Netlink for FreeBSD

Alexander Chernikov <melifaro@FreeBSD.org>, September 2022

Agenda

- Motivation
- Netlink overview
- Implementation overview
- Next steps
- Q&A

Network configuration actors

- Who manages runtime network configuration?
  - Users?
  - Software?

Network configuration actors

- Who manages runtime network configuration?
  - Users
  - **Software**
    - NetworkManager, dhcp, ..
    - Routing Software (Bird,Frr,GoBGP,mpd,…)
    - Containers (containerd, …)
    - UIs (pfsense/OpnSense, …)

Existing management APIs

- system(“/sbin/ifconfig”)
  - Some tools provide structured output
  - Limited portability
- Ioctls / sysctls
  - Primary kernel interface
  - Low-level, limited documentation
  - Limited portability (partially compatible with other *BSDs)
  - Synchronous
  - Limited extendability

Existing management APIs #2

- Routing socket
  - Compatible across *BSDs
  - Largely non-extendable
  - RTM_VERSION was last bumped 28 years ago
  - Synchronous
- Libraries
  - libifconfig, libjail
  - Largely exposes the same low-level API
  - FreeBSD-specific
- Devd
  - Event notifications via system(“”)

Management APIs we provide #3

- Kernel Notifications - rtsock
  - Compatible across *BSDs
  - Largely non-extendable
- Kernel Notifications - Devd
  - Event notifications via system(“/some/script”)
  - “Subscription” requires configuration changes
  - Undocumented devd.pipe interface

Users of the APIs

- Management software is not only C:
  - Go, Python, Rust
  - APIs need to ported/tested to start their adoption
- Making new (or old) APIs usable is an effort
- Making APIs *easily* usable is a significant effort

Summary

- Software layer is the important management mechanism
- APIs we provide are extremely important
- We can do better with APIs

APIs: Netlink

- User<>kernel TLV-based communication protocol
- Defined in [RFC 3549]
- Supports large dumps and notifications
- Fully asynchronous
- Easily extendable
- Offers reasonable network object models
  - High-level CRUD-like APIs
- Can serve as API broker between drivers and userland

APIs: Netlink #2

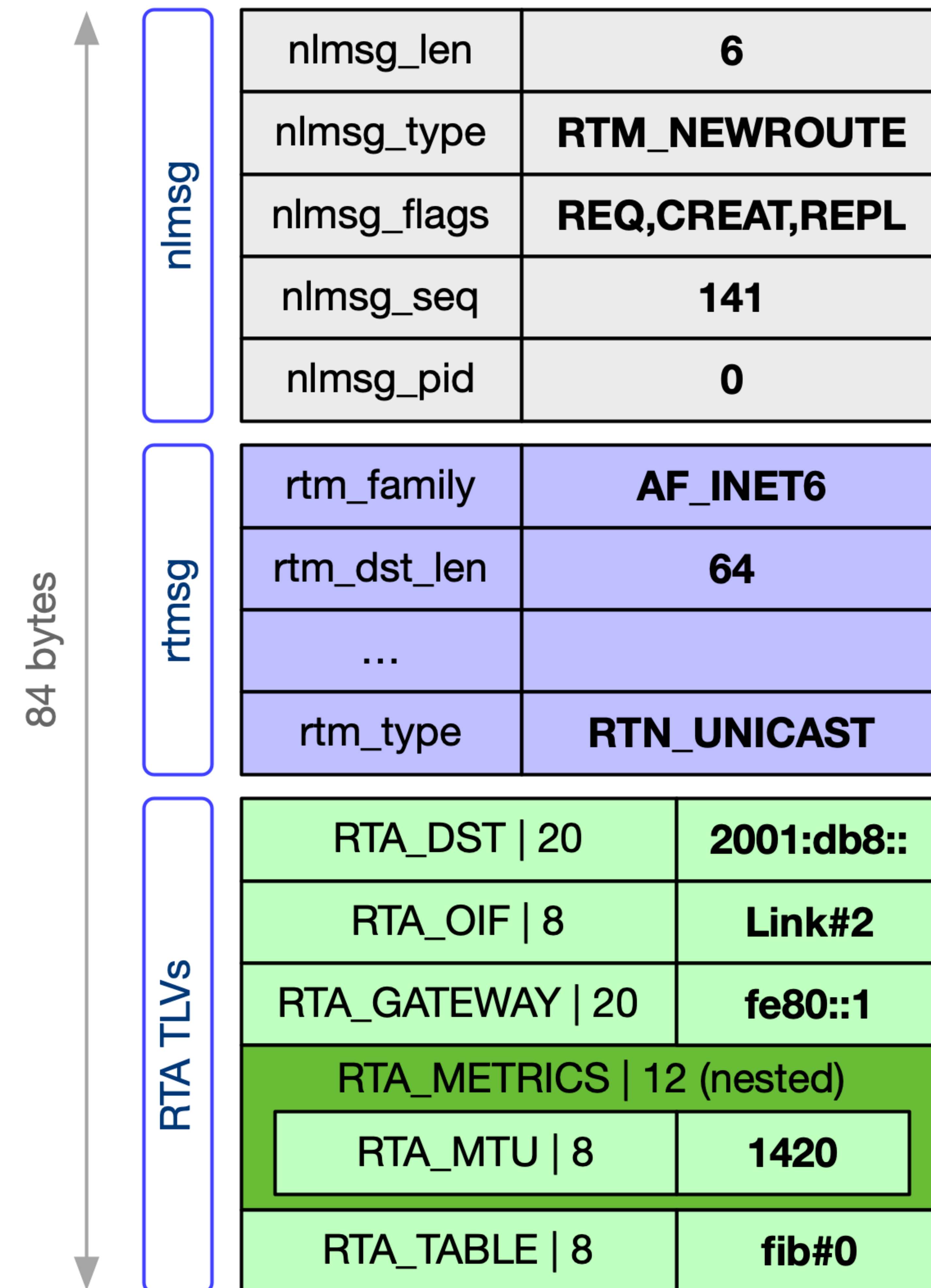
- De-facto standard in Linux networking
  - interfaces, routes, neighbors, smb, macsec, gtp, wireguard, tcp_metrics,..
- Used in other areas
  - acpi, vfs, raid, thermal, devlink

