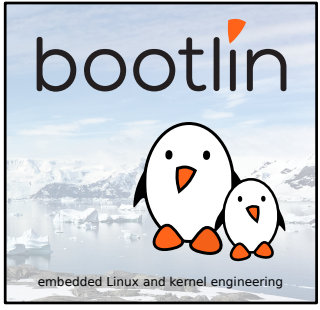
Embedded Linux networking training



Latest update: July 03, 2025.

Document updates and training details:
https://bootlin.com/training/networking

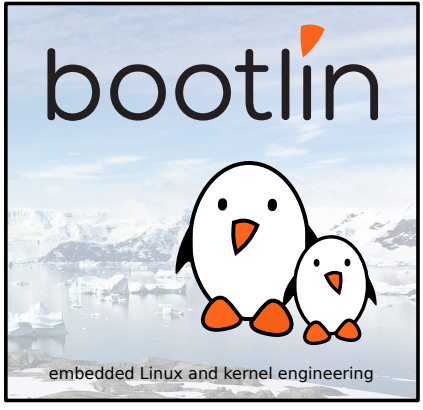
Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

Embedded Linux networking training

- These slides are the training materials for Bootlin's *Embedded Linux networking* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/networking
- Contact: training@bootlin.com

Icon by Eucalyp, Flaticon

About Bootlin



Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

bootlin

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development On-site: 4 or 5 days Online: 7 * 4 hours	Linux kernel driver development On-site: 5 days Online: 7 * 4 hours	Yocto Project system development On-site: 3 days Online: 4 * 4 hours	Buildroot system development On-site: 3 days Online: 5 * 4 hours	Embedded Linux networking On-site: 3 days Online: 4 * 4 hours
Understanding the Linux graphics stack On-site: 2 days Online: 4 * 4 hours	Embedded Linux audio On-site: 2 days Online: 4 * 4 hours	Real-Time Linux with PREEMPT_RT On-site: 2 days Online: 3 * 4 hours	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours	

All our training materials are freely available
under a free documentation license (CC-BY-SA 3.0)
See https://bootlin.com/training/

Bootlin, an open-source contributor

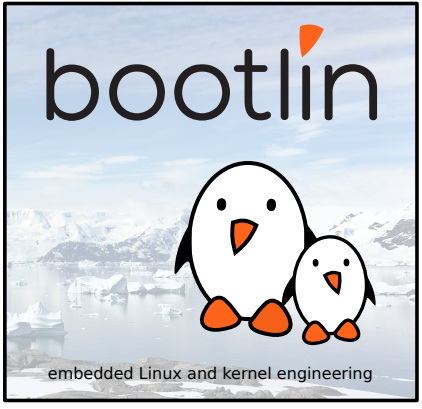
- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 9000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 6000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

Icon by Freepik, Flaticon

Generic course information



Supported hardware

Espressobin from Globalscale

- Marvell Armada 3720 SoC (Dual ARM64 Cortex-A53 CPU)
- SoC with powerful Network Controller (up to 2.5Gbps), SATA, PCIe
- 1 GB of RAM
- 8 GB of on-board eMMC storage
- Marvell 88e6341 Switch with 3 Gbps interfaces

Training quiz and certificate

- You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.

Participate!

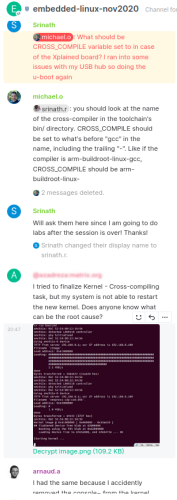
During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please always keep your camera on!
  - Also make sure your name is properly filled.
  - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

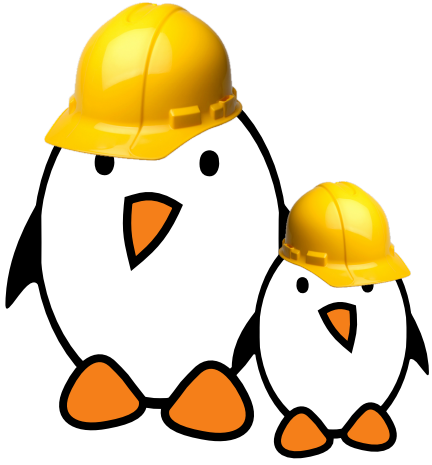
Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.



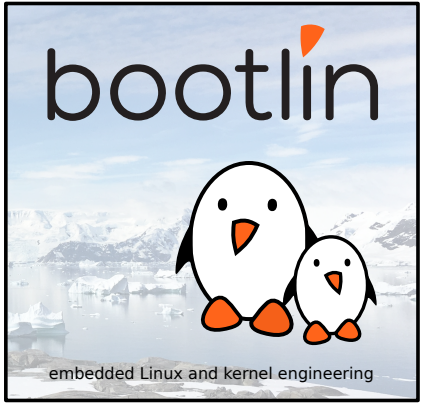
Practical lab - Training Setup



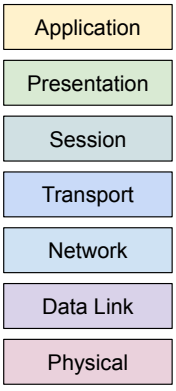
Prepare your lab environment

- Download and extract the lab archive

Introduction - Networking Technologies

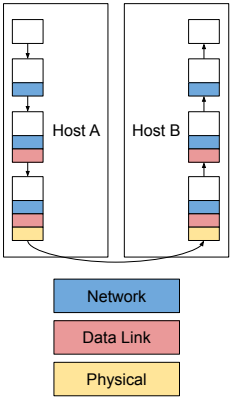


OSI Model



- **O**pen **S**ystems **I**nterconnection
- Reference model to design network protocols, created in the late 1970s
- Defines 7 **layers**, with specific functions and semantics
- Each layer relies on the layer below, and provides features to the layer above
  - In practise, this is done through **encapsulation**
  - Each layer add its required data at the front of the data array
    - This is the layer's **header**
  - A Layer N's header is part of Layer N-1's payload

OSI layer communication



- The payload sent by a layer targets the same layer on the receiving end
- Each layer only cares about its specific header and treats the payload as a **black box**
- When the **peer** receives it, it **decapsulates** the received data
- Every layer has its own semantics about the data it manipulates
- Every layer's unit is called **P**rotocol **D**ata **U**nit
- Every layer has a **M**aximum **T**ransmit **U**nit per PDU

