FreeBSD 5 Network Enhancements

André Oppermann <andre@FreeBSD.org>

SUCON 04
Zürich, 3. September 2004

About

This talk gives an overview on what is new or has changed in FreeBSD 5 Networking Code compared to FreeBSD 4-Series. It is by no means exhaustive and touches only the most important improvements.

Areas covered:

- Routing Table
- Interface Handling
- IPv4 Processing
- Packet Filters
- TCP Processing
- New Network Stacks
- Network Stack (General)

Routing Table

PRCLONING removed.

PRCLONING was previously done for two reasons. TCP stored/cached certain observations (for example RTT and RTT Variance) per remote host. For every host that has/had a TCP session with us it would create/clone a route to store these informations. Reduced rt_metrics from 14 to 3 fields and saving 11 times sizeof(u_long), on i386 44 Bytes. Most of the 11 removed fields are now in the tcp_hostcache (see later): (By: andre)

```
struct rt_metrics {
        u_long   rmx_locks;       /* Kernel must leave these values alone */
        u_long   rmx_mtu;         /* MTU for this path */
        u_long   rmx_hopcount;    /* max hops expected */
        u_long   rmx_expire;      /* lifetime for route, e.g. redirect */
        u_long   rmx_recvpipe;    /* inbound delay-bandwidth product */
        u_long   rmx_sendpipe;    /* outbound delay-bandwidth product */
        u_long   rmx_ssthresh;    /* outbound gateway buffer limit */
        u_long   rmx_rtt;         /* estimated round trip time */
        u_long   rmx_rttvar;      /* estimated rtt variance */
        u_long   rmx_pksent;      /* packets sent using this route */
        u_long   rmx_filler[4];   /* will be used for T/TCP later */
};

struct rt_metrics_lite {
        u_long   rmx_mtu;         /* MTU for this path */
        u_long   rmx_expire;      /* lifetime for route, e.g. redirect */
        u_long   rmx_pksent;      /* packets sent using this route */
};
```

Routing Table

Removed pointer to route from INPCB

The backpointer from the INPCB into the routing table was complicating the locking of the routing table and locking of the INPCB itself. When a route has to be removed from the table but it was referenced by a INPCB it was necessary to sequentially scan through ALL INPCB's and remove the refence(es). A very expensive operation because we had to lock every INPCB to do the lookup. Now instead of directly using the route pointer a very fast routing table lookup is done on any packet sent out. (By: andre)

RTENTRY's allocated with UMA

(Universal Memory Allocator, SLAB/Zone Type). Instead of kernel malloc. Much more efficient memory usage now. 130 Bytes instead of 256 Bytes allocated per Route (on i386), savings of 49%. (By: andre)
With about 200MB kmem is was possible to load 1.2 million routes into the kernel.

Interface Handling

Interface Link State Notification via RTSOCKET and KQUEUE

At the moment this is only implemented for Ethernet Type interfaces. When the link state goes down because the cable to the switch was unplugged you get a RTMESSAGE and a KQUEUE event. The same when the link comes up again. This is very useful for routing daemons. (By: andre, RTSOCKET, OpenBSD)

```
sys/net/if.h:

struct if_data {
     ...
     u_char    ifi_link_state;     /* current link state */
     ...
};

#define   LINK_STATE_UNKNOWN  0    /* link invalid/unknown */
#define   LINK_STATE_DOWN     1    /* link is down */
#define   LINK_STATE_UP       2    /* link is up */
```

Interface Handling

Interface renaming

Very cool. Allows to you to change any interface name to any arbitrary string of max. 15 characters. (By: brooks)

```
# ifconfig -a
bge0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=1a<TXCSUM,VLAN_MTU,VLAN_HWTAGGING>
        inet 62.48.1.1 netmask 0xffffff00 broadcast 62.48.1.255
        inet6 fe80::2e0:81ff:fe27:e0a9%bge0 prefixlen 64 scopeid 0x2
        ether 00:e0:81:27:e0:a9
        media: Ethernet autoselect (100baseTX <full-duplex>)
        status: active
# ifconfig bge0 name office
# ifconfig -a
office: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=1a<TXCSUM,VLAN_MTU,VLAN_HWTAGGING>
        inet 62.48.1.1 netmask 0xffffff00 broadcast 62.48.1.255
        inet6 fe80::2e0:81ff:fe27:e0a9%office prefixlen 64 scopeid 0x2
        ether 00:e0:81:27:e0:a9
        media: Ethernet autoselect (100baseTX <full-duplex>)
        status: active
```

