

# Linux Kernel Networking

Rami Rosen

[ramirose@gmail.com](mailto:ramirose@gmail.com)

Haifux, August 2007

# Disclaimer

Everything in this lecture shall not, under any circumstances, hold any legal liability whatsoever.

Any usage of the data and information in this document shall be solely on the responsibility of the user.

This lecture is not given on behalf of any company or organization.

# Warning



- This lecture will deal with design functional description side by side with many implementation details; some knowledge of “C” is preferred.

# General

- The Linux networking kernel code (including network device drivers) is a large part of the Linux kernel code.
- **Scope:** We will not deal with wireless, IPv6, and multicasting.
  - Also not with user space routing daemons/apps, and with security attacks (like DoS, spoofing, etc.) .
- 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.
- There is a 10 pages Linux kernel networking walkthrouh document

# General - Contd.

- Though it deals with 2.4.20 Linux kernel, most of it is relevant.
- This lecture will concentrate on this walkthrough (design and implementation details).
- References to code in this lecture are based on linux-2.6.23-rc2.
- There was some serious cleanup in 2.6.23

# Hierarchy of networking layers

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

Transport Layer (L4) (udp,tcp...)

Network Layer (L3) (ip)

Link Layer (L2) (ethernet)

# Networking Data Structures

- The two most important structures of linux kernel network layer are:
  - sk\_buff (defined in *include/linux/skbuff.h*)
  - netdevice (defined in *include/linux/netdevice.h*)
- It is better to know a bit about them before delving into the walkthrough code.

# SK\_BUFF

- sk\_buff represents data and headers.
- sk\_buff API (examples)
  - sk\_buff allocation is done with *alloc\_skb()* or *dev\_alloc\_skb()*; drivers use *dev\_alloc\_skb()*; (free by *kfree\_skb()* and *dev\_kfree\_skb()*).
- *unsigned char\* data* : points to the current header.
- *skb\_pull(int len)* – removes data from the start of a buffer by advancing data to data+len and by decreasing len.
- Almost always sk\_buff instances appear as “skb” in the kernel code.

## SK\_BUFF - contd

- sk\_buff includes 3 unions; each corresponds to a kernel network layer:
- **transport\_header** (previously called h) – for layer 4, the transport layer (can include tcp header or udp header or icmp header, and more)
- **network\_header** – (previously called nh) for layer 3, the network layer (can include ip header or ipv6 header or arp header).
- **mac\_header** – (previously called mac) for layer 2, the link layer.
- skb\_network\_header(skb), skb\_transport\_header(skb) and skb\_mac\_header(skb) return pointer to the header.

## SK\_BUFF - contd.

- **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.

## SK\_BUFF - contd.

- 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.
- **tstamp** (of type `ktime_t` ) : time stamp of receiving the packet.
  - *net\_enable\_timestamp()* must be called in order to get values.

# net\_device

- net\_device represents a network interface card.
- 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);
  - In OpenSolaris there is a special pseudo driver called “vnic” which enables bandwidth allocation (project CrossBow).
- Important members:

## net\_device - contd

- **unsigned int mtu** – Maximum Transmission Unit: the maximum size of frame the device can handle.
- Each protocol has mtu of its own; the default is **1500** for Ethernet.
- you can change the mtu with ifconfig; for example, like this:
  - *ifconfig eth0 mtu 1400*
  - You cannot of course, change it to values higher than 1500 on 10Mb/s network:
  - *ifconfig eth0 mtu 1501* will give:
  - *SIOCSIFMTU: Invalid argument*

## net\_device - contd

- **unsigned int flags** - (which you see or set using ifconfig utility): for example, RUNNING or NOARP.
- **unsigned char dev\_addr[MAX\_ADDR\_LEN]** : the MAC address of the device (6 bytes).
- **int (\*hard\_start\_xmit)(struct sk\_buff \*skb, struct net\_device \*dev);**
  - a pointer to the device transmit method.
- **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.)

## net\_device - contd

- You are likely to encounter macros starting with IN\_DEV like:  
IN\_DEV\_FORWARD() or IN\_DEV\_RX\_REDIRECTS(). How are they related to net\_device ? How are these macros implemented ?
- **void \*ip\_ptr**: IPv4 specific data. This pointer is assigned to a pointer to in\_device in *inetdev\_init()* (*net/ipv4/devinet.c*)

## net\_device - Contd.

- struct `in_device` have a member named `cnf` (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=0`
- *But probably this is not so on your ADSL router.*

# network interface drivers

- Most of the nics are 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 **Idd3**.
- There are two modes in which a NIC can receive a packet.
  - The traditional way is interrupt-driven : each received packet is an asynchronous event which causes an interrupt.

# NAPI

- NAPI (new API).
  - The NIC works in polling mode.
  - 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 non-NAPI driver they pass into the stack)
  - in Solaris, polling is built into the kernel (no need to build drivers in any special way)

# User Space Tools

- iputils (including ping, arping, and more)
- net-tools (ifconfig, netstat, , route, arp and more)
- IPROUTE2 (ip command with many options)
  - Uses rtnetlink API.
  - Has much wider functionalities; 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 and the routing cache 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(const struct flowi *flp, struct fib_result *res)`*
- 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`*.

# Routing Subsystem - contd.

- 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 is one routing cache, regardless of how many routing tables there are.
- You can see the routing cache by running *"route -C"*.
- Alternatively, you can see it by : *"cat /proc/net/route"*.
  - con: this way, the addresses are in hex format

# Routing Cache

- The routing cache is built of **rtable** elements:
- struct rtable (see: */include/net/route.h*)

```
{
```

```
union {
```

```
    struct dst_entry dst;
```

```
    } u;
```

```
...
```

```
}
```

## Routing Cache - contd

- The **dst\_entry** is the protocol-independent part.
  - Thus, for example, we have a `dst_entry` member (also called `dst`) in `rt6_info` in `ipv6`. ( *include/net/ip6\_fib.h*)
- The key for a lookup operation in the routing cache is an IP address (whereas in the routing table the key is a subnet).
- Inserting elements into the routing cache by : *rt\_intern\_hash()*
- There is an alternate mechanism for route cache lookup, called **fib\_trie**, which is inside the kernel tree (*net/ipv4/fib\_trie.c*)

## Routing Cache - contd

- It is based on extending the lookup key.
- You should set: `CONFIG_IP_FIB_TRIE` (=y)
  - (instead of `CONFIG_IP_FIB_HASH`)
- By Robert Olsson et al (see links).

# 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*.

## Routing Cache - Contd.

- 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.

# Policy Routing (multiple tables)-contd.

- 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

# Policy Routing (multiple tables)-contd.

- You can add a rule to the **routing policy database (*RPDB*)** by “*ip rule add ...*”
  - The rule can be based on input interface, TOS, fwmark (from netfilter).
- *ip rule list* – show all rules.

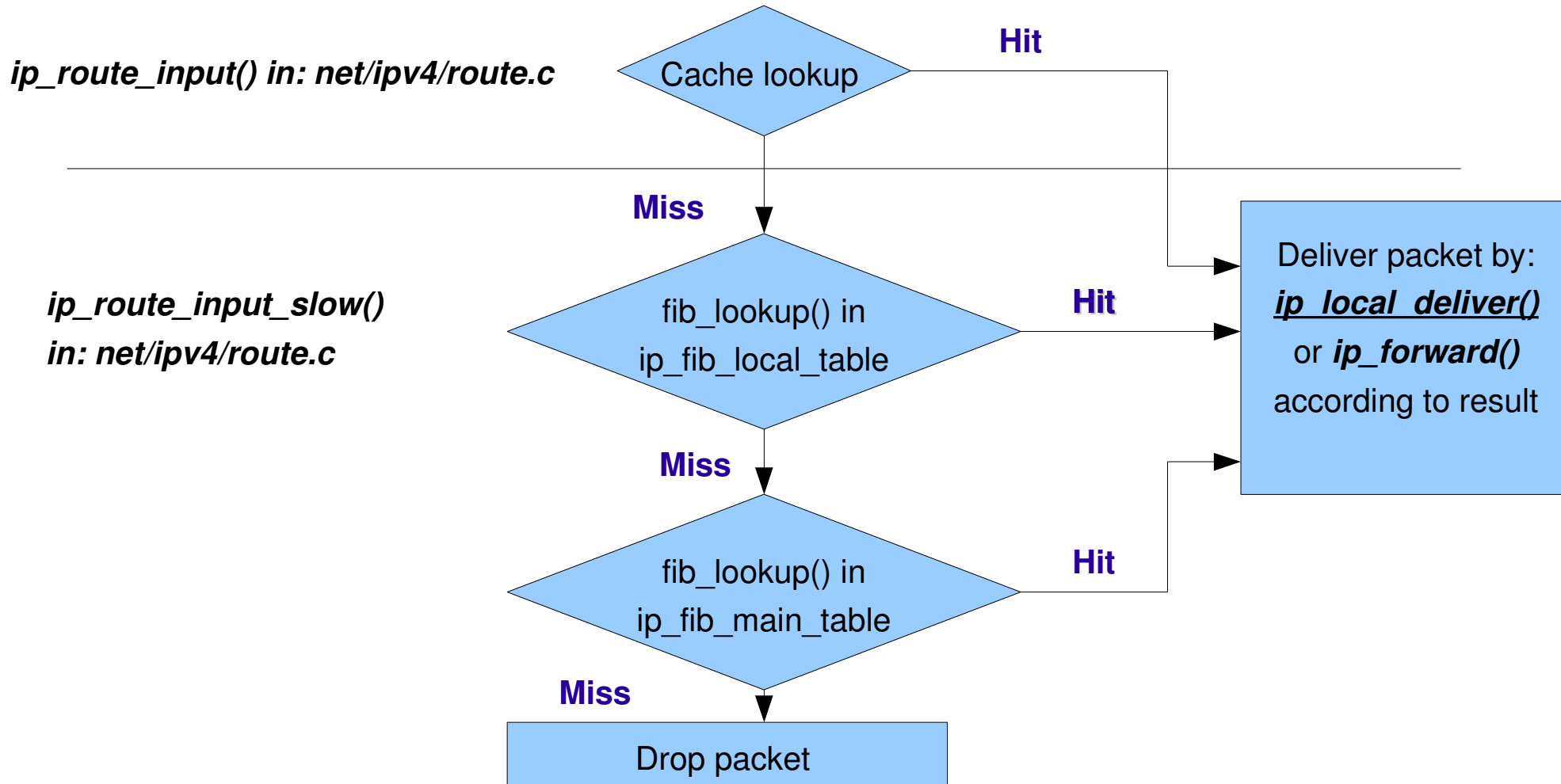
# Policy Routing: add/delete a rule - example

- *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
  - ...

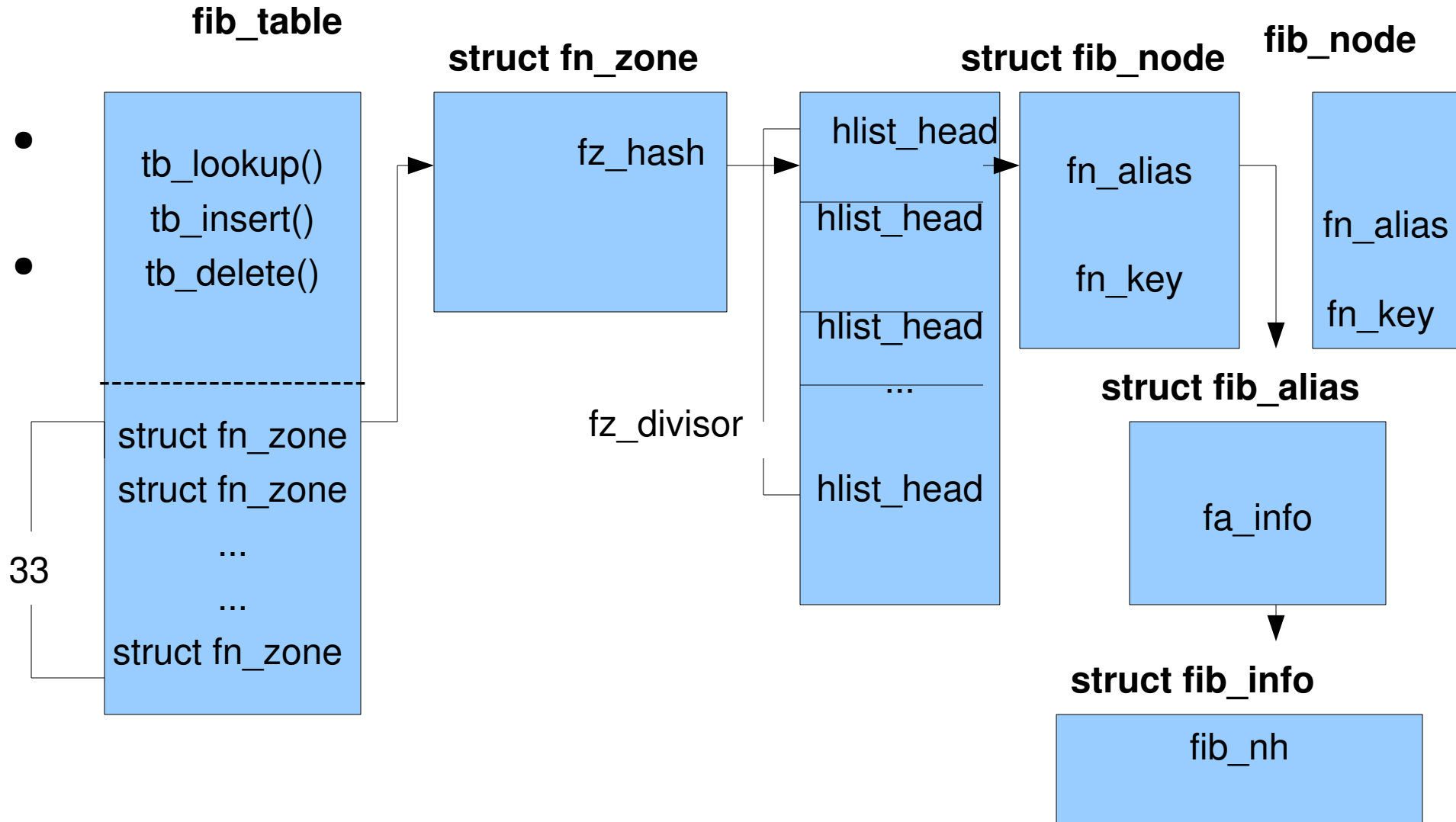
# Policy Routing: add/delete a rule - example

- Delete a rule : *ip rule del tos 0x04 table 252*

# Routing Lookup



# Routing Table Diagram



# Routing Tables

- 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

## Routing Tables - contd

- Each ***fib\_node*** element represents a unique subnet.
  - The ***fn\_key*** member of fib\_node is the subnet (32 bit)

## Routing Tables - contd

- 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.

## Routing Table lookup - contd.

- 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_lookup` 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).

## Receiving a packet - contd

- 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.

# Receiving a packet - contd

- The handler for receiving a packet is *ip\_rcv()*. (*net/ipv4/ip\_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)

# Receiving a packet - contd

- 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.

# Receiving a packet - contd

- *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);
```

# Receiving a packet - contd

- *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.)

# Receiving a packet - contd

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  $\leq 1$  , the methods send ICMP message (ICMP\_TIME\_EXCEEDED) and drops the packet.*
  - *Calls NF\_HOOK(PF\_INET,NF\_IP\_FORWARD, skb, skb->dev, rt->u.dst.dev, ip\_forward\_finish);*

## Forwarding- Contd

- *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*)

# Sending a Packet

- Handling of sending a packet is done by ***ip\_route\_output\_key()***.
- We need to perform routing lookup also in the case of transmission.
- In case of a cache miss, we call *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()*

# Sending a Packet-contd

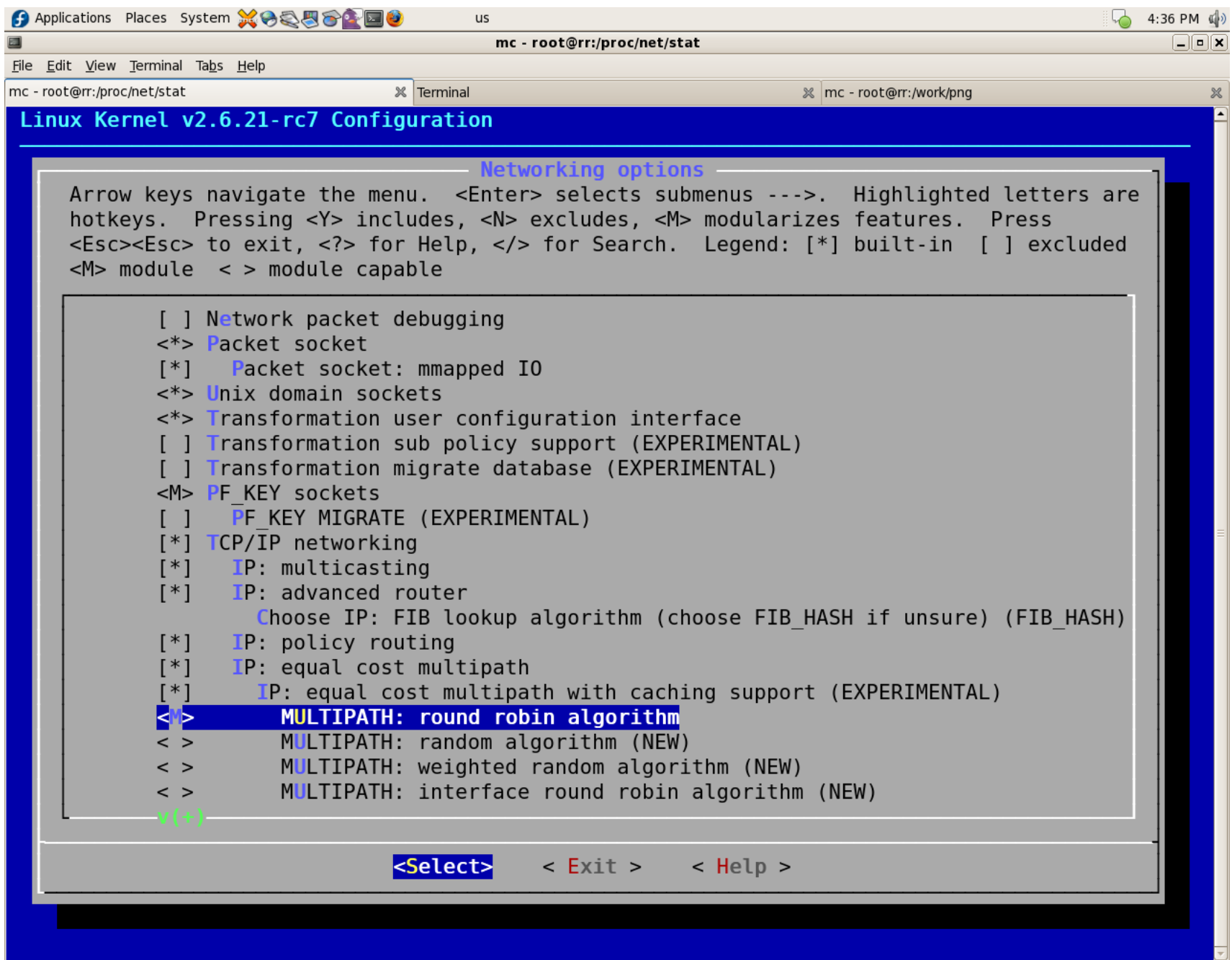
- *ip\_output()* will call *ip\_finish\_output()*
  - This is the NF\_IP\_POST\_ROUTING point.
- *ip\_finish\_output()* will eventually send the packet from a neighbor by:
  - *dst->neighbour->output(skb)*
  - *arp\_bind\_neighbour()* sees to it that the L2 address of the next hop will be known. (*net/ipv4/arp.c*)

## Sending a Packet - Contd.

- 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*

# 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).



# Netfilter

- Netfilter is the kernel layer to support applying iptables rules.
  - It enables:
    - Filtering
    - Changing packets (masquerading)
    - Connection Tracking

# 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()*

## Netfilter rule - example (contd)

- 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*.

# 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)**.

# ICMP redirect message - contd.

- 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.

# ICMP redirect message - contd.

- 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	<b>Gateway</b>	Genmask	Flags	Metric	Ref	Use	Iface
192.168.0.10	<b>192.168.0.121</b>	255.255.255.255	UGH	0	0	0	eth0

## ICMP redirect message - contd.

- 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:
-

## ICMP redirect message - contd.

- 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*)

## ICMP redirect message - contd.

- 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).

## ICMP redirect message - contd.

- 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)
    - see: *include/linux/if\_ether.h*

# Neighboring Subsystem - contd

- 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

- You can define a bridge and add NICs to it (“enslaving ports”) using *brctl* (from bridge-utils).
- You can have up to 1024 ports for every bridge device (BR\_MAX\_PORTS) .
- Example:
- *brctl addbr mybr*
- *brctl addif mybr eth0*
- *brctl show*

## Bridging Subsystem - contd.

- 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 we receive a frame, *netif\_receive\_skb()* calls *handle\_bridge()*.

## Bridging Subsystem - contd.

- 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.

## Bridging Subsystem- contd

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

# 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.
- Transformation bundles.
- Chain of dst entries; only the last one is for routing.

# IPSec-cont.

- User space tools: <http://ipsec-tools.sf.net>
- Building VPN : <http://www.openswan.org/> (Open Source).
- 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.

# IPSec-contd.

- Show the security policies:
  - *ip xfrm policy show*
- Create RSA keys:
  - *ipsec rsasigkey --verbose 2048 > keys.txt*
  - *ipsec showhostkey --left > left.publickey*
  - *ipsec showhostkey --right > right.publickey*

# IPSec-contd.

Example: Host to Host VPN (using openswan)

in */etc/ipsec.conf*:

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

# IPSec-contd.

- *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/ip-sysctl.txt: networking kernel tunables
- 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).

# Tips for hacking - Contd.

- 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 `tracert` (see `man tracert`).
  - `Tracert` is from `iputils`.

# Tips for hacking - Contd.

- 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. */  
  
};
```

## Tips for hacking - Contd.

- NIPQUAD() : macro for printing hex addresses
- CONFIG\_NET\_DMA is for TCP/IP offload.
- When you encounter: xfrm / CONFIG\_XFRM this has to do with IPSEC. (transformers).

# New and future trends

- IO/AT.
- NetChannels (Van Jacobson and Evgeniy Polyakov).
- TCP Offloading.
- RDMA.
- Multiqueus. : some new nics, like e1000 and IPW2200, allow two or more hardware Tx queues. There are already patches to enable this.

## New and future trends - contd.

- See: “Enabling Linux Network Support of Hardware Multiqueue Devices”, OLS 2007.
- Some more info in: `Documentation/networking/multiqueue.txt` in recent Linux kernels.
- Devices with multiple TX/RX queues will have the `NETIF_F_MULTI_QUEUE` feature (`include/linux/netdevice.h`)
- MQ nic drivers will call *`alloc_etherdev_mq()`* or *`alloc_netdev_mq()`* instead of *`alloc_etherdev()`* or *`alloc_netdev()`*.

# Links and more info

1) Linux Network Stack Walkthrough (2.4.20):

[http://gicl.cs.drexel.edu/people/sevy/network/Linux\\_network\\_stack\\_wa](http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_wa)

2) Understanding the Linux Kernel, Second Edition

By Daniel P. Bovet, Marco Cesati

Second Edition December 2002

chapter 18: networking.

- Understanding Linux Network Internals, Christian benvenuti

Oreilly , First Edition.

## Links and more info

3) Linux Device Driver, by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

Third Edition February 2005.

- Chapter 17, Network Drivers

4) Linux networking: (a lot of docs about specific networking topics)

- [http://linux-net.osdl.org/index.php/Main\\_Page](http://linux-net.osdl.org/index.php/Main_Page)

5) netdev mailing list: <http://www.spinics.net/lists/netdev/>

## Links and more info

6) Removal of multipath routing cache from kernel code:

<http://lists.openwall.net/netdev/2007/03/12/76>

<http://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/>

## Links and more info

### 9) **Writing Network Device Driver for Linux:** (article)

- <http://app.linux.org.mt/article/writing-netdrivers?locale=en>

## Links and more info

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>
- Next one: Linux Conf Australia, January 2008, Melbourne
- David S. Miller, James Morris , Rusty Russell , Jamal Hadi Salim , Stephen Hemminger , Harald Welte, Hideaki YOSHIFUJI, Herbert Xu , Thomas Graf , Robert Olsson , Arnaldo Carvalho de Melo and others

## Links and more info

### 11) **Policy Routing With Linux** - Online Book Edition

- by Matthew G. Marsh (Sams).
- <http://www.policyrouting.org/PolicyRoutingBook/>

### 12) THRASH - A dynamic LC-trie 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.ipsec-howto.org/t1.html>

## Links and more info

14) Openswan: Building and Integrating Virtual Private Networks , by Paul Wouters, Ken Bantoft

<http://www.packtpub.com/book/openswan/mid/061205jqdnh2by>

publisher: Packt Publishing.

# Linux Kernel Networking- advanced topics: Neighboring and IPsec



Rami Rosen  
[ramirose@gmail.com](mailto:ramirose@gmail.com)  
Haifux, January 2008  
[www.haifux.org](http://www.haifux.org)

# Contents

- Short rehearsal (4 slides)
- Neighboring Subsystem
  - struct neighbour
  - arp
  - arp\_bind\_neighbour() method
  - Duplicate Address Detection (DAD)
  - LVS (Linux Virtual Sever)
  - ARPD – arp user space daemon
  - Neighbour states
  - Change of IP address/Mac address
- IPsec

# Scope

- We will not deal with multicast and with ipv6 and with wireless.
- The L3 network protocol we deal with is ipv4, and the L2 Link Layer protocol is Ethernet.

# Neighboring Subsystem

- All code in this lecture is taken from linux-2.6.24-rc4
- 04-Dec-2007
- Can be obtained from <http://www.kernel.org/pub/linux/kernel/v2.6/testing/> (and mirrors)

# Short rehearsal (4 slides)

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

Transport Layer (L4) (udp,tcp...)

Network Layer (L3) (ip)

Link Layer (L2) (ethernet)

## Short rehearsal (4 slides)

- Two most Important data structures: sk\_buff and net\_device.

### **sk\_buff:**

- dst is an instance of dst\_entry; dst is a member in sk\_buff.
- The lookup in the routing subsystem constructs dst.
- It decides how the packet will continue its traversal.
- This is done by assigning methods to its input()/output() functions
- Each dst\_entry has a neighbour member.(with IPsec it is NULL).
- When working with IPsec, the dst in fact represents a linked list of dst\_entries. Only the last one is for routing; all previous dst\_entries are for IPsec transformers.

# Short rehearsal (4 slides)

## net\_device

- net\_device represents a Network Interface Card.
- net\_device has members like mtu, dev\_addr (device MAC address), promiscuity, name of device (eth0, eth1, lo, etc), and more.
- An important member of net\_device is flags.
- You can disable ARP replies on a NIC by setting IFF\_NOARP flag:
- ***ifconfig eth0 -arp***
  - ***ifconfig eth0*** will show:
    - UP BROADCAST RUNNING **NOARP** MULTICAST ...
  - Enabling ARP again is done by: ***ifconfig eth0 arp***.

## Short rehearsal (4 slides)

- ***ip\_input\_route()*** method: performs a lookup in the routing subsystem for each incoming packet. Looks first in the routing cache; in case there is a cache miss, looks into the routing table and inserts an entry into the routing cache. Calls ***arp\_bind\_neighbour()*** for UNICAST packets only. Returns 0 upon success.
- ***dev\_queue\_xmit(struct sk\_buff \*skb)*** is called to transmit the packet, when it is ready. (has L2 destination address)  
(*net/core/dev.c*)
  - ***dev\_queue\_xmit()*** passes the packet to the nic device driver for transmission using the device driver ***hard\_start\_xmit()*** method.

# Neighboring Subsystem

- **Goals: what is the neighboring subsystem for?**
- *“The world is a jungle in general, and the networking game contributes many animals.”* (from RFC 826, ARP, 1982)
- In IPV4 implemented by **ARP**; in IPv6: **ND**, neighbour discovery.
- Ethernet header is 14 bytes long:
- Source Mac address and destination Mac address - 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:



# Neighboring Subsystem – struct neighbour

- Implementation - important data structures
- `struct neighbour` (*/include/net/neighbour.h*)
  - *ha* - 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).