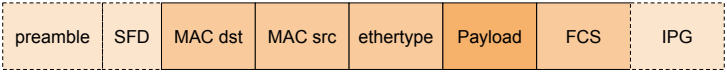
Layer 1 - PHY Layer

- Defines how data is sent to a peer trough a **physical medium**
- PDU is **symbols** and **bits**
- IEEE 802.3 "Ethernet" defines a lot of Layer 1 technologies
  - 1000BaseT4 : Transmit data at 1000Mbps over 4 twisted copper pairs
  - 1000BaseFX : Transmit data at 1.25Gsps / 1Gbps over an optics fiber
  - 10BaseT1S : Transmit data at 10Mbps over a single twisted copper pair
  - Ethernet protocols may use the same medium
    - Protocol selection can be done through **autonegotiation**
    - Link detection is done by sending **Idle** words
- IEEE 802.11 "Wifi" also defines Layer 1 technologies using 2.4/5/60GHz radio modulation
- Many more exists : IEEE 802.15.4, "Bluetooth", NFC, etc.
- Usually handled by a dedicated hardware component : a **PHY**

Layer 2 - Data Link Layer

- Sometimes called **MAC** layer
- PDU is a **Frame**
- In charge of Point-to-point communication
- IEEE 802.3 "Ethernet" defines a Layer 2 standard as well
  - Source address, Destination address, Layer 3 type
- IEEE 802.11 "Wifi" also defines a Layer 2
  - 2, 3 or 4 addresses
  - Receiver, Transmitter, Source and Destination

Ethernet - Layer 2

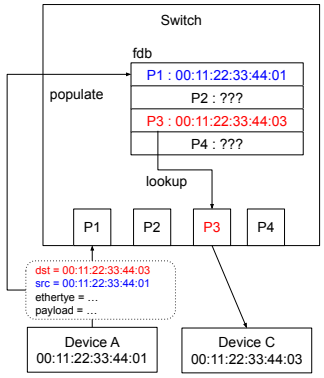


- Point-to-point **frames** are sent on the medium, separated by a **gap**
- Regardless of the speed and medium, frames have the same structure :
  - 7 bytes **Preamble**: Used to synchronize both equipments
  - 1 byte **SFD** (Start Frame Delimiter): Ends the preamble
  - 6 bytes **Destination address**, identifying the destination equipment
  - 6 bytes **Source address**, identifying the source equipment
  - 2 bytes **ethertype**, identifying the encapsulated protocol
  - A **payload**
  - 4 trailing bytes **FCS** (Frame Check Sequence): Checksum of the frame
- Each frame must be separated by at least 12 bytes, named the **I**nter **P**acket **G**ap
- $header + payload + fcs <= 1522$ bytes, the payload's size is at most **1504 bytes**
- $header + payload + fcs >= 64$ bytes, the payload must be *zero-padded* otherwise

Network Bridging

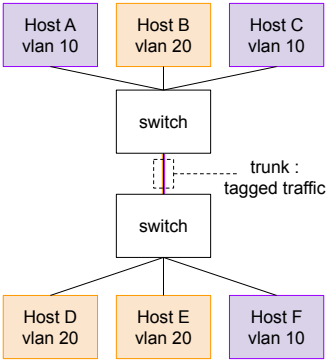
- A **Network bridge** is a Layer 2 device interconnecting multiple network segments
- We usually use a dedicated **Network Switch** for this purpose
  - Using a **ASIC** chip
  - Using a software implementation
- Ethernet switches are usually **transparent switches**
  - They monitor Layer 2 header to **learn** the Port to Address asscciation
  - This is stored in the FDB : **F**orwarding **D**ata**B**ase
- Some switches can be highly configurable, and support VLANs

Transparent bridge



- Ports are monitored, the **source address** saved
- The **destination address** is looked-up
- If no match is found, the frame is **flooded** on all ports
- Advanced switches can do **port mirroring**
  - Duplicate traffic going forwarded a port to another port
  - Used for administration and troubleshooting

VLANs



- **V**irtual **L**ocal **A**rea **N**etwork
- Multiple VLAN technologies exist:
  - 802.1Q (*dot1q*): Layer 2
  - 802.1AD (*QinQ*): Layer 2
  - VxLan : Layer 4 (UDP)
  - MACVlan : Based only on MAC addresses
- Logical segmentation of the network
- Used for isolation, priorisation and bandwitdh optimisation
- The same conduit can be used to convey multiple VLANs
  - We talk about a **trunk** interface
  - Frames are **tagged** to indicate the VLAN it belongs to

VLAN - 802.1Q

- A 802.1Q frame has includes an extra **4 bytes** tag in the Ethernet header
- The **ethertype** is set to `0x8100`, the real ethertype is stored after the tag
- A 16 bits value identifies the Tag : **T**ag **C**ontrol **I**nformation
  - 3 bits indicate a **priority**, between 0 and 7
    - Also called **C**lass **o**f **S**ervice
  - 1 bit **D**rop **E**ligible **I**ndicator
  - 12 bits represent the ID of the vlan, between 1 and 4094
    - ID 0 means **no tag**, only the priorty is consided
    - ID **4095** is reserved.

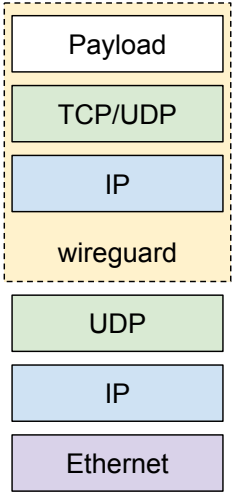
Layer 3 - Network Layer

- PDU is **packet** or **Segment**
- Handles routing between multiple machines
- Defined by **subnets**, linked tothegher by **routers**
- Main technologies are **IPv4** and **IPv6**
  - IPv4 : 32-bits addresses, IPv6 : 128-bits addresses
- Layer 2 to Layer 3 addresses can be associated
  - e.g. the **A**ddress **R**esolution **P**rotocol
  - MAC to IP tables are named **ARP** or **neighbouring** tables
- Layer 3 can perform **fragmentation**
  - e.g. if an IPv4 packet is too big to fit within the Ethernet MTU, it is split into multiple IPv4 packets
  - each packet is individually routable
  - Re-assembly is done by the peer

Transport Layer

- Communication between endpoints over a routed network
  - *e.g.* Multple applications on the same machine
  - Each end-point is further identified within the host
  - on TCP and UDP, **ports** are used
- TCP : Connection-oriented, reliable, guarantees ordering.
  - Stream of data, boundaries may be be preserved
- UDP : Connection-less, not reliable, no ordering guarantee
  - Sends datagrams with clear boundaries
- QUIC : Based on UDP, introduced by Google. Connection-oriended, reliable, guarantees ordering
  - Can batch Acknowledgments, supports encryption (i think)

