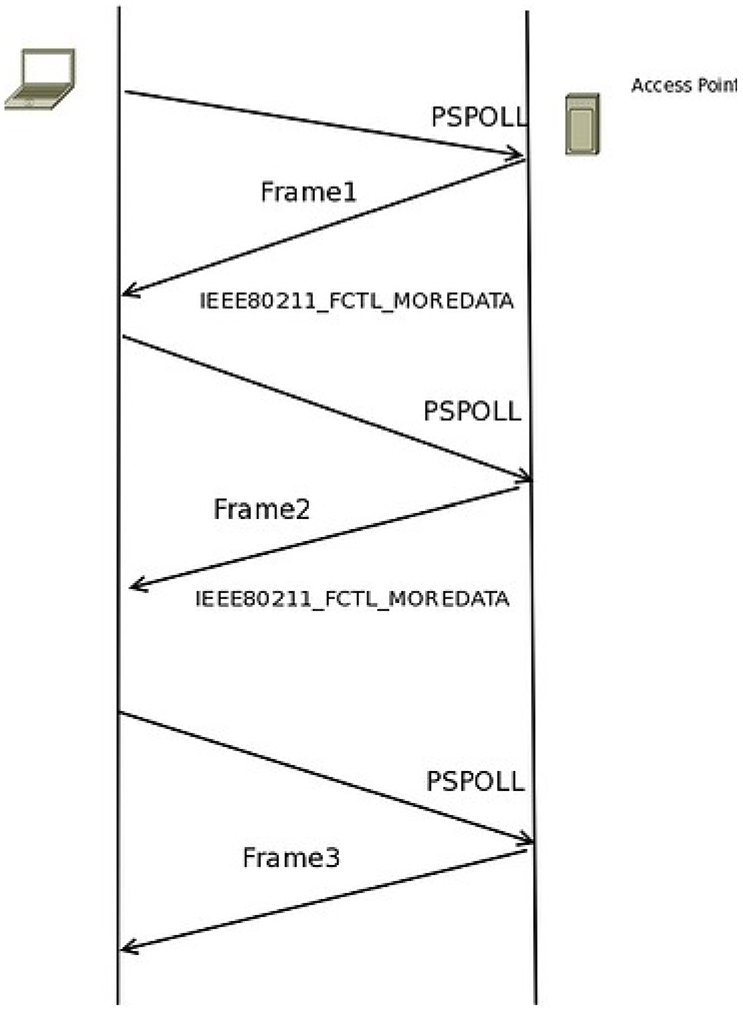
This method calls ieee80211_check_tim(), to see if the TIM has traffic for this station AID.

(ieee80211_check_tim() is implemented in include/linux/ieee80211.h)

ieee80211_send_pspoll() in order to send PSPOLL packet to the AP.

PSPOLL are control packets.

Note that in the following diagram, we do not show the ACK packets.



AD Hoc

Implementation of 802.11 AD Hoc is mainly in: net/mac80211/ibss.c

ieee80211_if_ibss structure represents an AD hoc station.

80211.n

80211.n started with the High Throughput Study Group in about 2002.

In 802.11, each packet should be acknowledged. In 802.11nm we grouping packets in a block and acknowledging this block instead acknowledging each packet separately. This improves performance.

Grouping packets in a block in this way is called "packet aggregation" in 802.11n terminology.

There are two forms of aggregation:

- AMPDU (The more common form)

AMPDU aggregation requires the use of block acknowledgement or BlockAck, which was introduced in 802.11e and has been optimized in 802.11n.

802.11e is the quality-of-service extensions amendment.

The 802.11e amendment deals with QoS; it introduced four queues for different types of traffic: voice traffic, video traffic, best-effort traffic and background traffic. The Linux implementation of 802.11e uses multiqueues. Traffic in higher priority queue is transmitted before traffic in a lower priority queue.

MPDU stand for: MAC protocol data units

- AMSDU

With AMSDU, you make one big packet out of some packets.

This big packet should be acked.

Disadvantage: more risk of corruption of the big packet.

Less in usage, fading out.

MSDU stands for: MAC service data units.

