montavista VISION 2007

EMBEDDED LINUX DEVELOPERS CONFERENCE

Advanced Networking:
From the Inside Looking Out

Mark Huth

Network Architect

montavista

Introduction

- Linux™ Advanced Networking
  - Assumes you have all coded up some version of an application using sockets
  - Know AF_INET from AF_INET6
  - Streams from Datagrams
  - Own at least one volume of Stevens Networking tomes
  - Scoured it looking for enlightenment
    - Only to be confused by SO_REUSEADDR
- Not Unix System V™, NetBSD™, FreeBSD™, Solaris™, Windows Vista™
- Been flamed by David Miller

What Linux Networking Provides

1. Large selection of common protocols
2. In-kernel packet forwarding
3. Comprehensive Firewalling/Filtering
4. In-kernel Authentication and Encryption
5. Extensible Stack via Kernel Modules
6. QOS – Differentiated and Integrated/RSVP
7. Traffic Shaping and Policing for Egress and Ingress
8. Packet Debugging and Statistics
9. Tuning and Behavioral Controls
10. Load Management via NAPI and RT-Preemption
11. Logging
12. User space Extensions
13. And more …

Advanced Networking with Linux

Goal –

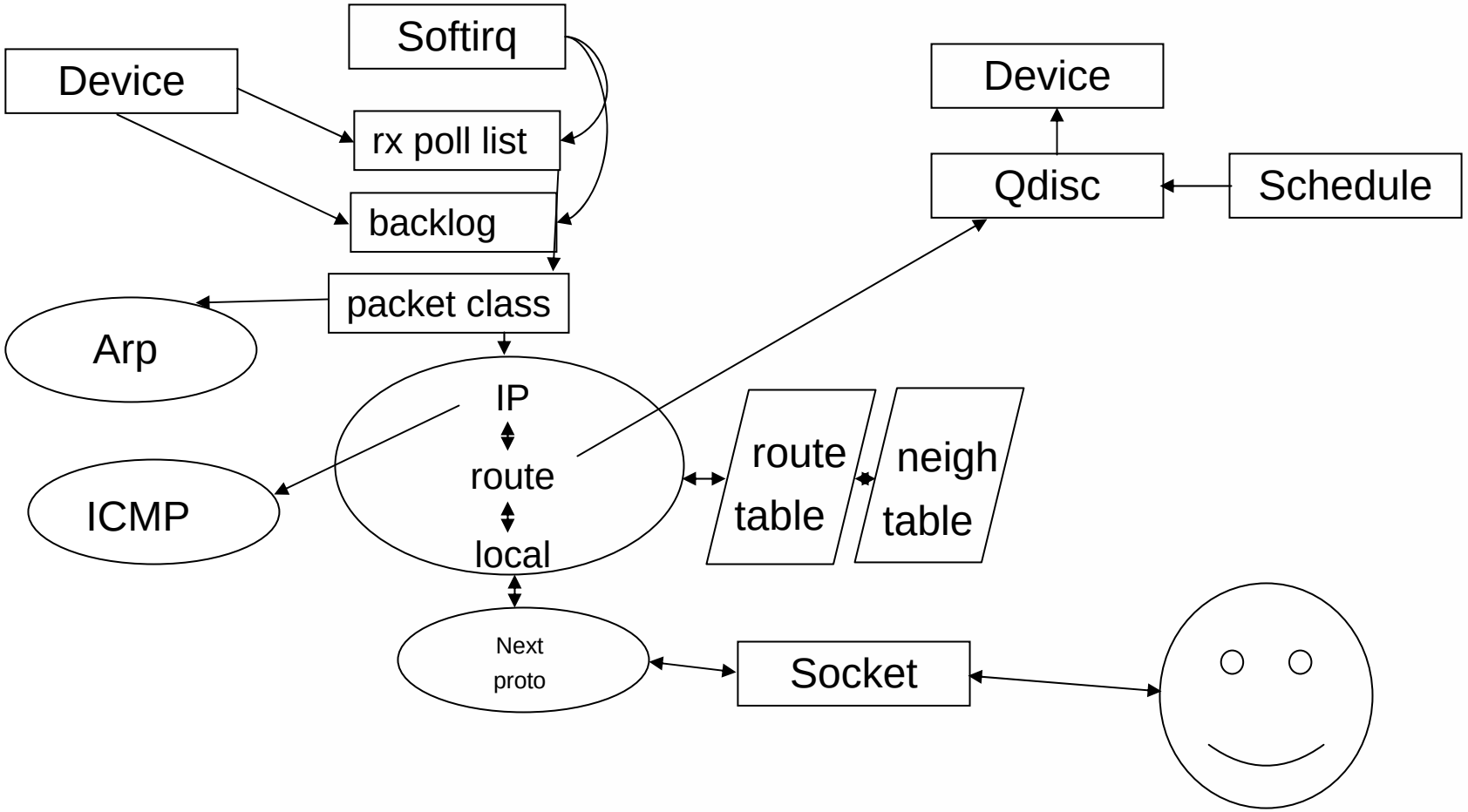
Give network application developers a place to start when doing custom applications after it starts to look like some custom kernel work is required.