Tunneling

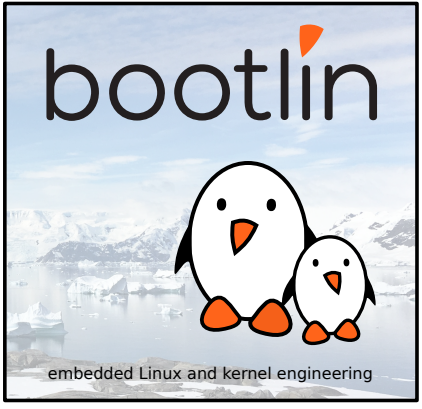


- Encapsulate a Lower-level layer into a higher one
- Used to virtualise networks (e.g. VXLAN is Ethernet over UDP)
- Also used for encryption (IPSec, Wireguard, OpenVPN)
  - *e.g.* Wireguard Encrypts and encapsulates IP packets into UDP packets
- There may therefore be more headers to decapsulate than there are ISO layers

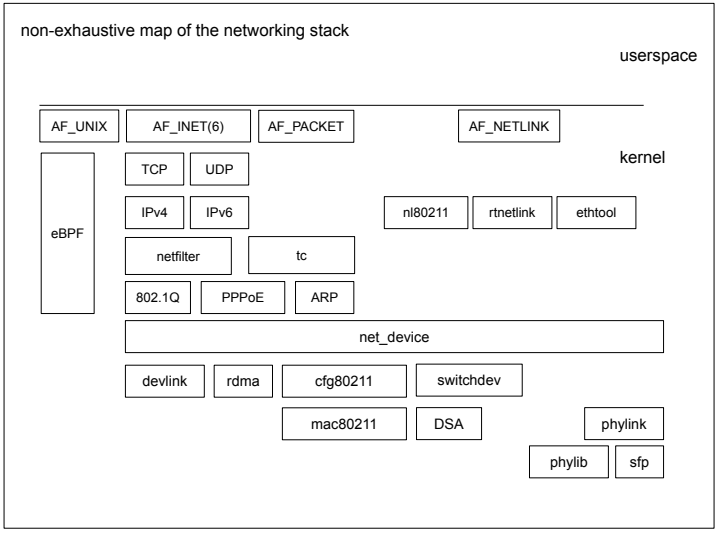
Layers 5, 6 and 7

- Session : Handles the connection, authentication and lifetime of data exchanges
  - *e.g.* RPC, SOCKS
- Presentation : Handles the data conversion and serialization for interoperability
  - *e.g.* character transcoding depending on the locale
  - *e.g.* serializing user data in `JSON` or `XML`
- Application : Communication between user applications
  - *e.g.* HTTP for Web applications
- *less relevant for this training, as they aren't handled by the linux kernel*

The Linux Kernel Networking Stack



The Linux Kernel Networking Stack



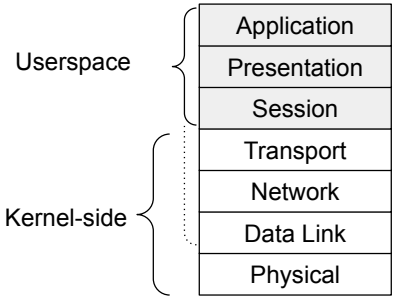
As of v6.16-rc1 :

- over 209000 commits
- over 7750 files
- over 993000 LoC
- around 100 maintainers
- Around 880 drivers (Layer 2)

Some history

- First support was introduced in v0.96 (may 1992) !
- Already exposing a socked-based API
- IPv4 : v0.98 - September 1992
- TCP/UDP : v0.98 - September 1992
- IPv6 : v2.2 - January 1999 (IPv6 was created in 1998)
- BPF : v2.5 - 2003, was then known as **L**inux **S**ocket **F**iltering
  - eBPF : v3.15 - June 2014
- PHYlib : v2.6.13 - August 2005, before that PHYs were handled in MAC drivers
- XDP : v4.8 - September 2016
- phylink : v4.13 - September 2017

Networking in the Linux Kernel



- Abstracts the Network Devices
- Implements some OSI Layers :
  - Layer 1 (PHY) : Ethernet, WiFi, CAN, etc.
  - Layer 2 (MAC) : Bridging, VLANs, etc.
  - Layer 3 (Network) : IPv4, IPv6, etc.
  - Layer 4 (Transport) : TCP, UDP, etc.
- Provides a set of APIs to userspace :
  - Socket API and `io_uring`
  - Control through ioctl and Netlink

Physical and MAC support

- The Networking stack provides a framework for Layer 2 drivers : `struct net_device`
- Used by Ethernet, Wifi, Bluetooth, CAN, 802.15.4, Radio, etc.
- PHY drivers have their dedicated frameworks
  - phylib for Ethernet PHYs
  - mac80211 and wiphy for 802.11 PHYs
- A lot of communication technologies are handled through the network stack
  - Ethernet
  - Wifi
  - Bluetooth and Bluetooth Low Energy
  - Infiniband
  - 802.15.4, radio, X.25
  - CAN Bus

Ethernet

- Ethernet MAC controllers are supported through regular `struct net_device` as well as `ethtool`
- Switch drivers are supported, with offload operation going through Switchdev
- Standalone Ethernet Switches are handled through DSA
- Ethernet PHYs are supported via phylib, and the MAC to PHY link via phylink
- SFF and SFP cages and modules are also supported
- Supports 802.3 frames and Ethernet II
- Multiple 802.1 and 802.3 Low-Level aspects are supported :
  - Vlan with 802.1Q and 802.1AD
  - Bridging and Switching
  - MACSec (802.1ae) for Ethernet-level encryption
  - Teaming, Bonding, HSR and PRP for link redudancy
- Raw Ethernet frames can be sent and received in userspace API with `AF_PACKET`

Wireless subsystem

- Wifi (802.11) Stack :
  - Supports Wifi chips with internal MAC stack (**hardmac**)
  - Also provides a 802.11 MAC stack for **softmac** drivers in `mac80211`
  - The main implementation is in `cfg80211`, configured via `nl80211`
- Bluetooth stack :
  - Low-level support for Bluetooth and BLE
  - Exposes a socket-based API for management and data
  - Profiles are implemented either in the kernel or userspace
  - BlueZ is the main userspace companion stack
- 802.15.4 stack :
  - Also provides **hardmac** and **softmac** support
  - Has its own PHY layer
  - Complemented by the **6lowpan** stack for upper levels
    - 6lowpan can also be used with Bluetooth Low Energy

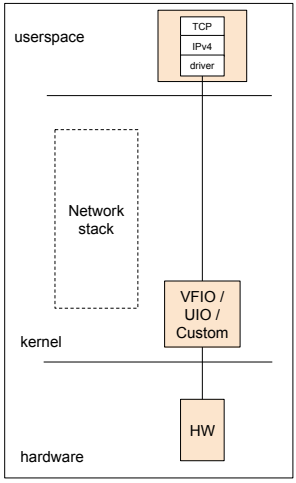
Other Technologies

- X25 / AX25 : Amateur radio protocols. Long-standing support in Linux.
- Infiniband : Used for very high speed link, usually in datacenters
  - Layer 1 and Layer 2 technology (like Ethernet)
  - Allows using RDMA : Remote Direct Memory Access
  - provides a VERBS-based API (IB Verbs) , not sockets
- RoCE : RDMA over Converged Ethernet
  - RDMA over Ethernet-based networks (instead of Infiniband)
  - Works on top of Layer 4 (RoCE v2)
- CAN bus : Controller Area Network Bus
  - Widely used in automotive and industrial equipment
  - Support implemented using the Network Stack, socket-based