Packet aggregation

- There are two sides to a block ack session: originator and recipient. Each block session has a different TID (traffic identifier).
- The originator starts the block acknowledge session by calling ieee80211_start_tx_ba_session() (net/mac80211/agg-tx.c)

ieee80211_tx_ba_session_handle_start() is a callback of ieee80211_start_tx_ba_session(). In this callback we send an ADDBA (add Block Acknowledgment) request packet, by invoking ieee80211_send_addba_request() method (Also in net/mac80211/agg-tx.c) ieee80211_send_addba_request() method builds a management action packet

(The sub type is action, IEEE80211_STYPE_ACTION).

The response to the ADDBA request should be received within 1 HZ, which is one millisecond in x86_64 machines (ADDBA_RESP_INTERVAL, defined in net-next/net/mac80211/sta_info.h)

In case we do not get a response in time,
the sta_addba_resp_timer_expired() will stop the BA session by calling ieee80211_stop_tx_ba_session().

When the other side (the recipient) receives the ADDBA request, it first sends an ACK. Then it processes the ADDBA request by calling ieee80211_process_addba_request(); (net/mac80211/agg-rx.c)

if everything is ok, it sets the aggregation state of this machine to operational
(HT_AGG_STATE_OPERATIONAL), and sends an ADDBA Response by calling
ieee80211_send_addba_resp().

After a session was started, a data block, containing multiple MPDU packets is sent. Consequently, the originator sends a Block Ack Request (BAR) packet by
calling ieee80211_send_bar(). (net/mac80211/agg-tx.c)

The BAR is a control packet with Block Ack Request subtype (IEEE80211_STYPE_BACK_REQ).

The bar packet includes the SSN (start sequence number), which is the sequence number of the oldest MSDU in the block which should be acknowledged.

The BAR (HT Block Ack Request) is defined in include/linux/ieee80211.h.

Its start_seq_num member is initialized to the proper SSN.

There are two types of Block Ack: Immediate Block Ack and Delayed Block Ack.

Mac80211 debugfs support:

In order to have mac80211 debugfs support, kernel should be built with CONFIG_MAC80211_DEBUGFS (and CONFIG_DEBUG_FS)

Then after:

mount -t debugfs debugfs /sys/kernel/debug

You can see debugfs entries under:

/sys/kernel/debug/ieee80211/phy*

Open Firmware

The Atheros 802.11n USB chipset (AR9170) has open firmware;

see http://www.linuxwireless.org/en/users/Drivers/ar9170.fw

802.11 AC

The next generation of 802.11 is AC.

Support for 802.11AC was added in mac80211 stack. For example,

ieee80211_ie_build_vht_cap() in net/mac80211/util.c.

struct ieee80211_vht_capabilities and struct ieee80211_vht_operation

in include/linux/ieee80211.h.

VHT stands for: Very High Throughput

Development:

Sending patches should be done against the wireless-testing tree

git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-testing.git

The maintainer of compat wireless is Luis R. Rodriguez.

Mesh networking (802.11s)

- 802.11s started as a Study Group of IEEE in September 2003, and became a

Task Group named TGs in 2004.

In 2006, two proposals, out of 15, (the "SEEMesh" and "Wi-Mesh" proposals) were merged into one. This is draft D0.01.

There are two topologies for mesh networks.

- The first is full mesh; with full mesh, each node is connected to all the other nodes.
- The second mesh topology is partial mesh. With partial mesh, nodes are connected to only some of the other nodes, not all. This topology is much more common in wireless mesh networks.

In 2.6.26, the network stack added support for the draft of wireless mesh networking (802.11s), thanks to the open80211s project. The open80211s project goal was to create the first open implementation of 802.11s. The project got some sponsorship from the OLPC project. Luis Carlos Cobo and Javier Cardona and other developers from Cozybit
developed the Linux mac80211 mesh code. This code was merged into the Linux
Kernel from 2.6.26 release (July 2008). There are some drivers in the linux kernel with
support to mesh networking (ath5k, b43, libertas_tf, p54, zd1211rw).

HWMP protocol.

802.11s defines a default routing protocol called HWMP (Hybrid Wireless Mesh Protocol). The HWMP protocol works with layer 2 (Mac addresses) as opposed to IPV4 routing protocol, for example, which works with layer 3 (IP addresses). HWMP routing
is based on two types of routing (hence it is called hybrid). The first is on demand routing and the second is proactive, dynamic routing. Currently only on demand routing
is implemented in the Linux Kernel. We have three types of messages with on demand routing. The first is PREQ (Path Request). This type of messages is sent as a broadcast when we look for some destination, which we still do not have a route to. This PREQ
message is propagated in the mesh until it gets to its destination. On each station until the final destination is reached, a lookup is performed (by mesh_path_lookup(), net/mac80211/mesh_pathtbl.c).