# Neighboring Subsystem – struct neighbour contd

- A neighbour can change its state to NUD\_REACHABLE by one of three ways:
- L4 confirmation.
- Receive ARP reply for the first time or receiving an ARP reply in response to an ARP request when in NUD\_PROBE state.
- Confirmation can be done also by issuing a sysadmin command (but it is rare).

# Neighboring Subsystem – struct neighbour contd

- *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, then 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.

# Neighboring Subsystem – struct neighbour contd

- `timer` (The callback method is `neigh_timer_handler()`).
- `struct hh_cache *hh` (defined in `include/linux/netdevice.h`)
- `confirmed` – confirmation timestamp.
  - Confirmation can be 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()`.

# Neighboring Subsystem – struct neighbour contd

- *dev* (net\_device from which we send packets to the neighbour).
- *struct neigh\_parms \*parms;*
  - parms include mostly timer tunables, net structure (network namespaces), etc.
  - network namespaces enable multiple instances of the network stack to the user space.
- A network device belongs to exactly one network namespace.
- CONFIG\_NET\_NS when building the kernel.

# Neighboring Subsystem – struct neighbour contd

- *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\*](#)

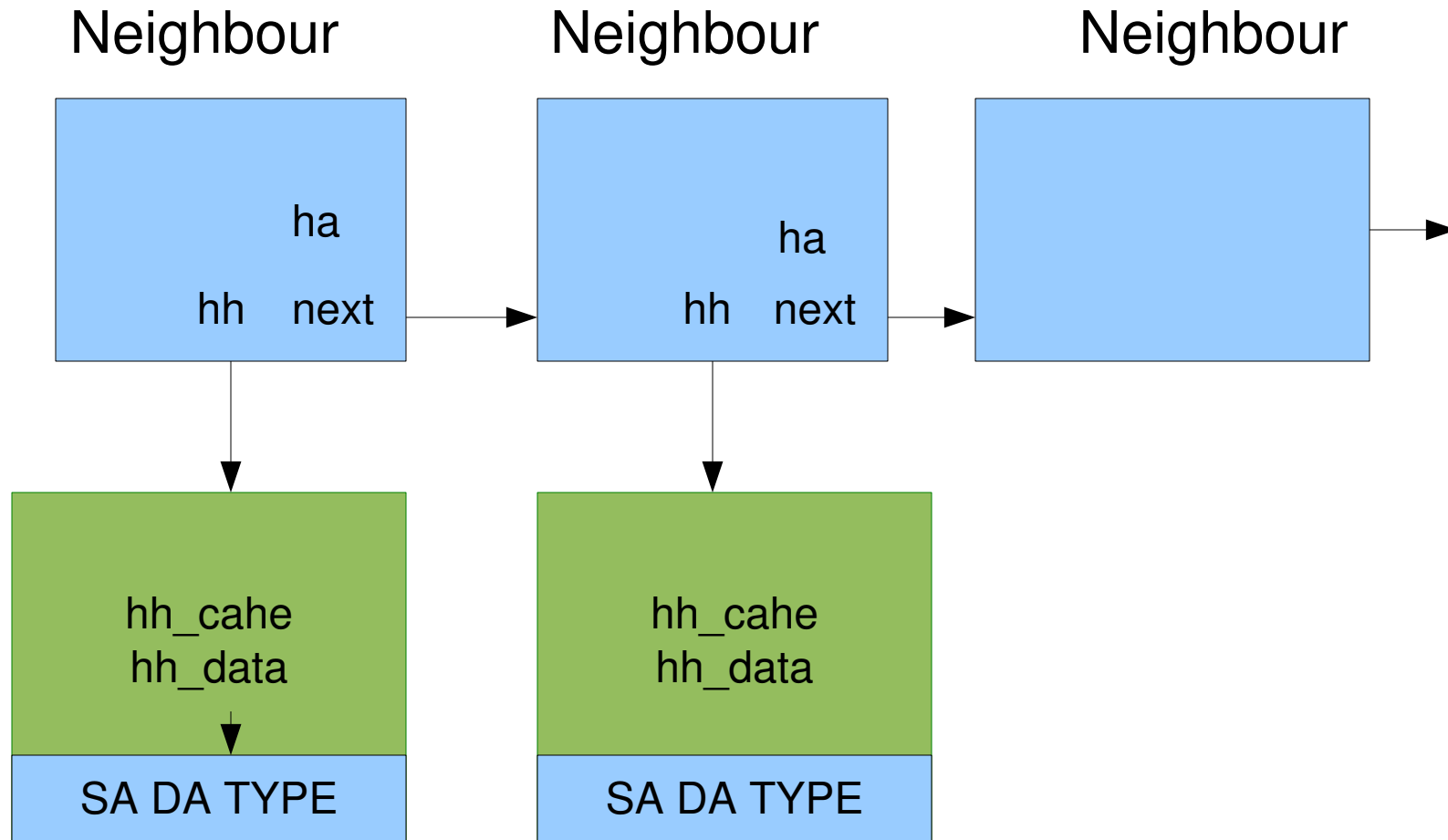
# struct neigh\_table

- `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*.

# ARP table



# Neighboring Subsystem - arp

- You can see the contents of the arp table by running:  
“*cat /proc/net/arp*” or by running the “arp” from a command line .
- *ip neigh show* is the new method to show arp (from IPROUTE2)
- 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).

# Neighboring Subsystem – arp table

- 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 show neigh

- 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

192.168.0.98 dev eth0 INCOMPLETE

# Neighboring Subsystem – arp

- *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;
```

# Neighboring Subsystem - arp

(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 an 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.

# Neighboring Subsystem - arp

- 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()*.

# Neighboring Subsystem - arp

- 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_UN SOLICITED_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.

# arp\_bind\_neighbour() method

- *neigh\_create()* creates a neighbour with NUD\_NONE state
  - setting nud\_state to NUD\_NONE is done in neigh\_alloc()

# Neighboring Subsystem – 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 FC8,for example)
- `/etc/sysconfig/network-scripts/ifcfg-eth1`

...

`IPADDR=192.168.0.122`

...

and then run:

- `ifup eth1`

# Changing IP address - contd.

- 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 ?
- ifup is from the initscripts package.

# 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.

eth0: Capturing - Wireshark

File Edit View Go Capture Analyze Statistics Help

Filter:  + Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
1	0.0	AbitComp_93:ac:af	Broadcast	ARP	Who has 192.168.0.10? Tell 0.0.0.0

Frame 1 (42 bytes on wire, 42 bytes captured)

- Ethernet II, Src: AbitComp\_93:ac:af (00:50:8d:93:ac:af), Dst: Broadcast
  - Destination: Broadcast (ff:ff:ff:ff:ff:ff)
  - Source: AbitComp\_93:ac:af (00:50:8d:93:ac:af)
  - Type: ARP (0x0806)
- Address Resolution Protocol (request)
  - Hardware type: Ethernet (0x0001)
  - Protocol type: IP (0x0800)
  - Hardware size: 6
  - Protocol size: 4
  - Opcode: request (0x0001)
  - Sender MAC address: AbitComp\_93:ac:af (00:50:8d:93:ac:af)
  - Sender IP address: 0.0.0.0 (0.0.0.0)
  - Target MAC address: Broadcast (ff:ff:ff:ff:ff:ff)
  - Target IP address: 192.168.0.10 (192.168.0.10)

0010 08 00 06 04 00 01 00 50 8d 93 ac af 00 00 00 00 .....P .....

Sender IP address (arp.src.proto\_ipv4), 4 bytes P: 1 D: 1 M: 0

# Duplicate Address Detection -contd

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” in the example above).

# Neighboring Subsystem – Asynchronous Garbage Collection

- *neigh\_forced\_gc()* performs synchronous 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.

# Neighboring Subsystem – Garbage Collection

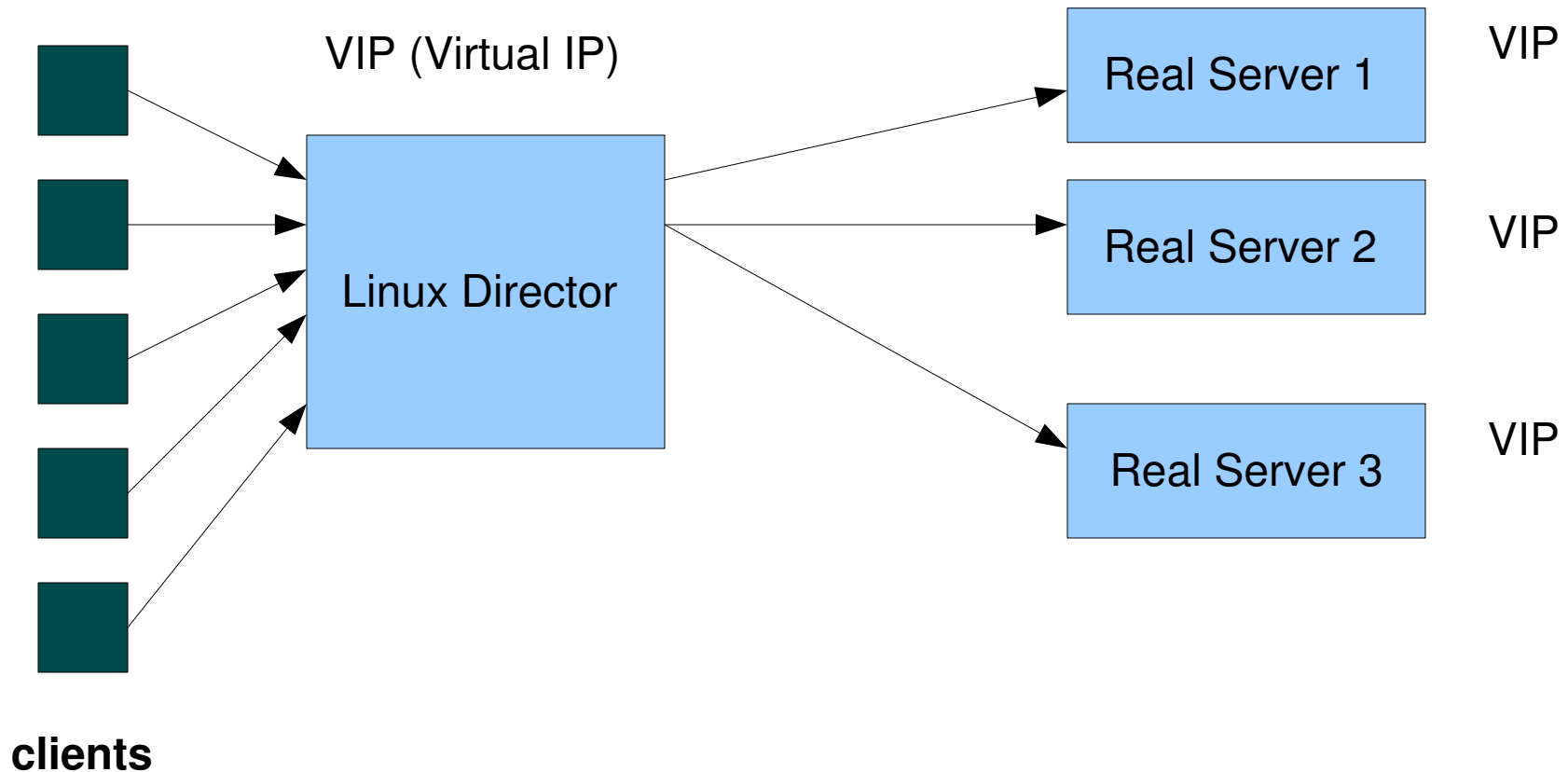
- Changing the neighbour state is done only in *neigh\_timer\_handler()* .

# LVS (Linux Virtual Server)

- <http://www.linuxvirtualserver.org/>
- Integrated into the Linux kernel (in 2.4 kernel it was a patch).
- Located in: *net/ipv4/ipvs* in the kernel tree. No IPV6 support.
- 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 Virtual IP (VIP).



# LVS and ARP

- 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.

54.eth - Wireshark

File Edit View Go Capture Analyze Statistics Help

Filter:  + Expression... Clear Apply

Destination	Protocol	Info
Broadcast	ARP	Who has 192.168.0.151? Tell 192.168.0.54
aa:ab:ac:ad:ae:af	ARP	192.168.0.151 is at 00:00:00:aa:bb:cc
aa:ab:ac:ad:ae:af	ARP	192.168.0.151 is at 00:00:00:cc:bb:aa

▼ Ethernet II, Src: aa:ab:ac:ad:ae:af (aa:ab:ac:ad:ae:af), Dst: Broadcast

- Destination: Broadcast (ff:ff:ff:ff:ff:ff)
- Source: aa:ab:ac:ad:ae:af (aa:ab:ac:ad:ae:af)
- Type: ARP (0x0806)

▼ Address Resolution Protocol (request)

- Hardware type: Ethernet (0x0001)
- Protocol type: IP (0x0800)
- Hardware size: 6
- Protocol size: 4
- Opcode: request (0x0001)
- Sender MAC address: aa:ab:ac:ad:ae:af (aa:ab:ac:ad:ae:af)
- Sender IP address: 192.168.0.54 (192.168.0.54)
- Target MAC address: Broadcast (ff:ff:ff:ff:ff:ff)
- Target IP address: 192.168.0.151 (192.168.0.151)

0000 ff ff ff ff ff ff aa ab ac ad ae af 08 06 00 01 .....  
0010 08 00 06 01 00 01 aa ab ac ad ae af 00 28 00 36

File: "/work/doc/54.eth" 234 Bytes 00:00:00 P: 3 D: 3 M: 0

# 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*)

# LVS and ARP

- <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. (It is considered “Experimental”).
- **see:** /usr/share/doc/iproute-2.6.22/arpd.ps (Alexey Kuznetsov).

# ARPD

- 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.
- We can tune kernel parameters as we like; in fact, we can tune it so that `arp` requests will be send only from the daemon and not from the kernel at all.

# ARPD

- 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`)

# ARPD

- 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.
  - Technion (?)
  - 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, ARP spoofing).
- **Arpwatch** is an open source tool;helps to detect such 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: ...

# 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
  - Sometimes you should only bring down the nic before.

# Flushing the arp table

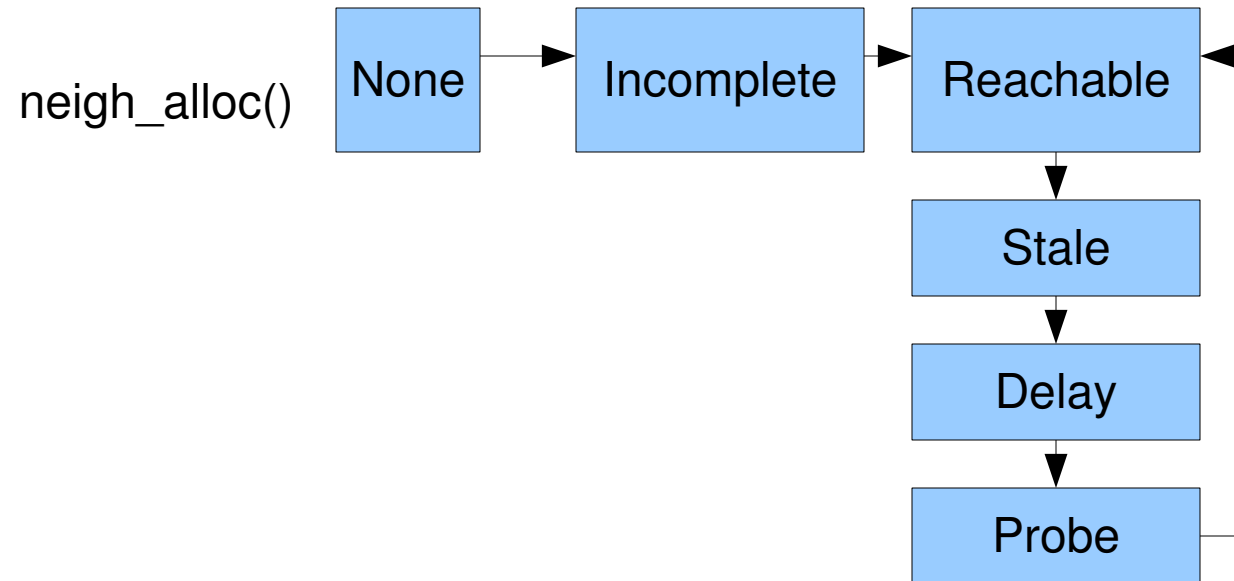
- Flushing the arp:
- `ip -statistics neigh flush dev eth0`
- `*** Round 1, deleting 7 entries ***`
- `*** Flush is complete after 1 round ***`

# Flushing the arp table -contd

- 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`

# Neighbour states

- neighbour states



# Neighboring Subsystem – states

- NUD states
  - NUD\_NONE
  - NUD\_REACHABLE
  - NUD\_STALE
  - NUD\_DELAY
  - NUD\_PROBE
  - NUD\_FAILED
  - NUD\_INCOMPLETE

# Neighboring Subsystem – states

- From the beginning of core/neighbour.c:
- Is it a (latent) bug ?

```
if (!(state & NUD_IN_TIMER)) {  
    #ifndef CONFIG_SMP  
        printk(KERN_WARNING "neigh: timer & !nud_in_timer\n");  
    #endif  
    goto out;  
}
```

# Neighboring Subsystem – states

- 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)
  - When removing a neighbour, we stop the timer (call *del\_timer()*) only if the state is NUD\_IN\_TIMER.
- NUD\_VALID (NUD\_PERMANENT|NUD\_NOARP|NUD\_REACHABLE|NUD\_PROBE|NUD\_STALE|NUD\_DELAY)
- NUD\_CONNECTED (NUD\_PERMANENT|NUD\_NOARP|NUD\_REACHABLE)

# Neighbour states

- 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.

# Neighbour states

- 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 *hard\_start\_xmit()* method of the NIC device driver.
- The *hard\_start\_xmit()* method actually puts the frame on the wire.

# 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.
- In 2.6 kernel : implemented by Dave Miller and Alexey Kuznetsov.
- Transformation bundles.
- Chain of dst entries; only the last one is for routing.
- The dst entries in the chain have A NULL Neighbor as a member.
  - (except the last one)

# IPSec-cont.

- RFC2401

# IPSec-cont.

- User space tools: <http://ipsec-tools.sf.net>
- Building VPN : <http://www.openswan.org/> (Open Source).
- There are also non IPSec solutions for VPN
  - OpenVPN uses ssl/tls.
  - 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.

# IPSec-contd.

- Show the security policies:
  - *ip xfrm policy show*
- Create RSA keys:
  - *ipsec rsasigkey --verbose 2048 > keys.txt*
  - *ipsec showhostkey --left > left.publickey*
  - *ipsec showhostkey --right > right.publickey*

# IPSec-contd.

Example: Host to Host VPN (using openswan)

in */etc/ipsec.conf*:

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

# IPSec-contd.

- *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/ip-sysctl.txt: networking kernel tunables
- 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).
- A BASH script for getting MAC address from IP address: (ipToHex.sh)

```
#!/bin/sh
```

```
IP_ADDR=$1
```

```
for I in $(echo ${IP_ADDR}| sed -e "s\./ /g"); do
```

```
    printf '%02X' $I
```

```
done
```

```
echo
```

```
usage example: ./ipToHex.sh 192.168.0.1 => C0A80001
```

# Tips for hacking - Contd.

- 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 `tracert` (see `man tracert`).
  - `Tracert` is from `iputils`.

## Tips for hacking - Contd.

- **inet\_addr\_type()** method: returns the address type; the input to this method is the IP address. The return value can be RTN\_LOCAL, RTN\_UNICAST, RTN\_BROADCAST, RTN\_MULTICAST etc.  
See: `net/ipv4/fib_frontend.c`

## Tips for hacking - Contd.

- In case you want to send a packet from a user space application through a specified device without altering any routing tables:

```
struct ifreq interface;
```

```
strncpy(interface.ifr_ifrn.ifrn_name, "eth1", IFNAMSIZ);
```

```
if (setsockopt(s, SOL_SOCKET, SO_BINDTODEVICE, (char  
    *)&interface, sizeof(interface)) < 0)
```

```
{
```

```
    printf("error setting SO_BINDTODEVICE");
```

```
    exit(1);
```

```
}
```

# Tips for hacking - Contd.

- 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. */  
};
```

# Tips for hacking - Contd.

- NIPQUAD() : macro for printing hex addresses
- Printing mac address (from net\_device):

```
printk("sk_buff->dev =%02x:%02x:%02x:%02x:%02x:%02x\n",  
((skb)->dev)->dev_addr[0], ((skb)->dev)->dev_addr[1],  
((skb)->dev)->dev_addr[2],((skb)->dev)->dev_addr[3],  
((skb)->dev)->dev_addr[4], ((skb)->dev)->dev_addr[5]);
```

- Printing IP address (primary\_key) of a neighbour (in hex format):

```
printk("neigh->primary_key =%02x.%02x.%02x.%02x\n",  
neigh->primary_key[0], neigh->primary_key[1],  
neigh->primary_key[2],neigh->primary_key[3]);
```

# Tips for hacking - Contd.

- Or:

```
printk("***neigh->primary_key= %u.%u.%u.%u\n",  
        NIPQUAD(*(u32*)neigh->primary_key));
```

- CONFIG\_NET\_DMA is for TCP/IP offload.
- When you encounter: xfrm / CONFIG\_XFRM this has to do with IPSEC. (transformers).

# Tips for hacking - Contd.

- Showing arp statistics by:
- ***cat /proc/net/stat/arp\_cache***

entries allocs destroys hash\_grows lookups hits res\_failed  
rcv\_probes\_mcast rcv\_probes\_ucast periodic\_gc\_runs  
forced\_gc\_runs

**periodic\_gc\_runs**: statistics of how many times the  
*neigh\_periodic\_timer()* is called.

# Links and more info

1) Linux Network Stack Walkthrough (2.4.20):

[http://gicl.cs.drexel.edu/people/sevy/network/Linux\\_network\\_stack\\_wa](http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_wa)

2) Understanding the Linux Kernel, Second Edition

By Daniel P. Bovet, Marco Cesati

Second Edition December 2002

chapter 18: networking.

- Understanding Linux Network Internals, Christian benvenuti

Oreilly , First Edition.

# Links and more info

3) Linux Device Driver, by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

Third Edition February 2005.

- Chapter 17, Network Drivers

4) Linux networking: (a lot of docs about specific networking topics)

- [http://linux-net.osdl.org/index.php/Main\\_Page](http://linux-net.osdl.org/index.php/Main_Page)

5) netdev mailing list: <http://www.spinics.net/lists/netdev/>

# Links and more info

6) Removal of multipath routing cache from kernel code:

<http://lists.openwall.net/netdev/2007/03/12/76>

<http://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/>

# Links and more info

## 9) **Writing Network Device Driver for Linux:** (article)

- <http://app.linux.org.mt/article/writing-netdrivers?locale=en>

# Links and more info

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>
- Next one: Linux Conf Australia, January 2008, Melbourne
- David S. Miller, James Morris , Rusty Russell , Jamal Hadi Salim , Stephen Hemminger , Harald Welte, Hideaki YOSHIFUJI, Herbert Xu , Thomas Graf , Robert Olsson , Arnaldo Carvalho de Melo and others

# Links and more info

## 11) **Policy Routing With Linux** - Online Book Edition

- by Matthew G. Marsh (Sams).
- <http://www.policyrouting.org/PolicyRoutingBook/>

## 12) THRASH - A dynamic LC-trie 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.ipsec-howto.org/t1.html>

## Links and more info

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) a book including chapters about LVS:

“The Linux Enterprise Cluster- Build a Highly Available Cluster with Commodity Hardware and Free Software”, By Karl Kopper.

<http://www.nostarch.com/frameset.php?startat=cluster>

15) <http://www.vyatta.com> - Open-Source Networking

# Links and more info

16) Address Resolution Protocol (ARP)

– <http://linux-ip.net/html/ether-arp.html>

17) ARPWatch – a tool for monitor incoming ARP traffic.

Lawrence Berkeley National Laboratory -

<ftp://ftp.ee.lbl.gov/arpwatch.tar.gz>.

18) arptables:

<http://ebtables.sourceforge.net/download.html>

19) TCP/IP Illustrated, Volume 1: The Protocols

By W. Richard Stevens

<http://www.informit.com/store/product.aspx?isbn=0201633469>

## Links and more info

20) Unix Network Programming, Volume 1: The Sockets  
Networking API (3rd Edition) (Addison-Wesley Professional  
Computing Series) (Hardcover)

by W. Richard Stevens (Author), Bill Fenner (Author), Andrew M.  
Rudoff (Author)

# Questions

- Questions ?
- Thank You !

# IPV6

## Linux Kernel Networking (3)- advanced topics



Rami Rosen  
[ramirose@gmail.com](mailto:ramirose@gmail.com)  
Haifux, April 2008  
[www.haifux.org](http://www.haifux.org)

# Linux Kernel Networking (3)- advanced topics

- Note:
- This lecture is a sequel to the following two lectures I gave:
- **Linux Kernel Networking lecture**
  - <http://www.haifux.org/lectures/172/>
  - **slides**:<http://www.haifux.org/lectures/172/netLec.pdf>
- **Advanced Linux Kernel Networking - Neighboring Subsystem and IPSec lecture**
  - <http://www.haifux.org/lectures/180/>
  - **slides**:<http://www.haifux.org/lectures/180/netLec2.pdf>

# Contents

- IPV6
  - General
  - ICMPV6
  - Radvd
  - Autoconfiguration
- Network Namespaces
- Bridging Subsystem
- Pktgen kernel module.
- Tips
- Links and more info

# Scope

- We will not deal with wireless.
- The L3 network protocol we deal with is ipv4/ipv6, and the L2 Link Layer protocol is Ethernet.

# IPV6 -General

- Discussions started at IETF in 1992 (IPng).
- First Specification: RFC1883 (1995).
- Was deprecated by RFC2460 (1998)
- Main reason for IPV6: shortage of IPv4 addresses.
  - The address space is enlarged in IPV6 from  $2^{32}$  to  $2^{128}$ . (which is by  $2^{96}$ ).
- Secondary reason: improvements over IPV4.
  - For example: using ICMPV6 as a neighbour protocol instead of ARP.
  - Fixed IP header (40 bytes) (in IPV4 it is 20-60 bytes).

# IPV6 -General

- Usually in IPV4, Mobile devices are behind NAT.
- Using mobile IPV6 devices which are not behind a NAT can avoid the need to send Keep-Alive.
- Growing market of mobile devices
  - Some say number of mobile devices will exceed 4 billion in the end of 2008.
- IPSec is mandatory in IPV6 and optional in IPV4.
  - Though most operating systems implemented IPSec also in IPv4.

# IPV6 - history

- In the end of 1997 **IBM's AIX 4.3** was the first commercial platform that supported IPv6
- **Sun Solaris** has IPv6 support since Solaris 8 in February 2000.
- 2007: **Microsoft Windows Vista** (2007) has IPv6 supported and enabled by default.
- February 2008: **IANA** added DNS records for the IPv6 addresses of **six of the thirteen root name servers** to enable Internet host to communicate by IPV6.

- Around the world
  - There was a big IPV6 experiment in China
  - USA, Japan, Korea, France: sites which operate with IPV6.
  - Israel: experiment in Intenet Zahav
  - <http://www.m6bone.net/>
  - Freenet6: <http://go6.net/4105/freenet.asp>

# IPV6 in the Linux Kernel

- IPV6 Kernel part was started long ago - 1996 (by Pedro Roque), based on the BSD API; It was **Linux kernel 2.1.8**.
- When adding IPV6 support to the Linux kernel, almost only the Network layer (L3) is changed.
  - Almost no other layer needs to be changed because each layer operates independently.

# IPV6 in the Linux Kernel-contd.

- **USAGI** Project was founded in 2000 in Japan.
  - “Universal Playground for Ipv6”.
  - Held by volunteers, mostly from Japan. The **USAGI** aimed at providing missing implementation according to the new IPV6 RFCs.
  - Awarded with IPV6 ready logo
- Yoshifuki Hideaki-member of USAGI Project; Keio University
- comaintainer of IPV6 in the Linux Kernel.(also maintainer of iputils)



# IPV6 -General

- Yoshfuji git tree.
  - From time to time, the main networking tree pulls this git tree.
  - `git-clone git://git.linux-ipv6.org/gitroot/yoshfuji/linux-2.6-dev.git inet-2.6.26`
  - This git tree supports IPV6 Multicast Routing. (with pim6sd daemon, ported from kame; PIM-SM stands for Protocol Independent Multicast—Sparse Mode).
    - *Based on Mickael Hoerdts IPv6 Multicast Forwarding Patch.*
    - <http://clarinet.u-strasbg.fr/~hoerdt/>
    - Hoerdts patch is partially based on mrouted.
- Many patches in 2.6.\* kernel are from the USAGI project.

