1) Dynamic Protocol Stacks for Linux
2) Network Debugging Toolkit Netsniff-NG

Daniel Borkmann
<daniel.borkmann@tik.ee.ethz.ch>

Swiss Federal Institute of Technology Zurich (ETH Zurich)
Computer Engineering and Networks Laboratory
Communication Systems Group

EPiCS Project (http://www.epics-project.eu/)

GTALUG, October 9, 2012
Zurich, Switzerland
Zurich, Switzerland
Zurich, Switzerland
Communication Systems Group

- Performs research in ...
  - Wireless mobile networks and social networks
  - Network measurement and security
  - Future Internet architectures
The Internet Architecture ...

- Static, layered model
- IP used as “glue” between application and physical layer
... and its Limitations!

- Originally designed for a fixed infrastructure
- No provisions for resource limitations
- Difficult to integrate/deploy completely new network functionality
  - E.g. first IPv6 RFC from 1995
- Impossible to bypass unnecessary layers

→ Leads to dedicated network architectures for e.g. sensor networks
How would an Alternative look like?

- Here: clean slate architecture
- Meta-architecture does not enforce the protocols to be used
- Protocol functionality divided into “functional blocks” (FBs)
What are the Benefits?

- One unified framework, where in each situation the optimal protocol stack can be built
  - Sensor networks vs. supercomputer interconnect
  - Servers, desktop nodes, embedded systems, ...
  - Pervasive computing: mobile phones, clothes, fridges, ...

- The protocol stack can be changed on the fly
  - Insert encryption, compression, reliability, ... on demand
  - Reboot-less updating of functionality, bugfixing

- New protocols can be added/deployed to the stack easily

- Probably, we can also put intelligence into the stack
Implemented in the Linux kernel space

Stack configuration from user space

Question: when/how is our code invoked?
1. PCI MSI-X IRQ Signal
2. Device driver 3c59x
3. Network card (hardware)

1. do_IRQ
2. irq_exit
3. irq_enter

4. netpoll framework
5. netif_rx
6. corkscrew_interrupt
7. boomerang_rx

8. DMA Receive Ring Buffer

__raise_softirq_irqoff(NET_RX_SOFTIRQ)

__napi_schedule

---

Daniel Borkmann (ETHZ) Dynamic Protocol Stacks / Netsniff-NG October 9, 2012 14 / 50
Linux Packet Processing Path, Ingress

For all registered NAPI functions:
- `net_rx_action`
- `drivers_napi_rx_function`

Assuming NAPI is implemented in the driver:
- `netif_receive_skb`
- `enqueue_backlog`
- `__netif_receive_skb`

Assuming the packet came via `netif_rx` or receive packet steering was triggered:
- `process_backlog`

- For all packets:
  1. `sk_buff`
  2. `CPU sk_buff backlog queue; if wrong receive packet steering`

- `CPU-local softirq thread`:
  - `ksoftirqd`

- Depending on Kernel setup:
  - `kernel Process Scheduler`
  - `wakeup_softirqd`

- Depending on Kernel setup:
  - `do_softirq`
  - `__do_softirq`
  - `invoke_softirq`
  - `irq_exit`

- `netpoll framework`
- `deliver_skb`
- `VLAN processing`
- `rx_handler`
- `IPv4, IPv6, ... protocol handler` (entrance to the protocol stack)
- `E.g. Bridging`

- `Entrace Point of our Stack`
Packet Processing Path, Egress

Dynamic Protocol Stack

Design, Software

Userspace

buffer; userspace

Kernel space

buffer; kernelspace

memcpy_fromiovec

Socket / Systemcall Context

Or called from somewhere within the kernel

dev_queue_xmit

dev_hard_start_xmit

netif_rx

Virtual network device or loopback device

Loopback device

Qdisc of RED or HTB or ...
implementation

Processing in our Protocol Stack

Daniel Borkmann (ETHZ)
Packet Processing Path, Egress

1. If network device is busy and Packet quota exceeded or CPU is needed elsewhere

2. For all packets

   - _qdisc_run
      - qdisc_restart
      - dequeque_skb(qdisc)
      - sch_direct_xmit

   - __netif_schedule
     - _netif_reschedule(qdisc)

   - raise_softirq_irqoff(NET_TX_SOFTIRQ)

   - skb_checksum_start_offset
     - _skb_linearize
     - ndo_start_xmit

   - ksoftirqd

   Software Interrupt Context

   - NET_TX_SOFTIRQ
     - do_softirq
       - __do_softirq
     - net_tx_action
     - qdisc_run

   CPU-local softirq thread

   - dev_queue_xmit_nit
     - Copies and hands packet to network sniffer / TAP applications