In case the lookup fails, the PREQ is forwarded (as a broadcast).

The PREQ message is sent in a management packet; its subtype is a

Then a PREP (Path Reply) unicast packet is sent. This packet is sent in the reverse path.
The PREP message is also sent in a management packet; its subtype is also action.
(IEEE80211_STYPE_ACTION). It is handled by
hwmp_prep_frame_process(). Both
PREQ and PREP are sent in the mesh_path_sel_frame_tx() function. If there is some
failure on the way, a PERR is sent.(Path Error). A PERR message is handled by
mesh_path_error_tx().
The route take into consideration a radio-aware metric (airtime metric). The airtime
metric is calculated in the airtime_link_metric_get() method ,
net/mac80211/mesh_hwmp.c(based on rate and other
hardware parameters). Mesh Points continuously monitor their links and update metric
values with neighbours.

The station which sent the PREQ may try to send packets to the final destination while
still not knowing the route to that destination; these packets are kept in a buffer called
frame_queue, which is a member of mesh_path struct;
net/mac80211/mesh.h) in such a case, when a PREP finally arrives,
the pending packets of this buffer are sent to the final destination
(by calling mesh_path_tx_pending()). The maximum number of frames
buffered per destination for unresolved destinations is 10
(MESH_FRAME_QUEUE_LEN, defined in net/mac80211/mesh.h).

The advantages of mesh networking are:
● Rapid deployment.
● Minimal configuration; inexpensive.

● Easy to deploy in hard-to-wire environments.
● Connectivity while nodes are in motion.

The disadvantages:
● Many broadcasts limit network performance
● Not all wireless drivers support mesh mode at the moment.

Tip for hacking mac80211 with openwrt:

The WRTG54L LinkSys wireless router comes out of factory with Linux.

In case you want to hack mac80211 with OpenWrt, you can do it
with backfire or

with kamikaze, which are versions of OpenWrt. In case of
kamikaze, you will soon
find out that with recent kamikaze releases (8.09.1 and 8.09.2),
the wireless driver does not exist (kmod-b43).

For this reason "opkg install kmod-b43" fails on kamikaze 8.09.1 and
kamikaze 8.09.2.

You can use also kamikaze 9.0.2 and build the broadcom wireless
driver
as a kernel module.
A simple way of achieving this is thus:
"make kernel_menuconfig"
Then:
select driver/network/wireless/B43 by

(Broadcom 43xx wireless support (mac80211 stack))

  CONFIG_B43 should be "m".

Make sure that you create also mac80211.ko and cfg80211.ko
backfire_svn/build_dir/linux-brcm47xx/compat-wireless-2011-12-

The source files for b43 drivers selected in this way are under

build_dir/linux-brcm47xx/compat-wireless-2011-12-01/drivers/net/wireless/b43

Tip:

When working with b43 kernel module (b43.ko) it is enough to run

make target/linux/compile

in order to create b43.ko (under build_dir/linux-brcm47xx/linux-2.6.32.27/drivers/net/wireless/b43/) and copy it.

The hostapd sources are under:

build_dir/target-mipsel_uClibc-0.9.30.1/hostapd-full

Copy cfg80211.ko, mac80211.ko and b43.ko to the linksys device.

Insert them by this order:

```
insmod cfg80211.ko
insmod mac80211.ko
insmod b43.ko
```

iwconfig should show "wlan0".

When trying "ifconfig wlan0 up", in case you get an error about firmware,
like this error message about missing firmware file,
"b43-phy0 ERROR: Firmware file "b43/ucode5.fw" not found or load failed."

do as described in:

http://linuxwireless.org/en/users/Drivers/b43#devicefirmware

In case you will try to scan, you will get:

```
ifconfig  wlan0 up
iwlist wlan0 scan
wlan0     Interface doesn't support scanning : Operation not supported
```

It **is** included in kamikaze 8.09.

(so when booting with kamikaze 8.09 you do see wireless interface when
running iwconfig).