Userspace Networking

- Userspace applications can also access traffic at various points in the stack through sockets :
- `AF_PACKET` Sockets allows raw Layer 2 access
  - Can be used for custom protocol support in userspace
  - Used by `libpcap` and traffic monitoring tools like **tcpdump** and **wireshark**
- `socket(AF_PACKET, SOCK_DGRAM, 0)` sockets expose raw IPv4 and IPv6 packets
- Some protocols only have userspace implementations by design :
  - QUIC : Not in the kernel when the protocol was first intrduced
    - Rationale was to prevent ossification
    - Recent kernel-side implementation submitted for inclusion in the kernel

Kernel Bypass



- Contrary to AF_PACKET, Kernel Bypass techniques circumvent the networking stack
- Allows using an alternative implementation of the network stack, entirely running in userspace
  - For use-case optimised scenarios
- DPDK : Data Plate Development Kit
  - Re-implement the drivers in userspace as well as a custom stack
- Not supported by the linux kernel community
- Implies re-writing a full driver in userspace
- With XDP + AF_XDP, we now have a fully upstream solution

Netdev community

- The networking stack is made of around 1M lines, 7000 distinct files
- 4 Maintainers share the top-level load :
  - Jakub Kicinski, David S. Miller, Eric Dumazet and Paolo Abeni
- Lots of maintainers for specific aspects of the networking stack
  - Wireless, Bluetooth, TC, Ethernet framework, PHY framework, individual drivers...
- Very active subsystem with lots of contributions and reviews.

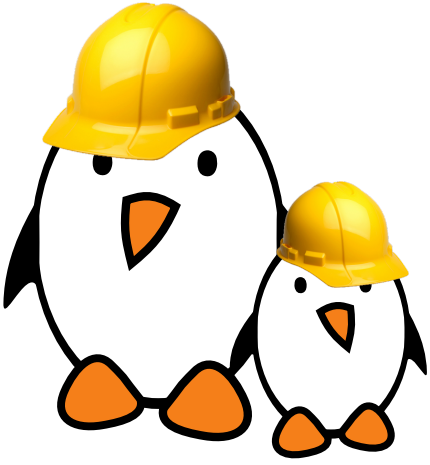
Contributing

- Development occurs on the `netdev@vger.kernel.org` mailing list, see archives
- Follows the kernel development cycle, with a 2 weeks break during the merge window
- 2 git repositories are used as a development basis:
  - net-next : For **new features**, development stops during the Merge Window
    - Check the status page before sending patches !
  - net : For **fixes**, always open to patches.
- Very fast-paced development, replies arrive quickly, for quick iterations
- Patch status can be tracked on patchwork
- Automated build-test and runtime tests are run with NIPA, results are published here

Conferences

- The **Linux Plumbers Conference** hosts a Networking Track
  - Main maintainers attend and host the track
  - For **ongoing development**, to discuss current issues and future work
  - Very technical topics
  - Usually single-day track on a multi-day event
  - LPC 2025 will be in Tôkyo, Japan, in December
- The **Netdev Conference** is dedicated to kernel networking development
  - Main maintainers also attend
  - Hosted by a dedicated group of individuals (the Netdev Society)
  - 4 or 5 days, mixing remte and on-site sessions
  - Very technical topics as well, not many are embedded-oriented
  - Netdev 2025 was in Zagreb, Croatia, in April.

Practical lab - Setup lab



- Build the image used for the whole training
- Setup the host machine

Network Devices

Network devices in Linux

- In UNIX systems, the common saying is that "*everything is a file*"
- Most classes of devices follow that rule, and expose **block** and **char** devices in `/dev`
  - `/dev/mmcblk0` : eMMC device 0
  - `/dev/i2c-3` : I2C bus number 3
  - `/dev/input/*` : HID devices
- Network devices don't follow that rule, as they are rarely directly accessed
- The Linux Kernel provides access to Layers 2, 3 and 4 through the **socket** API
- Network devices appear as **interfaces** under `/sys/class/net`
- The `sysfs` API is only for limited control and device information

struct net_device

- The `struct net_device` structure represents a conduit
- Used for **physical** interfaces and **virtual** interfaces
- Abstract interfaces can be used for **vlan**, **bridging**, **tap**, **veth**, etc.
- Every `struct net_device` objects can transmit and receive packets :
  - Physically, in which case it is managed by a device driver
  - or Logically, by passing them to another component in the stack after potientally altering them
- Instances of `struct net_device` are often called `netdev` in the Documentation
- Variables of that type are usually named `dev`
  - Unfortunately, this is als the usual name of `struct device` objects

struct net_device (2)

- Userspace sees a `netdev` as an **interface**
  - Listed with `ip link show` or `ifconfig`
  - Also appearing under `/sys/class/net/`
- Interfaces have a **name**, which may change
- Interfaces also have an index (**ifindex**) that uniquely identifies them
- They have attributes, changeable or not, depending on their type :
  - Addresses : IPv4, IPv6, MAC, etc.
  - Properties : MTU, Queue length, etc.
  - Statistics : RX/TX packets, link events, etc.
  - State : Link up or down, admin state, Promiscuous, etc.

Network driver

- Creating a new Network Interface driver is similar to any other driver :
- The driver registers a **device driver** on its underlying bus :

```
static const struct of_device_id mvneta_match[] = {
        { .compatible = "marvell,armada-3700-neta" },
        { }
};

static struct platform_driver mvneta_driver = {
        .probe = mvneta_probe,
        .remove = mvneta_remove,
        .driver = {
                .name = MVNETA_DRIVER_NAME,
                .of_match_table = mvneta_match,
        },
};
module_platform_driver(mvneta_driver);
```

- In the `.probe()` function, allocate a `struct net_device` :

```
dev = devm_alloc_etherdev_mqs(&pdev->dev, sizeof(struct mvneta_port),
                              txq_number, rxq_number);
```

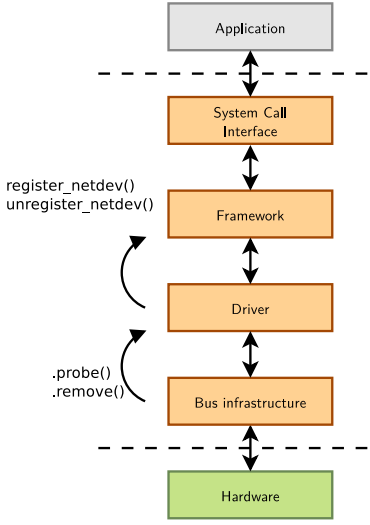
- The `netdev` is **registered** to the networking subsystem :

```
register_netdev(dev);
```

Reminder - Device Model and Device Drivers

In Linux, a driver is always interfacing with:

- a **framework** that allows the driver to expose the hardware features in a generic way.
- a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.