Approach

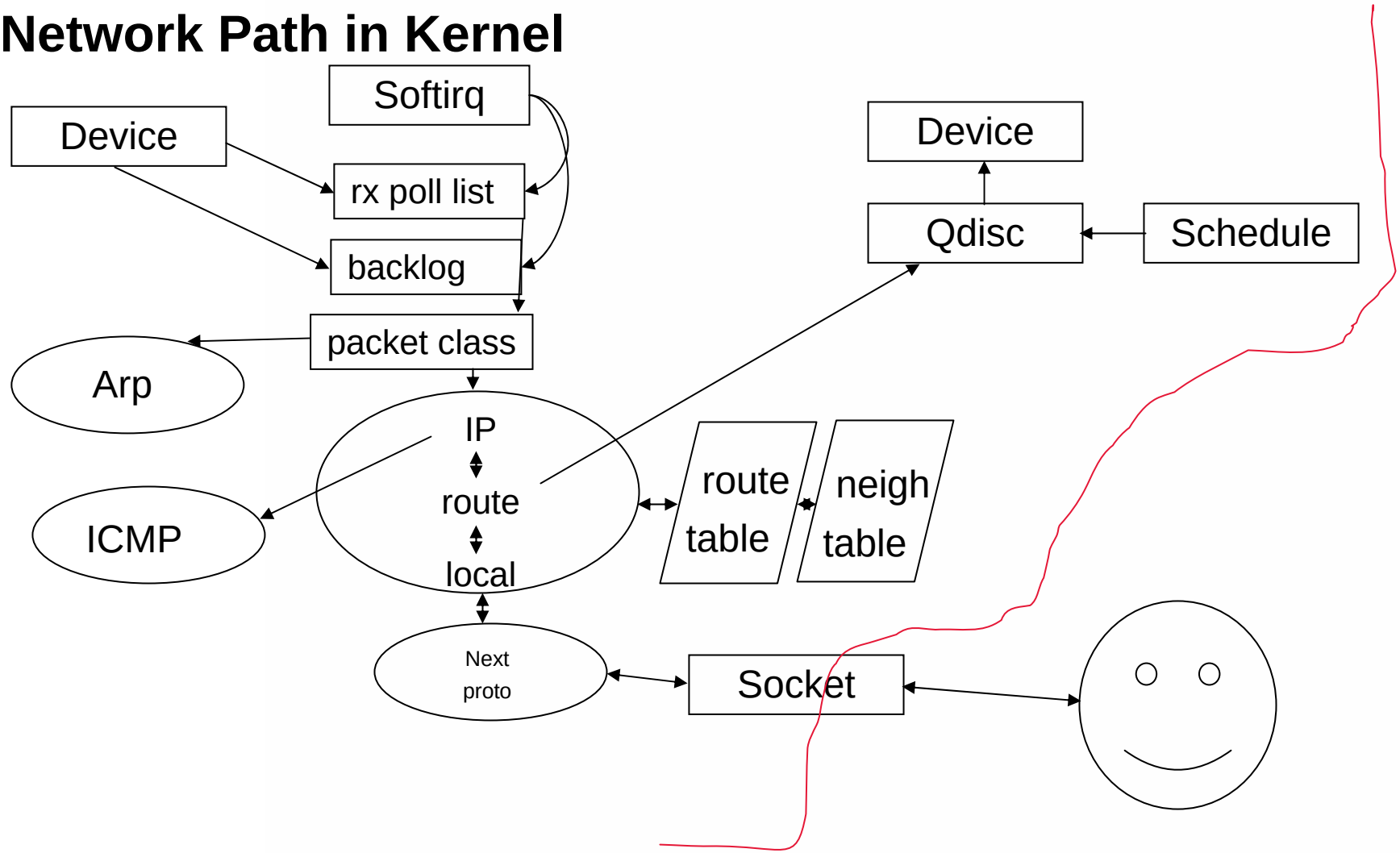
- Take a fairly detailed look at the packet path from interface to interface
- Do an introduction to the Netlink API
- Routing Tables in Detail