See this thread:
https://forum.openwrt.org/viewtopic.php?id=22103
"Why is b43 driver missing in recent releases?"

See also under:
http://downloads.openwrt.org/kamikaze/

Another tip:

In order to use , in /etc/hostapd.conf,

driver=nl80211,

you should have in hostapd .config, before running "make",

CONFIG_DRIVER_NL80211=y

Not that in some distributions CONFIG_DRIVER_NL80211 is not set in hostapd package.

- In case there are any problems with burning an image and you cannot access

the WRT54GL linksys device, you can burn an image via tftp,

in this way:

```
tftp 192.168.1.1
```

```
bin
trace
timeout 60

rexmt 1
put nameOfFirmareFile
```

- When using this way, you should download the firmware from linksys site:

http://homesupport.cisco.com/en-us/support/routers/WRT54GL

- In case you will try to burn an openwrt image, most likely you will get errors;

like:

```
....
...
tftp> put openwrt-brcm47xx-squashfs.trx
received ACK <block=0>
sent DATA <block=1, 512 bytes>
received ACK <block=0>
received ERROR <code=4, msg=code pattern incorrect>
Error code 4: code pattern incorrect
...
...
```

OpenFWWF website:

http://www.ing.unibs.it/~openfwwf/

 - also for wrt54GL.

building a firmware for b43 is simple:

you download b43-tools and b43 firmware.

From b43-tools/assembler you run "make && make install".

(you only need assembler for building the b43 firmware)

In case you get the following error:

```
b43-tools/assembler># make
CC b43-asm.bin
/usr/bin/ld: cannot find -lfl
```

make sure that flex-static and flex are installed. (yum install flex-static flex)

Then simply go to the folder where you extracted the firmware, and run "make".

A file name "ucode5.fw" will be generated.

With b43 on the WRT54GL, we use SSB_BUSTYPE_SSB

This means that
in b43_wireless_core_start() (drivers/net/wireless/b43/main.c),

dev->dev->bus->bustype is SSB_BUSTYPE_SSB and we call

request_threaded_irq() and not b43_sdio_request_irq().

(The other possibilities are SSB_BUSTYPE_PCI, SSB_BUSTYPE_PCMCIA or

SSB_BUSTYPE_SDIO).

b43/b43legacy Linux driver discussions:

http://lists.infradead.org/mailman/listinfo/b43-dev

 - Patches which are sent to this mailing list are also sent to Linux kernel wireless mailing list.

ath9k-devel mailing list:

http://www.mail-archive.com/ath9k-devel@lists.ath9k.org/index.html

TBD:

The following downloads 8.09.2 and not 8.09; how you get 8.09 and not 8.09.2?

You can download kamikaze 8.09 by:

svn co svn://svn.openwrt.org/openwrt/branches/8.09

OpenWrt repositories are in the following link:

https://dev.openwrt.org/wiki/GetSource

RFKILL

rfkill is a simple tool for accessing the Linux rfkill device interface, which is used to enable and disable wireless networking devices, typically
WLAN, Bluetooth and mobile broadband.

rfkill list will list the status of rfkill.

rfkill block to set a soft lock

rfkill unblock to clear a soft lock

see:

http://www.linuxwireless.org/en/users/Documentation/rfkill

WiMAX

LTE will undoubtedly be the 4G technology. There is WimMAX solution in

Linux kernel though.

WiMAX (Worldwide interoperability for Microwave access) is based on IEEE802.16

standard. It is a wireless solution for broadband WAN (Wide Area Network).
about 200 WiMAX projects around the world. WiMAX products can accommodate fixed and mobile usage models.
There is a WiMAX Linux git tree, maintained by Inaky Perez-Gonzalez from Intel.

In the past, Inaky was involved in developing the Linux USB stack and the Linux UWB
(Ultra Wideband) stack. The WiMAX stack and driver have been accepted in mainline
for 2.6.29 in January 2009. The WiMAX support in Linux consists of a Kernel module
(net/wimax/wimax.ko), device-specific drivers under it, and a user space management
stack, WiMAX Network Service. There was in the past an initiative from Nokia for a
WiMAX stack for Linux, but it is not integrated currently. Also
work was done on D-Bus interface to the WiMAX stack, which will help user space tools manage the WiMAX stack. There is currently one WiMAX driver in the Linux tree, the Intel
WiMAX Connection 2400 over USB driver (which supports any of the Intel Wireless
WiMAX/WiFi Link 5x50 series). The WiMAX stack uses generic netlink protocol
mechanism to send and receive netlink messages to and from userspace. Free form
messages can be sent back and forth between driver/device and user space

batman-adv

"B.A.T.M.A.N. Advanced Meshing Protocol is
a routing protocol for multi-hop ad-hoc mesh networks. The networks may be wired or wireless.