   END HERE

   - dev_hard_start_xmit
     - If device is busy
      - dev_requeue_skb
      - __vlan_put_tag

   End here
Implementation, General

- Concepts: modularity, layers of indirection
  - Functional blocks (fb)
  - Access to fbs via mapping (idp → pointer)

- PPE invoked from two different execution contexts
  - Per-CPU ksoftirqd (ingress)
  - CPU local socket system call (egress)

- Lightweight locking mechanisms and memory access all over the place
  - RCU locks
  - Sequential locks
  - Per-CPU “backlog” queues in PPE
Implementation, Functional Blocks

- Current repository: fb_eth, fb_ire, fb_bpf, fb_counter, fb_dummy, fb_otp, fb_tee, fb_pflana

- (Manual) configuration example:
  - fbctl add fb0 ch.ethz.csg.ire
  - fbctl add fb1 ch.ethz.csg.bpf
  - fbctl bind fb1 fb0
  - fbctl unbind fb1 fb0
  - fbctl rm fb1
Implementation, Summary

Application (via syscall)

Network Interface (via ksoftirqd)

- rcv_ctrl
- rcv_data
- pf_lana

- rcv_ctl
- rcv_data
- fb_eth

- backlog
- proc_data
- PPE

- rcv_ctrl
- rcv_data
- fb_x

- netlink config
- device file config
- add_ctrl
- notification chain

user space

kernel space
Implementation, Packet Handler Example

```c
static int fb_dummy_netrx(const struct fblock * const fb,
       struct sk_buff * const skb, enum path_type * const dir)
{
    int drop = 0;
    unsigned int seq;
    struct fb_dummy_priv *fb_priv = rcu_dereference_raw(fb->private_data);

    do {
        seq = read_seqbegin(&fb_priv->lock);
        write_next_idp_to_skb(skb, fb->idp, fb_priv->port[*dir]);
        if (fb_priv->port[*dir] == IDP_UNKNOWN)
            drop = 1;
    } while (read_seqretry(&fb_priv->lock, seq));

    if (drop) {
        kfree_skb(skb);
        return PPE_DROPPED;
    }

    /* Do some protocol processing ... */
    return PPE_SUCCESS;
}
```

Daniel Borkmann (ETHZ)
Implementation, Event Handler Example

```c
static int fb_dummy_event(struct notifier_block *self,
                           unsigned long cmd, void *args)
{
    int ret = NOTIFY_OK;
    struct fblock *fb;
    struct fb_dummy_priv *fb_priv;

    rcu_read_lock();
    fb = rcu_dereference_raw(container_of(self, struct fblock_notifier, nb)->self);
    fb_priv = rcu_dereference_raw(fb->private_data);
    rcu_read_unlock();

    switch (cmd) {
    case FBLOCK_BIND_IDP:
        ...
        write_seqlock(&fb_priv->lock);
        fb_priv->port[msg->dir] = msg->idp;
        write_sequnlock(&fb_priv->lock);
        ...
        break;
    ...
    }
    ...
    return ret;
}
```
Evaluation, Stacks

Kernel space LANA versus Linux, Receive and Drop

Packets per Second
Packet Size in Bytes (RFC2544)

- Physical max
- Linux max
- LANA max
- LANA, 3 FBs
- LANA, 5 FBs
- LANA, 10 FBs
Evaluation, Sockets

User space BSD Socket Sniffer: PF_LANA versus PF_PACKET

- Physical max
- Linux max
- Linux PF_PACKET
- LANA max
- LANA PF_LANA

Packets per Second vs. Packet Size in Bytes (RFC2544)
Taking this one Step further: Dynamic Hardware/Software Mapping of FBs

- During runtime, we want to dynamically map software FBs into hardware FBs and vice versa
- Reasons:
  - Better performance
  - Energy efficiency
- Hence, a node’s stack can partly be in hardware, partly in software
- Xilinx ML605 (FPGA) Board, Xilinx partial reconfiguration (icap)
Architecture Design, Big Picture

Applications

BSD Socket Interface

PPE

Device Driver

Interconnect

Physical Interface

Linux userspace

Linux kernelspace

Software

Hardware

Ethernet

FB_1

FB_n

FB_s

FB_s2h

FB_h2s

FB_x

FB_y
Implementation, General

- Line-rate Network on Chip (NoC) implementation in VHDL
- Possibility for runtime partial reconfiguration of FPGA
- Usage of ReconOS, reconfigurable OS based on Linux
Lets go one layer up and have a look at the user space!