Will not teach you to program or how to do kernel work

Network Path in Kernel



Network Path in Kernel



Network Path in Kernel: Media Access/Data Link

Input Actions: Non-NAPI

Device Interrupt routine

- Does link level sanity checks
- Separates the Link Layer type
- Passes packet to the backlog via netif_rx(skb)
- Performs other interrupt functions – e.g. Error recovery, TX completion

Input Actions: NAPI

Device ISR

- schedules the device poll routine on the poll queue
- may handle other interrupt sources
- disables at least the rx interrupt

Device Poll Routine

- Performs device receive processing as in Non-NAPI case
- But calls netif_receive_skb(skb) since already in softirq
- May do tx completion

Network Path in Kernel: Media Access/Data Link

```
void __netif_rx_schedule(struct net_device *dev)
{
        unsigned long flags;

        local_irq_save(flags);
        dev_hold(dev);
        list_add_tail(&dev->poll_list,
             &__get_cpu_var(softnet_data).poll_list);
        if (dev->quota < 0)
                dev->quota += dev->weight;
        else
                dev->quota = dev->weight;
        raise_softirq_irqoff(NET_RX_SOFTIRQ);
        local_irq_restore(flags);
}
```

Network Path in Kernel: Soft IRQ

- Per processor queue has the device poll callbacks
  - Called in sequence, with a budget for the device
  - Device callback completes in budget, leaves list, enables interrupt
  - Or is returned to the end of the poll list.
- Non-NAPI packets queued to pseudo-device; behaves similarly
- netif_receive_skb calls the receive method for the type (IP, ARP, etc.)
- SoftIRQ runs just before return from real irq
  - until done
  - or jiffy is incremented twice
- Also runs when BH re-enabled.
- Performance tradeoffs – NAPI vs. OldAPI

Network Path in Kernel: Soft IRQ

```
static void net_rx_action(struct softirq_action *h)
{
        struct softnet_data *queue;
        unsigned long start_time = jiffies;
        int budget = netdev_budget;
        void *have;

        local_irq_disable();
        queue = &__get_cpu_var(softnet_data);

        while (!list_empty(&queue->poll_list)) {
                struct net_device *dev;

                if (budget <= 0 || jiffies - start_time > 1)
                        goto softnet_break;

                local_irq_enable();
                if (unlikely(cond_resched_softirq_context())) {
                        local_irq_disable();
                        continue;
                }
```

Network Path in Kernel: Soft IRQ

```
dev = list_entry(queue->poll_list.next, struct net_device, poll_list);
have = netpoll_poll_lock(dev);

        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
                netpoll_poll_unlock(have);
                local_irq_disable();
                list_move_tail(&dev->poll_list, &queue->poll_list);
                if (dev->quota < 0)
                        dev->quota += dev->weight;
                else
                        dev->quota = dev->weight;
        } else {
                netpoll_poll_unlock(have);
                dev_put(dev);
                local_irq_disable();
        }
}
out:
```

Network Path in Kernel: IP Receive

- Input protocol checks and statistics done in ip_rcv
- Thence to ip_rcv_finish via the NF_IP_PRE_ROUTING hook
  - NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish);
  - Usually the skb->dst is NULL, resulting in a call to ip_route_input
  - It's possible that the netfilter module decided to divert the packet, in which case the skb->dst structure may be filled in to be executed immediately
- Once the destination is filled in, the destination input routine is called
  - IP options processed
  - dst->input is usually ip_local_deliver
- Fragments re-assembled – placed on hash list – parameters in /proc/… for controlling fraglists disorder, memory use and lifetime
- NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL, ip_local_deliver_finish);
  - Finds the upper layer protocol handler
  - Performs transforms, if needed