What FreeBSD gets from Netlink

- API compatibility with applications relying on Netlink
  - Support in go/rust/python netlink abstraction libraries
  - Reduce the barrier for app developers to support FreeBSD
    - Low-effort support the direct netlink consumers
      - net/bird required only headers change to switch to FreeBSD netlink
- Easy interface extendability without breaking old ioctl/rtsock interfaces
  - Reduce the barrier for bringing new functionality
  - Reduce the barrier for exporting data from drivers
- Standard way of providing userland notifications

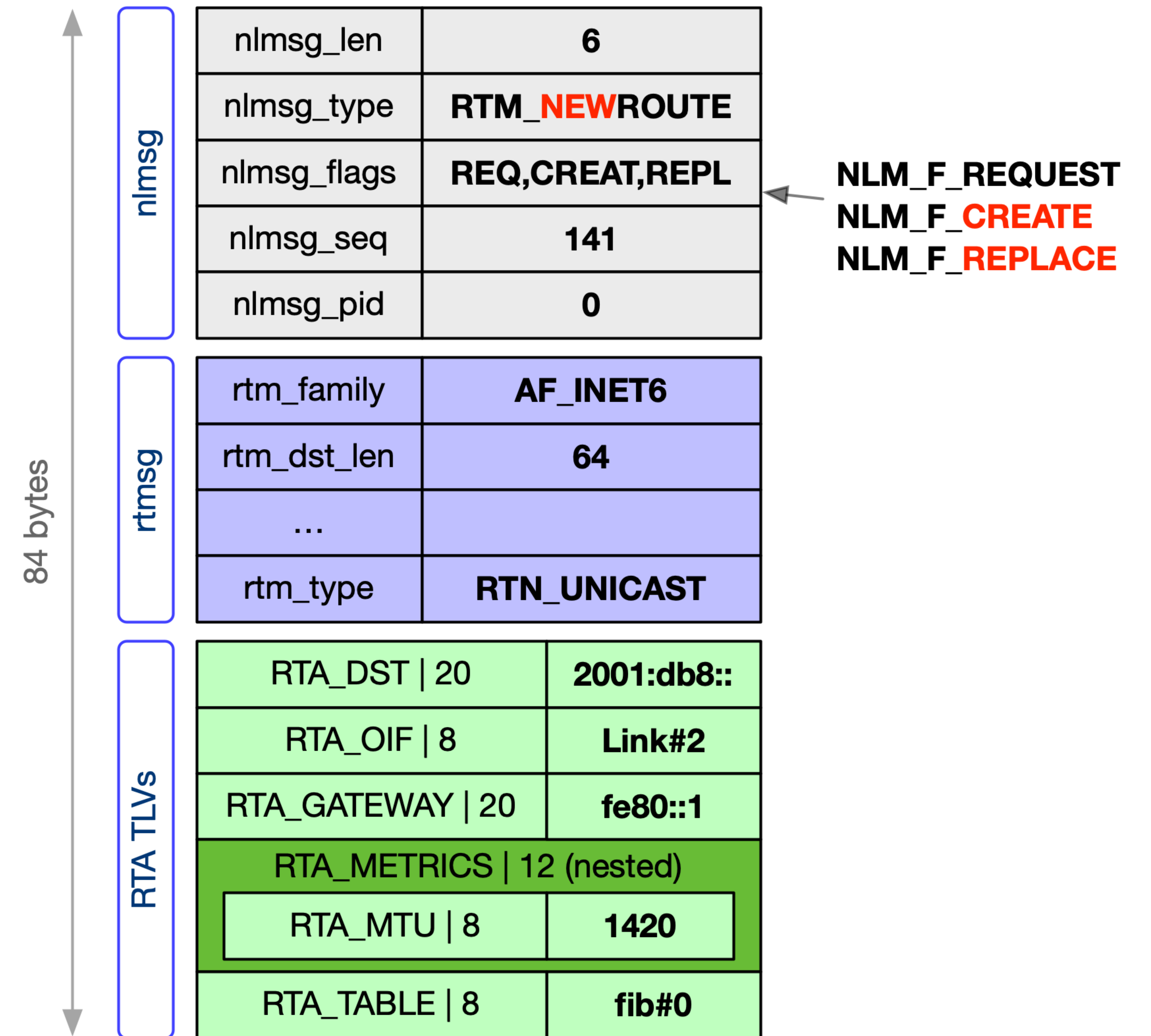
Netlink protocol

- Socket-based
- 16-byte netlink header
- Family header (8-16 bytes)
- Followed by the list of TLVs
- TLVs can be nested
- 32-bit aligned
- Fully async
    - operation result is a message