- Now that we have a nice reconfigurable backend, it’s time to make it more intelligent!
- Especially for sensor nodes, we could adapt the stack dynamically to environmental changes ...
  - E.g. battery lifetime vs. performance

- 2 new tools: sensord, configd
  - sensord: collection and analysis of sensor data
  - configd: stack builder, (re)configuration engine
sensord and configd
sensord Implementation

- Executable with collection of plugins as shared objects
- Plugins dynamically pluggable during runtime
- Scheduling via internal timer delta queue
- Collected data values stored in `timedb` round robin database (no, not RRDtools!)

- Clients can register for “threshold exceeded” notifications
  - Registration via Unix domain sockets
  - Notification alarm via SIGUSR1, SIGUSR2 (lower, upper thresholds)
  - Plugin and measurement value passing via shared memory
sensord Plugin Example, 1

```c
static void dummy_fetch(struct plugin_instance *self)
{
    int i;
    for (i = 0; i < self->cells_per_block; ++i) {
        self->cells[i] = ((double) rand() / (double) RAND_MAX);
    }
}

struct plugin_instance dummy_plugin = {
    .name = "dummy-1",
    .basename = "dummy",
    .fetch = dummy_fetch,
    .schedule_int = TIME_IN_SEC(1),
    .block_entries = 1000000,
    .cells_per_block = 2,
};
```
sensord Plugin Example, 2

static __init int dummy_init(void)
{
    struct plugin_instance *pi = &dummy_plugin;

    srand(time(NULL));
    pi->cells = xmalloc(pi->cells_per_block * sizeof(double));

    return register_plugin_instance(pi);
}

static __exit void dummy_exit(void)
{
    struct plugin_instance *pi = &dummy_plugin;
    free(pi->cells);
    unregister_plugin_instance(pi);
}

plugin_init(dummy_init);
plugin_exit(dummy_exit);

PLUGIN_AUTHOR("Daniel Borkmann <daniel.borkmann@tik.ee.ethz.ch>");
PLUGIN_DESC("A simple dummy sensord plugin");
Applications register their PF_LANA sockets to configd

configd registers itself to sensord, e.g. for wireless link quality notifications
configd, Application Registration

```c
int main(void)
{
    ... sock = socket(PF_LANA, SOCK_RAW, 0); if (sock < 0)
        panic("Cannot create socket!\n"); ...
    memset(buff, 0, sizeof(buff));
    bmsg = (struct bind_msg *) buff;
    strlcpy(bmsg->app, "temp-sensor");
    bmsg->props[0] = RELIABILITY;
    bmsg->flags = TYPE_CLIENT;

    /* Query kernel space fblock name */
    ret = ioctl(sock, SIOFBNAM, bmsg->name);
    ...
    ret = bind_config(bmsg);
    ...
    ret = sendto(sock, data, len, 0, NULL, 0);
    ...
    close(sock);
    return 0;
}
```
Internode Communication

Now we have the backend, sensord, configd, but how do nodes talk with each other?
Internode Stack Reconfiguration, 1

1. Stack builder determines possible protocols.
2. Data packets are sent.
3. Stack builder chooses a protocol and builds the stack.
4. The stack builder determines possible protocols at the destination.
5. The chosen protocol is sent.
6. The stack builder builds the stack.
7. The process repeats.

Source destination
possible protocols
chosen protocol
data packets
stack builder
choose protocol, build stack
stack builder
determine possible protocols
stack builder
build stack
stack builder
choose protocol, build stack
stack builder
build stack
Internode Stack Reconfiguration, 2

source

destination

chosen protocol

possible changes

packet 3 (hash2)

packet 1 (hash1)

packet 2 (hash1)

packet 4 (hash2)

APP

hash1

Ethernet

APP

FB1

hash1

hash2

Ethernet
destination

APP

hash1

Ethernet

FB1

hash1

hash2

Ethernet

APP

FB1

hash1

hash2

Ethernet
Internode Stack Reconfiguration, Impl.

node A: initiates a stack modification

sensor daemon

stack builder

socket

(Application)

(PF_UNIX, requirements)

write(stacks)

ctrl_msg(stacks)

pf_lana

(add ctrl tag)
(read ctrl tag)

/write(lana_re_cfg)

write(stack)

reconfig(stack)

send(stacks)

protocol stack

(add hash)
(read hash)

reconfig(stack)

read(stack)

write(stack)

request path

reply path

node A: initiates a stack modification
Example Scenario

- Two (or more) nodes, connected over wireless
- Periodically exchange (e.g.) temperature information
- Transmission has to be reliable
- Nodes run on battery, aim also to be energy-efficient

Testlab experiment:
- `sensord` with link-quality plugin, that notifies `configd`
- Triggers `fb Irr` to be included when needed resp. excluded when not
Source Code