Interface Handling

Interface cloning for virtual interfaces

You need a GRE tunnel interface? Just make one yourself: (By: brooks)

```
# ifconfig gre0 create
# ifconfig -a
gre0: flags=9010<POINTOPOINT,LINK0,MULTICAST> mtu 1476
# ifconfig gre0 destroy
```

Interface Handling

Automatic VLANS

Instead of cloning a vlan interface first and then specifying the parent interface and the 802.1Q tag you just do this: (By: brooks)

```
# ifconfig bge1.100 inet 192.168.1.1/24
# ifconfig -a
bge1.100: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    inet 192.168.1.1 netmask 0xffffff00 broadcast 192.168.1.255
    inet6 fe80::2e0:81ff:fe27:e08a%bge1.100 prefixlen 64 scopeid 0x6
    ether 00:e0:81:27:e0:8a
    media: Ethernet autoselect (100baseTX <full-duplex>)
    status: active
    vlan: 100 parent interface: bge1
```

Interface Handling

NDIS Binary Compatibility

a.k.a. “Project Evil”.

FreeBSD i386 can use binary Ethernet and WLAN network drivers written to the Windows XP NDIS 5.1 specification. It is a little cumbersome to convert a NDIS driver into a FreeBSD Kernel Loadable Module (KLD): (By: wpaul)

```
# ndiscvt -O -i neti557x.inf -s neti557x.sys -n intel0
# /* Compile and install new kernel with “options NDIS” */
# kldload intel0

Man ndis(4), ndisapi(9), ndiscvt(8).
```

IPv4 Processing

IP FastForward

IP FastForward processes a packet directly to completion (if it is not for the local host). The if_input directly calls into ip_fastfoward. All basic packet validation checks, routing table lookup, firewalling (pfil_hooks) and if_output steps are done in just one code path and function call. Compared to normal IP forwarding this can give a speedup of 40 to 60% in packet forwarding performance. (By: andre)

```
# sysctl net.inet.ip.fastforwarding=1
```

IPv4 Processing

IP FastForward (continued)

Thu, 2 Sep 2004 18:49:05 -0400 [zebra 21767]:
> > Indeed. We have a modified 5.3 kernel that broke **1Mpps** on a 2.8Ghz Xeon
> > using Smartbits.
>
> Have tried with 5.3 recently? I wrote a new ip_fastforward (don't look
> at the old man page, I haven't updated it yet) which processes packets
> directly to completion. Compared to normal forwarding via ip_input this
> should give you another 30% unless you have maxed out the bus bandwidth
> already. It's in every FreeBSD 5 kernel, just enable it with "sysctl
> net.inet.ip.fastforwarding=1".

Of course. Based on your new **fast forwarding** code is how **1Mpps** was achieved, btw ;-)

Also changed the old flow fastforwarding in 4.9 kernel with your code to improve performance on some routers that we run that are still in 4.x tree.. (just cant afford to upgrade the whole thing to 5.x just yet)

Your fast forwarding code is the solid positive step in terms of the way a real router should work. Your concept is pretty close to idea of cisco CEF implementation (with exception of real FIB yet, but that's being worked on anyways) in terms of running direct process to completion in the network interrupt routine. Great work! :)

James Jun <james@towardex.com> TowardEX Technologies, Inc.

IPv4 Processing

Random IP ID's

The IP_ID is used for packet reassembly and needs to be unique within a certain time frame specific to a certain host. Normally the IP_ID is assigned sequentially to each IP packet leaving the host. This makes it possible to gather for example the number of hosts behind a NAT device (track different sequences of IP_ID's). Enabling random IP_ID's assigns a random IP_ID to each packet rendering this kind of "attack" ineffective: (By: kris, dwmalone, OpenBSD)

```
# sysctl net.inet.ip.random_id=1
```

IPv4 Processing

IP Options Processing

IP Options do not have any practical use today. The only useful application is RR (Record Route) where it remembers the last 8 hops the packet traversed through. That allows you to check parts of the path back to you. IP options processing is rather expensive because the packet header has to be modified and expanded. In addition the only other use is to circumvent or trick firewalls thus it is normally blocked there. The options are these: (By: andre)

```
# sysctl net.inet.ip.process_options=0

Possible Modes:
    net.inet.ip.process_options=0  Ignore IP options and pass pkts unmodfied
    net.inet.ip.process_options=1  Process all IP options (default)
    net.inet.ip.process_options=2  Reject all pkts with IP options with ICMP
```