- There was also the **KAME** project in Japan, sponsored by six large companies.
- It aimed at providing a free stack of IPv6, IPsec, and Mobile IPv6 for BSD variants.
- <http://www.kame.net/>
- Sponsored by Hitachi and Toshiba and others.
- Mobile IPV6:
  - HUT - Helsinki University of Technology
  - <http://www.mobile-ipv6.org/>

# IPV6 -General

- Many router vendors support IPV6:
  - Cisco supports IPv6 since IOS 12.2(2)T
  - Hitachi
  - Nortel Networks
  - Juniper Networks, others.
  - <http://www.ietf.org/IESG/Implementations/ipv6-implementations.txt>
- Drawbacks of IPV6
  - Currently LVS is not implemented in IPV6.
  - Takes effort to port existing, IPV4 applications.
  - Tunnels, transitions (IPv6 and IPv4)

# IPv6 Addresses

- RFC 4291, IP Version 6 Addressing Architecture
- Format of IPv6 address:
- 8 blocks of 16 bits => 128 bits.
- xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx
- Where x is a hexadecimal digit.
- Leading zeroes can be replaced with "::" , but only once.
- Localhost address:
  - 0000:0000:0000:0000:0000:0000:0000:0001
  - Or, in compressed form:
  - ::1

# IPv6 Addresses - contd

- No broadcast address in IPv6 as in IPv4.
- Global addresses: 2001:....
  - There are more.
- Link Local : FE80:....

# IPV6 -General

- Caveat:
  - Sometimes ipv6 is configured as a module.
  - You cannot rmmod the ipv6 module.
- How can I know if my kernel has support for IPV6?
  - Run: *ls /proc/net/if\_inet6*
- Managing IP address:
- *ifconfig eth0 inet6 add 2001:db8:0:1:240:95ff:fe30:b0a3/64*
- *ifconfig eth0 inet6 del 2001:db8:0:1:240:95ff:fe30:b0a3/64*

# IPv6 -General

- Can be done also by “ip” command (IPROUTE2).
- `ip -6 addr`
- Using tcpdump to monitor ipv6 traffic:
  - `tcpdump ip6`
- or , for example:
  - `tcpdump ip6 and udp port 9999.`
- For wireshark fans:
  - `tethereal -R ipv6`

# IPV6 -General

- To show the Kernel IPv6 routing table :
  - *route -A inet6*
  - *Also: ip -6 route show*
- ssh usage: *ssh -6 2001:db8:0:1:230:48ff:fe61:e5e0*
- *traceroute6 -i eth0 fe80::20d:60ff:fe9a:26d2*
- *netstat -A inet6*
- *ip6tables* solution exist in IPV6.

# IPV6 -General

- *tracepath6* finds PMTU (path MTU).
  - This is done using IPV6\_MTU\_DISCOVER and IPV6\_PMTUDISC\_PROBE socket options.
  - using a UDP socket.

# ICMPV6

- In IPV6, the neighboring subsystem uses ICMPV6 for Neighboring messages (instead of ARP in IPV4).
- There are 5 types of ICMP codes for neighbour discovery messages:

Message	ICMPV6 code
NEIGHBOUR SOLICITATION	(135) -parallel to ARP request in IPV4
NEIGHBOUR ADVERTISEMENT	(136) -parallel to ARP reply in IPV4

ROUTER SOLICITATION	(133)	
ROUTER ADVERTISEMENT	(134)	// see sniff below
REDIRECT	(137)	

- ROUTER ADVERTISEMENT can be periodic or on demand.
- When ROUTER ADVERTISEMENT is sent as a reply to a ROUTER SOLICITATION, the destination address is **unicast**.  
When it is periodic, the destination address is a **multicast** (all hosts).

# Statefull and Stateless config

- There are two ways to configure IPV6 addresses on hosts (except configuring it manually):
- **Statefull**: DHCPV6 on a server.
  - RFC3315, Dynamic Host Configuration Protocol for IPv6 (DHCPv6).
- **Stateless**: for example, RADVD or Quagga on a server.
  - RFC 4862 - IPv6 Stateless Address Autoconfiguration (SLAAC) from 2007 ; Obsoletes RFC 2462 (1998).

In RADVD, you declare a prefix that only hosts (not routers) use. You can define more than one prefix.

- **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*

- 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).
- see <http://www.iana.org/assignments/ipv6-address-space>

- `ping6 -I eth0 FF02::2` or `ping6 -I eth0 ff02:0:0:0:0:0:0:2`  
will cause all the routers to reply.
- This means that all machines on which  
*/proc/sys/net/ipv6/conf/eth\*/forwarding* is 1 will reply.

# RADVD

- RADVD stands for ROUTER ADVERTISEMENT daemon.
- Maintainer: Pekka Savola
- <http://www.litech.org/radvd/>
- Sends ROUTER ADVERTISEMENT messages.
- The handler for all neighboring messages is *ndisc\_rcv()*.  
(*net/ipv6/ndisc.c*)
- When NDISC\_ROUTER\_ADVERTISEMENT message arrives from radvd, it calls *ndisc\_router\_discovery()*.  
(*net/ipv6/ndisc.c*)

# RADVD - contd

- If the receiving machine **is a router** (or is configured **not to accept router advertisement**), the **Router Advertisement** is not handled: see this code fragment from *ndisc\_router\_discovery()* (*net/ipv6/ndisc.c*)  

```
if (in6_dev->cnf.forwarding || !in6_dev->cnf.accept_ra) {  
    in6_dev_put(in6_dev);  
    return;  
}
```
- *addrconf\_prefix\_rcv()* eventually tries to create an address using the prefix received from radvd and the mac address of the machine (*net/ipv6/addrconf.c*).

# RADVD - contd

- Adding the IPV6 address is done in `ipv6_add_addr()`
  - How can we be sure that there is no same address on the LAN ?
  - We can't !
  - Therefore we set this address first to be tentative
  - In `ipv6_add_addr()`:
    - `ifa->flags = flags | IFA_F_TENTATIVE;`
    - *This means that initially this address cannot communicate with other hosts. (except for neighboring messages).*

# RADVD - contd

- Then we start DAD (*by calling `addrconf_dad_start()`*)
- **DAD** is “Duplicate Address Detection”.
- Upon successful completion of DAD, the IFA\_F\_TENTATIVE flag is removed and the host can communicate with other hosts on the LAN. The flag is set to be IFA\_F\_PERMANENT.
- Upon failure of DAD, the address is deleted.
- You see a message like this in the kernel log:
  - **eth0: duplicate address detected!**

# RADVD - contd

- Caveat:
- When using radvd official FC8 rpm, you will see, in /var/log/messages, the following message after starting the daemon:

...

radvd[2614]: version 1.0 started

radvd[2615]: failed to set CurHopLimit (64) for eth0

- You may ignore this message.
- This is due to that we run the daemon as user radvd.
- This was fixed in radvd 1.1 (still no Fedora official rpm).

# RADVD - contd

- radvd: sending router advertisement

// from *radvd-1.0/send.c*

```
send_ra()
```

```
{
```

```
...
```

```
    addr.sin6_family = AF_INET6;
```

```
    addr.sin6_port = htons(IPPROTO_ICMPV6);
```

```
    memcpy(&addr.sin6_addr, dest, sizeof(struct in6_addr));
```

```
    memset(&buff, 0, sizeof(buff));
```

```
radvert = (struct nd_router_advert *) buff;  
radvert->nd_ra_type = ND_ROUTER_ADVERT;
```

...

- and we have in /usr/include/netinet/icmpv6.h:
  - #define ND\_ROUTER\_SOLICIT 133
  - #define ND\_ROUTER\_ADVERT 134
- This Router Advertisement is sent to all hosts address:
  - **FF02::1**

- `nd_router_advert` structure is declared in */usr/include/netinet/icmp6.h*.
- It includes `icmpv6` header.
- Is there a protection against sending malicious Router Advertisements ?
- No thing as rejecting “unsolicited arp replies” as in IPV4 (which is the default behaviour, in order to prevent ARP Cache poisoning)

# radvd.conf - example:

```
interface eth0 #see man radvd.conf
```

```
{
```

```
    AdvSendAdvert on;
```

```
    MaxRtrAdvInterval 30;
```

```
    prefix 2002:db8:0:1::/64
```

```
    {
```

```
        AdvOnLink on;
```

```
        AdvAutonomous on;
```

```
    };
```

```
};
```

# radvd.conf – example (contd)

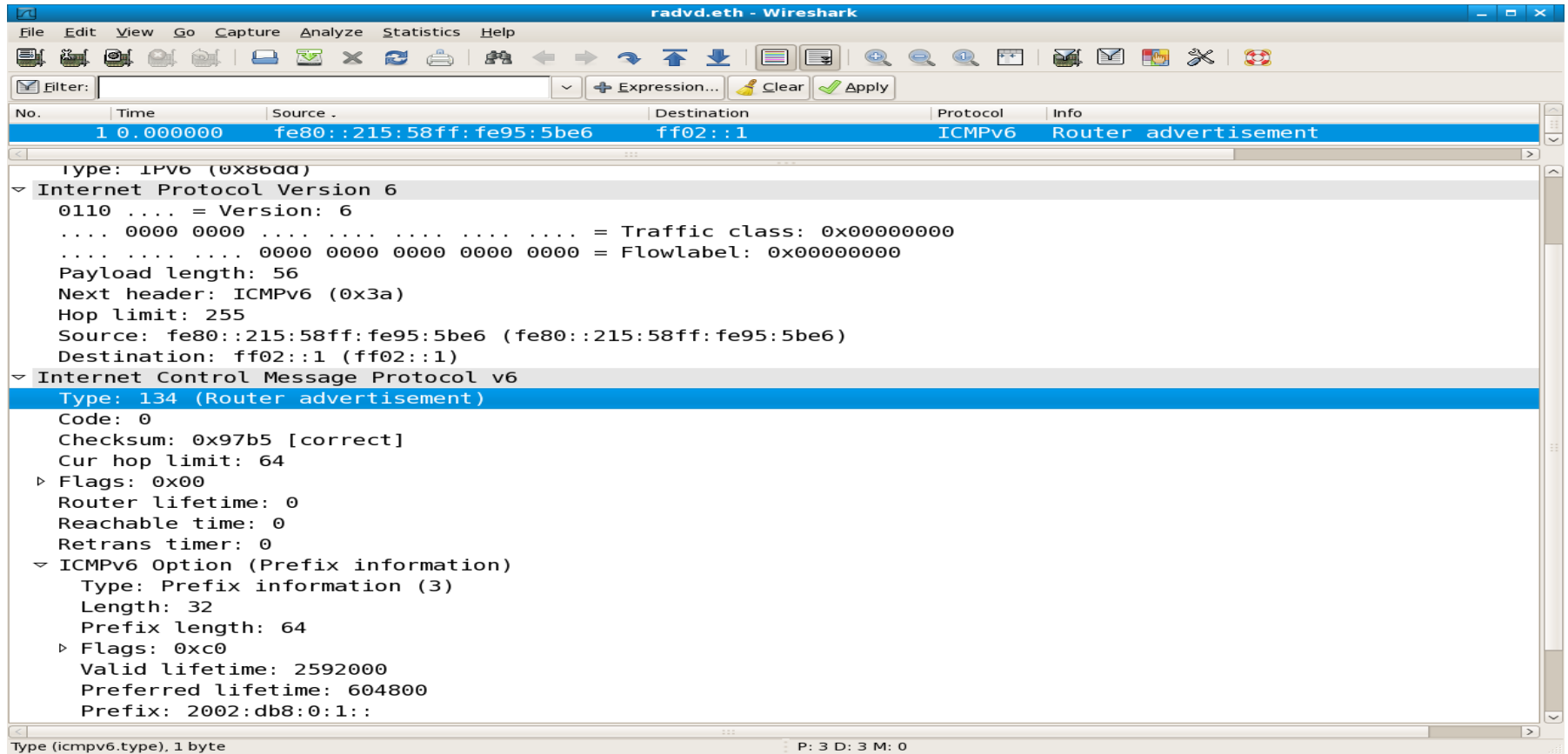
- The prefix length **MUST** be 64.
- See RFC2464, **Transmission of IPv6 Packets over Ethernet Networks** and RFC4291- **IP Version 6 Addressing Architecture**.
- Caveat:
- If the prefix length will be different than 64 than the Router Advertisement will be rejected.
- Caveat: You will not notice it, unless your syslog prints KERN\_DEBUG messages (see man syslog.conf)
- In case the syslog is configured for printing kernel debug messages, you will see this messages in the kernel log

IPv6 addrconf: prefix with wrong length

## radvd.conf – example (contd)

- Caveat 2:
- You cannot start radvd service if there is no link local address configured on your machine. Trying to do so will result with:  
  
radvd: no linklocal address configured for (null)  
  
radvd: error parsing or activating the config file:  
  
[FAILED]
- Even if it was possible, the kernel would reject a Router Advertisements originating from machines without link local IPV6 address.
- radvd -d 5 -m stderr (for starting with many debug messages)

# Router Advertisement with Prefix Information option sniff



- `valid_lft` - how long this prefix is valid, in seconds.
- `preferred_lft` - how long this address can be in preferred state, in seconds.
- `ip -6 addr` (note: `ifconfig` does not show these time values)

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu  
1500 qlen 1000
```

```
inet6 2004:db8:0:1:240:95ff:fe30:b0a3/64 scope global dynamic
```

```
    valid_lft 279sec preferred_lft 99sec
```

```
inet6 fe80::240:95ff:fe30:b0a3/64 scope link
```

```
    valid_lft forever preferred_lft forever
```

- When **preferred time** is finished, this IPv6 address will stop communicating. (will not answer ping6, etc).
- When the **valid time** is over, the IPV6 address is removed.
- ***ipv6\_del\_addr()** in `net/ipv6/addrconf.c` is responsible for deleting non valid addresses (called from **addrconf\_verify()**)*
- This is useful for renumbering.
  - RFC 2894 - Router Renumbering for IPv6

- Radvd also send its mac address of itself as part of the options.
- This enable the receiving host to add/update it neighbour table accordingly with the mac address of the router:
- From *ndisc\_router\_discovery()*: (*net/ipv6/ndisc.c*)

...

```
lladdr=ndisc_opt_addr_data(ndopts.nd_opts_src_lladdr,skb->dev);
```

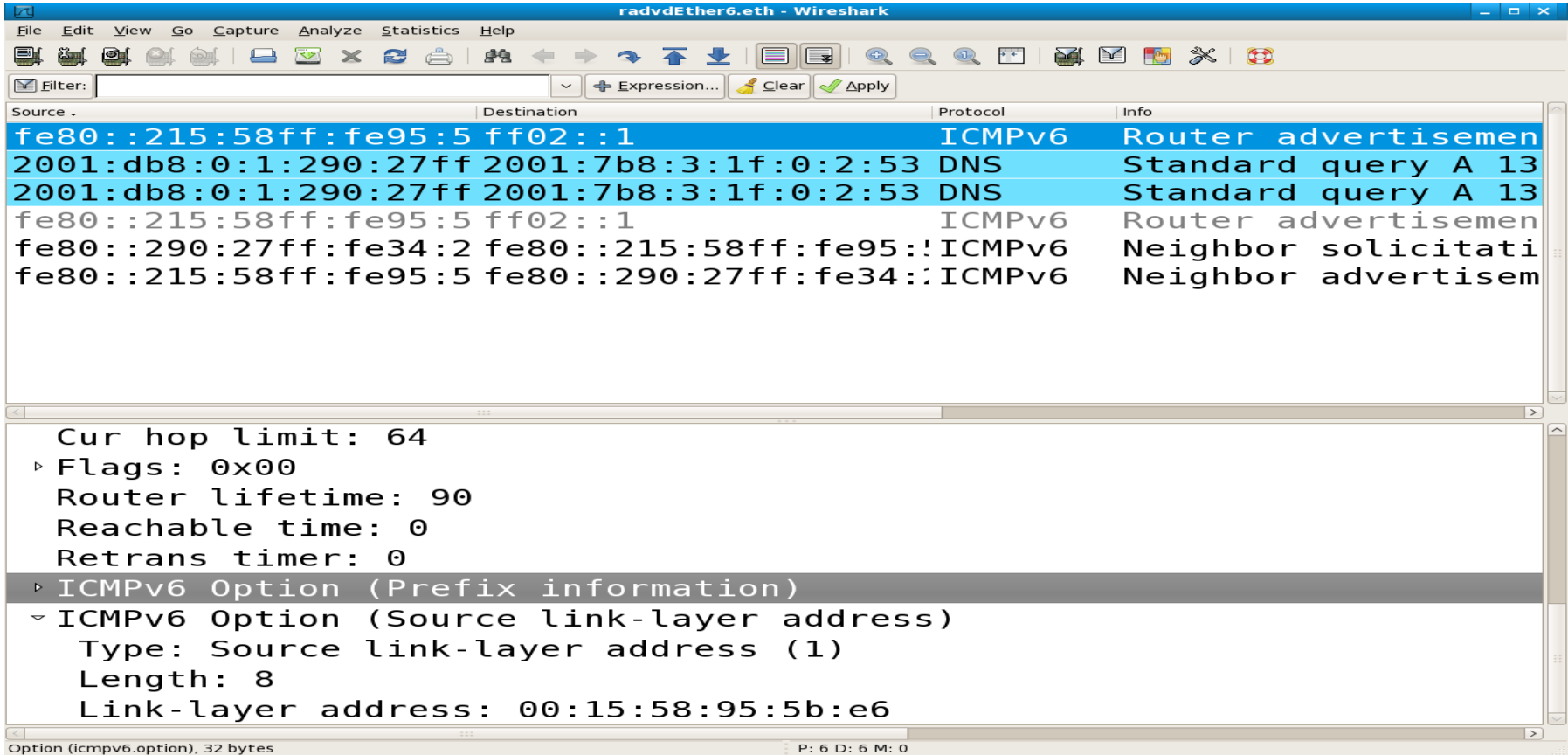
...

```
neigh_update(neigh, lladdr, ...)
```

...

- lladdr is the mac address of the router, passed in the options of Router Advertisement.

# Router Advertisement with Link Layer option sniff



The image shows a Wireshark packet capture window titled "radvdEther6.eth - Wireshark". The packet list pane displays several packets, with the first one selected. The packet details pane shows the structure of the selected ICMPv6 Router Advertisement packet.

Source	Destination	Protocol	Info
fe80::215:58ff:fe95:5	ff02::1	ICMPv6	Router advertisement
2001:db8:0:1:290:27ff	2001:7b8:3:1f:0:2:53	DNS	Standard query A 13
2001:db8:0:1:290:27ff	2001:7b8:3:1f:0:2:53	DNS	Standard query A 13
fe80::215:58ff:fe95:5	ff02::1	ICMPv6	Router advertisement
fe80::290:27ff:fe34:2	fe80::215:58ff:fe95::	ICMPv6	Neighbor solicitation
fe80::215:58ff:fe95:5	fe80::290:27ff:fe34::	ICMPv6	Neighbor advertisement

Cur hop limit: 64  
Flags: 0x00  
Router lifetime: 90  
Reachable time: 0  
Retrans timer: 0  
ICMPv6 Option (Prefix information)  
ICMPv6 Option (Source link-layer address)  
Type: Source link-layer address (1)  
Length: 8  
Link-layer address: 00:15:58:95:5b:e6

Option (icmpv6.option), 32 bytes P: 6 D: 6 M: 0

# radvd.conf and a default router

- Specifying AdvDefaultLifetime in radvd.conf will cause the host to add the radvd router as a default router.
  - Unless */proc/sys/net/ipv6/conf/eth0/accept\_ra\_defrtr* is 0.
- This default router has a limited lifetime. It will expire after the value specified for AdvDefaultLifetime.
- Maximum AdvDefaultLifetime value is 18.2 hours.

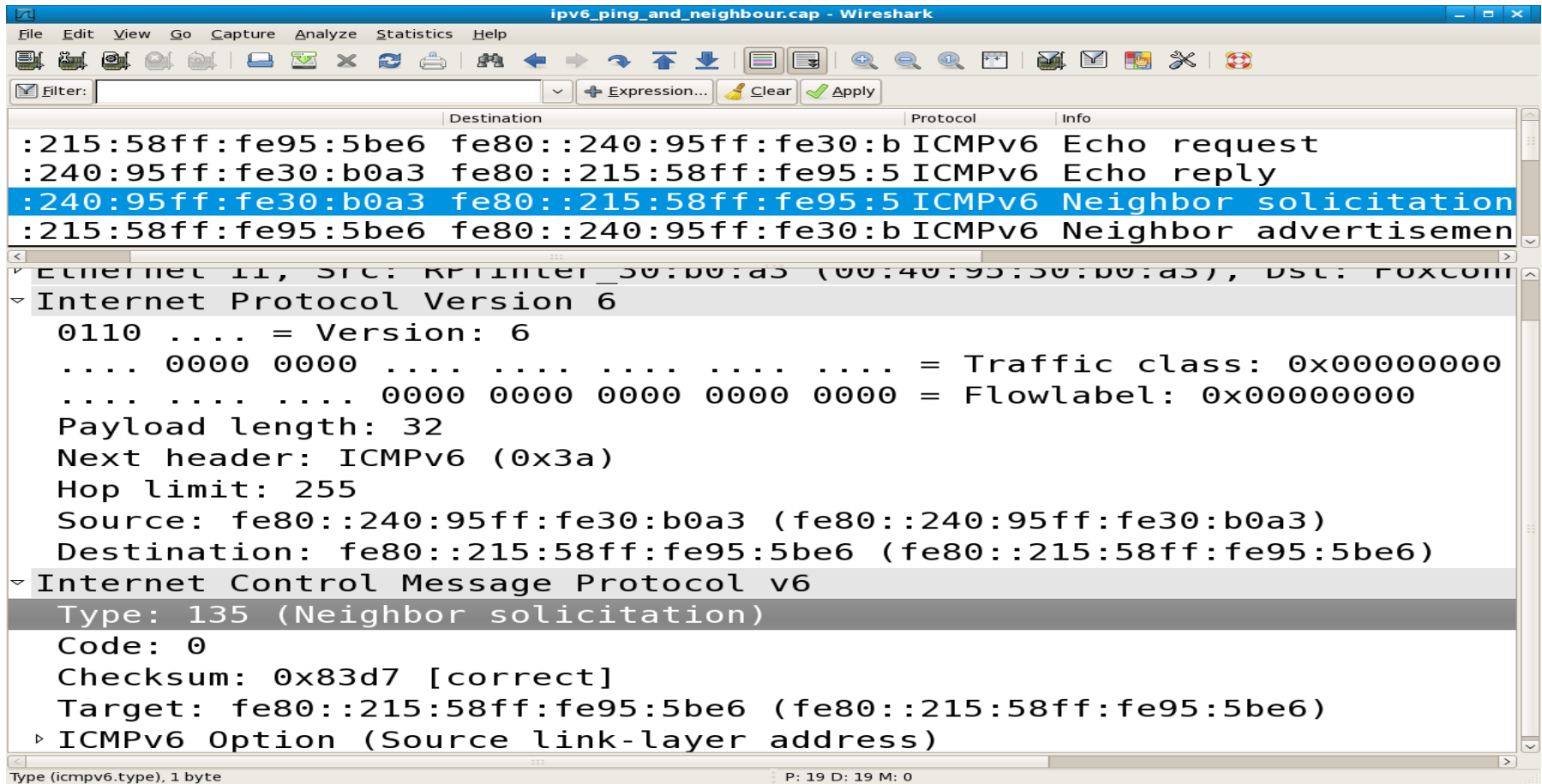
## Example (after setting AdvDefaultLifetime to 8000)

- *ip -6 route show default*
- default via fe80::215:58ff:fe95:5be6 dev eth0 proto kernel metric 1024 **expires 7996sec** mtu 1420 advmss 1360 hoplimit 64

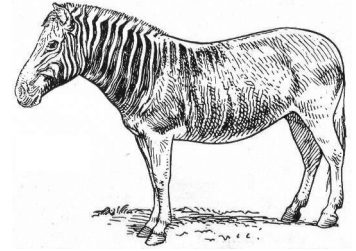
# radvd.conf and default router-cont.

- When we stop the radvd daemon this will send a Neighbour Advertisement with Router Lifetime as 0.
- This will cause the hosts which receive this message to delete the default router.
- Implemented by: *ip6\_del\_rt()* called from *ndisc\_router\_discovery()* in *net/ipv6/ndisc.c*
- You can also set MTU in radvd.conf
- This is not the MTU you see in ifconfig, but you see it in */proc/sys/net/ipv6/conf/eth0/mtu*.
- Radvd builtin util: **radvdump** prints out the contents of incoming router advertisements sent by radvd.

# IPV6 :Neighboring Solicitation sniff



# Quagga



- Quagga replaces Zebra
- <http://www.quagga.net/>
- Many routing protocols (BGP, OSPF, RIP, others)
- Supports IPV6
- Supports sending Router Advertisements.

interface eth0

ipv6 nd send-ra

ipv6 nd prefix-advertisement 2001:0db8:0005:0006::/64

# 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 */proc/sys/net/ipv6/conf/all/accept\_ra* to 0.
- Hosts can request Router Advertisements by sending a **Router Solicitation** message.

# 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)

# Autoconfiguration - contd.

- 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.

# Autoconfiguration - contd.

- 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*

- If a host boots when its cable is disconnected it will not get an IPV6 address.
- Connecting the cable will trigger an event (NETDEV\_CHANGE) in *addrconf\_notify()* and will result in sending Router Solicits (calling `ndisc_send_rs`) and eventually autoconfiguration will set an IPV6 address to the host.

# Optimistic DAD



- Do not wait till DAD is completed, and allow hosts to communicate with peers before DAD has finished successfully
- Target: to reduce latencies in the DAD process.
- The kernel should be build with: **CONFIG\_IPV6\_OPTIMISTIC\_DAD**.
- Very few apps need Optimistic DAD ; Usually the DAD process of DAD takes **less than 2 seconds**.
- RFC 4429 , Optimistic Duplicate Address Detection (DAD) for IPv6.

# IPv6 Fragmentation

- In IPv6, fragmentation is **not** done by routers (as in IPv4).
- The Minimum MTU is IPv6 is 1280.
- It is the responsibility of the host (**sender**) to fragment packets.
- Path MTU discovery is done by ICMPv6
  - ICMPv6\_PKT\_TOOBIG messages.
  - RFC 1981, Path MTU Discovery for IP version 6.

- Lookup in the IPV6 routing tables is done by *fib6\_lookup()*
  - *(net/ipv6/ip6\_fib.c)*
- The parameters for the lookup are the root of the table and the source and destination IPV6 address. (struct in6\_addr)
- The result of the lookup is saved in rt6\_info.
- rt6\_info is the parallel of rtable in IPV4.

// from *include/net/ip6\_fib.h*

```
struct rt6_info  
{  
    union {  
        struct dst_entry dst;  
    } u;
```

# IPV6 - contd

- Enable forwarding:
- `echo "1" > /proc/sys/net/ipv6/conf/all/forwarding`
- For Multicast Routing forwarding, there will be in the future:
  - *`/proc/sys/net/ipv6/conf/all/mc_forwarding`*

# IPv6 header – 40 bytes

- `include/linux/ipv6.h`

Version (4)	Priority/Traffic Class (4)	Flow Label (24)
Payload Length (16)	Next Header (8)	Hop Limit (8)
Source Address (128 bits=>16 bytes)		
Destination Address (128 bits=>16 bytes)		

# IPv6 header - contd.

- The IPv6 header length is **fixed**: 40 bytes.
- Therefore there is no header length field as in IPv4.
- In IPv4 the ip header is of **variable size**: 20 - 60 bytes; so we need the header length field. We can add to the base ip header by multiplications of 4 bytes up to 60 bytes.
  - Extension headers in ipv6.

# IPv6 header – Hop Limit

- The hop limit is by default 64.
  - `IPV6_DEFAULT_HOPLIMIT` is 64.
- This is the parallel of ttl field in ip header.
- *ip6\_forward()* checks the hop\_limit ; when it reaches 0 ,  
it sends an ICMP message:
  - (`ICMPV6_TIME_EXCEED`, `ICMPV6_EXC_HOPLIMIT...`)
- and the packet is dropped.
- **Note:** there is **NO checksum field** (as in IPV4).

# Extension Headers

- Hop-by-hop options (IPPROTO\_HOPOPTS)
- Routing packet header extension (IPPROTO\_ROUTING)
- Fragment packet header extension (IPPROTO\_FRAGMENT)
- ICMPV6 options (IPPROTO\_ICMPV6)
- No next header (IPPROTO\_NONE)
- Destination options (IPPROTO\_DSTOPTS)
- Mobility options (IPPROTO\_MH)
- Other Protocols (TCP,UDP,...)
  - See *include/linux/in6.h*
  - *There are some types of Next Headers which cannot have a Next Header field. For example, ICMPV6, TCP, UDP, no next header (IPPROTO\_NONE).*

# Extension Headers - contd.

- All these protocols are registered by *inet6\_add\_protocol()*
- If a host tries to parse an extension header which it does not recognize, then an ICMP error will be sent, notifying about a parameter problem. (type: *ICMPV6\_PARAMPROB*, code: *ICMPV6\_UNK\_NEXTHDR*) **and the packet will be dropped.**
- For example, in *ip6\_input\_finish()* (in *net/ipv6/ip6\_input.c*)
- *// next header does not specified a registered protocol*

...