Netdevice allocation

- `alloc_netdev_mqs()` : Main allocation function :

```
struct net_device *
alloc_netdev_mqs(int sizeof_priv,               /* Size of driver-dedicated private data area */
                 const char *name,              /* Default name of the device */
                 unsigned char name_assign_type, /* Category of device name assignment */
                 void (*setup)(struct net_device *), /* Setup callback function */
                 unsigned int txqs,             /* Number of TX queues */
                 unsigned int rxqs);            /* Number of RX queues */
```

- The `setup` callback is called directly by `alloc_netdev_mqs()`
- `free_netdev()` is used to destroy the netdevice
- **device-managed** variants exist : `devm_alloc_etherdev_mqs()`

Netdevice naming

- Netdevice names can be changed dynamically, and the name source is tracked
- `dev->name_assign_type`, exposed in `/sys/class/net/xxx/name_assign_type`
- `NET_NAME_ENUM` : Name built sequentially by the kernel
  - e.g. `eth0, eth1`, etc.
- `NET_NAME_PREDICTABLE` : Name predictably assigned by the kernel
  - e.g. `label="lan1"` in devicetree for DSA switches
- `NET_NAME_USER` : Name assigned by the user during device creation
  - e.g. `ip link add link eth0.10 type vlan id 10`
- `NET_NAME_RENAMED` : Device was renamed by userspace
  - e.g. `ip link set dev eth0 name new-eth0`
  - Used by systemd's Predictable Network Interface Names

Ethernet-specific device allocation

- `alloc_etherdev_mqs()` : Allocate a new `struct net_device` for Ethernet :
- Sets the number of queues passed as parameters
- Creates a default name using the `"eth\%d"` template
- Sets all the Ethernet-specific default parameters :
  - MTU, Header len, Address len, etc.

Netdev ops

- Before registering, the driver populates a `struct net_device_ops`

mvneta.c - simplified

```
static const struct net_device_ops mvneta_netdev_ops = {
        .ndo_open            = mvneta_open,
        .ndo_stop            = mvneta_stop,
        .ndo_start_xmit      = mvneta_tx,
        .ndo_set_mac_address = mvneta_set_mac_addr,
        .ndo_change_mtu      = mvneta_change_mtu,
...
};

static int mvneta_probe(struct platform_device *pdev)
{
        ...
        dev->netdev_ops = &mvneta_netdev_ops;

        register_netdev(dev);
}
```

- These hooks are referred to as **NDO**s
- `.ndo_start_xmit` must be populated, all other are optional.

Common NDOs

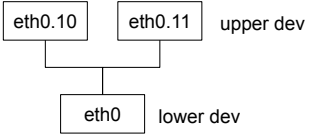
- `.ndo_open` and
- `.ndo_stop` : Bring the interface UP or DOWN
  - Call when using `ip link set eth0 up/down`
- `.ndo_start_xmit` : Send a packet
- `.ndo_set_rx_mode` : Configure the **rx filtering**
- `.ndo_set_mac_address` : Notify the driver that the MAC address was changed
- `.ndo_get_stats64` : Ask for hardware or driver statistics
- `.ndo_eth_ioctl` : Device-level `ioctl` handler, phased-out.

Netdev registration

- `register_netdevice()` : Registers the `struct net_device`, in the `netdev->net` namespace
  - Allocates the interface index
  - Makes the device visble from userspace
  - From this point on, NDOs may be called
  - Assumes `RTNL` is held.
  - `.ndo_init` is called at that stage, if provided
- `register_netdev()` : Calls `register_netdevice()` with RTNL held
  - Used mostly in drivers, as the device driver's `.probe()` doesn't hold RTNL

Stacking Network Devices

- Netdevices can be independent conduits, or stacked in a hierarchy
- e.g. a VLAN is represented as a dedicated `netdev`
  - A VLAN netdev's `lower_dev` is the physical device
  - The physical netdev's `upper_dev` is the VLAN device
- `struct net_device` has a list of `lower_dev` and `upper_dev`
- Packets may be passed between netdevs, which may modify them, e.g.
  - Encapsulation and Decapsulation (VLANs, tunnels)
  - Redirection and Routing (bridges)
  - Duplication and Redundancy (hsr, bond)
  - Encryption (macsec, wireguard)
- Can also be virtual interfaces, such as `veth`, `tun` and `tap`



Stacking Network Devices - 2

- Stacked devices show in userspace as `dev@lower`
  - *e.g.* DSA ports show as `lan1@eth0`
  - DSA uses stacking for the **conduit** interface
- The relationship is declared by calling :

```
int netdev_upper_dev_link(struct net_device *dev,
                          struct net_device *upper_dev,
                          struct netlink_ext_ack *extack)
```

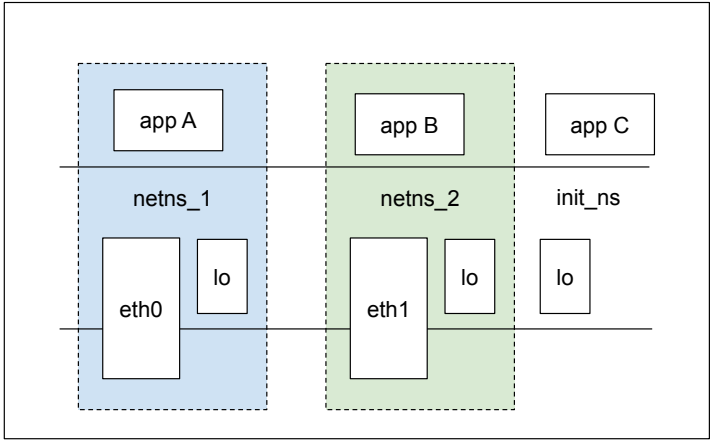
- A `netdev` can also have a `master` device
  - Similar to **upper**, except a `netdev` can only have one master
  - Used for **bridges**
  - `ip link set dev eth0 master br0`

Network Namespaces

- Netdevs can only view and pass traffic to other netdevs in the same **namespace**
- Net Namespaces, or **netns**, are represented internally by `struct net`
- A `netns` is created using `netlink`, e.g. through `iproute2` :
  - `ip netns add new_netns`
- Network namespace have their own set of resources :
  - Routing tables, ARP tables, caches, pools of memory, identifier pools...
- Netdevs are **moved** to a `netns` with `ip link set dev eth0 netns new_netns`
- All `netns` contain a **loopback** interface named `lo`, created when netdev is added to the netns
- Dedicated mechanisms such as **veth pairs** must be used for inter-netns communication
- Used by Containers for isolation