Packet Filters

PFIL_HOOKS

As they are found in NetBSD, have been implemented and enabled permanently in the ip_input and ip_output paths. The kernel config option is no longer needed and any kernel will allow a packet filter to be loaded at run-time. (By: mlaier, sam, andre)

Man pfil(9).

```
sys/netinet/ip_input.c:
        /* Jump over all PFIL processing if hooks are not active. */
        if (inet_pfil_hook.ph_busy_count == -1)
                goto passin;
        odst = ip->ip_dst;
        if (pfil_run_hooks(&inet_pfil_hook, &m, m->m_pkthdr.rcvif,
            PFIL_IN) != 0)
                return;
        if (m == NULL)                  /* consumed by filter */
                return;
        ip = mtod(m, struct ip *);
        dchg = (odst.s_addr != ip->ip_dst.s_addr);
#ifdef IPFIREWALL_FORWARD
        if (m->m_flags & M_FASTFWD_OURS) {
                m->m_flags &= ~M_FASTFWD_OURS;
                goto ours;
        }
        dchg = (m_tag_find(m, PACKET_TAG_IPFORWARD, NULL) != NULL);
#endif /* IPFIREWALL_FORWARD */

passin:
```

Packet Filters

PF (from OpenBSD) and ALTQ (KAME) have been imported and are fully functional. All features except CARP (coming soon) are available. (By: mlaier)

```
Man pf(4), pfctl(8), pf.conf(5), pfsync(4).
```

FreeBSD 5 has now three packet filters to chose from: IPFW (BSD origin), PF (OpenBSD) and IPFILTER (Darren Reed).

```
Man ipfw(8), pfctl(8), ipf(8).
```

IPFW has been converted to use the generic PFIL_HOOKS like the other two. Previously it was directly fitted ("hacked") into ip_input and ip_output. This conversion removes about 370 lines code from those two and significantly cleans up those functions. Nothing in the use of IPFW has changed. It is completely transparent to the user. (By: andre)

Packet Filters

IPFW has got a number of new functions:

Rule Sets which can be enabled and disabled together (By: luigi)

```
# ipfw add 10000 set 5 allow ip from any to any
# ipfw set disable 5
# ipfw set move 10000 5 to 3
# ipfw set swap 5 3
```

Rules to match on packets from/to jails (By: csjp)

```
# ipfw add 10000 allow ip from any to any jail foobar
```

Rule to verify if a packet arrived via the interface the back-route points to (verrevpath) (By: cjc)

```
# ipfw add 10000 deny ip from any to any not verrevpath in
```

Packet Filters

Rule to verify the source address of a packet that is being routed (versrcreach) (By: andre)

```
# ipfw add 10000 deny ip from any to any not versrcreach in
```

Rule to verify a packet with a source address from a connected network actually arrive through that interface (By: andre)

```
# ipfw add 10000 deny ip from any to any not antispoof in
```

Lists of IP's on command line (By: luigi)

```
# ipfw add 10000 deny ip from 192.168.0.0/16, 172.16.0.0/12 to any
```

Lookup tables (implemented like routing tables, very fast for large numbers of entries) (By: ru)

```
# ipfw table 5 add 192.168.0.0/16
# ipfw table 5 add 172.16.0.0/12
# ipfw add 10000 deny ip from table 5 to any
```

TCP Processing

TCP Hostcache

The TCP hostcache contains/caches the per remote host observations from TCP. This allows to remember the path characteristics from previous connections and to pre-tune new TCP sessions to the same remote host. For HTTP connections this can provide a significant speedup on consecutive connections. (By: andre)

```
struct hc_metrics {
        /* housekeeping */
        TAILQ_ENTRY(hc_metrics) rmx_q;
        struct  hc_head *rmx_head; /* head of bucket tail queue */
        struct  in_addr ip4;    /* IP address */
        struct  in6_addr ip6;   /* IP6 address */
        /* endpoint specific values for tcp */
        u_long  rmx_mtu;        /* MTU for this path */
        u_long  rmx_ssthresh;   /* outbound gateway buffer limit */
        u_long  rmx_rtt;        /* estimated round trip time */
        u_long  rmx_rttvar;     /* estimated rtt variance */
        u_long  rmx_bandwidth; /* estimated bandwidth */
        u_long  rmx_cwnd;       /* congestion window */
        u_long  rmx_sendpipe;   /* outbound delay-bandwidth product */
        u_long  rmx_recvpipe;   /* inbound delay-bandwidth product */
        struct  rmxp_tao rmx_tao; /* TAO cache for T/TCP */
        /* tcp hostcache internal data */
        int     rmx_expire;     /* lifetime for object */
        u_long  rmx_hits;       /* number of hits */
        u_long  rmx_updates;    /* number of updates */
};
```