```
icmpv6_send(skb, ICMPV6_PARAMPROB,  
             ICMPV6_UNK_NEXTHDR, nhoff, skb->dev);
```

# Extension Headers - contd

- **Router Alert** is a subtype of Hop-by-hop option, and it tells the router to process the packet besides forwarding it. It is used in multicasting.

# DHCPv6

- The DHCPv6 client runs on UDP port 546.
- The DHCPv6 server runs on UDP port 547.
- Projects:
- Dnsmasq:
- <http://klub.com.pl/dhcpv6/>
  - Linux and windows
- <https://fedorahosted.org/dhcpv6/>
- Maintained by David Cantrell (Red Hat)
- No mailing list...

- WIDE-DHCPv6
  - originally developed in KAME project,
  - for BSD and Linux
- DHCPV6 clients send SOLICIT requests in order to find DHCPV6 servers.
- Hosts send DHCPV6 solicit messages are sent on to the all-DHCPv6 multicast address (FF02::1:2).
- DHCPv6 Servers reply with advertisements.

# Socket API

- By default, the port space is not shared between IPV6 and IPV4.
- Simple example for creating TCP server with IPV6:

```
unsigned short port=9999;
```

```
struct sockaddr_in6 server;
```

```
struct sockaddr_in6 from;
```

```
sock = socket(AF_INET6, SOCK_STREAM, 0);
```

```
server.sin6_family = AF_INET6;
```

```
server.sin6_addr = in6addr_any;
```

# Socket API - contd.

```
server.sin6_port = htons(port);  
bind(sock,(struct sockaddr*)&server, sizeof(server));  
fromlen = sizeof(from);  
if (listen(sock,5)<0)  
printf("error listening\n");  
while (1)  
{  
newsock = (int*)malloc(sizeof(int));  
*newsock = accept(sock, (struct sockaddr *)&from,&fromlen);
```

# Socket API - contd.

- When trying to run a similar application in IPV4 on the same port simultaneously, you will get the following error:

*binding socket error*

*bind*

*: Address already in use*

- It will succeed if you set *IPV6\_V6ONLY* option in IPV6 socket:

*int on=1;*

*if (setsockopt(sock, IPPROTO\_IPV6, IPV6\_V6ONLY,  
(char \*)&on, sizeof(on)) == -1)*

# Receiving an IPv6 packet

- *ipv6\_rcv()* is the handler for IPv6 packet (*net/ipv6/ip6\_input.c*)
- Performs some sanity checks and then calls:
- *return NF\_HOOK(PF\_INET6, NF\_IP6\_PRE\_ROUTING, skb, dev, NULL, ip6\_rcv\_finish);*
- *ip6\_rcv\_finish()* performs a lookup in the routing subsystem by calling *ip6\_route\_input(skb)* in order to construct *skb->dst*.

# IPv6 – address types

- **Unicast**
  - The target is a single interface;
  - packet is delivered to a single interface.
- **Anycast** (new ! Does not exist in IPv4).
  - The target is a set of interfaces;
  - packet is delivered to a single interface.
- **Multicast**
  - The target is a set of interfaces;
  - packet is delivered to all the interfaces in this set.

- *ip6\_mc\_input()* currently only verifies that the packet is indeed for a multicast address of which the netdevice device is a member and calls *ip6\_input()*
- The Multicast Routing patch (CONFIG\_IPV6\_MROUTE) adds a call to *ip6\_mr\_input()*.
  - net/ipv6/ip6\_input.c
  - There is a user space daemon (pim6sd) which works in conjunction with multicast routing.
  - Configuration file: /usr/local/etc/pim6sd.conf
  - pim6sd is part of mcast-tools

# MLD

- **MLD - Multicast Listener Discovery**
  - also known as **Multicast Group Management**.
- MLD is similar to IGMP in IPV4 but used ICMPv6 messages
- MLD messages are sent via ICMPV6.
- MLDV2: RFC 3810 (added filtering abilities).
- The MLD is used by routers to discover the presence of Multicast listeners.
- MLDV2 is based on IGMPv3.

# MLD - contd.

- A host can belong to more than one multicast group.
- The Ethernet frame for a multicast address starts with 0x3333
- In IPV4, in multicast addresses, the first bit is 1 in the Ethernet frame (this is half of the MAC addresses!).
- The hop limit is always 1 in MLD messages so that a router will not forward them.
- `netstat -g -n` : show IPv6/IPv4 Group Memberships.
- `ip -6 maddr show`
- mcjoin: a util for joining an IPv6 Multicast Group
  - [http://www.benedikt-stockebrand.net/hacks\\_e.html](http://www.benedikt-stockebrand.net/hacks_e.html)

# MLD - contd.

- When a host boots, it first sends an MLDV2 message in ICMPV6. This is Type 143 message (ICMP code), and it is a Multicast Listener Discovery 2 **Report Message**. The report message tells routers and multicast-aware switches that the host wants to receive messages sent to the multicast address of the group it joined. This message has a hop limit of 1, so that it won't be forwarded outside. It is sent to **FF02::16** (A multicast address, which represents the all MLDv2-capable routers multicast group).

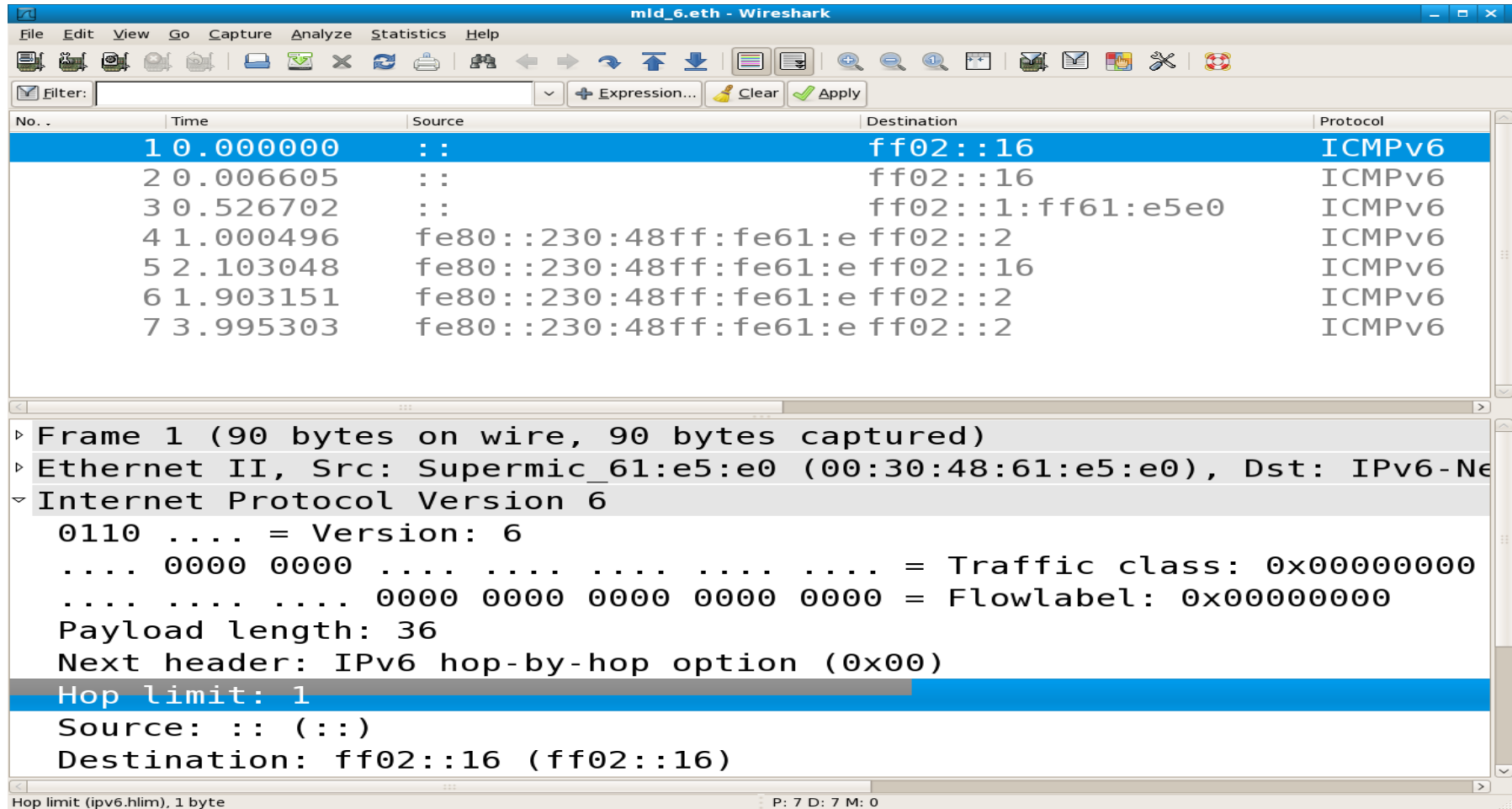
# MLD - contd.

- *addrconf\_add\_linklocal()* calls *ipv6\_add\_addr()* which eventually calls *igmp6\_group\_added()*, *mld\_newpack()* , setting type to **ICMPV6\_MLD2\_REPORT** and send an ICMP message.
- *(net/ipv6/mcast.c)*
- The source address of the MLD message can be a Link Local address or the unspecified address (::)
- When a host boots, it has a tentative address (until DAD is finished) and it sends an MLD report message to join the solicited node multicast group.
- *addrconf\_join\_solict()* calls *ipv6\_dev\_mc\_inc()* *net/ipv6/addrconf.c*

# MLD - contd.

- In this case , the source address of the MLD messages is the unspecified address (::)
- "Change to Exclude" in MLDV2 report.
- When a host leaves a group, it sends an MLDv2 ICMPV6\_MGM\_REDUCTION message
- *(igmp6\_leave\_group() in /net/ipv6/mcast.c)*
- This message is sent to the all routers multicast address (note the difference against REPORT, which is sent to FF02::16).

# MLD sniff



The image shows a Wireshark capture of MLD traffic. The main packet list shows seven packets, all of type ICMPv6. The first packet is highlighted. The packet details pane shows the structure of the first packet, which is an MLDv2 message (ICMPv6 Hop-by-hop option).

No.	Time	Source	Destination	Protocol
1	0.000000	::	ff02::16	ICMPv6
2	0.006605	::	ff02::16	ICMPv6
3	0.526702	::	ff02::1:ff61:e5e0	ICMPv6
4	1.000496	fe80::230:48ff:fe61:e	ff02::2	ICMPv6
5	2.103048	fe80::230:48ff:fe61:e	ff02::16	ICMPv6
6	1.903151	fe80::230:48ff:fe61:e	ff02::2	ICMPv6
7	3.995303	fe80::230:48ff:fe61:e	ff02::2	ICMPv6

Frame 1 (90 bytes on wire, 90 bytes captured)  
Ethernet II, Src: Supermic\_61:e5:e0 (00:30:48:61:e5:e0), Dst: IPv6-Ne  
Internet Protocol Version 6  
0110 .... = Version: 6  
.... 0000 0000 .... = Traffic class: 0x00000000  
.... 0000 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000  
Payload length: 36  
Next header: IPv6 hop-by-hop option (0x00)  
Hop limit: 1  
Source: :: (:::)  
Destination: ff02::16 (ff02::16)

Hop limit (ipv6.hlim), 1 byte

# MLD - contd.

- A router will recognize this MLD message by the hop-by-hop option in the extended header.
- *NEXTHDR\_HOP* in *include/net/ipv6.h*
- *RFC 2711 - IPv6 Router Alert Option.*

# MLD - contd.

- There are two types of messages in MLDV2:
  - Query (130) (ICMPV6\_MGM\_QUERY)
    - In icmp6 header, icmp6\_type = 130
  - Report (143) ICMPV6\_MLD2\_REPORT
    - In icmp6 header, icmp6\_type = 143
  - Reports are sent by MLDV2 with destination address of **FF02::16** (FF02:0:0:0:0:0:0:16)

# Network Namespaces

- Two types of virtualization in the Linux Kernel
- OS virtualization.
- process/container virtualization. (Like solaris zones).
- OS virtualization:
  - Xen
  - Kvm (hardware virtualization)
  - Lguest (Only 32 bit; there is a RedHat trial to write 64 bit version).

# Network Namespaces - contd.

- OpenVZ project (<http://openvz.org/>).
  - Currently only for Linux (the FreeBSD port was dropped).
  - There is a serious effort to integrate it into mainline Linux kernel.
- Many patches recently to netdev kernel mailing list.
- struct net (*include/net/net\_namespace.h*)
- Adding support for namespaces in IPV4 is finished.
  - Currently work is being done on adding support for namespaces in IPV6.

# Packet Generator

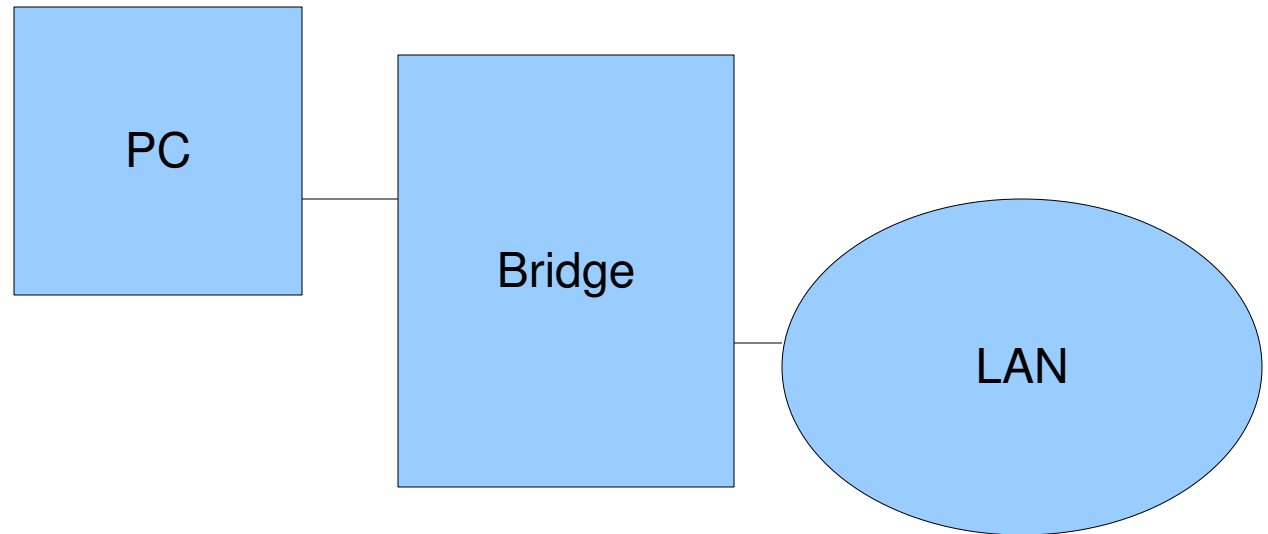
- Pktgen kernel module (Robert ollson)
  - Works also with IPV6.

# Bridging Subsystem

- You can define a bridge and add NICs to it (“enslaving ports”) using `brctl` (from `bridge-utils`).
- You can have up to 1024 for every bridge device (`BR_MAX_PORTS`) .
- Example:
- `brctl addbr mybr`
- `brctl addif mybr eth0` #adding interface to a bridge
- `brctl show`

# Simple example

- In this simple example, you can connect a PC to a bridge without any configuration on the PC.



# Bridging Subsystem-contd

- There are devices which you cannot add to a bridge (by `addif`); like another bridge or a loopback device or a tunnel device or any other device which has no HW address.
- You can add a tap device (but not a tun device) (?)
  -
- 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 we receive a frame, `netif_receive_skb()` calls `handle_bridge()`.

# Bridging Subsystem-contd

- *br\_handle\_frame()* is invoked (*net/bridge/br\_input.c*)
- `NF_HOOK(PF_BRIDGE, NF_BR_PRE_ROUTING, skb, skb->dev, NULL, br_handle_frame_finish);`
- *br\_handle\_frame\_finish()* checks the MAC destination of the packet.
  - If the packet is for the local machine, we do not forward the packet but call *br\_pass\_frame\_up()*.
  - *br\_pass\_frame\_up()* calls:
    - `NF_HOOK(PF_BRIDGE, NF_BR_LOCAL_IN, skb, indev, NULL, netif_receive_skb);`

# Bridging Subsystem-contd

- If the packet is for the local machine we forward the packet:
  - by *br\_forward()* if the address is in the forwarding DB.
  - *br\_flood\_forward()* if the address is in not the forwarding DB.

# Bridging Subsystem-contd

- 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 ebttables mechanism is the L2 parallel of L3 Netfilter.

# Bridging Subsystem-contd

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

# Tips for hacking

- Documentation/networking/ip-sysctl.txt: networking kernel tunables
- 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).
- A BASH script for getting MAC address from IP address: (ipToHex.sh)

```
#!/bin/sh
```

```
IP_ADDR=$1
```

```
for I in $(echo ${IP_ADDR}| sed -e "s\./ /g"); do
```

```
    printf '%02X' $I
```

```
done
```

```
echo
```

```
usage example: ./ipToHex.sh 192.168.0.1 => C0A80001
```

# Tips for hacking - Contd.

- 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 `tracert` (see `man tracert`).
  - `Tracert` is from `iputils`.

## Tips for hacking - Contd.

- **inet\_addr\_type()** method: returns the address type; the input to this method is the IP address. The return value can be RTN\_LOCAL, RTN\_UNICAST, RTN\_BROADCAST, RTN\_MULTICAST etc.  
See: `net/ipv4/fib_frontend.c`

## Tips for hacking - Contd.

- In case you want to send a packet from a user space application through a specified device without altering any routing tables:

```
struct ifreq interface;
```

```
strncpy(interface.ifr_ifrn.ifrn_name, "eth1", IFNAMSIZ);
```

```
if (setsockopt(s, SOL_SOCKET, SO_BINDTODEVICE, (char  
    *)&interface, sizeof(interface)) < 0)
```

```
{
```

```
    printf("error setting SO_BINDTODEVICE");
```

```
    exit(1);
```

```
}
```

# Tips for hacking - Contd.

- 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. */  
};
```

# Tips for hacking - Contd.

- NIPQUAD() : macro for printing hex addresses
- Printing mac address (from net\_device):

```
printk("sk_buff->dev =%02x:%02x:%02x:%02x:%02x:%02x\n",  
((skb)->dev)->dev_addr[0], ((skb)->dev)->dev_addr[1],  
((skb)->dev)->dev_addr[2],((skb)->dev)->dev_addr[3],  
((skb)->dev)->dev_addr[4], ((skb)->dev)->dev_addr[5]);
```

- Printing IP address (primary\_key) of a neighbour (in hex format):

```
printk("neigh->primary_key =%02x.%02x.%02x.%02x\n",  
neigh->primary_key[0], neigh->primary_key[1],  
neigh->primary_key[2],neigh->primary_key[3]);
```

# Tips for hacking - Contd.

- Or:

```
printk("***neigh->primary_key= %u.%u.%u.%u\n",  
        NIPQUAD(*(u32*)neigh->primary_key));
```

- CONFIG\_NET\_DMA is for TCP/IP offload.
- When you encounter: xfrm / CONFIG\_XFRM this has to do with IPSEC. (transformers).

# Tips for hacking - Contd.

- Showing arp statistics by:
- ***cat /proc/net/stat/arp\_cache***

entries allocs destroys hash\_grows lookups hits res\_failed  
rcv\_probes\_mcast rcv\_probes\_ucast periodic\_gc\_runs  
forced\_gc\_runs

**periodic\_gc\_runs**: statistics of how many times the  
*neigh\_periodic\_timer()* is called.

# Links and more info

- IPV6 howto (Peter Bieringer) :  
<http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Linux+IPv6-HOWTO.pdf>
- USAGI Project - Linux IPv6 Development Project
  - <http://www.linux-ipv6.org/>
- **Porting applications to IPv6 HowTo BY Eva M. Castro:**
  - <http://gsyc.es/~eva/IPv6-web/ipv6.html>
  -
- RFC 3493: Basic Socket Interface Extensions for IPv6.
- RFC 3542: Advanced Sockets Application Program Interface (API) for IPv6.

# Links and more info

- **Books:**
- **IPv6 Essentials, Second Edition (O'Reilly)**
  - A book By Silvia Hagen
  - Second Edition May 2006
  - Pages: 436
  - ISBN 10: 0-596-10058-2 | ISBN 13: 9780596100582

# Links and more info

- **IPv6 in Practice: A Unixer's Guide to the Next Generation Internet**
- by Benedikt Stockebrand (Author) ; Springer; 1 edition, 2006.
- Talks about implementation of IPv6 in Linux, Solaris, BSD.
- [http://www.benedikt-stockebrand.net/books\\_e.html](http://www.benedikt-stockebrand.net/books_e.html)

# Links and more info

- 1) **IPv6 Advanced Protocols Implementation (2007)**
- 2) **IPv6 Core Protocols Implementation (2006)**

Both books were written by Qing Li, Tatuya Jinmei and Keiichi Shima

- published by Morgan Kaufmann Series in Networking.
  - Both books discuss the **Kame** implementation of IPV6. (in BSD).

# Links and more info

- IPv6 Information Page!
  - <http://www.ipv6.org/>
- What's up in the Linux IPv6 Stack
- Lecture slides by Hideaki YOSHIFUJI from lca2008.
  - Keio University
  - USAGI/WIDE Project
- <http://mirror.linux.org.au/pub/linux.conf.au/2008/slides/131-200801-LCA2008-LinuxIPv6.pdf>
- Html: <http://www.linux-ipv6.org/materials/200801-LCA2008/>

# Links and more info

Linux Network Stack Walkthrough (2.4.20):

[http://gicl.cs.drexel.edu/people/sevy/network/Linux\\_network\\_stack\\_wa](http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_wa)

Understanding the Linux Kernel, Second Edition

By Daniel P. Bovet, Marco Cesati

Second Edition December 2002

chapter 18: networking.

- Understanding Linux Network Internals, Christian benvenuti

Oreilly , First Edition.

# Links and more info

Linux Device Driver, by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

Third Edition February 2005.

- Chapter 17, Network Drivers

Linux networking: (a lot of docs about specific networking topics)

- [http://www.linux-foundation.org/en/Net:Main\\_Page](http://www.linux-foundation.org/en/Net:Main_Page)
- 

netdev mailing list: <http://www.spinics.net/lists/netdev/>

# Links and more info

Removal of multipath routing cache from kernel code:

<http://lists.openwall.net/netdev/2007/03/12/76>

<http://lwn.net/Articles/241465/>

Linux Advanced Routing & Traffic Control :

<http://lartc.org/>

ebtables – a filtering tool for a bridging:

<http://ebtables.sourceforge.net/>

# Links and more info

**Writing Network Device Driver for Linux:** (article)

- <http://app.linux.org.mt/article/writing-netdrivers?locale=en>

# Links and more info

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>
- Next one: Linux Conf Australia, January 2008, Melbourne
- David S. Miller, James Morris , Rusty Russell , Jamal Hadi Salim , Stephen Hemminger , Harald Welte, Hideaki YOSHIFUJI, Herbert Xu , Thomas Graf , Robert Olsson , Arnaldo Carvalho de Melo and others

# Links and more info

## **Policy Routing With Linux** - Online Book Edition

- by Matthew G. Marsh (Sams).
- <http://www.policyrouting.org/PolicyRoutingBook/>

THRASH - A dynamic LC-trie and hash data structure:

Robert Olsson Stefan Nilsson, August 2006

<http://www.csc.kth.se/~snilsson/public/papers/trash/trash.pdf>

IPSec howto:

<http://www.ipsec-howto.org/t1.html>

## Links and more info

Openswan: Building and Integrating Virtual Private Networks ,  
by Paul Wouters, Ken Bantoft

<http://www.packtpub.com/book/openswan/mid/061205jqdnh2by>

publisher: Packt Publishing.

a book including chapters about LVS:

“The Linux Enterprise Cluster- Build a Highly Available Cluster  
with Commodity Hardware and Free Software”, By Karl  
Kopper.

<http://www.nostarch.com/frameset.php?startat=cluster>

<http://www.vyatta.com> - Open-Source Networking

# Links and more info

Address Resolution Protocol (ARP)

- <http://linux-ip.net/html/ether-arp.html>

ARPWatch – a tool for monitor incoming ARP traffic.

Lawrence Berkeley National Laboratory -

<ftp://ftp.ee.lbl.gov/arpwatch.tar.gz>.

arptables:

<http://ebtables.sourceforge.net/download.html>

TCP/IP Illustrated, Volume 1: The Protocols

By W. Richard Stevens

<http://www.informit.com/store/product.aspx?isbn=0201633469>

# Links and more info

Unix Network Programming, Volume 1: The Sockets    Networking  
API (3rd Edition) (Addison-Wesley Professional Computing Series)  
(Hardcover)

by W. Richard Stevens (Author), Bill Fenner (Author), Andrew M.  
Rudoff (Author)

Linux Ethernet Bridging mailing list:

<http://www.spinics.net/lists/linux-ethernet-bridging/>

# Questions

- Questions ?
- Thank You !

# Linux Wireless - Linux Kernel Networking (4)- advanced topics

Rami Rosen  
[ramirose@gmail.com](mailto:ramirose@gmail.com)  
Haifux, March 2009  
[www.haifux.org](http://www.haifux.org)



# Linux Kernel Networking (4)- advanced topics

- Note:
- This lecture is a sequel to the following 3 lectures I gave:

## 1) Linux Kernel Networking lecture

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

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

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

# Linux Kernel Networking (4)- advanced topics

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

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

# Contents:

- General.
  - IEEE80211 specs.
  - SoftMAC and FullMAC; mac80211.
- Modes: (802.11 Topologies)
  - Infrastructure mode.
    - Association.
    - Scanning.
    - Hostapd
    - Power save in Infrastructure mode.
  - IBSS (Ad Hoc mode).
  - Mesh mode (80211s).

- 802.11 Physical Modes.
- Appendix: mac80211- implementation details.
- Tips.
- Glossary.
- Links.
- Images
  - Beacon filter – Wireshark sniff.
  - edimax router user manual page (BR-6504N).

- **Note:** we will not deal with security/encryption, regulation, fragmentation in the linux wireless stack and not deal with tools (NetworkManager, kwifimanager,etc). and not with billing (Radius, etc).
- You might find help on these topics in two Haifux lectures:
- Wireless management (WiFi (802.11) in GNU/Linux by Ohad Lutzky):
  - <http://www.haifux.org/lectures/138/>
- Wireless security (Firewall Piercing, by Alon Altman):
  - <http://www.haifux.org/lectures/124/>
- Note: We will not delve into hardware features.

# General

- Wireless networks market grows constantly
- Two items from recent month newspaper: (ynet.co.il)
  - Over 12,000 wireless room hotels in Israel.
  - Over 50,000 wireless networks in Europe.
- In the late nineties there were discussions in IEEE committees regarding the 802.11 protocol.
- **1999** : The first spec (about 500 pages).
  - (see no 1 in the list of links below).
- **2007**: A second spec (1232 pages) ; and there were some amendments since then.