Netlink ops model

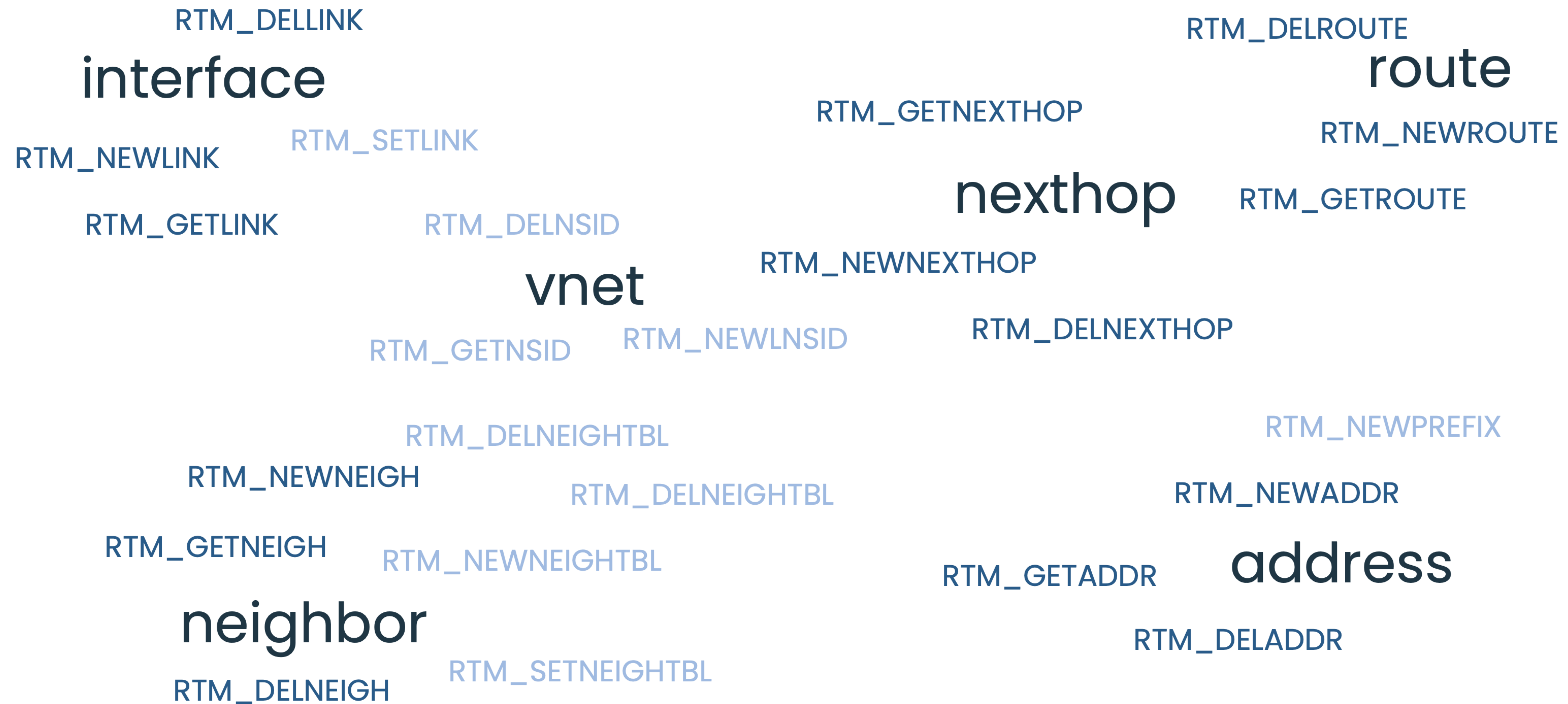
- Netlink core suggests CRUD-like object model
- Commands are informally classified into GET/NEW/DELETE
- Command flags extends the meaning
- Create (“NEW” / “UPDATE”)
    - REPLACE / EXCL to deal with existing object
    - CREATE to create if not exists
    - APPEND to extend an object
- GET (“READ”)
    - Dumps all or matching entries
- DELETE
    - Deletes matching object