Implementation is in net/batman-adv

See http://www.open-mesh.org/

Wireless Summit (2012)

http://wireless.kernel.org/en/developers/Summits/Barcelona-2012

Will deal with 802.11, **802.15.4 stack (6lowpan),** Bluetooth, NFC, and more.

lecture slides of the 802.15.4 lecture by Alan Ott: **802.15.4 stack (6lowpan)**

6LoWPAN stands for: "IPv6 over Low power Wireless Personal Area Networks"

http://elinux.org/images/7/71/Wireless_Networking_with_IEEE_802.15.4_and_6LoWPAN.pdf

IEEE 802.15.4

IEEE standard 802.15.4 is for wireless personal area network (WPAN).

Implementation in the Linux kernel tree: net/ieee802154/

The maintainers of IEEE 802.15.4 SUBSYSTEM are Alexander Smirnov and Dmitry Eremin-Solenikov.

Web site: http://sourceforge.net/apps/trac/linux-zigbee

Git tree: git://git.kernel.org/pub/scm/linux/kernel/git/lowpan/lowpan.git

compat-wireless

compat-wireless is a backport of the wireless stack from newer kernels to older ones.

Wi-Fi Direct, previously known as Wi-Fi P2P, is a standard that allows Wi-Fi devices to connect to each other without the need for an Access Point.

CRDA stands for "Central Regulatory Domain Agent". It is based on nl80211 and udev.

You can download the source code by:

git clone git://github.com/mcgrof/crda.git

Mostly written by Luis R. Rodriguez (mcgrof@qca.qualcomm.com).

see:

http://www.linuxwireless.org/en/developers/Regulatory/CRDA

Links:

"Linux wireless networking", article from 2004
http://www.ibm.com/developerworks/library/wi-enable/index.html

Updated standard (2012)
http://standards.ieee.org/getieee802/download/802.11-2012.pdf

Books:

802.11 Wireless Networks: The Definitive Guide, 2nd Edition
By Matthew Gast
Publisher: O'Reilly Media, 2005

802.11n: A Survival Guide
By Matthew Gast
Publisher: O'Reilly Media, 2012

TBD:
ACS (Automatic Channel Selection)

Useful tips:

Printing IP address:

```
__be32 ipAddr;
printk("ipAddr = %pI4\n", &ipAddr);
```

when

```
u32 ipAddr;
```

TBD!

If you want immediate UDP traffic, you can use traceroute.

Remember that the destination port is incremented by 1 for each sent packet.

You can also generate raw UDP traffic with traceroute, by:

tarceroute -P (Default protocol is 253 , see rfc3692).

`wireshark tip:`

Sometimes you see in wireshark sniffer,
that the amount of "Bytes on wire" is larger then the MTU
of the network card.
This is probably due to using Jumbo packets or offloading.

Links and more info

1) Understanding the Linux Kernel, Third Edition By Daniel P. Bovet, Marco Cesati.

Understanding Linux Network Internals, Christian Benvenuti, O'reilly contains all details of the Linux networking stack.

2) Linux Device Drivers, by Jonathan Corbet, Alessandro Rubini, Greg Kroah Hartman Third Edition February 2005.

– Chapter 17, Network Drivers

3) Linux networking: (a lot of docs about specific networking topics)
– http://linuxnet.osdl.org/index.php/Main_Page

26) LCE: Challenges for Linux networking , By Jonathan Corbet , November 7, 2012:

http://lwn.net/Articles/523058/

5) netdev mailing list: http://www.spinics.net/lists/netdev/

6) Removal of multipath routing cache from kernel code: http://lists.openwall.net/netdev/2007/03/12/76http://lwn.net/Articles/241465/

7) Linux Advanced Routing & Traffic Control : http://lartc.org/

8) ebtables – a filtering tool for a bridging: http://ebtables.sourceforge.net/

9) Writing Network Device Driver for Linux: (article) – http://app.linux.org.mt/article/writingnetdrivers?locale=en

10) Netconf – a yearly networking conference; first was in 2004.