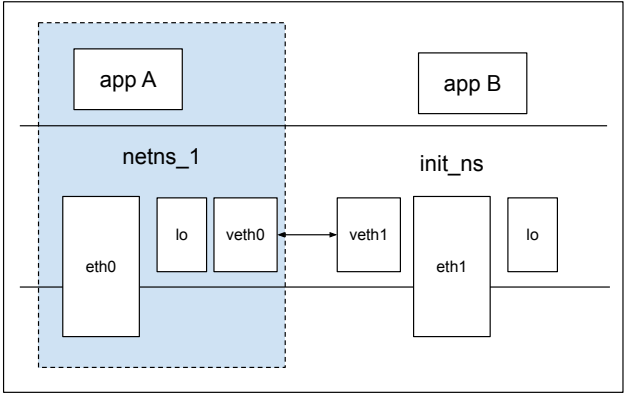
Network Namespaces - 2

- User processes run within a given **netns** and cannot see other interfaces
  - `ip netns <ns> exec <cmd>` : Run `cmd` in the `ns` namespace
- By default, netdevs are created in the **init_ns**

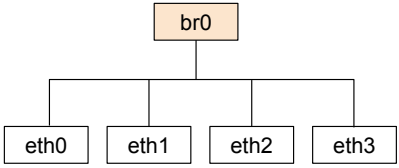


veth : Virtual Ethernet Pairs

- `ip link add type veth` : creates `veth0@veth1` and `veth1@veth0`
- Both `veth0` and `veth1` are linked together, traffic flows between the 2
- Main way to traverse namespaces, heavily used by containers



Bridges

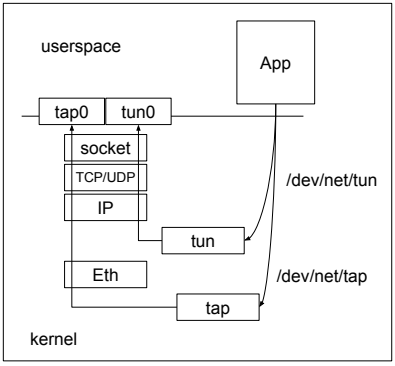


- A **bridge** represents a logical switch.
- If there is a hardware switch, its ports should act as **standalone** interfaces
  - A logical switch corresponding to the hardware can be re-created
  - switching operations can be **offloaded** in hardware with **switchdev**
- Bridges are represented with their own `struct net_device`
  - Created with `ip link add name br0 type bridge`
  - It acts as the **master** of all the switch ports
  - Ports are added with ip link set dev lan0 master br0
- The bridge interface maintains the `fdb` and handles forwarding

Vlan

- Multiple types of Vlans are supported in Linux through dedicated drivers
- 802.1Q : Layer 2 tag-based VLANs
  - `ip link add link eth0 name eth0-100 type vlan id 100`
- 802.1AD (Q in Q) : Allows using Vlans withing Vlans, with multiple tags
  - `ip link add link eth0 name eth0-100 type vlan id 100 protocol 802.1ad`
- VxLAN : VLAN using UDP encapsulation
  - `sudo ip link add link eth0 name vxlan100 type vxlan id 100 \`
    `local 192.168.42.1 remote 192.168.42.2`
- MACVlan : Virtual interface with a different MAC address than the physical one
  - `ip link add macvlan1 link eth0 type macvlan mode bridge`
  - Used a lot by containers

tun and tap interfaces



- Create virtual interfaces where a userspace program feeds and receives data from the `netdev`
  - Data is sent and received by accessing `/dev/net/tun`
- Used for userspace tunnel implementations, such as VPNs
- `ip tuntap add dev tun0 mode tun`
- `ip tuntap add dev tap0 mode tap`

Control interfaces for the Network Stack

Networking stack control path

- The Networking stack is very highly configurable, at all levels :
- Controller and driver behaviour, through `ethtool`, *e.g.* set the link speed
- Interface configuration, with `iproute2`, *e.g.* configure the IP address
- System-wide configuration, *e.g.* enable IP forwarding
- Per-connection configuration, *e.g.* select the TCP congestion-control algorithm
  - The `setsockopts()` syscall is covered later in this training.

ioctl interface

- The `ioctl` syscall is used to perform device-specific configuration
- `ioctl()` acts on a **file descriptor**.
  - For hardware configuration, we usually use `ioctl` on `/dev/xxx` descriptors
- We don't have any `fd` that corresponds to a specific `struct net_device`
- Network admin `ioctl` uses a `fd` corresponding to a **socket** with unspecified family : `AF_UNSPEC`
- Any socket **fd** can be used for network ioctls.

Example ioctl - Get interface name

```
struct ifreq ifr;
ifr.ifr_ifindex = ifindex;
ioctl (fd, SIOCGIFNAME, &ifr);
```

ioctl API

- Network-related `ioctl` have the `SIOC` prefix :
  - *e.g.* `SIOCGIFNAME` : Returns the name of an interface from its index
  - *e.g.* `SIOCADDMULTI` : Add to the multicast address list
  - *e.g.* `SIOCSHWTSTAMP` : Contigure hardware timestamping
- Most of the `ioctl` API is now frozen, and maintained for compatibility
- Replaced with `Netlink`, which offers more flexibility

sysctl interface

- The **sysctl** parameters are global, kernel-level parameters tunable at runtime
- **sysctl** is equivalent to writing into the corresponding files under `/proc/sys/`
- e.g. `systcl net.ipv4.ip_forward=1` is equivalent to `echo 1 > /proc/sys/net/ipv4/ip_forward`
- Values can be stored in `/etc/sysctl.d/*.conf`, and loaded with `sysctl -p`
- **sysctl** values are per-namespace, inheriting values from the `init_net`
- `net.core` : Core and net_device level configuration
  - `sysctl net.core.netdev_budget` : Displays the default NAPI budget
- `net.ipv4` IPv4 and Layer 4 configuration
  - `sysctl net.ipv4.ip_forward` : Allow IP forwarding (Router mode)
  - `sysctl net.ipv4.tcp_fin_timeout` : Set the TCP connection timeout (even for IPv6)
- `net.ipv6` IPv6 configuration

Netlink interface

- More flexible kernel to userspace communication mechanism, based on sockets

  ```
  fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_GENERIC);
  ```

- Allows easy extension of the userspace API without breaking compatibility
- User applications must open a **netlink socket** and send specially-formatted messages
- The socket can also be listened to for Kernel to userspace notificatins
- Netlink messages are grouped in **families**, grouping message types per class.
  - routing, ethtool, 802.11, team, macsec, etc.
- Netlink messages have a well-defined and stable format, but extensible.