# SoftMAC and FullMAC

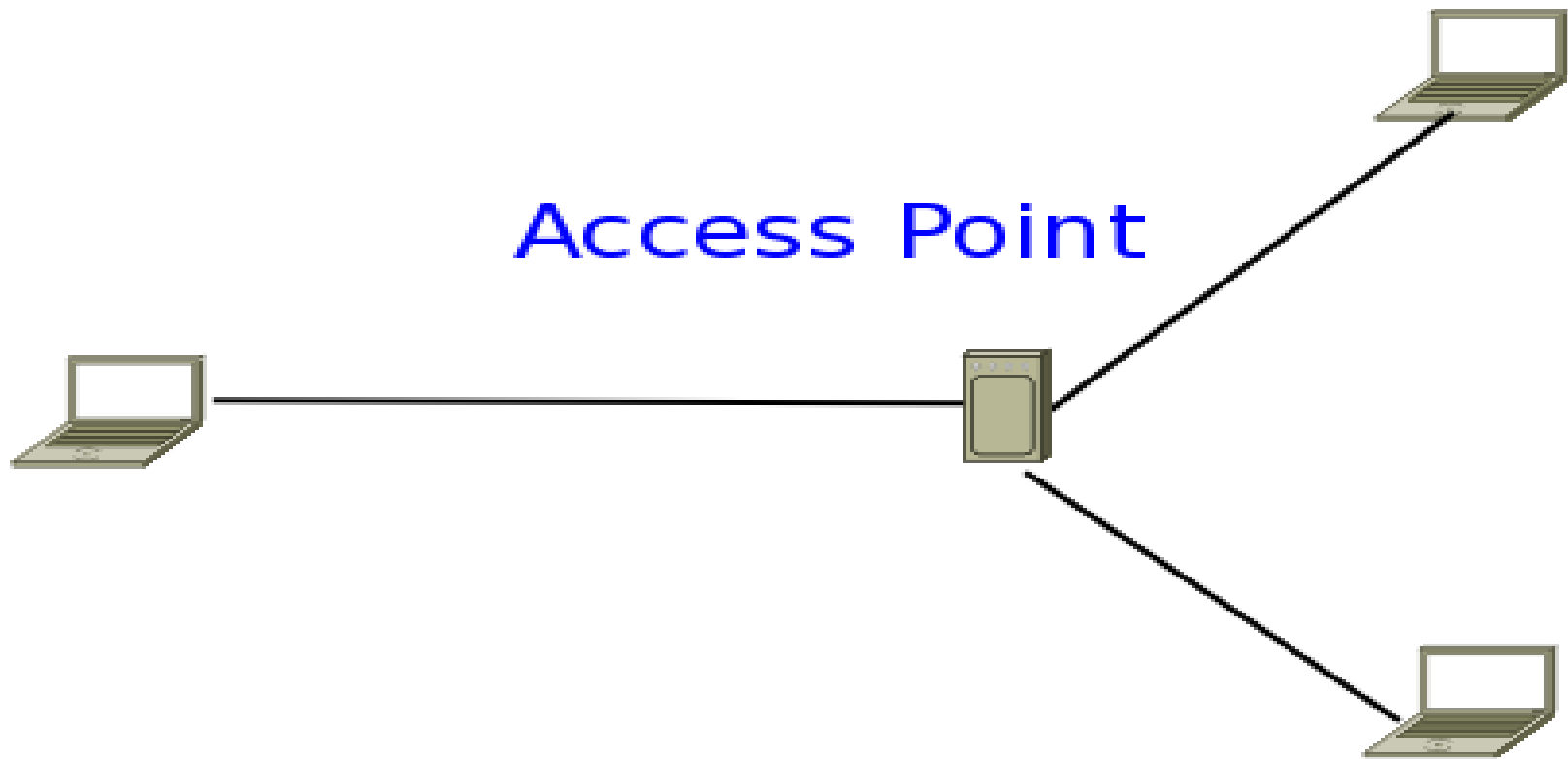
- In 2000-2001, the market became abound with laptops with wireless nics.
- It was important to produce wireless driver and wireless stack Linux solutions in time.
- The goal was then, as Jeff Garzik (the previous wireless Maintainer) put it: "They just want their hardware to work...".
- **mac80211** - new Linux softmac layer.
  - formerly called d80211 of Devicescape)
- Current mac80211 maintainer: Johannes Berg from sipsolutions.

- Mac80211 merged into Kernel mainstream (upstream) starting 2.6.22, July 2007.
- Drivers were adjusted to use mac80211 afterwards.
- Devicescap is a wireless networking company.
  - <http://devicescap.com/pub/home.do>
- Location in the kernel tree: net/mac80211.
- A kernel module named mac80211.ko.

- Most wireless drivers were ported to use mac80211.
  - There is a little number of exceptions though.
- Libertas (Marvell) does **not** work with mac80211.
- libertas\_tf (Marvell) uses thin firmware ; so it **does use** mac80211.
  - libertas\_tf supports Access Point and Mesh Point.
  - Both are in OLPC project.
- When starting development of a new driver, most chances are that it will use mac80211 API.

Modes: Infrastructure BSS

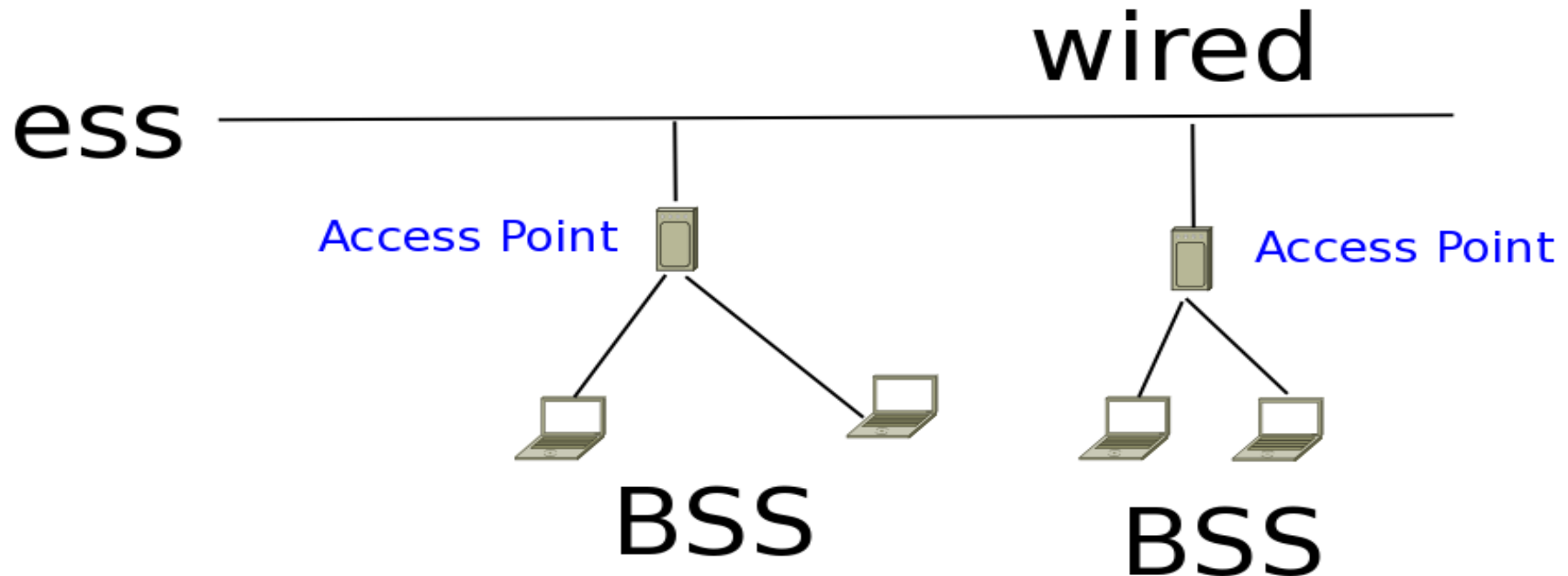
# Infrastructure BSS



# Classic ESS (Extended Service Set)

ESS = two or more BSSs.

Infrastructure BSS



- What is an Access Point ?
- Edimax MIMO nMax BR-6504n Router



- Linksys WRT54GL 54Mbps Route



- **NOTE: Infrastructure BSS != IBSS**
  - **IBSS = Independent BSS. (Ad-Hoc mode)**
- **Access Point:** A wireless device acting in **master mode** with some hw enhancements and a management software (like **hostapd**).
  - A wireless device in master mode cannot scan (as opposed to other modes).
    - Also a wireless device in monitor mode cannot scan.
- Master Mode is one of 7 modes in which a wireless card can be configured.

- All stations must **authenticate** and **associate** with the Access Point prior to communicating.
- Stations sometimes perform **scanning** prior to **authentication** and **association** in order to get details about the Access Point (like mac address, essid, and more).

# Scanning

- Scanning can be:
  - **Active** (send broadcast Probe request) scanning.
  - **Passive** (Listening for beacons) scanning.
  - Some drivers support passive scanning. ( see the IEEE80211\_CHAN\_PASSIVE\_SCAN flag).
  - Passive scanning is needed in some higher 802.11A frequency bands,as you're not allowed to transmit anything at all until you've heard an AP beacon.
- scanning with "*iwlist wlan0 scan*" is in fact sending an IOCTL (SIOCSIWSCAN).

# Scanning-contd.

- It is handled by *ieee80211\_ioctl\_siwscan()*.
- This is part of the Wireless-Extensions mechanism. (aka WE).
- Also other operations like setting the mode to Ad-Hoc or Managed can be done via IOCTLs.
- The Wireless Extensions module; see: `net/mac80211/wext.c`
- Eventually, the scanning starts by calling *ieee80211\_sta\_start\_scan()* method ,in `net/mac80211/mlme.c`.
- **MLME** = MAC Layer Management Entity.

# Scanning-contd.

- Active Scanning is performed by sending Probe Requests on **all the channels** which are supported by the station.
  - One station in each BSS will respond to a Probe Request.
  - That station is the one **which transmitted the last beacon in that BSS.**
    - In **infrastructure BSS**, this stations is the Access Point.
    - Simply because there are no other stations in BSS which send beacons.
    - In **IBSS**, the station which sent the last beacon can change during time.

# Scanning-contd.

- You can also sometimes scan for a specific BSS:
  - *iwlist wlan1 scan essid homeNet.*
  - Also in this case, a broadcast is sent.
  - (sometimes, this will return homeNet1 also and homeNet2).

# Example of scan results

## iwlist wlan2 scan

wlan2 Scan completed :

Cell 01 - Address: 00:16:E3:F0:FB:39

ESSID:"SIEMENS-F0FB39"

Mode:Master

Channel:6

Frequency:2.437 GHz (Channel 6)

Quality=5/100 Signal level:25/100

Encryption key:on

IE: Unknown: 000E5349454D454E532D463046423339

IE: Unknown: 010882848B962430486C

IE: Unknown: 030106

IE: Unknown: 2A0100

IE: Unknown: 32040C121860

IE: Unknown: DD06001018020000

Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; 18 Mb/s

24 Mb/s; 36 Mb/s; 54 Mb/s; 6 Mb/s; 9 Mb/s

12 Mb/s; 48 Mb/s

Extra:tsf=00000063cbf32479

Extra: Last beacon: 470ms ago

Cell 02 - Address: 00:13:46:73:D4:F1

ESSID:"D-Link"

Mode:Master

Channel:6

Frequency:2.437 GHz (Channel 6)

# Authentication

- Open-system authentication (WLAN\_AUTH\_OPEN) is the only mandatory authentication method required by 802.11.
- The AP does not check the identity of the station.
- Authentication Algorithm Identification = 0.
- Authentication frames are management frames.

# Association

- 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).

# Association-contd.

- Trying this:
  - *iwconfig wlan0 essid AP1 ap macAddress1*
  - *iwconfig wlan0 essid AP2 ap macAddress2*
- Will cause first associating to AP1, and then disassociating from AP1 and associating to AP2.
- AP will not receive any data frames from a station before it is associated with the AP.

# Association-contd.

- 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. The Access Point also define a Listen Interval.
- When a station associates to an Access Point, it gets an ASSOCIATION ID (**AID**) in the range 1-2007.

# Association-contd.

- Trying unsuccessfully to associate more than 3 times results with this message in the kernel log:
- “apDeviceName: association with AP apMacAddress timed out” and the state is changed to **IEEE80211\_STA\_MLME\_DISABLED**.
- Also if does not match security requirement, will return **IEEE80211\_STA\_MLME\_DISABLED**.

# 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.
  - Example of a very simple hostapd.conf file:

```
interface=wlan0  
driver=nl80211  
hw_mode=g  
channel=1  
ssid=homeNet
```

# Hostapd-cont.

- Launching hostapd:
  - *./hostapd hostapd.conf*
  - *(add -dd for getting more verbose debug messages)*
- 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-cont.

- Hostapd uses the nl80211 API (netlink socket based , as opposed to ioctl based).

# Hostapd-cont.

- The hostapd starts the device in monitor mode:

*drv->monitor\_ifidx =*

*nl80211\_create\_iface(drv, buf, NL80211\_IFTYPE\_MONITOR, NULL);*

The hostapd opens a raw socket with this device:

*drv->monitor\_sock = socket(PF\_PACKET, SOCK\_RAW, htons(ETH\_P\_ALL));*  
*(hostapd/driver\_nl80211.c)*

The packets which arrive at this socket are handled by the AP.

- Receiving in monitor mode means that a special header (RADIOTAP) is added to the received packet.
- The hostapd changes management and control packets.
- The packet is sent by the `sendmsg()` system call:
- *sendmsg(drv->monitor\_sock, &msg, flags);*

# Hostapd-cont.

- This means sending directly from the raw socket (PF\_PACKET) and putting on the transmit queue (by *dev\_queue\_xmit()*), without going through the 80211 stack and without the driver).
- When the packet is transmitted, an “INJECTED” flag is added. This tells the other side, which will receive the packet, to remove the radiotap header. (IEEE80211\_TX\_CTL\_INJECTED)

# Hostapd-cont.

- Hostapd manages:
  - Association/Disassociation requests.
  - Authentication/deauthentication requests.
- The Hostapd keeps an array of stations; When an association request of a new station arrives at the AP, a new station is added to this array.

# Hostapd-cont.

- There are three types of IEEE80211 packets:
- The type and subtype of the packet are represented by the **frame control** field in the 802.11 header.
  - **Management** (IEEE80211\_FTYPE\_MGMT)
  - Each management frame contains information elements (IEs). For example, beacons has the ssid (network name) ,ESS/IBSS bits (10=AP,01=IBSS), and more.
  - (WLAN\_CAPABILITY\_ESS/WLAN\_CAPABILITY\_IBSS in ieee80211.h.)
  - There are 47 types of information elements (IEs) in current implementation
  - All in /include/linux/ieee80211.h.

- Association and Authentication are management packets.
  - Beacons are also management frames.
  - IEEE80211\_STYPE\_BEACON

# Hostapd-cont.

- **Control** (**IEEE80211\_FTYPE\_CTL**)
  - For example, PSPELL  
IEEE80211\_STYPE\_PSPOLL
    - Also ACK, RTS/CTS.
- **Data** (**IEEE80211\_FTYPE\_DATA**)
  - See: [include/linux/ieee80211.h](#)
  - The hostapd daemon sends special management packets called **beacons** (Access Points send usually 10 beacons in a second; this can be configured (see the router manual page at the bottom)).
- The area in which these beacons appear define the basic service area.

From /net/mac80211/rx.c (with remarks)

\* IEEE 802.11 address fields:

ToDS	FromDS	Addr1	Addr2	Addr3	Addr4	
0	0	DA	SA	BSSID	n/a	AdHoc
0	1	DA	BSSID	SA	n/a	Infra (From AP)
1	0	BSSID	SA	DA	n/a	To AP (Infra)
1	1	RA	TA	DA	SA	WDS (Bridge )

# My laptop as an access point

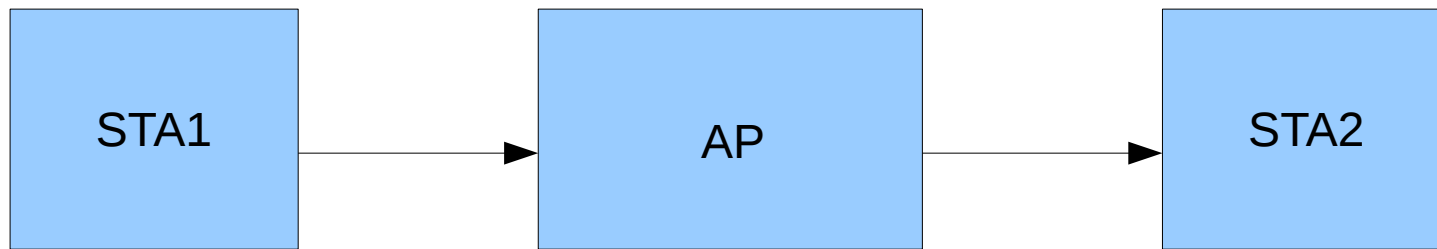
- My laptop as an access point: There is an Israeli Start Up company which develops free access point Windows sw which enables your laptop to be an access point.
- <http://www.bzeek.com/static/index.html>
- Currently it is for Intel PRO/Wireless 3945.
- In the future: Intel PRO/Wireless 4965.

# Power Save in Infrastructure Mode

- Power Save is a hot subject.
- Intel linux Power Save site:
  - <http://www.lesswatts.org/>
  - PowerTOP util:
    - PowerTOP is a tool that helps you find which software is using the most power.

# Power Save in Infrastructure Mode- cont

- Usual case (Infrastructure BSS).



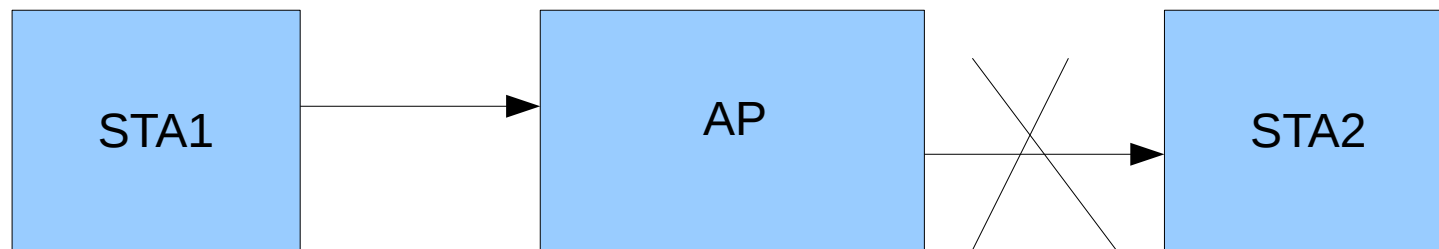
- Mobile devices are usually battery powered most of the time.
- A station may be in one of two different modes:
  - Awake (fully powered)
  - Asleep (also termed “dozed” in the specs)
- Access points never enters power save mode and does not transmit Null packets.
- In power save mode, the station is not able to transmit or receive and consumes very low power.

- Until recently, power management worked only with devices which handled power save in firmware.
- From time to time, a station enters **power save** mode.
- This is done by:
  - firmware, or
  - by using mac80211 API
    - Dynamic power management patches that were recently sent by Kalle Valo (Nokia).

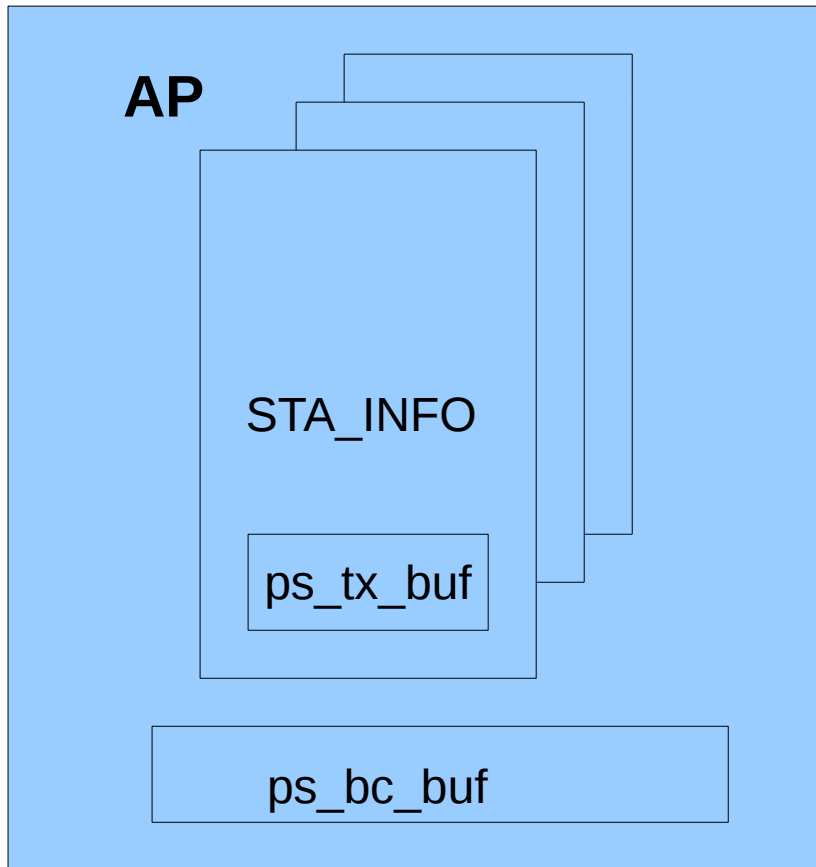
- How do we initiate power save?
- *iwconfig wlan0 power timeout 5*
  - *Sets the timeout to 5 seconds.*
- *Note: this can be done only with the beta version of Wireless Tools (version 30-pre7 (beta) ):*
- [http://www.hpl.hp.com/personal/Jean\\_Tourrilhes/Linux/Tools.html](http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html)

- In case the firmware has support for power save, drivers can disable this feature by setting **IEEE80211\_HW\_NO\_STACK\_DYNAMIC\_PS** flag in the driver configuration.
- The Access Point is notified about it by a **null frame** which is sent from the client (which calls *ieee80211\_send\_nullfunc()* ). The **PM** bit is set in this packet (Power Management).

- When STA2 is in power saving mode:
- AP has two buffers: (a doubly linked list of sk\_buff structures, sk\_buff\_head).
  - For unicast frames (ps\_tx\_buf in sta; one queue for each station).
  - For multicast/broadcast frames. (ps\_bc\_buf ,one for AP).



- Each AP has an array of its associated stations inside (sta\_info objects). Each one has **ps\_tx\_buf** queue inside, (for unicasts), and **ps\_bc\_buf** (for multicast/broadcasts)



- The size of `ps_tx_buf` and of `ps_bc_buf` is 128 packets
- `#define STA_MAX_TX_BUFFER 128` in `net/mac80211/sta_info.h`
- `#define AP_MAX_BC_BUFFER 128` in `net/mac80211/ieee80211_i.h`
- Adding to the queue: done by `skb_queue_tail()`.
- There is however, a common counter (`total_ps_buffered`) which sums both buffered unicasts and multicasts.
- 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.**

- When buffering in AP, every packet (unicast and multicast) is saved in the corresponding key.
- The only exception is when strict ordering between unicast and multicast is enforced. This is a service which MAC layer supply. However, it is rarely in use.

- From net/mac80211/tx.c:

```
ieee80211_tx_h_multicast_ps_buf() {
```

```
...
```

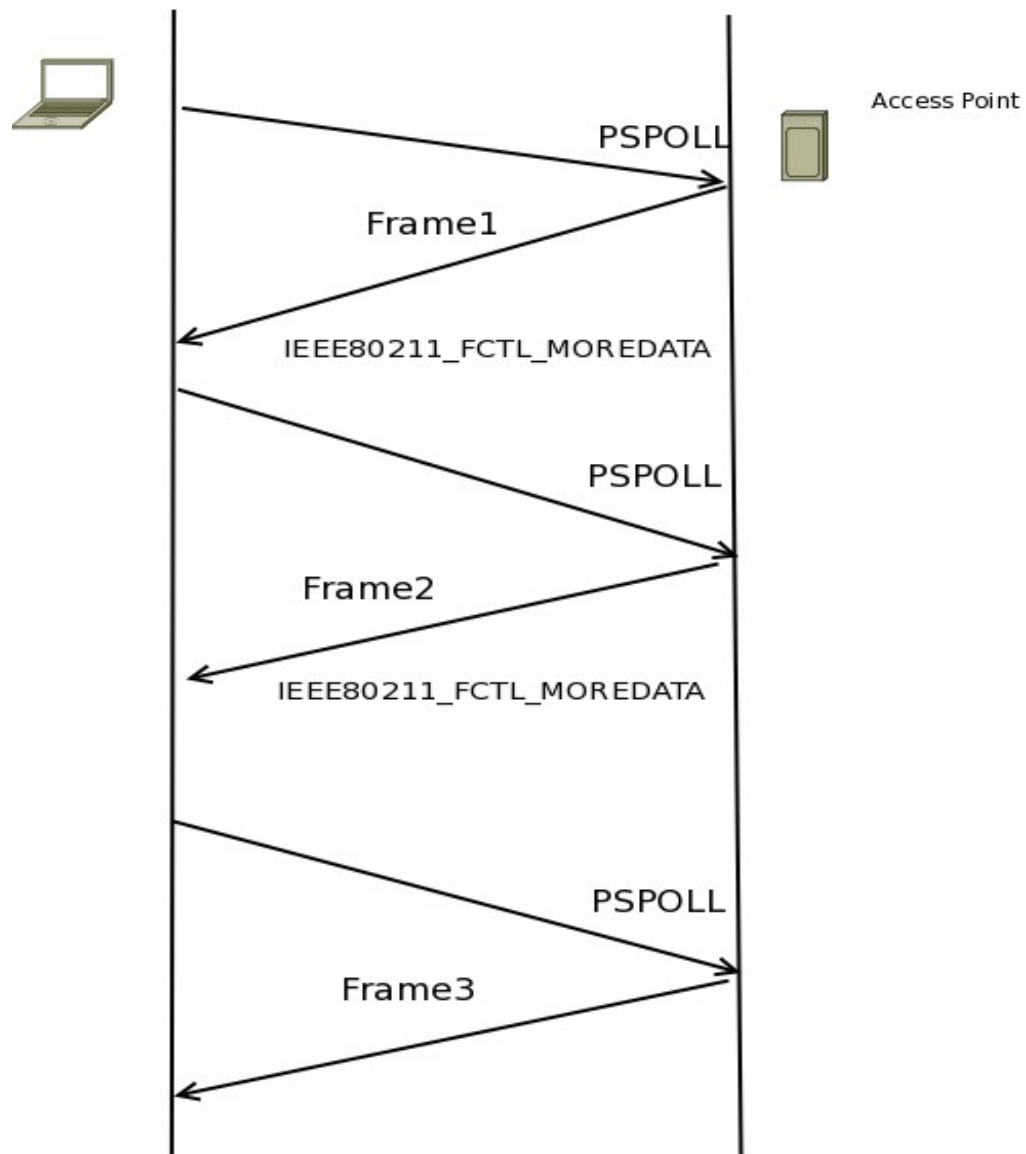
```
/* no buffering for ordered frames */
```

```
if (ieee80211_has_order(hdr->frame_control))
```

```
    return TX_CONTINUE;
```

- The AP sends a **TIM** (Traffic Indication Map) with each beacon.
- Beacons are sent periodically from the AP.
- $TIM[i]=1 \Rightarrow$  The AP has buffered traffic for a station with Association ID=i.
  - In fact, a partial virtual bitmap is sent – which is a smaller data structure in most cases.
- The STA sends a **PS-POLL** packet (Power Saving Poll) to tell the AP that it is awake.
- AP sends the buffered frame.

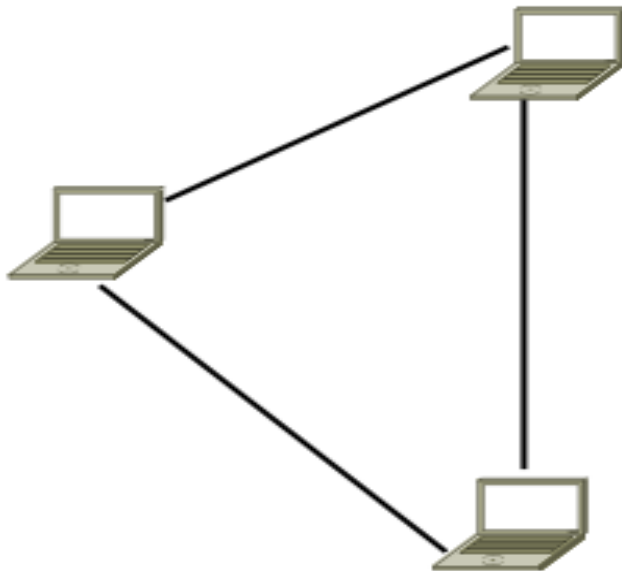
# pspoll diagram



# IBSS Mode

- IBSS – **without** an access point.

IBSS (Independent BSS)



# IBSS Mode - contd

- IBSS network is often formed without pre-planning, for only as long as the LAN is needed.
- This type of operation is often referred to as an **Ad Hoc** network.
  - Also sometimes called “Peer To Peer” network.