– http://vger.kernel.org/netconf2004.html

– http://vger.kernel.org/netconf2005.html

– http://vger.kernel.org/netconf2006.html

– Linux Conf Australia, January 2008,Melbourne

http://vger.kernel.org/netconf2010.html

http://vger.kernel.org/netconf2011.html

11) http://www.policyrouting.org/PolicyRoutingBook/

12) THRASH A dynamic LCtrie and hash data structure:

Robert Olsson Stefan Nilsson, August 2006

http://www.csc.kth.se/~snilsson/public/papers/trash/trash.pdf

13) IPSec howto:

http://www.ipsechowto.org/t1.html

14) Openswan: Building and Integrating Virtual Private Networks , by Paul Wouters, Ken Bantoft

http://www.packtpub.com/book/openswan/mid/061205jqdnh2by publisher: Packt Publishing.

15) http://www.vyatta.com/ Open-Source Networking

16) For a very basic description of the network stack, see [1].

17) http://www.ibm.com/developerworks/linux/library/l-linux-networking-stack/ gives an overview of the networking stack.

18) http://www.makelinux.net/reference is a general reference for Linux kernel internals.

19) This Linux Journal article by Alan Cox is an overall introduction to the networking kernel.

20) Receive packet steering (RPS)

http://lwn.net/Articles/362339/

RPS and RFS
http://lwn.net/Articles/398385/

Receive flow steering
http://lwn.net/Articles/382428/

xps: Transmit Packet Steering
http://lwn.net/Articles/412062/

21) application for zero copy:

http://netsniff-ng.org/

(trafgen; uses PF_PACKET RAW sockets and sendto() sys call)

22) splice tools: http://brick.kernel.dk/snaps/splice-git-latest.tar.gz

network splice receive:

http://lwn.net/Articles/236918/

23) Network namespaces - by Jonathan Corbet:

http://lwn.net/Articles/219794/

24) The initial change to napi_struct is explained in
ttp://lwn.net/Articles/244640/

25) “**How GRO works”** by David Miller:

http://vger.kernel.org/~davem/cgi-bin/blog.cgi/2010/08/30

26) A JIT for packet filters By Jonathan Corbet, April 12, 2011

http://lwn.net/Articles/437981/

27) dynamic seccomp policies (using BPF filters)
http://lwn.net/Articles/475019

28) LAN Ethernet Maximum Rates, Generation, Capturing & Monitoring

http://wiki.networksecuritytoolkit.org/nstwiki/index.php/LAN_Ethernet_Maximum_Rates,_Generation,_Capturing_%26_Monitoring

29) Network data flow through kernel - diagram:

http://www.linuxfoundation.org/images/1/1c/Network_data_flow_through_kernel.png

30) The TCP/IP Guide: online book: http://www.tcpipguide.com/free/index.htm

31) Quagga: http://www.nongnu.org/quagga/

32) Communicating between the kernel and user-space in Linux using Netlink sockets Netlink article:

http://1984.lsi.us.es/~pablo/docs/spae.pdf

33) generic netlink sockets:

https://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto

34) Convert and locate IP addresses:

http://www.kloth.net/services/iplocate.php

kernel networking repositories:

35) Intel SR-IOV Explanation **By Patrick Kutch, Intel.**

http://www.youtube.com/watch?v=hRHsk8Nycdg

http://www.windowsitpro.com/article/systems-management/sriov-single-root-io-virtualization-142151

36) Tuning 10Gb network cards on Linux: Breno Henrique Leitao, IBM:

http://www.kernel.org/doc/ols/2009/ols2009-pages-169-184.pdf

37) Netdev IRC:

http://www.spinics.net/lists/netdev/msg225170.html

36) **Monitoring tools:**

Zabbix is an enterprise-class open source distributed monitoring solution for networks and applications.

http://www.zabbix.com/

ntop

http://www.ntop.org/

VSOCK - TBD

To clone the stable tree you should run:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/davem/net.git
```

To clone net-next you should run:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-
next.git
```

Rami Rosen