Network Path in Kernel: Input Routing

- ip_route_input
  - cache is randomized hash of destination and source address and interface, using a full key of (daddr, saddr, iif, (oif==0), mark, tos) under RCU lock
  - If found, the skb->dst is set to the route's dst_entry
  - Not found – then checked for multicast destination, or MC forwarding, which are processed via ip_route_input_mc()
  - Otherwise ip_route_input_slow is called.
- ip_route_input_slow
  - Does some basic validation, filters out "martians"
  - Does fib_lookup on flowi – failure is host unreachable or no route
  - Result of fib_lookup is processed a bit more, source is validated
  - Route is added to the route cache
  - skb->dst is pointed to the route. Input is either ip_forward or ip_local_deliver

Network Path in Kernel: IP Forwarding

- ip_forward
  - Does some validation and setup
  - Checks xfrm policies
  - skb_cow the packet, as it is about to be “mangled”
  - decrements TTL
  - sends a redirect if needed
  - maps the skb priority from TOS
  - NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, rt->u.dst.dev, ip_forward_finish);
- ip_forward_finish
  - handles options if needed
  - calls the destination output function – which is ip_output

Network Path in Kernel: IP Output

- ip_output
  - Pulls up dst info to skb fields, sets protocol
  - NF_HOOK_COND(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev, ip_finish_output, !(IPCB(skb)->flags & IPSKB_REROUTED));
- ip_finish_output
  - Goes around again if SNAT from hook has a new policy
  - Calls either ip_fragment or ip_finish_output2
- ip_finish_output2
  - Checks for room in skb
  - Uses cached hardware header if available via neigh_hh_output
  - Or calls dst->neighbour->output method – either ultimately calls the dev.c routine dev_queue_xmit

Locally generated traffic comes through ip_queue_xmit which routes and returns NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev, dst_output), which is normally ip_output.

IP Routing in Kernel: Routing Cache

When Routes are Used they are placed in cache

- Route Cache is a hash table – buckets are struct rtable pointers
- Hash is keyed Jenkins hash
  - re-keyed every 10 minutes
  - uses source and destination address and the interface index
  - match key is flowi
- Routes in cache are fully qualified flows – unidirectional
- Cache can be quite large

IP Routing in Kernel: Routing Cache

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

        struct flowi            fl;
        struct in_device        *idev;
        unsigned                rt_flags;       /*RTCF_ flags in linux/in_route.h */
        __u16                   rt_type;        /* RTN_ types in linux/rtnetlink.h */
        __u16                   rt_multipath_alg;

        __be32                  rt_dst; /* Path destination     */
        __be32                  rt_src; /* Path source          */
        int                     rt_iif;

        /* Info on neighbour */
        __be32                  rt_gateway;
        __be32                  rt_spec_dst;                    /* RFC1122 specific destination */
       struct inet_peer        *peer;                           /* long-living peer info */
};
```

IP Routing in Kernel: Cache - Destinations

```
struct dst_entry
{
        struct rcu_head         rcu_head;
        struct dst_entry        *child;     /* stackable dsts for IPSec */
        struct net_device       *dev;       /* egress net_device */
        short                   error;      /* lookup error for ICMP */
        short                   obsolete;   /* various – 2 means don't use */
        int                     flags;
#define DST_HOST                1
#define DST_NOXFRM              2
#define DST_NOPOLICY            4
#define DST_NOHASH              8
#define DST_BALANCED            0x10
#define DST_OTBUNDLE            0x20
        unsigned long           expires;    /* time when this entry expires */
```

IP Routing in Kernel: Cache - Destinations

```
unsigned short      header_len;     /* more space at head required  - IPSec */
unsigned short      nfheader_len;   /* more non-fragment space at head required */
unsigned short      trailer_len;    /* space to reserve at tail */

u32                 metrics[RTAX_MAX];      /* Mostly TCP metrics */
struct dst_entry    *path;              /* Points to dst for actual xmit – IPSec */

unsigned long       rate_last;      /* rate limiting for ICMP */
unsigned long       rate_tokens;

struct neighbour    *neighbour;
struct hh_cache     *hh;
struct xfrm_state   *xfrm;              /* IPSec */

int                 (*input)(struct sk_buff*);              /* last routing ops */
int                 (*output)(struct sk_buff*);
```

IP Routing in Kernel: Cache - Destinations

```
#ifdef CONFIG_NET_CLS_ROUTE
        __u32                   tclassid;               /* Supplies route class to netfilter/tc */
#endif

        struct  dst_ops         *ops;       /* destination management routines */

        unsigned long           lastuse;
        atomic_t                __refcnt;       /* client references    */
        int                     __use;
        union {
                struct dst_entry        *next;
                struct rtable           *rt_next;
                struct rt6_info         *rt6_next;
                struct dn_route         *dn_next;
        };
        char                    info[0];
};
```

IP Routing in Kernel: Flowi