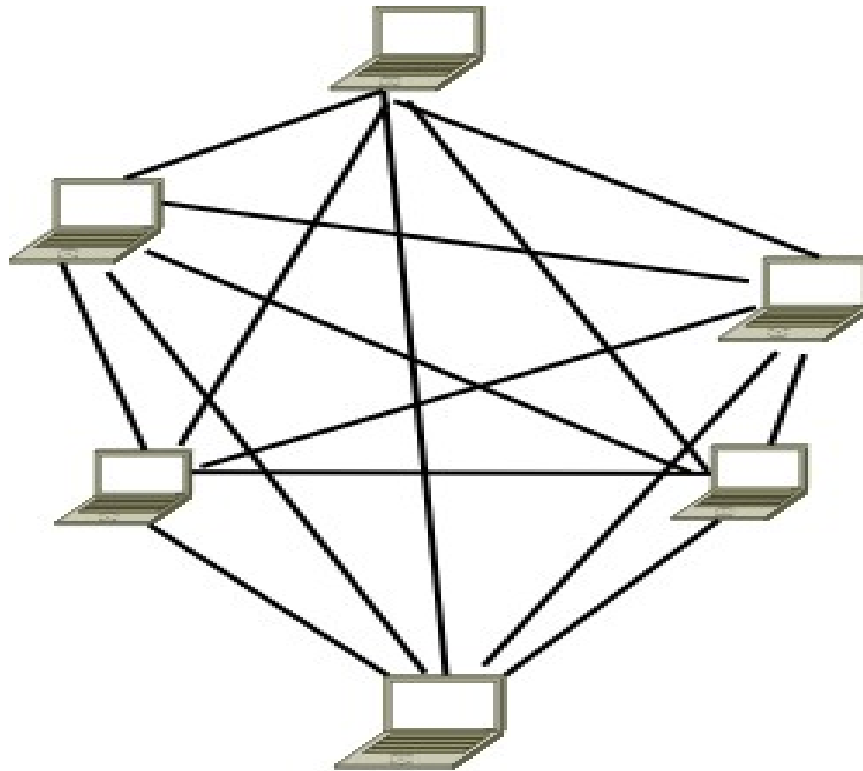
- Creating Ad-Hoc network:
  - iwconfig wlan0 mode ad-hoc
  - (note: if the nic is running, you should run before this: *ifconfig wlan0 down*)
  - iwconfig wlan0 essid myEssid
  - The essid has to be distributed manually (or otherwise) to everyone who wishes to connect to the Ad-Hoc network.
- The BSSID is a random MAC address.
  - (in fact, 46 bits of it are random).

- “iwconfig wlan0 essid myEssid” triggers ibss creation by calling *ieee80211\_sta\_create\_ibss()*
  - *net/mac80211/mlme.c*

- Joining an IBSS:
  - All members of the IBSS participate in beacon generation.
  - The members are synchronized (TSF).
  - The beacon interval within an IBSS is established by the STA that instantiates the IBSS.
  - *ieee80211\_sta\_create\_ibss()* (mlme.c)
  - The bssid of the ibss is a random address (based on mixing get\_random\_bytes() and MAC address).

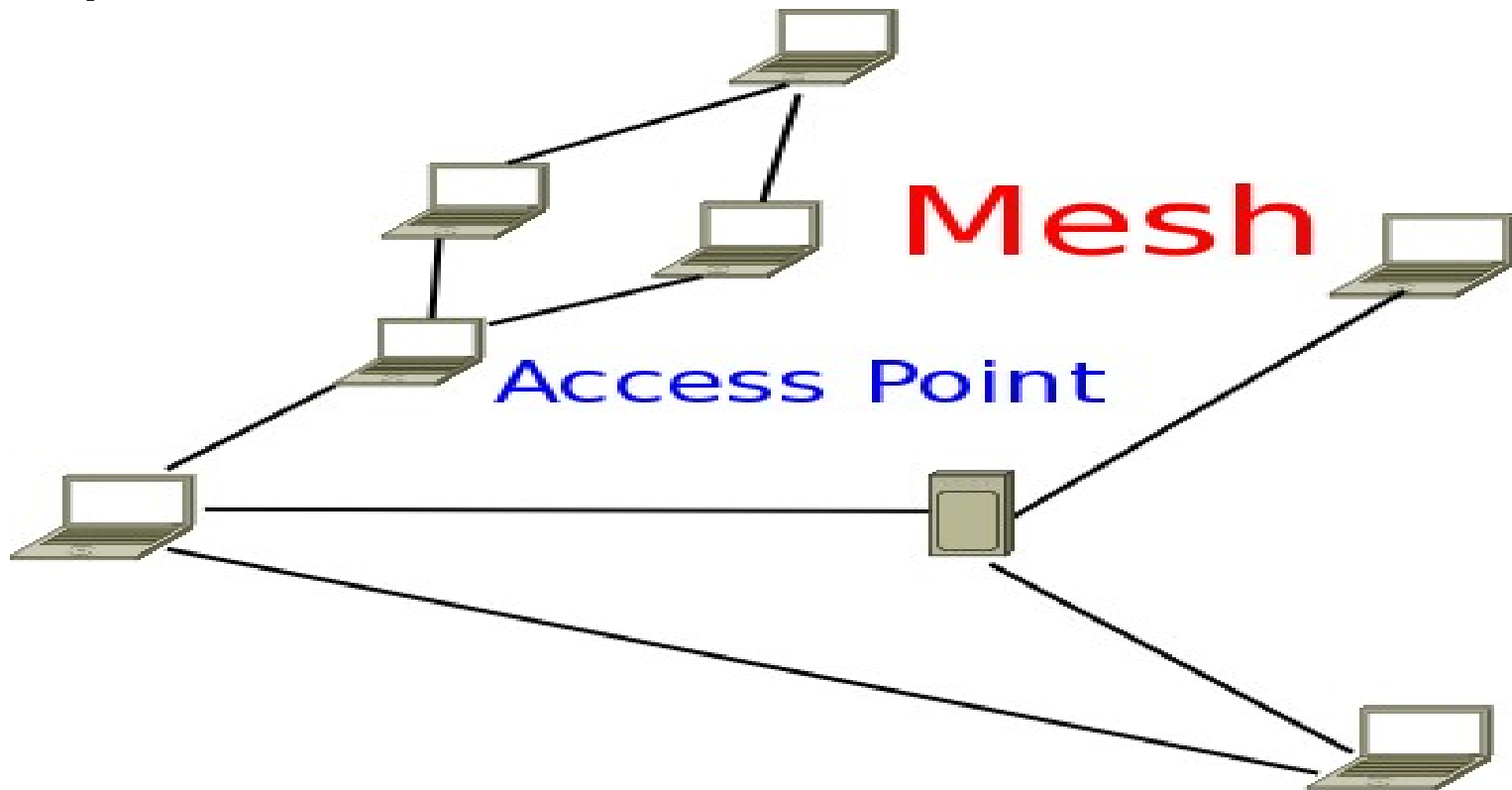
# Mesh Mode (80211s)

**Full Mesh:** In the full mesh topology, each node is connected directly to each of the others.



# Mesh Mode (80211s)

**Partial Mesh: nodes are connected to only some, not all.**



# 802.11s (Mesh)

- 802.11s started as a Study Group of IEEE 802.11 in September 2003, and became a TG (Task Group) in 2004. (name: TGs)
- In 2006, two proposals, out of 15, (the "SEE-Mesh" and "Wi-Mesh" proposals) were merged into one. This is draft D0.01.
- Wireless Mesh Networks are also called WMN.
- Wireless mesh networks forward data packets over multiple wireless hops. Each mesh node acts as relay point/router for other mesh nodes.

- In 2.6.26, the network stack added support for the draft of wireless mesh networking (802.11s), thanks to the open80211s project (<http://www.open80211s.org/>).
  - There is still no final spec.
  - There are currently **five** drivers in linux with support to mesh networking (ath5k,b43,libertas\_tf,p54,zd1211rw), and **one** is under development (rt2x00).

- Open80211.s
- Goal: To create the first open implementation of 802.11s.

- Sponsors:

- OLPC project.



- Cozybit (<http://www.cozybit.com/>), the company that developed the mesh software on the OLPC Laptop.
    - Luis Carlos Cobo and Javier Cardona (both from Cozybit) developed the Linux mac80211 mesh code.
  - Nortel

- 80211.s defines a default routing protocol called **HWMP** (Hybrid Wireless Mesh Protocol)
- Based on: Ad Hoc Demand Distance Vector (AODV) routing (C. Perkins); rfc3561.
- The HWMP protocol works with layer 2 (Mac addresses).
- The 80211 header was extended:
  - A ttl field was added to avoid loops.

- The current implementation uses **on demand** path selection.
- The draft also talks about proactive path selection.
  - This is not implemented yet in the Linux Kernel.
  - Uses Root Announcement (RANN) messages and Mesh Portal as a root.

- As with IPV4 static routes, you can force a specific next hop for a mesh station (MESH\_PATH\_FIXED flag)
  - (*mesh\_path\_fix\_nexthop()* in *mesh\_pathtbl.c*)
- Every station is called an **MP**. (Mesh Point)
- MPP is a Mesh Portal. (For example, when an MP is used to connect to external network, like the Internet).
- Each station holds a routing table (*struct mesh\_table*) – helps to decide which route to take.

- In the initial state, when a packet is sent to another station, there is first a lookup in the mesh table; there is no hit, so a **PREQ (Path Request)** is sent as a broadcast.
  - When the **PREQ** is received on all stations except the final destination, it is forwarded.
  - When the **PREQ** is received on the final station, a PREP is sent (**Path Reply**).
  - If there is some failure on the way, a **PERR** is sent. (**Path Error**).
    - Handled by *mesh\_path\_error\_tx()*, mesh\_hwmp.c
- The route take into consideration an airtime metric
  - Calculated in *airtime\_link\_metric\_get()* (based on rate and other hw parameters).
- POWER SAVING in the MESH spec is optional.

- **Advantage:**

- Rapid deployment.
- Minimal configuration; inexpensive.
- Easy to deploy in hard-to-wire environments.

- **Disadvantage:**

- Many broadcasts limit network performance
- You can set a wireless device to work in mesh mode only with the iw command (You cannot perform this with the wireless tools).
- Example: setting a wireless nic to work in mesh mode:
  - *iw dev wlan1 interface add mesh type mp mesh\_id 1*
  - *(type = mp => Mesh Point)*

## 802.11 Physical Modes

- 802.11 (WiFi) is a set of standards for wireless networking, which were defined in 1997 but started to become popular in the market around 2001.
- **802.11a** (1999) at 5 GHz, 54MBit maximum speed; range about 30m.
- **802.11b** (1999) at 2.4GHz, 11Mbit maximum speed, range about 30m.
- **802.11g** (2003) at 2.4GHz, 54Mbit maximum speed, range about 30m.

- **802.11n** (2008) at 2.4GHz/5GHz, 200 Mbit (typical), range about 50m.
- is planned to support up to about 540Mbit/ 600 Mbit.
- Improves the previous 802.11 standards by adding multiple-input multiple-output (MIMO)
  - multiple antennas.
  - High Throughput (**HT**).
  - Use packet aggregation
    - The ability to send several packets together at one time.

- Still is considered a proposal.
  - Expected to be approved only in [December 2009](#) or later.
- iwlagm and ath9k are the only drivers that support 80211.n in the Linux kernel at the moment.
- Tip: how can I know whether my wireless nic supports 80211.n?
  - Run: *iwconfig*
  - You should see : "IEEE 802.11abgn" or somesuch.

# Appendix: mac80211 implementation details

- BSSID = Basic Service Set Identification.
- Each BSS has an BSSID.
- BSSID is an 48 bit number (like MAC address).
  - This avoids getting broadcasts from other networks which may be physically overlapping.
  - In infrastructure BSS, the BSSID is the MAC address of the Access Point which created the BSS.
  - In IBSS, the BSSID is generated from calling a random function (generating 46 random bits; the other 2 are fixed).

# Modes of operation

- A wireless interface always operates in one of the following modes:
- **Infrastructure mode:** with an AccessPoint (AP)
  - The access point hold a list of associated stations.
  - also called managed)
- **IBSS** (Independent BSS,Ad-Hoc) mode
  - When using ad-hoc, an access point is not needed.
- **Monitor** mode
- **WDS** (Wireless Distribution System)

# Modes of operation - contd.

- Wireless Distribution System (WDS) - allows access points to talk to other access points.
- **Mesh**

see: include/linux/nl80211.h:

```
enum nl80211_iftype {  
    NL80211_IFTYPE_UNSPECIFIED,  
    NL80211_IFTYPE_ADHOC,  
    NL80211_IFTYPE_STATION,  
    NL80211_IFTYPE_AP,  
    NL80211_IFTYPE_AP_VLAN,  
    NL80211_IFTYPE_WDS,  
    NL80211_IFTYPE_MONITOR,  
    NL80211_IFTYPE_MESH_POINT,  
}
```

# cfg80211 and nl80211

- Wireless-Extensions has a new replacement;
- It is cfg80211 and nl80211 (message-based mechanism, using netlink interface).
- iw uses the nl80211 interface.
  - You can compare it the the old ioctl-based net-tools versus the new rtnetlink IPROUTE2 set of tools.
  - You cannot set master mode with iw.
  - You cannot change the channel with iw.

- Wireless git trees:
- Wireless-testing
- Was started on February 14, 2008 by John Linville.
  - primary development target.
  - the bleeding edge Linux wireless developments.
- wireless-next-2.6
- Wireless-2.6
- Daily compat-wireless tar ball in:
- <http://www.orbit-lab.org/kernel/compat-wireless-2.6/>
- The compat-wireless tar ball includes only part of the kernel
  - (Essentially it includes wireless drivers and wireless stack)

- Fedora kernels are usually up-to-date with wireless-testing git tree.
- There is usually at least one pull request (or more) in a week, to the netdev mailing list (main Linux kernel networking mailing list).
- The Maintainer of the wireless (802.11) in the Linux kernel is John Linville (RedHat), starting from January 2006.

- For helping in delving into the mac80211 code little help.
- Important data structures:
- `struct ieee80211_hw` – represents hardware information and state (include/net/mac80211.h).
  - Important member: `void *priv` (pointer to private area).
  - Most drivers define a struct for this private area , like *lbtf\_private* (Marvell) or *iwl\_priv* (iwlwifi of Intel) or *mac80211\_hwsim\_data* in mac80211\_hwsim.
  - Every driver allocates it by `ieee80211_alloc_hw()`
  - *A pointer to ieee80211\_ops (see later) is passed as a parameter to `ieee80211_alloc_hw()`.*
  - Every driver calls `ieee80211_register_hw()` to create wlan0 and wmaster0 and for various initializations.

- You set the machine mode prior to calling *ieee80211\_register\_hw()* by assigning flags for the interface\_modes flags of wiphy member
  - wiphy itself is a member of ieee80211\_hw structure.
  - For example,

```
hw->wiphy->interface_modes =  
    BIT(NL80211_IFTYPE_STATION) |  
    BIT(NL80211_IFTYPE_AP);
```

- This sets the machine to be in Access Point mode.

- `struct ieee80211_if_ap` – represents an access point. (see `ieee80211_i.h`)
- Power saving members of `ieee80211_if_ap`:
  - `ps_bc_buf` (multicast/broadcast buffer).
  - `num_sta_ps` (number of stations in PS mode).

- **struct ieee80211\_ops** – The drivers use its members. (include/net/mac80211.h).
- For example, **config** (to change a channel) or **config\_interface** to change bssid.
- Some drivers upload firmware at the start() method, like lbt\_f\_op\_start() in libetras\_tf driver or zd\_op\_start() (which calls zd\_op\_start() to upload firmware zd1211rw
- All methods of this struct get a pointer to struct ieee80211\_hw as a first parameter.
  - There are 24 methods in this struct.
  - **Seven** of them are mandatory:  
**tx, start, stop, add\_interface, remove\_interface, config** and **configure\_filter**.
  - (If anyone of them is missing, we end in BUG\_ON())

- Receiving a packet is done by calling *ieee80211\_rx\_irqsafe()* from the low level driver. Eventually, the packet is handled by *\_\_ieee80211\_rx()*:
- *\_\_ieee80211\_rx()*(struct ieee80211\_hw \*hw,  
                  struct sk\_buff \*skb,  
                  struct ieee80211\_rx\_status \*status);
- *ieee80211\_rx\_irqsafe()* can be called from interrupt context.
  - There is only one more mac80211 method which can be called from interrupt context:
  - *ieee80211\_tx\_status\_irqsafe()*

- Data frames
  - Addr1 – destination (receiver MAC address).
  - Addr2 – source (transmitter MAC address).
  - Addr3 - DS info
  - Addr4 – for WDS.
- Management frames
  - Addr1 – destination (receiver MAC address).
  - Addr2 – source (transmitter MAC address).
  - Addr3 - DS info

# Firmware

- Firmware:
  - Most wireless drivers load firmware in the probe method (by calling *request\_firmware()*)
  - Usually the firmware is not open source.
  - Open FirmWare for WiFi networks site:
    - <http://www.ing.unibs.it/openfwf/>
      - Written in assembler.
  - B43 firmware will be replaced by open source firmware.
  - ath5k/athk9k driver doesn't load firmware. (its fw is burnt into an onchip ROM)

# Wireless Future trends (WiMax)

- WiMax - IEEE 802.16.
- There are already laptops which are sold with
- WiMax chips (Toshiba, Lenovo).
- WiMax and Linux:
- <http://linuxwimax.org/>
- Inaky Perez-Gonzalez from Intel
  - (formerly a kernel USB developer)
- Location in the kernel tree: *drivers/net/wimax*.

# Wireless Future trends (WiMax) - contd

- Two parts:
- Kernel module driver
- User space management stack, WIMAX Network Service.
- A request to merge linux-wimax GIT tree with the netdev GIT tree was sent in 26.11.08
- <http://www.spinics.net/lists/netdev/msg81902.html>
- There is also an initiative from Nokia for a WiMax stack for Linux.

# Tips

- How can I know if my wireless nic was configured to support power management ?
  - Look in *iwconfig* for “Power Management” entry.
- How do I know if my USB nic has support in Linux?
  - <http://www.qbik.ch/usb/devices/>
- How do I know which Wireless Extensions does my kernel use?
- Grep for `#define WIRELESS_EXT` in `include/linux/wireless.h` in your kernel tree.

- How can I know the channel number from a sniff?
  - Look at the radiotap header in the sniffer output; channel frequency translates to a channel number (1 to 1.)
  - See also Table 15-7—DSSS PHY frequency channel plan , in the 2007 80211
  - Often, the channel number appears in square brackets. Like:
  - channel frequency 2437 [BG 6]
  - BG stands for 802.11B/802.11G, respectively

- Channel 14 for example would show as B, because you're not allowed to transmit 802.11G on it.
- Israel regdomain:
  - <http://wireless.kernel.org/en/developers/Regulatory/Database?alpha2=IL>
  - IL is in the range 1-13.
  - With US configuration, only channel 1 to 11 are selectable. Not 12,13.
  - Many Aps are shipped on a US configuration.

- What is the MAC address of my nic?
  - `cat /sys/class/ieee80211/phy*/macaddress`
  - 
  - **Common Filters for wireshark sniffer:**

Management Frames `wlan.fc.type eq 0`

Control Frames `wlan.fc.type eq 1`

Data Frames `wlan.fc.type eq 2`

Association Request `wlan.fc.type_subtype eq 0`

Association response `wlan.fc.type_subtype eq 1`

Reassociation Request `wlan.fc.type_subtype eq 2`

Reassociation Response `wlan.fc.type_subtype eq 3`

Probe Request `wlan.fc.type_subtype eq 4`

Probe Response wlan.fc.type\_subtype eq 5

Beacon wlan.fc.type\_subtype eq 8

Announcement Traffic Indication Map (ATIM) wlan.fc.type\_subtype eq 9

Disassociate wlan.fc.type\_subtype eq 10

Authentication wlan.fc.type\_subtype eq 11

Deauthentication wlan.fc.type\_subtype eq 12

Action Frames wlan.fc.type\_subtype eq 13

Block Acknowledgement (ACK) Request wlan.fc.type\_subtype eq 24

Block ACK wlan.fc.type\_subtype eq 25

Power-Save Poll wlan.fc.type\_subtype eq 26

Request to Send wlan.fc.type\_subtype eq 27

# Sniffing a WLAN

- You could sniff with Wireshark
- Sometime you can't put the wireless interface to promiscuous mode (or it is not enough). You should set the interface to work in monitor mode (For example: `iwconfig wlan0 mode monitor`).
- If you want to capture traffic on networks other than the one with which you're associated, you will **have to** capture in **monitor** mode.

# Sniffing a WLAN - contd.

- See the following wireshark wiki page, talking about various wireless cards and sniffing in Linux;
- WLAN (IEEE 802.11) capture setup:
  - <http://wiki.wireshark.org/CaptureSetup/WLAN#head->
- Using a filter from command line:
  - `tshark -R wlan -i wlan0`
  - `tethereal -R wlan -i wlan0 -w wlan.eth`
  - You will see this message in the kernel log:
    - “device wlan0 entered promiscuous mode”

# Sniffing a WLAN - contd.

- Sometimes you will have to set a different channel than the default one in order to see beacon frames (try channels 1,6,11)
  - `iwconfig wlan1 channel 11`
  - Tip: usefull wireshark display filter:
    - For showing only beacons:
    - *wlan.fc.type\_subtype eq 8*
  - For tshark command line:
    - *tshark -R "wlan.fc.type\_subtype eq 8" -i wlan0*
    - *(this will sniff for beacons).*

# Glossary

- AMPDU=Application Message Protocol Data Unit.
- CRDA = Central Regulatory Domain Agent
- CSMA/CA = Carrier Sense Multiple Access with Collision Avoidance
- CSMA/CD Carrier Sense Multiple Access with Collision Detection
- DS = Distribution System
- EAP = The Extensible Authentication Protocol
- ERP = extended rate PHY

- HWMP = Hybrid Wireless Mesh Protocol
- MPDU = MAC Protocol Data Unit
- MIMO = Multiple-Input/Multiple-Output
- PSAP = Power Saving Access Points
- PS = Power Saving.
- RSSI = Receive signal strength indicator.
- TIM = Traffic Indication Map
- WPA = Wi-Fi Protected Access
- WME = Wireless Multimedia Extensions



# Links

- 1) IEEE 80211 specs:
  - <http://standards.ieee.org/getieee802/802.11.html>
- 2) Linux wireless status June - 2008
  - <http://www.kernel.org/pub/linux/kernel/people/mcgrof/presentations/linux-wireless-status.pdf>
- 3) official Linux Wireless wiki hosted by Johannes Berg.
  - <http://wireless.kernel.org/>
  - or <http://linuxwireless.org/>

- 4) A book:
  - 802.11 Wireless Networks: The Definitive Guide
  - by Matthew Gast
  - Publisher: O'Reilly
- 5) Wireless Sniffing with Wireshark - Chapter 6 of Syngress Wireshark and Ethereal Network Protocol Analyzer Toolkit.
- 6) <http://www.lesswatts.org/>
  - Saving power with Linux (an Intel site)

- 7) A book: Wireless Mesh Networking:  
Architectures, Protocols And Standards  
by Yan Zhang, Jijun Luo, Honglin Hu (Hardcover  
– 2006)

Auerbach Publications

8) <http://www.radiotap.org/>

# Images

- Beacon wireshark filter:
- wlan.fc.type\_subtype eq 8
  - shows only beacons.

# Beacon filter – sniff

The image shows a Wireshark capture window titled "adHocHome.eth - Wireshark". The filter bar at the top contains the expression "wlan.fc.type\_subtype eq 8". The packet list shows a series of IEEE 802.11 Beacon frames, all with a source address of 00:21:91:80:ba:2d and a destination of Broadcast. Packet 45 is highlighted in blue.

No.	Time	Source	Destination	Protocol	Info
35	2009-02-01 21:19:26.869991	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=34, FN=0, F1
36	2009-02-01 21:19:26.971987	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=35, FN=0, F1
37	2009-02-01 21:19:27.075004	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=36, FN=0, F1
38	2009-02-01 21:19:27.177043	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=37, FN=0, F1
39	2009-02-01 21:19:27.280007	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=38, FN=0, F1
40	2009-02-01 21:19:27.382008	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=39, FN=0, F1
41	2009-02-01 21:19:27.484010	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=40, FN=0, F1
42	2009-02-01 21:19:27.587003	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=41, FN=0, F1
43	2009-02-01 21:19:27.689025	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=42, FN=0, F1
44	2009-02-01 21:19:27.792008	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=43, FN=0, F1
45	2009-02-01 21:19:27.894004	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=44, FN=0, F1
46	2009-02-01 21:19:27.996001	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=45, FN=0, F1
47	2009-02-01 21:19:28.098987	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=46, FN=0, F1
48	2009-02-01 21:19:28.200998	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=47, FN=0, F1
49	2009-02-01 21:19:28.304004	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=48, FN=0, F1
50	2009-02-01 21:19:28.406004	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=49, FN=0, F1
51	2009-02-01 21:19:28.508992	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=50, FN=0, F1
52	2009-02-01 21:19:28.611003	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=51, FN=0, F1
53	2009-02-01 21:19:28.714004	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=52, FN=0, F1
54	2009-02-01 21:19:28.816010	00:21:91:80:ba:2d	Broadcast	IEEE 802	Beacon frame, SN=53, FN=0, F1

Packet 45 details:

- ▶ Radiotap Header v0, Length 24
- ▼ IEEE 802.11 Beacon frame, Flags: .....C
  - Type/Subtype: Beacon frame (0x08)
  - ▶ Frame Control: 0x0080 (Normal)
  - Duration: 0
  - Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
  - Source address: 00:21:91:80:ba:2d (00:21:91:80:ba:2d)
  - BSS Id: 22:3f:b7:97:3e:45 (22:3f:b7:97:3e:45)
  - Fragment number: 0

Packet 45 hex dump:

```
0000  00 00 18 00 0e 58 00 00 10 02 6c 09 a0 00 00 5b  ....X.. ..l....[
0010  00 00 00 00 00 00 00 00 80 00 00 00 ff ff ff ff  ....
0020  ff ff 00 21 91 80 ba 2d 22 3f b7 97 3e 45 c0 02  ....!...- "?..>E..
0030  a6 a3 0f 00 00 00 00 00 64 00 02 00 00 07 68 6f  ....d....ho
```

File: "adHocHome.eth" 6994 Bytes 00:01:27 Packets: 68 Displayed: 34 Marked: 0

# Beacon interval and DTIM period in edimax router (BR-6504N) (From the manual)

The screenshot shows the Edimax BR-6504N router's web interface. The top navigation bar includes links for Quick Setup, General Setup, Status Info, and System Tools. The left sidebar contains a menu with System, WAN, LAN, Wireless (selected), QoS, NAT, and Firewall. Under the Wireless menu, Basic Settings, Advance Settings (highlighted), Security Settings, Access Control, and WPS are listed. The main content area is titled 'Advance Settings' and contains a warning: 'These settings are only for more technically advanced users who have a sufficient knowledge about wireless LAN. These settings should not be changed unless you know what effect the changes will have on your Broadband router.'

Fragment Threshold:	2346	(256-2346)	
RTS Threshold:	2347	(0-2347)	
Beacon Interval:	100	(20- 1024 ms)	
DTIM Period:	3	(1-10)	
Data Rate:	Auto		
N Data Rate:	Auto		
Channel Width:	<input checked="" type="radio"/> Auto 20/40 MHZ	<input type="radio"/> 20 MHZ	
Preamble Type:	<input checked="" type="radio"/> Short Preamble	<input type="radio"/> Long Preamble	
Broadcast Essid:	<input checked="" type="radio"/> Enable	<input type="radio"/> Disable	
CTS Protect:	<input type="radio"/> Auto	<input type="radio"/> Always	<input checked="" type="radio"/> None
Tx Power:	100 %		
Turbo Mode:	<input checked="" type="radio"/> Enable	<input type="radio"/> Disable	
WMM:	<input type="radio"/> Enable	<input checked="" type="radio"/> Disable	

# Thank You !



# Linux Kernel Networking – advanced topics (5)

## Sockets in the kernel

Rami Rosen

[ramirose@gmail.com](mailto:ramirose@gmail.com)

Haifux, August 2009

[www.haifux.org](http://www.haifux.org)

All rights reserved.