TCP Processing

TCP Hostcache (continued)

```
# sysctl net.inet.tcp.hostcache        Shows the status of hostcache
# sysctl net.inet.tcp.hostcache.list   Shows all entries in the hostcache
```

TCP Processing

Inflight Bandwidth-Delay Limiter

TCP maintains a send-window of how many data it can send out (inflight) before it receives an acknowledge from the remote host. When everything goes well the window opens pretty quickly until the path in between is overloaded and packet gets lost. This is then detected and the window shrinks in response. And then it opens again, resulting in the saw-tooth pattern. The Inflight code observes the timing of the ACK's and computes the delay-bandwidth product of the path. It then limits the opening of the window to exactly that amount to prevent the loss/shrink cycle which slows down the transmission. For long standing connections like FTP, FileSharing and large HTTP downloads this can provide a much smoother packet transport and thus a significant speedup. (By: dillon)

```
# sysctl net.inet.tcp.inflight.enable=1    Enabled by default.
```

TCP Processing

Compressed TIME_WAIT2 State

TCP connections in TIME_WAIT2 state (connection closed) waiting for the 2MSL timeout maintain only a minimal set of necessary information instead of a full blown TCP control block. This saves about 80% memory per connection in that state. Especially for HTTP servers this give a far better kernel memory resource usage and a higher number of concurrent connections that can be served within a short time frame ("Slashdot effect"). (By: jlemon, silby)

RFC3042 Limited Transmit

Speeds up the recovery from packet losses by sending more data faster if double ACK's are reveived. (By: hsu)

```
# sysctl net.inet.tcp.rfc3042=1        Enabled by default.
```

TCP Processing

RFC3390 Increased initial TCP congestion Window

Normally TCP will start with a window of just one packet to send out and then wait for the first ACK to arrive. RFC3390 allows for up to four packets to be sent out right away. On connections with large RTT's this give a significant speedup and allows the window to grow faster after the first ACK's are received. Especially for HTTP servers with many short connections this makes a noticeable difference. (By: hsu)

```
# sysctl net.inet.tcp.rfc3390=1        Enabled by default.
```

TCP Processing

SACK, Selective TCP ACK's

Normally when TCP experiences packet loss almost all packets from the point of the loss have to be resent even if most/all of them made it to the remote host. With SACK the remote host will ACK the packets received after the lost one as successfully arrived and thus indicate which one is missing. The sender then resend's just the missing one and continues from the highest received packet. On lossy links (like WLAN) this significantly speeds up packet loss recovery and general connection throughput. (By: ps, jayanth, Yahoo!, OpenBSD)

```
# sysctl net.inet.tcp.sack.enable=1        Enabled by default.
```

TCP Processing

TCP_MD5

Only partly implemented at the moment. But allows Zebra/Quagga routing daemons to connect to Cisco's and Junipers with MD5 signed TCP connections. (By: bms)

Needs a few things compiled into the kernel:

```
options  FAST_IPSEC
options  crypto
options  TCP_MD5
```

New Network Stacks

Bluetooth Netgraph Framework

This includes almost the entire Bluetooth specification and works with 3com Bluetooth cards and any USB Bluetooth adapter. (By: emax)

```
Man ng_bluetooth(4), ng_btsocket(4), ng_hci(4), ng_l2cap(4), bluetooth(3),
ng_bt3c(4), ng_h4(4), ng_ubt(4).
```

ATM Netgraph Framework

A new and very throughout ATM framework implementing almost all aspects of ATM packet networking and works with Fore/Marconi ATM155 and ATM622 cards, IDT77252 and Midway based cards. Drivers for the Mindspeed ATM155/622 chips are in the works. (By: harti)

```
Man ng_atm(4), natm(4), ng_atmpif(4), ng_sscfu(4), ng_sscop(4), ng_uni(4),
natmip(4).
```

Network Stack (General)

Locking concept of the network stack is per data structure.

Different concepts lock per code path (kernel threads, as in DragonFlyBSD) and/or do a fixed per-CPU split of packets and connections.

Both have advantages and disadvantages. Time and experience will tell.

That’s it. Any questions?