```
struct flowi {
        int     oif;
        int     iif;
        __u32   mark;

        union {
                struct {
                        __be32                  daddr;
                        __be32                  saddr;
                        __u8                    tos;
                        __u8                    scope;
                } ip4_u;

                struct {
                        struct in6_addr         daddr;
                        struct in6_addr         saddr;
                        __be32              flowlabel;
                } ip6_u;

                struct {
                        __le16                  daddr;
                        __le16                  saddr;
                        __u8                    scope;
                } dn_u;
        } nl_u;
```

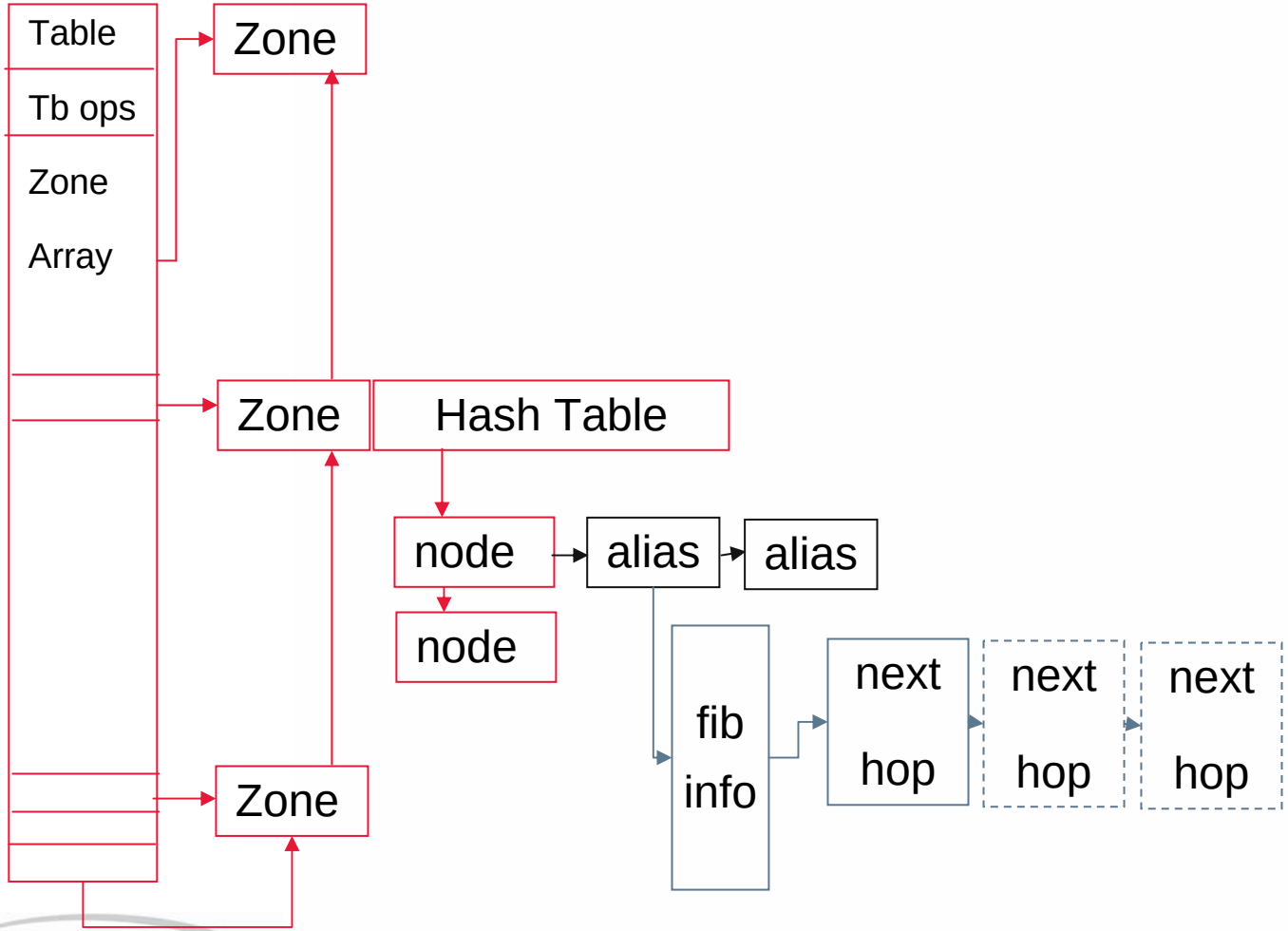
```
#define fld_dst         nl_u.dn_u.daddr
#define fld_src         nl_u.dn_u.saddr
#define fld_scope       nl_u.dn_u.scope
#define fl6_dst         nl_u.ip6_u.daddr
#define fl6_src         nl_u.ip6_u.saddr
#define fl6_flowlabel   nl_u.ip6_u.flowlabel
#define fl4_dst         nl_u.ip4_u.daddr
#define fl4_src         nl_u.ip4_u.saddr
#define fl4_tos         nl_u.ip4_u.tos
#define fl4_scope       nl_u.ip4_u.scope
```

IP Routing in Kernel: Flowi