Relevant netlink families

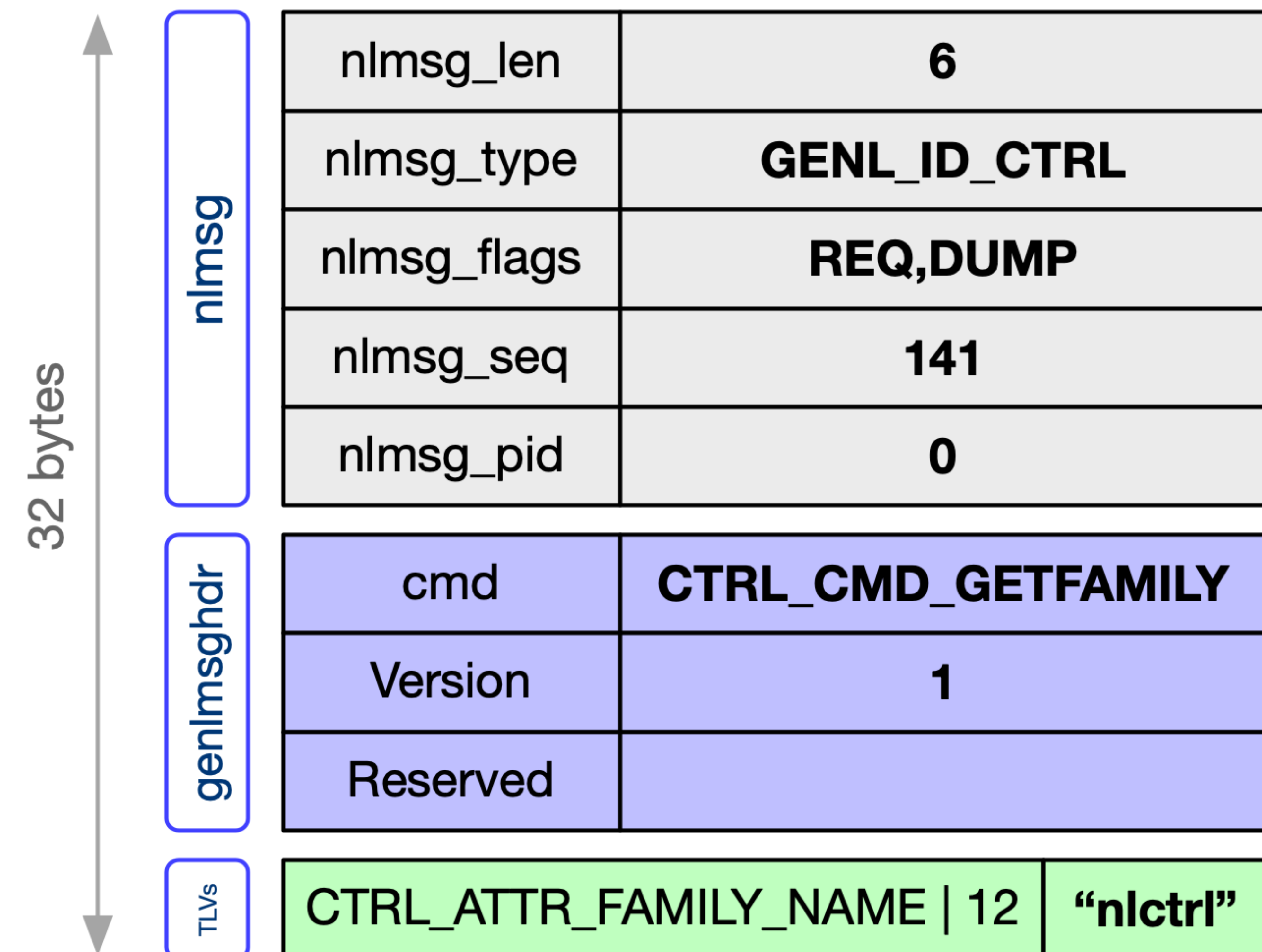
- NETLINK_ROUTE
  - First and the biggest (100+ messages)
  - Most of “classic” network management is here
- NETLINK_GENERIC
  - “Container” family
  - Used to declare other families “on the fly”
  - String family / group names
  - Single socket can interface with any sub-family

NETLINK_ROUTE



Generic netlink

- New NETLINK_GENERIC family
  - App can communicate within multiple families
  - Single socket
  - Tiny family header
  - Families and notification groups are strings
  - All new customers adopts GENERIC netlink



FreeBSD implementation

Implementation overview

- Derived from 2021 GSoC project by Ng Peng Nam Sean
- Kernel module
- Implements subset of NETLINK_ROUTE family
    - Routes, nexthops, interfaces, neighbours
    - Notifications for all of the above
- NETLINK_GENERIC framework
    - Base “nlctrl” family implemented
    - KPI for loading/unloading families
- Code: D36002

Implementation overview #2

- Async processing
  - Per-socket dispatch taskque
  - Allows to call code with M_WAITOK
    - Useful to call interface ioctls
- Locking
  - Per-socket lock & sockbuf lock
  - Reading/writing does not block message dispatching
  - No global locks on fast path