Netlink Classic vs Netlink Generic

- Most netlink users today use **generic netlink**
- This replaces **classic netlink**, which has statically allocated familiy ID
- Generic Netlink (`genetlink`) allows dynamic family regstration
  - Allows easy implementation of custom families
  - Families are looked-up by **name** (a string) instead of ID.
- Example families :
  - "ethtool": Ethtool commands, also called **ethnl**
  - "wireguard": Wireguard tunneling configuration
  - "nl80211": Wifi-configuration netlink commands

Netlink Messages

- Transmission of **netlink messages** :
  - A fixed-format Header begins the message
  - The information is conveyed through TLV items : **Type**, **Length**, **Value**

```
struct nlmsghdr {
        __u32   nlmsg_len;      /* Length of message including headers */
        __u16   nlmsg_type;     /* Generic Netlink Family (subsystem) ID */
        __u16   nlmsg_flags;    /* Flags - request or dump */
        __u32   nlmsg_seq;      /* Sequence number */
        __u32   nlmsg_pid;      /* Port ID, set to 0 */
};
struct genlmsghdr {
        __u8    cmd;            /* Command, as defined by the Family */
        __u8    version;        /* Irrelevant, set to 1 */
        __u16   reserved;       /* Reserved, set to 0 */
};
/* TLV attributes follow... */
```

Netlink commands

- There are multiple types of Netlink requests based on the `nlmsg_flags`
- Commands may be used to Get or Set some kernel attributes
- Netlink `Get` commands can target one or several objects
  - A single object request is a `.doit()` request
    - `ip link show eth0`
  - An object listing request is a `.dumpit()` request
    - `ip link show`
- Netlink also exposes **multicast notifications**
- The message content is made of a set of pre-defined Attributes, based on the Command and Family
  - *e.g.* Command `ETHTOOL_MSG_LINKMODES_GET` for family "ethtool"
  - Contains `ETHTOOL_A_LINKMODES_SPEED`

Netlink specifications

- Message content used to be specified directly in the kernel uAPI headers
- Formats are now defined as **Netlink Specs** written in YAML
- Specifications are written per-family in Documentation/netlink/specs

Documentation/netlink/specs/ethtool.yaml

```yaml
name: ethtool
protocol: genetlink-legacy
doc: Partial family for Ethtool Netlink.

definitions:
  - ...
attribute-sets:
  - ...
operations:
  - ...
```

Netlink specifications - 2

- Netlink specs are used internally to **generate** the uAPI headers
  - Generated in `include/uapi/linux/ethtool_netlink_generated.h`
  - Included by `include/uapi/linux/ethtool_netlink.h`
- When modifying the specs, headers can be regenerated with `${KDIR}/tools/net/ynl/ynl-regen.sh`
- The **ynl** tool included in the kernel's sources can be used to sent hand-crafted messages
  - `make -C tools/net/ynl`
- Uses the Netlink Specs to derive the format and family :

```
ynl --family ethtool --no-schema --do linkinfo-get \
--json '{"header" : { "dev-name" : "eth0"}}'
```

Netlink monitor

- **Netlink Monitoring** car refer to 2 distinct operations :
- One can listen to **netlink notifications**
  - Emitted by the kernel upon configuration change
  - Applications can listen for specific notifications (Address change, link up, etc.)
  - e.g. `ip monitor`, `ethtool --monitor`, etc.
- It is also possible to listen to **All Netlink Traffic**
  - It includes All netlink messages, requests, replies and notifications
  - Done through a dedicated virtual interface : `nlmon`
  - e.g. `ip link add name nlmon0 type nlmon`
  - Tools such as `tcpdump` and `wireshark` can be used on the nlmon interface
- All these mechanisms still go through network namespaces

Configuration serialization in the kernel

- Actions triggered by `ioctl` or `netlink` messages often need serialization
  - Some actions impact multiple devices (e.g. netns removal)
  - Actions may be performed on multiple CPUs concurrently
- The main lock used to serialize the configuration is the **rtnl lock**
  - Global `struct mutex`, taken with `rtnl_lock()` and released with `rtnl_unlock()`
  - **R**ou**T**ing **N**et**L**ink
- `net_device.lock` : Mutex to protect *some* of the `struct net_device` fields
  - Very recent feature, introduced in `v6.14`
- The list of `struct net_device` is protected by RCU
- All `struct net_device` instance are reference-counted and reference-tracked

The RTNL lock

- Sometimes the Network Stack's Big Kernel Lock
  - Its scope is slowly getting removed, replaced with more specific locks
- Serializes most NDOs that aren't on the datapath
  - e.g. it does not protect `.ndo_start_xmit()`.
- Also serializes most `struct ethtool_ops`
- Protects some of the `struct net_device` fields
- For now, RTNL is **not** per-namespace, it is global. This is being reworked.
- Functions that rely on the caller holding rtnl ofen use `ASSERT_RTNL()`
- It is a `mutex` :
  - It is possible to sleep while holding rtnl
  - rtnl cannot be used when sleeping is forbidden (e.g. interrupt and softirq context)

Using Netlink in the kernel

- A new family can be registered by registering a `struct genl_family`
- This allows registering custom messages and associated handlers
  - e.g the macsec family
- Existing families already provide layers of abstractions :
  - The `struct rtnl_link_ops` is used for virtual netdev types
  - The ethnl abstraction is used for ethtool commands

Netlink Attributes

- As netlink is part of **userspace API**, all attribute values are heavily checked
- When declaring attributes, we can specify a **policy**
  - Allows specifying a range of acceptable values

```
const struct nla_policy ethnl_linkmodes_set_policy[] = {
        [ETHTOOL_A_LINKMODES_LANES] = NLA_POLICY_RANGE(NLA_U32, 1, 8),
};
```

- Handling messages is done in the genetlink `.doit()` callback
- Netlink attributes are represented as `struct nlattr`
- Attributes are passed as an array, indexed by attribute id, usually named `tb`
- Helpers are provided to get attribute values, *e.g.* `nla_get_u32()`

```
if (data[IFLA_MACSEC_WINDOW])
        secy->replay_window = nla_get_u32(data[IFLA_MACSEC_WINDOW]);^^I
```

struct netlink_ext_ack

- `struct netlink_ext_ack` allows reporting error messages to userspace
- It is included as part of the reply to netlink requests
- It can be found passed as a parameter to numerous internal kernel function

```
int dsa_port_mst_enable(struct dsa_port *dp, bool on, struct netlink_ext_ack *extack)
{
        if (on && !dsa_port_supports_mst(dp)) {
                NL_SET_ERR_MSG_MOD(extack, "Hardware does not support MST");
                return -EINVAL;
        }

        return 0;
}
```

Userpace libraries

- libmnl: Simple and lightweight library to access netlink
  - Used by `nftables`, `iproute2` and `ethtool`
- libnl: Higher level of abstraction but bigger binary
  - Useful for programs that "just work"

Userspace tooling

- **iproute2**
  - `ip` : Configure and query interfaces, routing and tunnels
  - `bridge` : Configures bridges (switching)
  - `tc` : Configures traffic control policing, shaping and filtering
  - `dcb` : Configures "Data Center Bridging" for traffic priorisation
  - These tools use the **Netlink** API
- **net-tools**
  - Obsolete ! replaced by **iproute2**
  - `ifconfig` : Configure interfaces (replaced by `ip`)
  - `brctl` : Configure bridges (replaced by `bridge`)
- **ethtool**
  - Used to configure Ethernet devices.
  - Can be compiled with `ioctl` or `netlink` support.