```
        __u8    proto;
        __u8    flags;
#define FLOWI_FLAG_NOTROUTE          0x02   /* not used
by routing */
        union {
                struct {
                        __be16  sport;
                        __be16  dport;
                } ports;

                struct {
                        __u8    type;
                        __u8    code;
                } icmpt;

                struct {
                        __le16  sport;
                        __le16  dport;
                } dnports;

                __be32          spi;

                struct {
                        __u8    type;
                } mht;
        } uli_u;
        __u32  secid;  /* used by xfrm; see secid.txt */
} __attribute__((__aligned__(BITS_PER_LONG/8)));
```

```
#define fl_ip_sport     uli_u.ports.sport
#define fl_ip_dport     uli_u.ports.dport
#define fl_icmp_type    uli_u.icmpt.type
#define fl_icmp_code    uli_u.icmpt.code
#define fl_ipsec_spi    uli_u.spi
#define fl_mh_type      uli_u.mht.type
```

IP Routing in Kernel: Routing Tables



IP Routing in Kernel: Routing Tables

```
struct fib_table {
        struct hlist_node           tb_hlist;
        u32             tb_id;
        unsigned        tb_stamp;
        int             (*tb_lookup)(struct fib_table *tb, const struct flowi *flp, struct fib_result
                                                                                                *res);
        int             (*tb_insert)(struct fib_table *, struct fib_config *);
        int             (*tb_delete)(struct fib_table *, struct fib_config *);
        int             (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
                                        struct netlink_callback *cb);
        int             (*tb_flush)(struct fib_table *table);
        void            (*tb_select_default)(struct fib_table *table,
                                        const struct flowi *flp, struct fib_result *res);

        unsigned char   tb_data[0];
};
```

IP Routing in Kernel: Routing Tables

```
struct fn_hash {
        struct fn_zone  *fn_zones[33];
        struct fn_zone  *fn_zone_list;
};
```

or

```
struct trie {
        struct node *trie;
#ifdef CONFIG_IP_FIB_TRIE_STATS
        struct trie_use_stats stats;
#endif
        int size;
        unsigned int revision;
};
```

IP Routing in Kernel: Routing Tables

```
struct fn_zone {
        struct fn_zone          *fz_next;               /* Next not empty zone  */
        struct hlist_head       *fz_hash;               /* Hash table pointer   */
        int                     fz_nent;                /* Number of entries    */

        int                     fz_divisor;             /* Hash divisor         */
        u32                     fz_hashmask;            /* (fz_divisor - 1)     */
#define FZ_HASHMASK(fz)         ((fz)->fz_hashmask)

        int                     fz_order;               /* Zone order           */
        __be32                  fz_mask;                /* Netmask */
#define FZ_MASK(fz)             ((fz)->fz_mask)
};

struct fib_node {
        struct hlist_node       fn_hash;
        struct list_head        fn_alias;
        __be32                  fn_key;
};
```

IP Routing in Kernel: Routing Tables

```
struct fib_alias {
        struct list_head        fa_list;
        struct rcu_head rcu;
        struct fib_info         *fa_info;
        u8                      fa_tos;     /*key*/
        u8                      fa_type;
        u8                      fa_scope; /*key*/
        u8                      fa_state;
};