- Prototype implementation released under GPLv2.0
- Directly included into ReconOS
- Github: https://github.com/EPiCS/reconos
- Research publications:
  - http://www.epics-project.eu/
  - http://www.csg.ethz.ch/people/arkeller
  - http://borkmann.ch/
High-Performance Network Debugging

netsniff-ng toolkit

- netsniff-ng, a high-performance zero-copy analyzer, pcap capturing and replaying tool
- trafgen, a high-performance zero-copy network traffic generator
- mausezahn, a packet generator and analyzer for HW/SW appliances with a Cisco-CLI
- bpfc, a Berkeley Packet Filter (BPF) compiler with Linux extensions
- ifpps, a top-like kernel networking and system statistics tool
- flowtop, a top-like netfilter connection tracking tool
- curvetun, a lightweight multiuser IP tunnel based on elliptic curve cryptography
- astraceroute, an autonomous system (AS) trace route utility
The Toolkit

- **Here**: focus on netsniff-ng, trafgen, mausezahn
- Used to debug and stress-test our dynamic protocol stack
- Rx/Tx zero-copy: no copies between kernel and user space
- Users reported higher capturing/transmission rates on 1-10 Gbps than commonly used tools (tcpdump/libpcap, Wireshark, ...)
- Part of all the big distributions, plus Backtrack, GRML, Xplico, NST, Alpine Linux, Scientific Linux/CERN
netsniff-ng

- High-performance traffic analyzer, replayer
- PCAP files compatible with tcpdump, Wireshark, ...

```
  netsniff-ng --in eth0 --out dump.pcap -s -b 0
  netsniff-ng --in wlan0 --rfraw --out dump.pcap -s -b 0
  netsniff-ng --in dump.pcap --mmap --out eth0 -s -b 0
  netsniff-ng --in eth1 --out /opt/probe/ -s -m -J --interval 30 -b 0
  netsniff-ng --in any --filter ip4tcp.bpf --ascii
```
netsniff-ng, Filtering

```c
ldh [12] ; Load Ethernet type field
ejq #0x800, Cont, Drop ; Check value against 0x800
Cont: ldb [23] ; Load IPv4 proto
jej #0x6, Keep, Drop ; Check against 0x6 (TCP)
Keep: ret #0xffffffff ; Return packet
Drop: ret #0 ; Discard packet
```

- `bpfc ip4tcp.bpfa > ip4tcp.bpf`, then pass it to `--filter`
- Or abuse tcpdump: `tcpdump -dd my-filter`
- Filtering done in the Linux kernel (BPF virtual machine)
- Newer kernels: BPF JIT for x86/x86_64, powerpc, sparc
trafgen

- Low-level, high-performance traffic generator
- `trafgen --dev eth0 --conf packets.txf -b 0`
- `trafgen --dev wlan0 --rfraw --conf beacon.txf -b 0`
- `trafgen --dev eth0 --conf trafgen.txf -b 0 --num 10 --rand`

- Own configuration language:

  ```
  { 0x00, 0x01, 0x03, fill(0xff, 60), 0x04 }
  { 0x00, 0x01, 0x03, rnd(60), 0x04 }
  { drnd(64) }
  { 0x00, 0b00110011, 0b10101010, rnd(60), 0x04 }
  ```
trafgen

- Uses PF_PACKET sockets with mmap(2)'ed TX_RING
- Users have reported wire-rate performance from user space
- Low-level packet configuration, more flexible than pktgen
mausezahn

- High-level, (not so) high-performance traffic generator
- Taken over development and maintainership
- Has a Cisco-like CLI, but also a normal cmdline interface
- Intended for HW/SW appliance in your lab

```bash
mausezahn eth0 -A rand -B 1.1.1.1 -c 0 -t tcp 
"dp=1-1023, flags=syn" -P "Good morning! This is a SYN Flood Attack. We apologize for any inconvenience."
```

```bash
mausezahn eth0 -M 214 -t tcp "dp=80" -P "HTTP..." -B myhost.com
```
Comparison of Traffic Generators

- trafgen
- mausezahn
- pktgen

Transmitted Packets per Second vs. Packet Size in Bytes
What’s next in netsniff-ng?

- The usual: cleanups, extend documentation, man-pages
- bpf-hla, high-level language for filtering
- DNS traceroute to detect malicious DNS injections on transit traffic
- Compressed on-the-fly bitmap indexing for large PCAP files
- New protocol dissectors/generators for netsniff-ng/mausezahn
- Further performance optimizations (OProfile is your friend)
- Hack net/packet/af_packet.c for a better performance
Source Code

- Toolkit released under GPLv2.0
- Website: http://www.netsniff-ng.org/
- Github: https://github.com/gnumaniacs/netsniff-ng
- Patches, feedback are welcomed!