Message parsing

- Framework handles all parsing
  - Pre-defined parsers for common types
  - Nested parsers supported
  - Detailed error reporting

```
struct nl_parsed_route {
        struct sockaddr         *rta_dst;
        struct sockaddr         *rta_gw;
        struct ifnet            *rta_oif;
        struct rta_mpath        *rta_multipath;
        uint32_t                rta_table;
        uint32_t                rta_rtflags;
        uint32_t                rta_nh_id;
        uint32_t                rtax_mtu;
        uint8_t                 rtm_family;
        uint8_t                 rtm_dst_len;
};
#define _IN(_field)     offsetof(struct rtmsg, _field)
#define _OUT(_field)    offsetof(struct nl_parsed_route, _field)
static struct nlattr_parser nla_p_rtmetrics[] = {
        { .type = NL_RTAX_MTU, .off = _OUT(rtax_mtu), .cb = nlattr_get_uint32 },
};
NL_DECLARE_ATTR_PARSER(metrics_parser, nla_p_rtmetrics);
static const struct nlattr_parser nla_p_rtmsg[] = {
        { .type = NL_RTA_DST, .off = _OUT(rta_dst), .cb = nlattr_get_ip },
        { .type = NL_RTA_OIF, .off = _OUT(rta_oif), .cb = nlattr_get_ifp },
        { .type = NL_RTA_GATEWAY, .off = _OUT(rta_gw), .cb = nlattr_get_ip },
        { .type = NL_RTA_METRICS, .arg = &metrics_parser, .cb = nlattr_get_nested },
        { .type = NL_RTA_MULTIPATH, .off = _OUT(rta_multipath), .cb = nlattr_get_multipath },
        { .type = NL_RTA_RTFLAGS, .off = _OUT(rta_rtflags), .cb = nlattr_get_uint32 },
        { .type = NL_RTA_TABLE, .off = _OUT(rta_table), .cb = nlattr_get_uint32 },
        { .type = NL_RTA_VIA, .off = _OUT(rta_gw), .cb = nlattr_get_ipvia },
        { .type = NL_RTA_NH_ID, .off = _OUT(rta_nh_id), .cb = nlattr_get_uint32 },
};
static const struct nlfield_parser nlf_p_rtmsg[] = {
        {.off_in = _IN(rtm_family), .off_out = _OUT(rtm_family), .cb = nlf_get_u8 },
        {.off_in = _IN(rtm_dst_len), .off_out = _OUT(rtm_dst_len), .cb = nlf_get_u8 },
};
#undef _IN
#undef _OUT
NL_DECLARE_PARSER(rtm_parser, struct rtmsg, nlf_p_rtmsg, nla_p_rtmsg);
...
        struct nl_parsed_route attrs = {};
        error = nl_parse_nlmsg(hdr, &rtm_parser, npt, &attrs);
        if (error != 0)
                return (error);
```