#define FA_S_ACCESSED   0x01
```

IP Routing in Kernel: Routing Tables

```
struct fib_info {
        struct hlist_node       fib_hash;
        struct hlist_node       fib_lhash;
        int                     fib_treeref;
        atomic_t                fib_clntref;
        int                     fib_dead;
        unsigned                fib_flags;
        int                     fib_protocol;
        __be32                  fib_prefsrc;
        u32                     fib_priority;
        u32                     fib_metrics[RTAX_MAX];
#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADVMSS-1]
        int                     fib_nhs;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
        int                     fib_power;
#endif
#ifdef CONFIG_IP_ROUTE_MULTIPATH_CACHED
        u32                     fib_mp_alg;
#endif
        struct fib_nh           fib_nh[0];
#define fib_dev                 fib_nh[0].nh_dev
};
```

IP Routing in Kernel: Routing Tables

```
struct fib_nh {
        struct net_device       *nh_dev;
        struct hlist_node       nh_hash;
        struct fib_info         *nh_parent;
        unsigned                nh_flags;
        unsigned char           nh_scope;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
        int                     nh_weight;
        int                     nh_power;
#endif
#ifdef CONFIG_NET_CLS_ROUTE
        __u32                   nh_tclassid;
#endif
        int                     nh_oif;
        __be32                  nh_gw;
};
```

Policy Routing - Tables

Policy Routing

- Routing decisions made on more than just the destination address
- Multiple Tables are used by Linux for Policy Routing
- Up to 255 tables, numbered 1-255
- Looks like work is going into removing the table limit (32 bit ID on hash)
- Tables may be user specified

- Kernel Created Tables
  - Without Policy Routing local and main route tables are created
  - With Policy Routing kernel creates three tables
    - Local = 255
    - Main = 254
    - Default = 253

Policy Routing - Rules

ip command allows setting up rules for the various tables, as well as table creation

Tables are created by reference in a route or rule add command

Three rules are created at init –

1. 0: from all lookup 255 - this is unchangeable – forces local table first
2. 32766: from all lookup main
3. 32767: from all lookup default

ip rule add unicast iif eth4 preference 5 table 5 –

ip rule [show | list]

1. 0: from all lookup 255
2. 5: from all iif eth4 lookup 5
3. 32766: from all lookup main
4. 32767: from all lookup default

Policy Routing - Rules

May be used for just about anything that can be discriminated via a simple rule

That rule includes the FW mark – so anything iptables/netfilter can do may be used to select the routing table that will be used.

Following Actions are available:

- unicast – lookup the route in the specified table
- blackhole – silently discard
- unreachable – drops packet and sends ICMP Network Unreachable
- prohibit – discard and send ICMP Administratively Prohibited
- may be others – see include/linux/fib_rules.h

Policy Routing - Rules

Rule selectors include:

- from – source prefix
- to – destination prefix
- iif – ingress interface name – loopback matches only local origin
- tos – TOS field
- fwmark – firewall mark to match

Additional parameters are

- realms – supplies routing realms if not supplied by route
- priority – order of the rule execution from 0 up
- table – which table for unicast action lookup

Again, look in include/linux/fib_rules.h for additional parameters available via netlink.

Need a route cache flush after the RPDB update:

```
ip route flush cache
```

Netlink Facility

Netlink Address Family

- Normal socket interface functions are used
- Includes netfilter, routing tables, arp/neighbour tables, security transforms, SELinux policy tables, IPC, kernel-user event passing, arbitrary extensions

Examples (from iproute2 package)

Provides library lib/libnetlink.c

```
int rtnl_open_byproto(struct rtnl_handle *rth, unsigned subscriptions,
                      int protocol)
…
    rth->fd = socket(AF_NETLINK, SOCK_RAW, protocol)
```

protocol = NETLINK_ {ROUTE, USERSOCK, FIREWALL, INET_DIAG, NFLOG, XFRM, SELINUX, ISCSI, AUDIT, FIB_LOOKUP, CONNECTOR, NETFILTER, IP6_FW, DNRTMSG, GENERIC, SCSITRANSPORT, ECRYPTFS, LTT}

Netlink Facility

```
…
memset(&rth->local, 0, sizeof(rth->local));
rth->local.nl_family = AF_NETLINK;
rth->local.nl_groups = subscriptions;

if (bind(rth->fd, (struct sockaddr*)&rth->local, sizeof(rth->local)) < 0) {
        perror("Cannot bind netlink socket");
        return -1;
}
```