# Linux Kernel Networking (5)- advanced topics

- Note:
- This lecture is a sequel to the following 4 lectures I gave in Haifux:

## **1) Linux Kernel Networking lecture**

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

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

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

# Linux Kernel Networking (5)- advanced topics

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

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

## 4) Wireless in Linux

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

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

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

TCP socket

UDP Socket

**Userspace**

---

Layer 4 (TCP,UDP,SCTP,...)

Layer 3 (Network layer: IPV4/IPV6)

Layer 2 (MAC layer)

**kernel**

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

- The **family** is PF\_INET for IPV4 or PF\_INET6 for IPV6.
  - The family is PF\_PACKET for Packet sockets, which operate at the device driver layer. (Layer 2).
- pcap library for Linux uses PF\_PACKET sockets:
  - pcap library is in use by sniffers such as tcpdump.
- Also hostapd uses PF\_PACKET sockets:
- (hostapd is a wireless access point management project)
- From hostapd:
  - `drv->monitor_sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));`

- Type:
  - SOCK\_STREAM and SOCK\_DGRAM are the mostly used types.
    - SOCK\_STREAM for TCP, SCTP, BLUETOOTH.
    - SOCK\_DGRAM for UDP.
    - SOCK\_RAW for RAW sockets.
    - There are cases where protocol can be either SOCK\_STREAM **or** SOCK\_DGRAM; for example, Unix domain socket (AF\_UNIX).
  - Protocol: usually 0 ( IPPROTO\_IP is 0, see: include/linux/in.h).
  - For SCTP, the protocol is **IPPROTO\_SCTP**:
    - sockfd=socket(AF\_INET, SOCK\_STREAM, **IPPROTO\_SCTP**);

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

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

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

```
struct sock {
```

```
...
```

```
    struct socket      *ssocket;
```

```
}
```

struct socket (include/linux/net.h)

```
struct socket {
```

```
    socket_state      state;
```

```
    short              type;
```

```
    unsigned long      flags;
```

```
    struct fasync_struct *fasync_list;
```

```
    wait_queue_head_t  wait;
```

```
    struct file         *file;
```

```
    struct sock         *sk;
```

```
    const struct proto_ops *ops;
```

- The state can be
  - SS\_FREE
  - SS\_UNCONNECTED
  - SS\_CONNECTING
  - SS\_CONNECTED
  - SS\_DISCONNECTING
- These states are not layer 4 states (like TCP\_ESTABLISHED or TCP\_CLOSE).
- The sk\_protocol member of struct sock equals to the third parameter (protocol) of the *socket()* system call.

- struct proto\_ops (interface of struct socket)

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

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

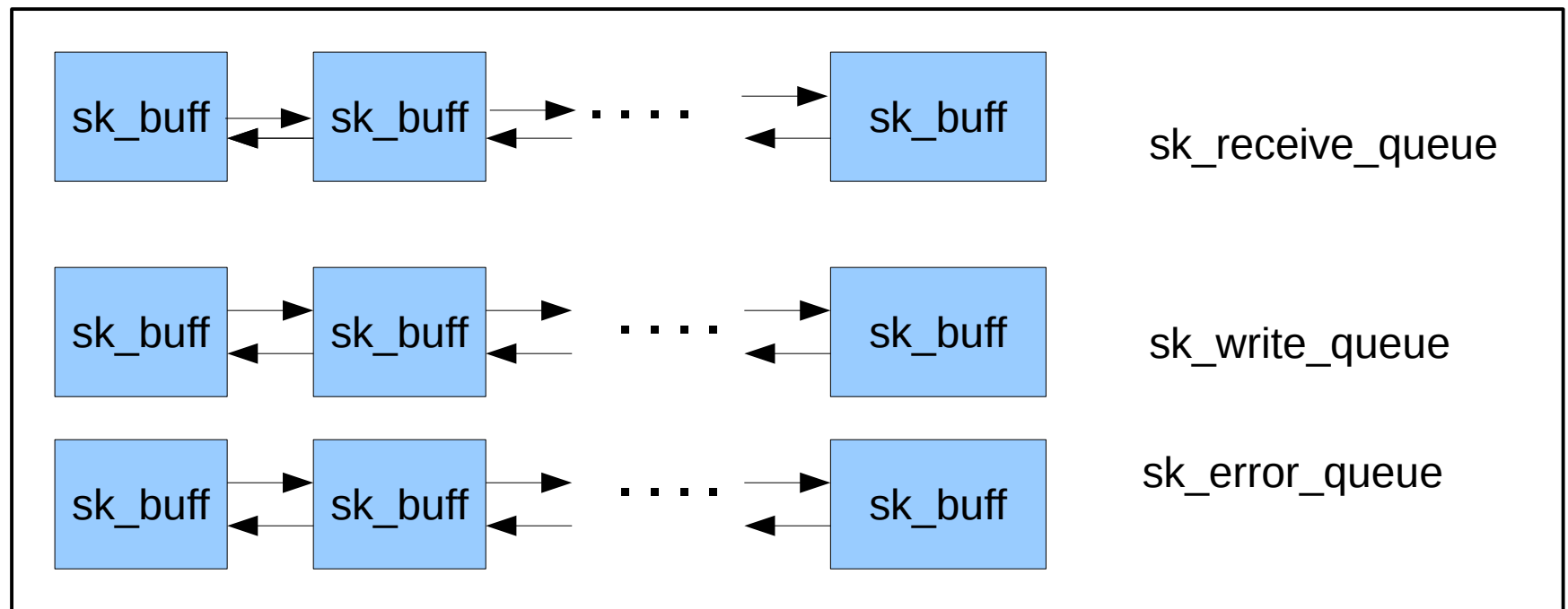
- Diagram: **struct inet\_sock**

struct sock (sk)

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

**inet\_sk(sock \*sk) => returns the inet\_sock which contains sk**

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



- Each queue has a lock (spinlock)

- *skb\_queue\_tail()* : Adding to the queue
- *skb\_dequeue()* : removing from the queue
  - With MSG\_PEEK, this is done in two stages:
    - *skb\_peek()*
    - *\_\_skb\_unlink()*. (to remove the sk\_buff from the queue).
- For the error queue: *sock\_queue\_err\_skb()* adds to its tail (include/net/sock.h). Eventually, it also calls *skb\_queue\_tail()*.
- Errors can be ICMP errors or EMSGSIZE errors.
- For more about errors, see APPENDIX F: UDP errors.

# UDP and TCP

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

# The udp header

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

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

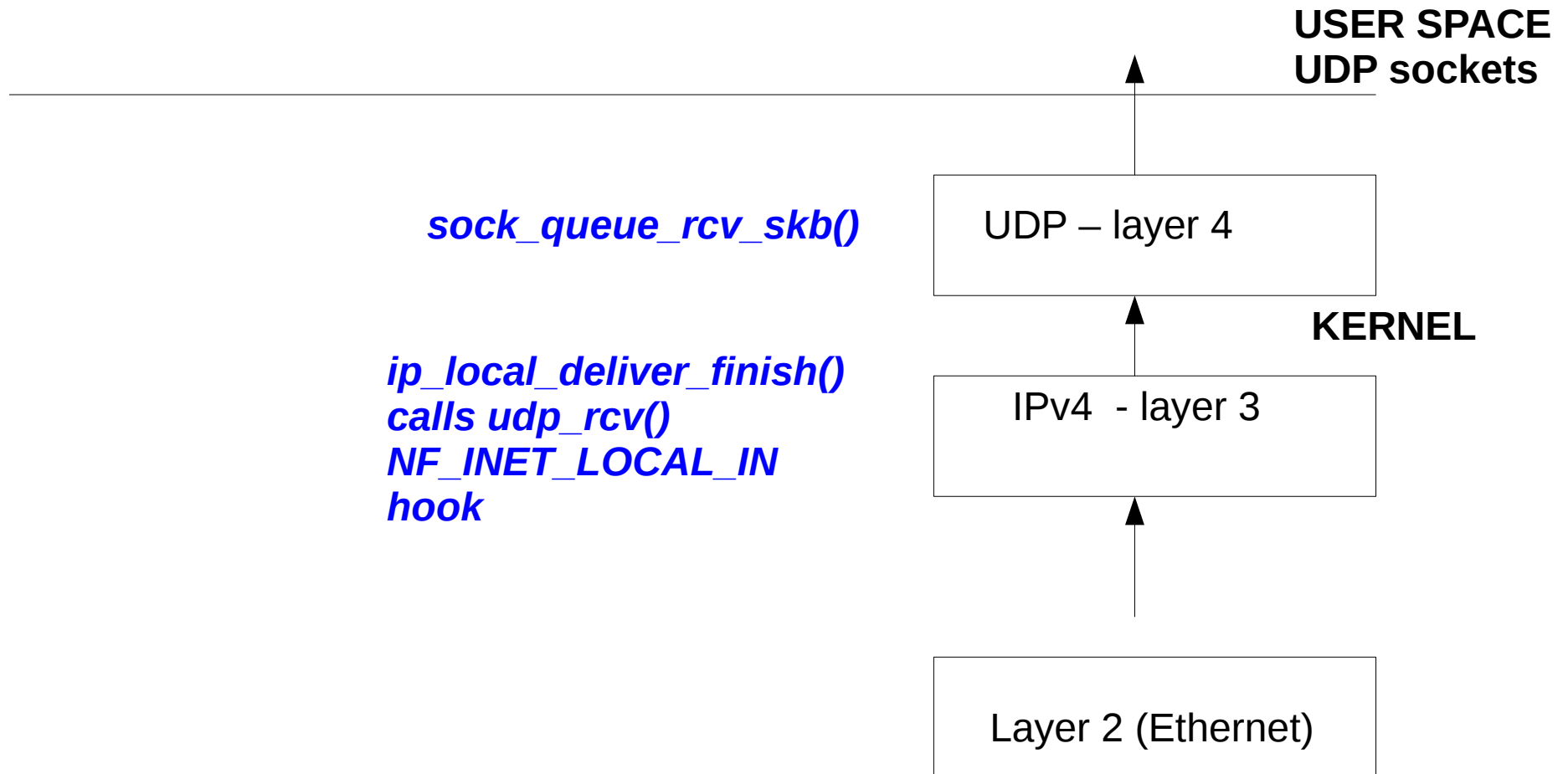
```
struct udphdr {  
    __be16 source;  
    __be16 dest;  
    __be16 len;  
    __sum16  check;  
};
```

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

source port	dest port
len	checksum
<b>Payload</b>	

# Receiving packets in UDP from kernel

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



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

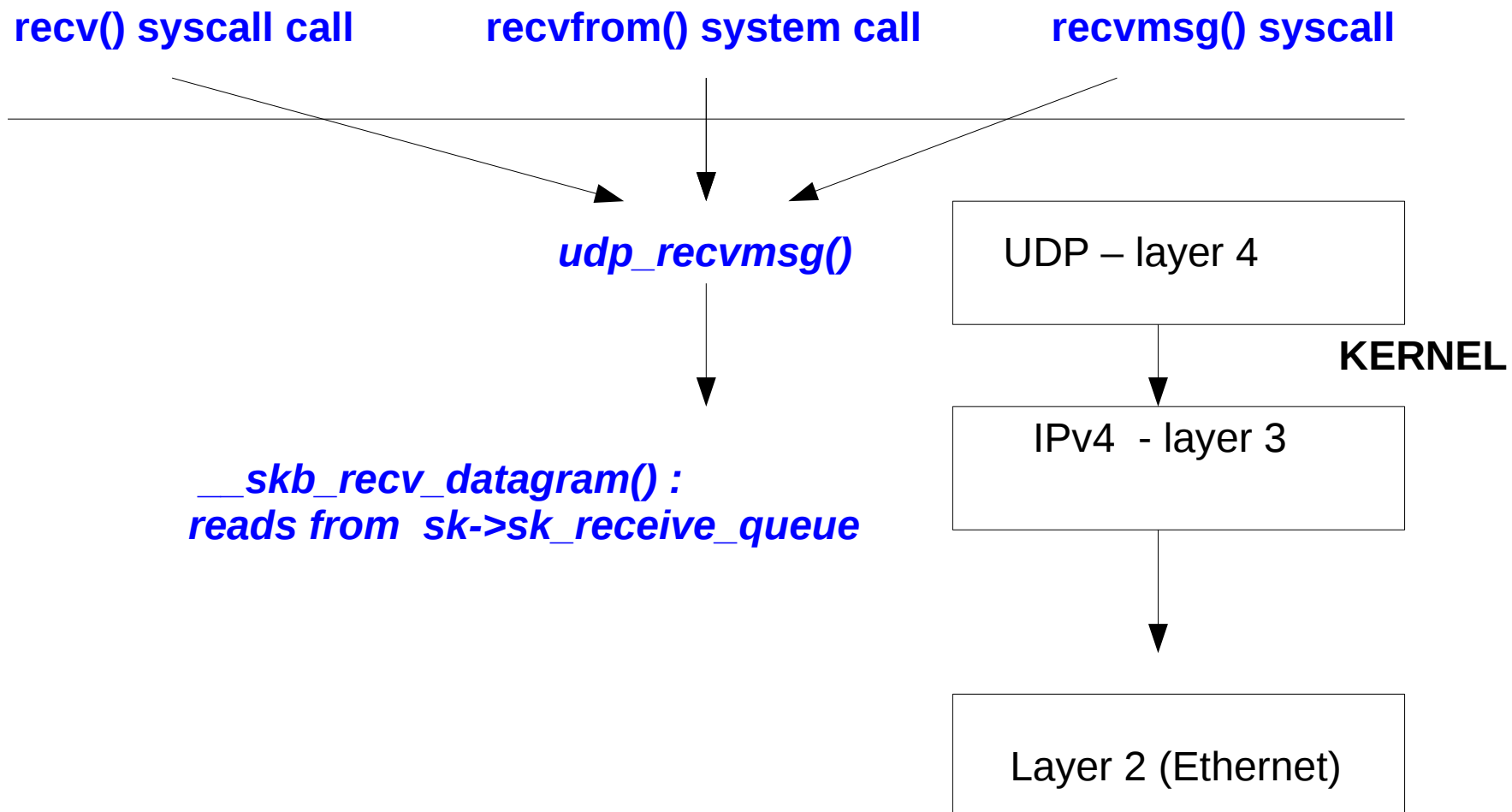
- For example, suppose you have a client-server udp applications, and the sender sends a packets which is longer than what the client had allocated for input buffer. The kernel than truncates the packet, and send **MSG\_TRUNC** flag. In order to retrieve it, you should use something like:

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

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

# Receiving packets in UDP from user space

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



# Receiving packets - udp\_rcv()

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

See the udp\_protocol definition: (net/ipv4/af\_inet.c)

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

- In the same way we have :
  - *raw\_rcv()* as a handler for raw packets.
  - *tcp\_v4\_rcv()* as a handler for TCP packets.
  - *icmp\_rcv()* as a handler for ICMP packets.
- Kernel implementation: the *proto\_register()* method registers a protocol handler.  
(net/core/sock.c)

## *udp\_rcv() implementation:*

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

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

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

# udp\_rcv()

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

*netstat -s*

.....

Udp:

...

**35** packets to unknown port received.

# udp\_rcv() - contd

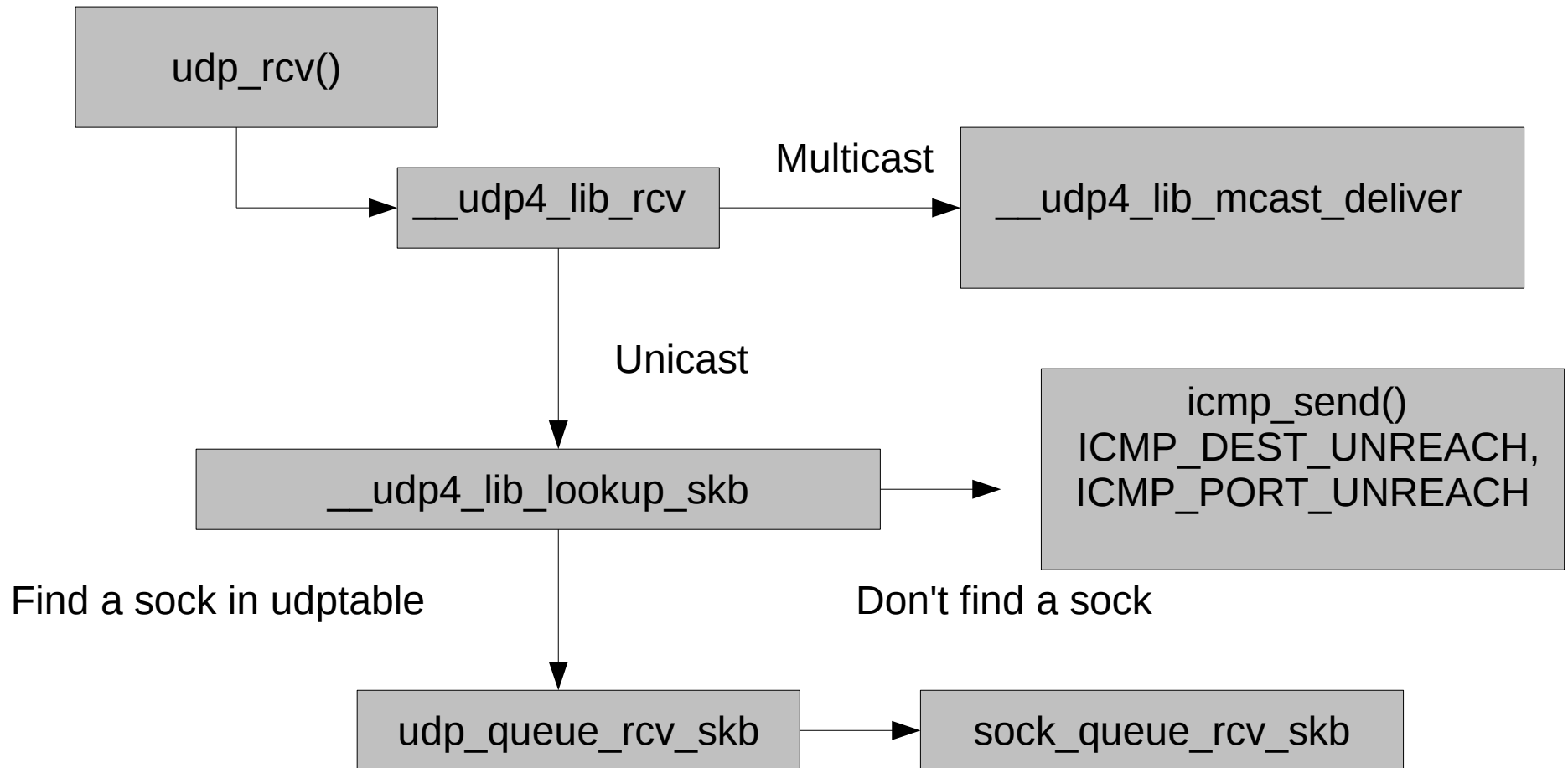
- Or, by:
- `cat /proc/net/snmp | grep Udp:`

Udp: InDatagrams **NoPorts** InErrors  
OutDatagrams RcvbufErrors SndbufErrors

Udp: 14 **35** 0 30 0 0

- If there is a sock listening on the destination port, call *udp\_queue\_rcv\_skb()*.
  - *Eventually calls sock\_queue\_rcv\_skb()*.
    - Which adds the packet to the **sk\_receive\_queue** by *skb\_queue\_tail()*

# udp\_rcv() diagram



- *udp\_recvmsg()*:
- Calls *\_\_skb\_recv\_datagram()* , for receiving one *sk\_buff*.
  - The *\_\_skb\_recv\_datagram()* may block.
  - Eventually, what *\_\_skb\_recv\_datagram()* does is read one *sk\_buff* from the *sk\_receive\_queue* queue.
- *memcpy\_toiovec()* performs the actual copy to user space by invoking *copy\_to\_user()*.
- One of the parameters of *udp\_recvmsg()* is a pointer to struct ***msghdr***. Let's take a look:

# MSGHDR

From include/linux/socket.h:

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

# Control messages (ancillary messages)

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

- cmsghdr members can mean different things based on the type of socket.
- There is a set of macros for handling cmsghdr like CMSG\_FIRSTHDR(), CMSG\_NXTHDR(), CMSG\_DATA(), CMSG\_LEN() and more.
- There are no control messages for TCP sockets.

# Socket options:

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

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

There are two levels of socket options:

To manipulate options at the sockets API level: **SOL\_SOCKET**

To manipulate options at a protocol level, that protocol number should be used;

- for example, for UDP it is **IPPROTO\_UDP** or **SOL\_UDP** (both are equal 17) ; see `include/linux/in.h` and `include/linux/socket.h`
  - **SOL\_IP** is 0.

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

See Appendix B for a full list of socket options.

- There is an option called `IP_PKTINFO`.
  - We will set the `IP_PKTINFO` option on a socket in the following example.

```
// from /usr/include/bits/in.h
```

```
#define IP_PKTINFO      8 /* bool */
```

```
/* Structure used for IP_PKTINFO. */
```

```
struct in_pktinfo
```

```
{  
    int ipi_ifindex;           /* Interface index */  
    struct in_addr ipi_spec_dst; /* Routing destination address */  
    struct in_addr ipi_addr;    /* Header destination address */  
};
```

```
const int on = 1;
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (setsockopt(sockfd, SOL_IP, IP_PKTINFO, &on,
    sizeof(on)) < 0)
    perror("setsockopt");
...
...
...
```

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

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

- In the kernel, this calls *ip\_cmsg\_recv()* in net/ipv4/ip\_sockglue.c. (which eventually calls *ip\_cmsg\_recv\_pktinfo()*).
- You can in this way retrieve other fields of the ip header:
  - For getting the TTL:
    - setsockopt(sockfd, SOL\_IP, IP\_RECVTTL, &on, sizeof(on))<0).
    - But: cmsg\_type == IP\_TTL.
  - For getting ip\_options:
    - setsockopt() with IP\_OPTIONS.

- Note: you cannot get/set ip\_options in Java app.

# Sending packets in UDP

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

- *udp\_sendpage() invokes udp\_sendmsg().*
  - *udp\_sendpage() will work only if the nic supports Scatter/Gather (NETIF\_F\_SG feature is supported).*

# Example – udp client

```
#include <stdio.h>
```

```
#include <arpa/inet.h>
```

```
#include <sys/socket.h>
```

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    int s;
```

```
    struct sockaddr_in target;
```

```
    int res;
```

```
    char buf[10];
```

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

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

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

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

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

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

- If you try *connect()* on an unbound UDP socket and then *bind()* you will also get the EINVAL error. The reason is that connecting to an unbound socket will call *inet\_autobind()* to automatically bind an unbound socket (on a random port). So after *connect()*, the socket is bounded. And the calling *bind()* again will fail with EINVAL (since the socket is already bonded).
- Binding in the kernel for UDP is implemented in *inet\_bind()* and *inet\_autobind()*
  - (in IPV6: *inet6\_bind()* )

# Non local bind

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

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

*`int flag = 1;`*

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

*`perror("setsockopt");`*

# UDP socket options

- For **IPPROTO\_UDP/SOL\_UDP** level, we have two socket options:
- **UDP\_CORK** socket option.
  - Added in Linux kernel 2.5.44.

```
int state=1;
```

```
setsockopt(s, IPPROTO_UDP, UDP_CORK, &state,  
          sizeof(state));
```

```
for (j=1;j<1000;j++)
```

```
    sendto(s,buf1,...)
```

```
state=0;
```

```
setsockopt(s, IPPROTO_UDP, UDP_CORK, &state,  
          sizeof(state));
```

- The above code fragment will call *udp\_sendmsg()* 1000 times **without** actually sending anything on the wire (in the usual case, when without *setsockopt()* with UDP\_CORK, 1000 packets will be send).
- Only after the second *setsockopt()* is called, with UDP\_CORK and state=0, one packet is sent on the wire.
- Kernel implementation: when using UDP\_CORK, *udp\_sendmsg()* passes MSG\_MORE to *ip\_append\_data()*.

- Implementation detail: UDP\_CORK is not in glibc-header (/usr/include/netinet/udp.h); you need to add in your program:
  - #define UDP\_CORK 1
- UDP\_ENCAP socket option.
  - For usage with IPSEC.
    - Used, for example, in ipsec-tools.
    - Note: UDP\_ENCAP does not appear yet in the man page of udp (UDP\_CORK does appear).
- Note that there are other socket options at the SOL\_SOCKET level which you can get/set on UDP sockets: for example, SO\_NO\_CHECK (to disable checksum on UDP receive). (see Appendix E).

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

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

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

```
int recv_err(int s)
{
    int res;
    char cbuf[512];
    struct iovec iov;
    struct msghdr msg;
    struct cmsghdr *cmsg;
    struct sock_extended_err *e;
    struct icmphdr icmph;
    struct sockaddr_in target;
```

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

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

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

```
    printf("Destination port unreachable\n");
```

```
    }
```

```
    }
```

```
}
```

```
}
```

```
}
```

# udp\_sendmsg()

- *udp\_sendmsg*(struct kiocb \*iocb, struct sock \*sk, struct msghdr \*msg, size\_t len)
- Sanity checks in *udp\_sendmsg*:

- The destination UDP port must not be 0.
- If we try destination port of 0 we get EINVAL error as a return value of *udp\_sendmsg()*
  - The destination UDP is embedded inside the msghdr parameter (In fact, msg->msg\_name represents a sockaddr\_in; **sin\_port** is sockaddr\_in is the destination port number).
- MSG\_OOB is the only illegal flag for UDP. Returns EOPNOTSUPP error if such a flag is passed. (only permitted to SOCK\_STREAM)
- MSG\_OOB is also illegal in AF\_UNIX.

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

# Sending packets in UDP (contd)

- In case the socket is not connected, we should find a route to it; this is done by calling *ip\_route\_output\_flow()*.
- In case it is connected, we use the route from the sock (*sk\_dst\_cache* member of *sk*, which is an instance of *dst\_entry*).
  - When the *connect()* system call was invoked, *ip4\_datagram\_connect()* find the route by *ip\_route\_connect()* and set *sk->sk\_dst\_cache* in *sk\_dst\_set()*
- Moving the packet to Layer 3 (IP layer) is done by *ip\_append\_data()*.

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

- *ip\_queue\_xmit()* is (naturally) a simpler method.
- Basically what the *udp\_sendmsg()* method does is:
- Finds the route for the packet by *ip\_route\_output\_flow()*
- Sends the packet with *ip\_local\_out(skb)*

# Asynchronous I/O

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

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

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

```
struct sockaddr_in source;
```

```
struct sigaction handler;
```

```
source.sin_family = AF_INET;
```

```
source.sin_port = htons(888);
```

```
source.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
servSocket = socket(AF_INET, SOCK_DGRAM, 0);
```

```
bind(servSocket, (struct sockaddr*)&source, sizeof(struct  
    sockaddr_in));
```

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

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

# Appendix B : Socket options

- Socket options by protocol:

**IP protocol (SOL\_IP) 19 socket options:**

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

Note: For BSD compatibility there is IP\_RECVRETOPTS (which is identical to IP\_RETOPTS).

- AF\_UNIX:
  - SO\_PASSCRED for AF\_UNIX sockets.
  - Note:For historical reasons these socket options are specified with a SOL\_SOCKET type even though they are PF\_UNIX specific.
- UDP:
  - UDP\_CORK (IPPROTO\_UDP level).
- RAW:
  - ICMP\_FILTER
- TCP:
  - TCP\_CORK
  - TCP\_DEFER\_ACCEPT
  - TCP\_INFO
  - TCP\_KEEPCNT

- TCP\_KEEPIIDLE
  - TCP\_KEEPINTVL
  - TCP\_LINGER2
  - TCP\_MAXSEG
  - TCP\_NODELAY
  - TCP\_QUICKACK
  - TCP\_SYNCNT
  - TCP\_WINDOW\_CLAMP
- AF\_PACKET
  - PACKET\_ADD\_MEMBERSHIP
  - PACKET\_DROP\_MEMBERSHIP

## **Socket options for socket level:**

SO\_DEBUG

SO\_REUSEADDR

SO\_TYPE

SO\_ERROR

SO\_DONTROUTE

SO\_BROADCAST

SO\_SNDBUF

SO\_RCVBUF

SO\_SNDBUFFORCE

SO\_RCVBUFFORCE

SO\_KEEPALIVE

SO\_OOBINLINE

SO\_NO\_CHECK

SO\_PRIORITY

SO\_LINGER

SO\_BSDCOMPAT

# Appendix C: tcp client

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

```
int main()
```

```
{
```

# tcp client - contd.

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

# tcp client - contd.

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

- You can send data on this socket by adding, for example:

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