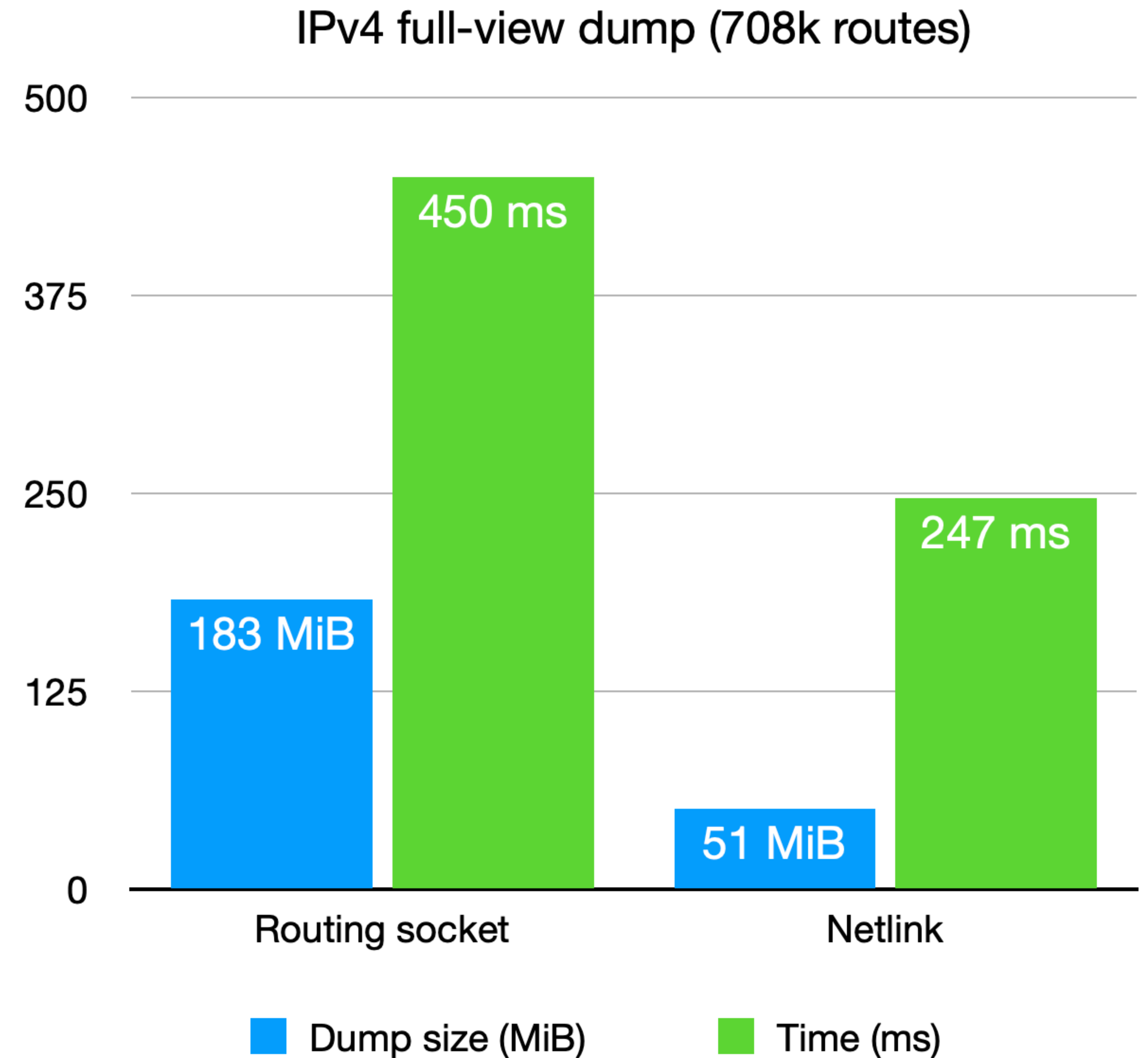
Message writing

- Convenient writing KPI
- No message size limits
- Contiguous message space
- Transparently uses mbufs / buffers

```
if (!nlmsg_reply(nw, hdr, sizeof(struct ifinfomsg)))
        goto enomem;

ifinfo = nlmsg_reserve_object(nw, struct ifinfomsg);
ifinfo->ifi_family = AF_UNSPEC;
ifinfo->__ifi_pad = 0;
ifinfo->ifi_type = ifp->if_type; // ARPHDR
ifinfo->ifi_index = ifp->if_index;
ifinfo->ifi_flags = ifp_flags_to_netlink(ifp);
ifinfo->ifi_change = 0;

nlattr_add_string(nw, IFLA_IFNAME, if_name(ifp));
struct if_state ifs = {};
get_operstate(ifp, &ifs);
nlattr_add_u8(nw, IFLA_OPERSTATE, ifs.ifla_operstate);
nlattr_add_u8(nw, IFLA_CARRIER, ifs.ifla_carrier);
if ((ifp->if_addr != NULL))
        dump_sa(nw, IFLA_ADDRESS, ifp->if_addr->ifa_addr);
if ((ifp->if_broadcastaddr != NULL))
        nlattr_add(nw, IFLA_BROADCAST, ifp->if_addrlen, ifp->if_broadcastaddr);
nlattr_add_u32(nw, IFLA_MTU, ifp->if_mtu);
get_stats(nw, ifp);

uint32_t val = (ifp->if_flags & IFF_PROMISC) != 0;
nlattr_add_u32(nw, IFLA_PROMISCUITY, val);

if (nlmsg_end(nw))
        return (true);
enomem:
NL_LOG(LOG_DEBUG, "unable to dump interface %s state (ENOMEM)", if_name(ifp));
nlmsg_abort(nw);
return (false);
```