```
struct sockaddr_nl
{
        sa_family_t         nl_family;     /* AF_NETLINK   */
        unsigned short      nl_pad;        /* zero         */
        __u32               nl_pid;        /* process pid  */
        __u32               nl_groups;     /* multicast groups mask */
};
```

Netlink Facility

After that the normal socket interface can be used to send/recv netlink messages of the following format:

```
struct nlmsghdr
{
        __u32       nlmsg_len;     /* Length of message including header */
        __u16       nlmsg_type;    /* Message content */
        __u16       nlmsg_flags;   /* Additional flags */
        __u32       nlmsg_seq;     /* Sequence number */
        __u32       nlmsg_pid;     /* Sending process PID */
};
```

where nlmsg_type is defined for each link type. In the case of NETLINK_ROUTE the choices are currently defined in include/linux/rtnetlink.h

Each link type defines own message types.

Netlink Facility - Sending

```
struct msghdr msg;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);

struct iovec iov;
iov.iov_base = (void *)nlh;
iov.iov_len = nlh->nlmsg_len;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;

sendmsg(fd, &msg, 0);
```

Netlink Facility - Receiving

```
struct sockaddr_nl nladdr;
struct msghdr msg;
struct iovec iov;
iov.iov_base = (void *)nlh;
iov.iov_len = MAX_NL_MSG_LEN;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
recvmsg(fd, &msg, 0);
```

See:

http://linux-net.osdl.org/index.php/Generic_Netlink_HOWTO

for details on creating generic netlink protocol extensions, complete with attribute passing

Policing and Shaping – Schedule Modules

```
static struct Qdisc_ops pfifo_fast_ops = {
        .id             =       "pfifo_fast",
        .priv_size      =       PFIFO_FAST_BANDS * sizeof(struct sk_buff_head),
        .enqueue        =       pfifo_fast_enqueue,
        .dequeue        =       pfifo_fast_dequeue,
        .requeue        =       pfifo_fast_requeue,
        .init           =       pfifo_fast_init,
        .reset          =       pfifo_fast_reset,
        .dump           =       pfifo_fast_dump,
        .owner          =       THIS_MODULE,
};
```

Policing and Shaping – Schedule Modules

```
static const u8 prio2band[TC_PRIO_MAX+1] =
        { 1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1 };

/* 3-band FIFO queue: old style, but should be a bit faster than
   generic prio+fifo combination.
 */

#define PFIFO_FAST_BANDS 3

static inline struct sk_buff_head *prio2list(struct sk_buff *skb,
                                             struct Qdisc *qdisc)
{
        struct sk_buff_head *list = qdisc_priv(qdisc);
        return list + prio2band[skb->priority & TC_PRIO_MAX];
}
```

Policing and Shaping – Schedule Modules

```
static int pfifo_fast_enqueue(struct sk_buff *skb, struct Qdisc* qdisc)
{
        struct sk_buff_head *list = prio2list(skb, qdisc);

        if (skb_queue_len(list) < qdisc->dev->tx_queue_len) {
                qdisc->q.qlen++;
                return __qdisc_enqueue_tail(skb, qdisc, list);
        }

        return qdisc_drop(skb, qdisc);
}

static struct sk_buff *pfifo_fast_dequeue(struct Qdisc* qdisc)
{
        int prio;
        struct sk_buff_head *list = qdisc_priv(qdisc);

        for (prio = 0; prio < PFIFO_FAST_BANDS; prio++) {
                if (!skb_queue_empty(list + prio)) {
                        qdisc->q.qlen--;
                        return __qdisc_dequeue_head(qdisc, list + prio);
                }
        }

        return NULL;
}
```

Policing and Shaping – Filters

Qdisc system has some built-in filter capabilities that can be specified via tc command.

There is no tc-filter man page

See http://lartc.org/howto/index.html for examples and many, many other good discussions concerning most aspects of linux advanced routing.

Resources

http://linux-net.osdl.org/index.php/Network_Overview (Aug 2007)
http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_walkthrough.html (2.4.20)
http://lartc.org/howto/index.html
http://linux-net.osdl.org/index.php/Generic_Netlink_HOWTO
http://www.linuxjournal.com/article/7356 (easy netlink interface)

Linux Firewals – Steve Suehring, Robert L. Ziegler (Novell Press)
Understanding Linux Network Internals – Christian Benvenuti (O'Reilly)
Linux iptables Pocket Reference – Gregor N. Purdy (O'Reilly)
Policy Routing Using Linux – Matthew G. March (Sams) old

Google is your Friend!