```
int length;
```

```
length = strlen(message)+1;
```

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

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

# Appendix D : ICMP options

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

ICMP\_ECHOREPLY

ICMP\_DEST\_UNREACH

ICMP\_SOURCE\_QUENCH

ICMP\_REDIRECT

ICMP\_ECHO

ICMP\_TIME\_EXCEEDED

ICMP\_PARAMETERPROB

ICMP\_TIMESTAMP

ICMP\_TIMESTAMPREPLY

ICMP\_INFO\_REQUEST

ICMP\_INFO\_REPLY

ICMP\_ADDRESS

ICMP\_ADDRESSREPLY

# APPENDIX E: flags for send/receive

MSG\_OOB

MSG\_PEEK

MSG\_DONTROUTE

MSG\_TRYHARD - Synonym for MSG\_DONTROUTE for DECnet

MSG\_CTRUNC

MSG\_PROBE - Do not send. Only probe path f.e. for MTU

MSG\_TRUNC

MSG\_DONTWAIT - Nonblocking io

MSG\_EOR - End of record

MSG\_WAITALL - Wait for a full request

MSG\_FIN

MSG\_SYN

MSG\_CONFIRM - Confirm path validity

MSG\_RST

MSG\_ERRQUEUE - Fetch message from error queue

MSG\_NOSIGNAL - Do not generate SIGPIPE

MSG\_MORE 0x8000 - Sender will send more.

# Example: set and get an option

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

```
#include <stdio.h>
```

```
#include <arpa/inet.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#include <string.h>
```

```
int main()
```

```
{
```

```
int s;
```

```
int opt;
```

```
int res;
```

```
int one = 1;
```

```
int size = sizeof(opt);
```

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

# Example: record route option

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

```
#define NROUTES 9
```

```
int main()
```

```
{
```

```
int s;
```

```
int optlen=0;
```

```
struct sockaddr_in target;
```

```
int res;
```

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

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

# APPENDIX F: UDP errors

Running :

```
cat /proc/net/snmp | grep Udp:
```

will give something like:

```
Udp: InDatagrams NoPorts InErrors OutDatagrams RcvbufErrors  
     SndbufErrors
```

```
Udp: 2625 1 0 2100 0 0
```

**InErrors** - (UDP\_MIB\_INERRORS)

**RcvbufErrors** – UDP\_MIB\_RCVBUFEERRORS:

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

**SndbufErrors** – (UDP\_MIB\_SNDBUFEERRORS)

- Incremented in `udp_sendmsg()`

- Another metric:
  - cat /proc/net/udp
  - The last column in: drops
    - Represents sk->sk\_drops.
    - Incremented in \_\_udp\_queue\_rcv\_skb()
      - net/ipv4/udp.c
- When do RcvbufErrors occur ?
  - The total number of bytes queued in sk\_receive\_queue queue of a socket is sk->sk\_rmem\_alloc.
  - The total allowed memory of a socket is sk->sk\_rcvbuf.
    - It can be retrieved with getsockopt() using SO\_RCVBUF.

- Each time a packet is received, the `sk->sk_rmem_alloc` is incremented by `skb->truesize`:
  - `skb->truesize` is the size (in bytes) allocated for the data of the `skb` plus the size of `sk_buff` structure itself.
  - This incrementation is done in `skb_set_owner_r()`
    - ...
    - `atomic_add(skb->truesize, &sk->sk_rmem_alloc);`
    - ...
  - see: `include/net/sock.h`

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

...

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

...

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

```
if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
    (unsigned)sk->sk_rcvbuf) {
    err = -ENOMEM;
```

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

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

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

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

...

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

...

- If it is true, everything is fine.
- If not, we end with setting **SOCK\_ASYNC\_NOSPACE** and **SOCK\_NOSPACE** flags of the socket:

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

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

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

...

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

...

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

*`sock_wfree()`*

*...*

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

*...*

# Tips

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

# END

- - Thank you!
  -
- [ramirose@gmail.com](mailto:ramirose@gmail.com)



# Linux Kernel Networking – advanced topics (6)

## Sockets in the kernel

Rami Rosen

[ramirose@gmail.com](mailto:ramirose@gmail.com)

Haifux, August 2009

[www.haifux.org](http://www.haifux.org)

All rights reserved.



# Linux Kernel Networking (6)- advanced topics

Note:

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

## **1) Linux Kernel Networking lecture**

<http://www.haifux.org/lectures/172/>

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

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

<http://www.haifux.org/lectures/180/>

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

# Linux Kernel Networking (6)- advanced topics

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

<http://www.haifux.org/lectures/187/>

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

## 4) Wireless in Linux

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

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

## 5) Sockets in the Linux Kernel

<http://www.haifux.org/lectures/217/>

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

# Note

Note: This is the second part of the “Sockets in the Linux Kernel” lecture which was given in Haifux in 27.7.09. You may find some background material for this lecture in its slides:

<http://www.haifux.org/lectures/217/netLec5.pdf>

# TOC

TOC:

RAW Sockets

UNIX Domain Sockets

Netlink sockets

SCTP sockets.

Appendices

Note: All code examples in this lecture refer to the recent **2.6.30** version of the Linux kernel.

# RAW Sockets

There are cases when there is no interface to create sockets of a certain protocol (ICMP protocol, NETLINK protocol) => use Raw sockets.

raw socket creation is done thus, for example:

```
sd = socket(AF_INET, SOCK_RAW, 0);
```

```
sd = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);
```

```
sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
```

ETH\_P\_IP tells to handle all IP packets.

When using AF\_INET family, as in the first two cases, the socket is added to kernel RAW sockets hash table (the hash key is the protocol number). This is done by `raw_hash_sk()`, (net/ipv4/raw.c), which is invoked by `inet_create()`, when creating the socket.

When using AF\_PACKET family, as in the third case, a socket is **not** added to the kernel RAW sockets hash table.

See Appendix F for an example of using packet raw socket.

Raw socket creation **MUST** be done as a super user.

In case an ordinary user try to create a raw socket, you will get:

“error: socket: Operation not permitted.” (**EPERM**).

You can set the CAP\_NET\_RAW capability to enable non root users to create raw sockets:

```
setcap cap_net_raw=+ep rawserver
```

# Usage of RAW socket: ping

You do not specify ports with RAW sockets; RAW sockets do not work with ports.

When the kernel receives a raw packet, it delivers it to all raw sockets.

Ping in fact is sending an ICMP packet.

The type of this ICMP packet is **ICMP ECHO REQUEST**.

# Send a ping implementation(simplified)

```
#define BUFSIZE 1500

char sendbuf[BUFSIZE];

struct icmp *icmp;

int sockfd;

struct sockaddr_in target;

int datalen=56;


target.sin_family = AF_INET;

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

icmp = (struct icmp *)sendbuf;

icmp->icmp_type = ICMP_ECHO;

icmp->icmp_code = 0;

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

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

```
icmp->icmp_cksum=0;
```

```
sockfd=socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

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

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

# Receive a ping- implementation(simplified)

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

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

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

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

The only SOL\_RAW option a Raw socket can get is ICMP\_FILTER.

This can be done thus:

```
#define ICMP_FILTER 1
```

```
struct icmp_filter {
```

```
    __u32 data;
```

```
};
```

```
filt.data = 1 << ICMP_DEST_UNREACH;
```

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

Adding this code in the receive Ping application above will prevent **Destination Unreachable** ICMP messages from received in user space by `recvmsg`.

There are quite a lot more ICMP options; by default, we do NOT filter any ICMP messages.

Among the other options you can set by `setsockopt` are:

ICMP\_ECHO (echo request)

ICMP\_ECHOREPLY (echo reply)

ICMP\_TIME\_EXCEEDED

And more (see Appendix D for a full list).

**Traceroute** also uses raw sockets.

Traceroute changes the TTL field in the ip header.

This is done by IP\_TTL and control messages in current Linux traceroute implementation (Dmitry Butskoy).

In the original traceroute (by Van Jacobson) it was done with the **IP\_HDRINCL** socket option:

(setsockopt(sndsock, IPPROTO\_IP, IP\_HDRINCL,...))

The `IP_HDRINCL` tells the IP layer **not** to prepare an IP header when sending a packet.

`IP_HDRINCL` is also applicable in IPV6.

When **receiving** a packet, the IP header is always included in the packet.

When **sending** a packet, by specifying the the `IP_HDRINCL` option you tell the kernel that the IP header is already included in the packet, so no need to prepare it in the kernel.

*`raw_send_hdrinc()` in `net/ipv4/raw.c`*

*The `IP_HDRINCL` option is applied only to the `SOCK_RAW` type of protocol.*

See Lawrence Berkeley National Laboratory traceroute:

<ftp://ftp.ee.lbl.gov/traceroute.tar.gz>

If a raw socket was created with protocol type of IPPROTO\_RAW , this implies enabling IP\_HDRINCL:

Thus, this call from user space:

```
socket(AF_INET,SOCK_RAW,IPPROTO_RAW)
```

invokes this code in the kernel:

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

(From inet\_create(), net/ipv4//af\_inet.c)

**Spoofing attack:** setting the IP address of packets to be different than the real ones.

UDP spoofing is easier since UDP is connectionless.

Following is an example of UDP spoofing with raw sockets and IP\_HDRINCL option:

We build an IP header.

We set the protocol field in this ip header to IP\_PROTOUDP.

We build a UDP header.

Note : when behind a NAT, this probably will not work

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

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

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

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

Note: what will happen if we specify an illegal source address, like “255.255.255.255”?

The packet will be sent.

If we want to log such packets on the receiver side, (to detect spoofing attempts), we must set the **log\_martians** kernel tunable thus:

```
echo "1" > /proc/sys/net/ipv4/conf/all/log_martians
```

Then we will see in the kernel syslog messages like this:

```
martian source 82.80.80.193 from 255.255.255.255, on  
dev eth0
```

Following will be the **ethernet header**:

II header:

# Raw sockets and sniffers

When you activate tshark (formerly tethereal) or wireshark or tcpdump, you call the *pcap\_open\_live()* method of the pcap library.

This method creates a raw socket thus:

```
socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))
```

*pcap\_open\_live() is implemented in* libpcap-0.9.8/pcap-linux.c.

PF\_PACKET sockets work with the network interface card.

Note:

When you open tshark thus:

**tshark -i any**

Then the socket is opened thus:

```
socket(PF_PACKET, SOCK_DGRAM,  
        htons(ETH_P_ALL))
```

This is called “cooked mode”

SLL. (Socket Link Layer).

With **SOCK\_DGRAM**, the kernel is responsible for adding ethernet header (when sending a packet) or removing ethernet header (when receiving a packet).

With SOCK\_RAW, the application is responsible for adding an ethernet header when sending.

Also you will get this message:

“Capturing on Pseudo-device that captures on all interfaces”

tshark: Promiscuous mode not supported on the "any" device

# Unix Domain Sockets

`AF_UNIX / PF_UNIX / AF_LOCAL / PF_LOCAL`.

A way for interprocess communication. (IPC)

the client and server are on the same host.

`AF_UNIX` sockets can be either `SOCK_STREAM` or `SOCK_DGRAM`.

And, since kernel 2.6.4, also `SOCK_SEQPACKET`.

Usage: in `rsyslogd(AF_UNIX/SOCK_DGRAM)` and `udev` daemons (`AF_LOCAL/SOCK_DGRAM`), `hald`, `crond`, and a lot more.

Unix domain sockets do not support the transmission of out-of-band data.

MSG\_OOB is not supported at all in Unix domain sockets

This applies For all 3 types,  
SOCK\_STREAM, SOCK\_DGRAM and SOCK\_SEQPACKET.

Usually uses files in the local filesystem.

Abstract namespaces.

Why not extend it to use between domains in virtualization which have access to shared filesystem ?

With rsyslogd, the path is under /dev:

`ls -al /dev/log`

```
srw-rw-rw- 1 root root 0 01-07-09 13:17 /dev/log
```

Notice the 's' in the beginning => for socket.

`ls -F /dev/log`

```
/dev/log=
```

(with ls, -F is for appending indicator to entries)

# Unix Domain Socket server Example

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

Calling *bind()* in the example above will create a file named `/work/test_unix`

```
ls -al /work/test_unix
```

```
srwxr-xr-x
```

Notice the “s” for socket.

Notice that with DGRAM Unix domain sockets, calling `sendto()` without calling `bind()` before, will **not call `autobind()` as opposed to what happens in udp under the same scenario.**

**In this case, the receiver cannot reply (because it does not know to who).**

**lsnf -U** : shows Unix domain sockets

Also: **netstat --unix -all**

Tip: use **netstat -ax** for short.

[ACC] in the third column means that the socket is  
unconnected and waiting for connection.  
(SO\_ACCEPTON).

And also:

**cat /proc/net/protocols | grep UNIX**

**cat /proc/net/unix**

**struct sockaddr\_un (/usr/include/linux/un.h)**

The pathname for a Unix domain socket should be an absolute pathname.

For abstract namespaces:

```
address.sun_path[0] = 0
```

The last column of `netstat --unix --all` is the path.

In case of abstract namespace, it will begin with `@`:

```
netstat --unix --all | grep udevd
```

```
unix 2      []          DGRAM          602  
    @/org/kernel/udev/udev
```

## Control messages in Unix domain sockets:

**SCM\_RIGHTS** - You can pass an open file descriptor from one process to another using Unix domain socket and control messages (ancillary data).

**SCM\_CREDENTIALS**- for passing process credentials (uid and gid).

You need to set the **SO\_PASSCRED** socket option with *setsockopt()* on the receiving side.

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

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

kernel: in include/linux/socket.h:

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

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

# Unix domain client example

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

If we will try to call `send()` in a stream-oriented socket after the stream-oriented server was closed, we will get `EPIPE` error:

`send: Broken pipe`

The kernel also sends the user space a `SIGPIPE` signal in this case.

In case the flags parameter in `send()` is `MSG_NOSIGNAL`, the kernel does NOT send a `SIGPIPE` signal.

In BSD, you can avoid signals by `setsockopt()` with `SO_NOSIGPIPE` (`SOL_SOCKET` option).

In IPV4, the only signal used is SIGURG for OOB in tcp.

In case of datagram-oriented sockets, SIGPIPE is not sent; we just get connection refused error.

If, in the above example, we tried to create a dgram client instead of stream client, thus;

```
socket_fd = socket(PF_LOCAL, SOCK_DGRAM, 0);
```

We would get:

connect: Protocol wrong type for socket (EPROTOTYPE)

see: *unix\_find\_other()*

The *socketpair()* system call:

Creates a pair of connected sockets.

On Linux, the only supported domain for this call is **AF\_UNIX** (or synonymously, **AF\_LOCAL**).

# Netlink sockets

**Netlink sockets:** a message mechanism from user-space to kernel and also between kernel ingredients.

Used widely in the kernel; mostly in networking, but also in other subsystems.

There are other mechanism for communication from user space to kernel:

- ioctl (drivers)

- /proc or /sys entries (VFS)

And there are of course signals from kernel to user space (like SIGIO, and more).

Creating netlink sockets is done (in the kernel) by *netlink\_kernel\_create()*.

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

```
static int rtnetlink_net_init(struct net *net)
{
    struct sock *sk;

    sk = netlink_kernel_create(net, NETLINK_ROUTE,
                               RTNLGRP_MAX, rtnetlink_rcv, &rtnl_mutex,
                               THIS_MODULE);
```

With generic netlink sockets, this is done using the NETLINK\_GENERIC protocol thus:

```
netlink_kernel_create(&init_net, NETLINK_GENERIC, 0,  
    genl_rcv, &genl_mutex, THIS_MODULE);
```

*See net/netlink/genetlink.c*

*The second parameter, [NETLINK\\_ROUTE](#), is the protocol. (kernel 2.6.30).*

There are currently 19 netlink protocols in the kernel:

<a href="#">NETLINK_ROUTE</a>	NETLINK_UNUSED	NETLINK_USERSOCK
<a href="#">NETLINK_FIREWALL</a>	NETLINK_INET_DIAG	NETLINK_NFLOG
NETLINK_XFRM	NETLINK_SELINUX	NETLINK_ISCSI
NETLINK_AUDIT	NETLINK_FIB_LOOKUP	NETLINK_CONNECTOR
NETLINK_NETFILTER	NETLINK_IP6_FW	NETLINK_DNRTMSG
NETLINK_KOBJECT_UEVENT	<a href="#">NETLINK_GENERIC</a>	NETLINK_SCSITRANSPORT
NETLINK_ECRYPTFS		

*(see `include/linux/netlink.h`).*

The fourth parameter, *rtnetlink\_rcv*, is the handler for netlink packets.

*rtnetlink\_rcv()* gets a packet (*sk\_buff*) as its parameter.

*NETLINK\_ROUTE* messages are not confined to the routing subsystem; they include also other types of messages (for example, neighboring)

*NETLINK\_ROUTE* messages can be divided into families. Most of these families has three types of messages. (New, Del and Get).

*For example:*

*RTM\_NEWROUTE – create a new route.*

*Handled by [inet\\_rtm\\_newroute\(\)](#).*

*RTM\_DELROUTE - delete a route.*

*Handled by [inet\\_rtm\\_delroute\(\)](#).*

*RTM\_GETROUTE – retrieve information about a route.*

*Handled by [inet\\_dump\\_fib\(\)](#).*

*All three methods are in net/ipv4/fib\_frontend.c.*

Another family of METLINK\_ROUTE is the NEIGH family:

RTM\_NEWNEIGH

RTM\_DELNEIGH

RTM\_GETNEIGH

How do these messages reach these handlers?

Registration is done by calling *rtnl\_register()*

in *ip\_fib\_init()*:

```
rtnl_register(PF_INET, RTM_NEWROUTE,  
    inet_rtm_newroute, NULL);
```

```
rtnl_register(PF_INET, RTM_DELROUTE,  
    inet_rtm_delroute, NULL);
```

```
rtnl_register(PF_INET, RTM_GETROUTE, NULL,  
    inet_dump_fib);
```

IPROUTE2 package is based on rtnetlink.

(IPROUTE2 is “ip” with subcommands, for example: *ip route show* to show the routing tables)

IPROUTE2 uses the libnetlink library.

See libnetlink.h (in the IPROUTE2 library)

*rtnl\_open()* to open a socket in user space.

*rtnl\_send()* to send a message to the kernel.

*rtnl\_open()* calls the `socket()` system call to create an rtnetlink socket:

```
socket(AF_NETLINK, SOCK_RAW, protocol);
```

*rtnl\_listen()* starts receiving messages by calling the *recvmsg()* system call.

*The AF\_NETLINK protocol is implemented in net/netlink/af\_netlink.c.*

*AF\_ROUTE is a synonym of AF\_NETLINK (due to BSD)*

```
#define AF_ROUTE AF_NETLINK (include/linux/socket.h)
```

*The kernel holds an array called *nl\_table*; it has up to 32 elements. (MAX\_LINKS).*

*Each element in this table corresponds to a protocol (in fact, the protocol is the index)*

# Example

```
#include "libnetlink.h"

int accept_msg(const struct sockaddr_nl *who, struct nlmsghdr *n, void *arg) {
    if (n->nmsg_type == RTM_NEWROUTE)
        printf("got RTM_NEWROUTE message \n");
}

int main() {
    int res;
    struct rtnl_handle rth;
    unsigned int groups = ~RTMGRP_TC | RTNLGRP_IPV4_ROUTE;
    if (rtnl_open(&rth, groups) < 0) {
        printf("rtnl_open() failed in %s %s\n", __FUNCTION__, __FILE__);
        return -1;
    }
}
```

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

Adding a route will be logged to stdout:

`ip route add 10.0.0.10 via 10.10.10.11`

will print:

`got RTM_NEWROUTE message`

- In this case, the `rtnl_open()` invokes

`socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);`

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

# Generic Netlink

The iw tools (wireless user space management) use the Generic Netlink API.

This API is based on Netlink sockets.

You register handlers in *nl80211\_init()*

*net/wireless/nl80211.c*

For example, for wireless **interfaces** we have these messages and handlers:

NL80211\_CMD\_GET\_INTERFACE

Handled by `nl80211_dump_interface()`

NL80211\_CMD\_SET\_INTERFACE

Handled by `nl80211_set_interface()`

NL80211\_CMD\_NEW\_INTERFACE

Handled by `nl80211_new_interface()`

NL80211\_CMD\_DEL\_INTERFACE

Handled by `nl80211_del_interface()`

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

See appendix A.

You can use the [NETLINK\\_FIREWALL](#) protocol for a netlink socket to catch packets in user space with the help of an iptables kernel module named ip\_queue.ko.

```
iptables -A OUTPUT -p UDP --dport 9999 -j  
NFQUEUE --queue-num 0
```

The user space application uses libnetfilter\_queue-0.0.17 API (which replaced the libipq lib).

Netlink sockets usage: xorp, (routing daemons: <http://www.xorp.org/>) , iproute2, iw.

# SCTP

## General:

Combines features of TCP and UDP.

Reliable (like TCP).

RFC 4960 (obsoletes RFC 2960).

Target: VoIP, telecommunications.

## People:

Randall Stewart (Cisco): co inventor, FreeBSD.

Peter Lei (Cisco)

Michael Tuxen (MacOS).

## Linux Kernel SCTP Maintainers:

Vlad Yasevich (HP)

Sridhar Samudrala (IBM).

SCTP support in the Linux kernel tree is from versions 2.5.36 and following.

Location in the kernel tree: net/sctp.

# SCTP

There are two types of SCTP sockets:

## **One to one socket**

`socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)`

Much like TCP connection.

## **One to many socket**

`socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)`

See for example, here:

[http://heim.ifi.uio.no/michawe/teaching/dipls/stefan\\_joerer.pdf](http://heim.ifi.uio.no/michawe/teaching/dipls/stefan_joerer.pdf)

- Much like UDP server with many clients.

You need to have lksctp-tools to use SCTP in userspace applications.

<http://lksctp.sourceforge.net>

In fedora,

lksctp-tools rpm.

lksctp-tools-devel rpm. (for /usr/include/netinet/sctp.h)

# Future lectures

Netfilter kernel implementation:

- NAT and connection tracking; dn timer, snat.

- MASQUERADING.

- Filter and mangle tables.

- Netfilter verdicts.

- The new generation: nftables

Network namespaces (Containers / OpenVZ).

DCCP

Virtio

IPVS/LVS (Linux Virtual Server).

Bluetooth, RFCOMM.

Multiqueues.

LRO (Large Receive Offload)

Multicasting.

TCP protocol.

# Appendix A : wireless messages

NL80211\_CMD\_GET\_WIPHY, NL80211\_CMD\_SET\_WIPHY,

NL80211\_CMD\_GET\_INTERFACE, NL80211\_CMD\_SET\_INTERFACE,  
NL80211\_CMD\_NEW\_INTERFACE, NL80211\_CMD\_DEL\_INTERFACE,

NL80211\_CMD\_GET\_KEY, NL80211\_CMD\_SET\_KEY, NL80211\_CMD\_NEW\_KEY, NL80211\_CMD\_DEL\_KEY,

NL80211\_CMD\_SET\_BEACON, NL80211\_CMD\_NEW\_BEACON, NL80211\_CMD\_DEL\_BEACON,

NL80211\_CMD\_GET\_STATION, NL80211\_CMD\_SET\_STATION, NL80211\_CMD\_NEW\_STATION,  
NL80211\_CMD\_DEL\_STATION,

NL80211\_CMD\_GET\_MPATH, NL80211\_CMD\_SET\_MPATH, NL80211\_CMD\_NEW\_MPATH, NL80211\_CMD\_DEL\_MPATH,

NL80211\_CMD\_SET\_BSS, NL80211\_CMD\_GET\_REG,

NL80211\_CMD\_SET\_REG, NL80211\_CMD\_REQ\_SET\_REG,

NL80211\_CMD\_GET\_MESH\_PARAMS, NL80211\_CMD\_SET\_MESH\_PARAMS,

NL80211\_CMD\_TRIGGER\_SCAN, NL80211\_CMD\_GET\_SCAN,

NL80211\_CMD\_AUTHENTICATE, NL80211\_CMD\_ASSOCIATE, NL80211\_CMD\_DEAUTHENTICATE,  
NL80211\_CMD\_DISASSOCIATE,

NL80211\_CMD\_JOIN\_IBSS, NL80211\_CMD\_LEAVE\_IBSS,

# Appendix B : Socket options

## Socket options by protocol:

### **IP protocol (SOL\_IP) 19 socket options:**

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

Note: For BSD compatibility there is IP\_RECVRETOPTS (which is identical to IP\_RETOPTS).

AF\_UNIX:

SO\_PASSCRED for AF\_UNIX sockets.

Note: For historical reasons these socket options are specified with a SOL\_SOCKET type even though they are PF\_UNIX specific.

UDP:

UDP\_CORK (IPPROTO\_UDP level).

RAW:

ICMP\_FILTER

TCP:

TCP\_CORK

TCP\_DEFER\_ACCEPT

TCP\_INFO

TCP\_KEEPCNT

TCP\_KEEPIDLE

TCP\_KEEPINTVL

TCP\_LINGER2

TCP\_MAXSEG

TCP\_NODELAY

TCP\_QUICKACK

TCP\_SYNCNT

TCP\_WINDOW\_CLAMP

AF\_PACKET

PACKET\_ADD\_MEMBERSHIP

PACKET\_DROP\_MEMBERSHIP

## **Socket options for socket level:**

SO\_DEBUG

SO\_REUSEADDR

SO\_TYPE

SO\_ERROR

SO\_DONTROUTE

SO\_BROADCAST

SO\_SNDBUF

SO\_RCVBUF

SO\_SNDBUFFORCE

SO\_RCVBUFFORCE

SO\_KEEPALIVE

SO\_OOBINLINE

SO\_NO\_CHECK

SO\_PRIORITY

SO\_LINGER

SO\_BSDCOMPAT

# Appendix C: tcp client

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

```
int main()
```

```
{
```

# tcp client - contd.

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

# tcp client - contd.

If on the other side (192.168.0.121 in this example) there is no TCP server listening on this port (853) you will get this error for the socket() system call:

connect: Connection refused.

You can send data on this socket by adding, for example:

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

```
int length;
```

```
length = strlen(message)+1;
```

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

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

# Appendix D : ICMP options

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

ICMP\_ECHOREPLY

ICMP\_DEST\_UNREACH

ICMP\_SOURCE\_QUENCH

ICMP\_REDIRECT

ICMP\_ECHO

ICMP\_TIME\_EXCEEDED

ICMP\_PARAMETERPROB

ICMP\_TIMESTAMP

ICMP\_TIMESTAMPREPLY

ICMP\_INFO\_REQUEST

ICMP\_INFO\_REPLY

ICMP\_ADDRESS

ICMP\_ADDRESSREPLY

# APPENDIX E: flags for send/receive

MSG\_OOB

MSG\_PEEK

MSG\_DONTROUTE

MSG\_TRYHARD - Synonym for MSG\_DONTROUTE for DECnet

MSG\_CTRUNC

MSG\_PROBE - Do not send. Only probe path f.e. for MTU

MSG\_TRUNC

MSG\_DONTWAIT - Nonblocking io

MSG\_EOR - End of record

MSG\_WAITALL - Wait for a full request

MSG\_FIN

MSG\_SYN

MSG\_CONFIRM - Confirm path validity

MSG\_RST

MSG\_ERRQUEUE - Fetch message from error queue

MSG\_NOSIGNAL - Do not generate SIGPIPE

MSG\_MORE0x8000 - Sender will send more.

# Example: set and get an option

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

```
#include <stdio.h>
```

```
#include <arpa/inet.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#include <string.h>
```

```
int main()
```

```
{
```

```
int s;
```

```
int opt;
```

```
int res;
```

```
int one = 1;
```

```
int size = sizeof(opt);
```

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

# Example: record route option

This example shows how to send a record route option.

```
#define NROUTES 9
```

```
int main()
```

```
{
```

```
int s;
```

```
int optlen=0;
```

```
struct sockaddr_in target;
```

```
int res;
```

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

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

# Appendix F: using packet raw socket

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

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

# Tips

To find out socket used by a process:

```
ls -l /proc/[pid]/fd|grep socket|cut -d: -f3|sed 's/\[//;s/\]//'
```

The number returned is the inode number of the socket.

Information about these sockets can be obtained from

```
netstat -ae
```

After starting a process which creates a socket, you can see that the inode cache was incremented by one by:

```
more /proc/slabinfo | grep sock
```

```
sock_inode_cache    476    485    768    5    1 : tunables    0    0
  0 : slabdata    97    97    0
```

The first number, 476, is the number of active objects.

END

Thank you!



ramirose@gmail.com