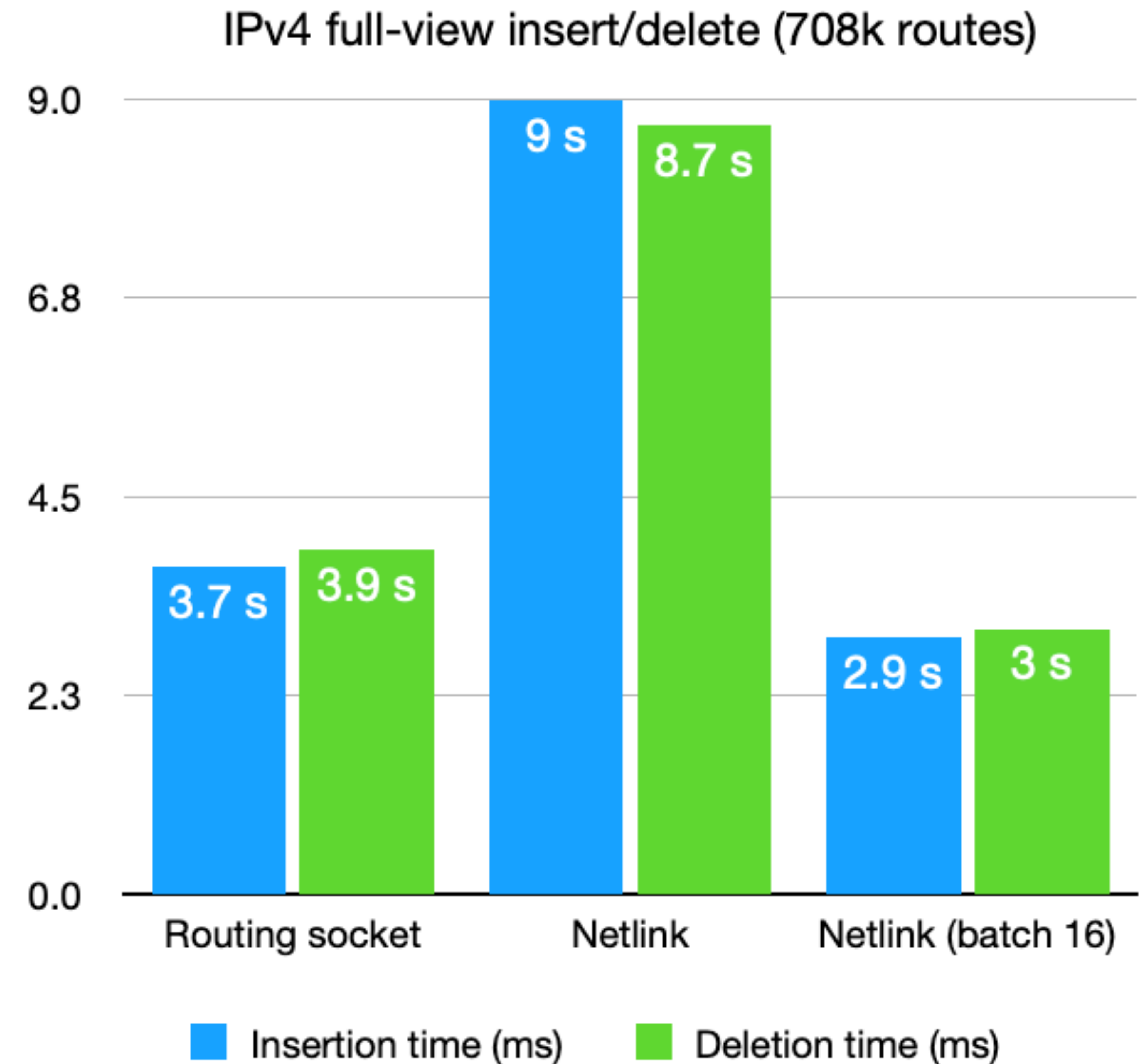
Performance: reading

- Biggest state dump: IPv4 full-view
- AMD Ryzen 7 3700X VM
- HEAD from September, with default debug options
- Dedicated C binary reading dump
- Size reported by binary, time - 'time'
- Netlink RX buffer size=8k



Performance: writing

- Biggest state insertion: IPv4 full-view
- AMD Ryzen 7 3700X VM
- HEAD from September, with default debug options
- net/bird with netlink patch
- Timing reported by bird IO cycle



Linux ABI compatibility

- Protocol is compatible, but some OS constants differs
    - Routing tables: fib 0 vs fib 254
    - AF_INET6 value is different
    - Interface, interface address flags are different
    - Error numbers are different
    - Need to rewrite messages both ways
- linux_common depends on netlink
    - 3 hooks to rewrite messages to/from Linux
    - Supports resizing (adding/deleting TLVs etc)
- ip(8) works fine for the supported netlink messages

Next steps

- Consider making Netlink the default management API in FreeBSD 14
  - Convert all our tools (route, netstat, ifconfig, apr, ndp)
  - netstat example: D36529
- **Keep rtsock and ioctls compatibility**
  - Events from rtsock commands are propagated to Netlink
  - And vice versa
- Make rtsock loadable module
- Compile under COMPAT_FREEBSD1<4|5>

Questions?

Alexander Chernikov <melifaro@FreeBSD.org> , September 2022