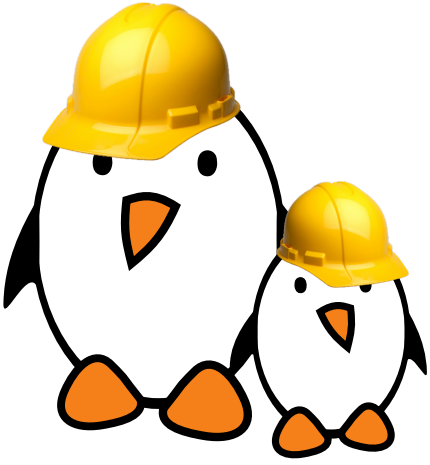
Userspace tooling - 2

- **NetworkManager**
  - Automatic configuration of interfaces based on config files
  - Handles IP assignment, low-level parameters
  - Very featureful, but bigger binary
  - Exposes a DBus API to interact with other software
  - See `man 8 NetworkManager`
- **Connman**
  - Alternative to NetworkManager, more embedded-oriented and lightweight
  - Also uses DBus to communicate with other software
- **Systemd-networkd**
  - Network configuration tool provided by SystemD
  - `.network` files are used to describe interfaces
  - See `man 8 systemd-networkd`

C library

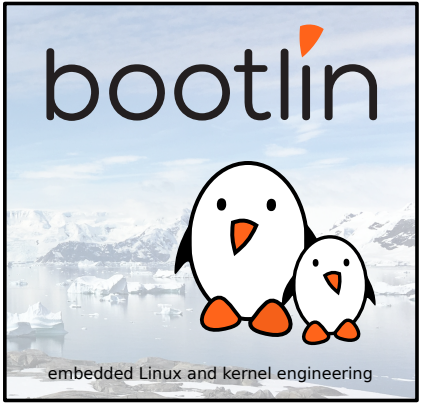
- The C library exposes a few helper functions to manipulate Network features
- `if_nametoindex`: Get the `ifindex` of a given interface
- `if_indextoname`: Get the name of an interface from its `index`
  - Can use either netlink or `ioctl`
- `inet_aton`, `inet_addr`: Convert IP address from string to binary
- `htons`, `ntohs`, `htonl`, `ntohl`: Endianness conversion
  - On most protocol headers, data is sent in **big endian** format
  - Also referred-to as Network Byte Order
  - **h**ost **to** **n**etwork **s**ort / **l**ong

Practical lab - Interacting with the Networking stack

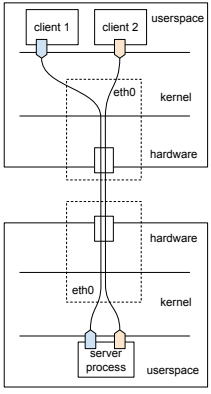


- Use `iproute2` and `ethtool`
- Experiment with namespaces
- Create stacked interfaces
- Use the netlink interface

Sockets and Data Path



Sockets



- The Socket programming model stems from **UNIX**
- It has been the main way for users to transmit data through the network since then
- Sockets are about more than networking, their behaviour depends on their attributes.
- A socket is represented from userspace as a **file descriptor** :

```
int socket(int domain, int type, int protocol);
```

  - see `man 2 socket`
- The `domain` or *family* defines the **underlying protocol** : IPv4, IPv6, Bluetooth, Netlink...
- The `type` defines the **semantics** : Connection-oriented, re-transmission, message ordering...
- The `protocol` depends on the domain and type, for further configuration.

Socket Families

```
int socket(int domain, int type, int protocol);
```

- Familes as defined by **UNIX**, **POSIX** or are **Linux**-specific
- In Linux, defined in `include/linux/socket.h`
  - `AF_UNIX`, `AF_LOCAL` : Unix Domain Sockets, for IPC. See `man 7 unix`
  - `AF_INET`, `AF_INET6` : IPv4 and IPv6 sockets, see `man 7 ip`
  - `AF_PACKET` (*raw* sockets) : Layer 2 sockets, see `man 7 packet`
  - `AF_NETLINK`, `AF_ROUTE` : Userspace to kernel sockets for configuration, see `man 7 netlink`
  - More specialised families : `AF_BLUETOOTH`, `AF_IEE802154`, `AF_NFC`, etc.
- Socket families are named `AF_xxx`, but equivalent names `PF_xxx` also exist
  - `PF` standing for **P**rotocol **F**amily, `AF` for **A**ddress **F**amily
  - legacy from the early UNIX days, `AF` and `PF` enums are equivalent on linux.

Socket Types

```
int socket(int domain, int type, int protocol);
```

- Socket types indicates the transmission semantics, which usually means Layer 4
- Its meaning depends on the selected **domain** :
  - `socket(AF_INET, SOCK_DGRAM, 0)` : UDP over IPv4 socket
  - `socket(AF_UNIX, SOCK_DGRAM, 0)` : Message-oriented Unix Socket
- `SOCK_STREAM` : Sequenced, reliable, two-way, connection-oriented
  - `socket(AF_INET, SOCK_STREAM, 0)` : TCP over IPv4 socket
- `SOCK_DGRAM` : Transmit datagrams of fixed maximum size, unreliable, connection-less
  - `socket(AF_INET, SOCK_DGRAM, 0)` : UDP over IPv4 socket
- `SOCK_RAW` : Raw sockets, usually containing the full **frame** including L2
- `SOCK_SEQPACKET`, `SOCK_RDM` : Other types with different ordering and message-length attributes
- `SOCK_NONBLOCK`, `SOCK_CLOEXEC` : Extra bitwise flags for configuration

Socket protocol

```
int socket(int domain, int type, int protocol);
```

- Complements the tuple <domain, type> to allow protocol selection.
  - socket(AF_INET, SOCK_STREAM, 0) : TCP over IPv4 socket
  - socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP) : SCTP over IPv4 socket
- For **raw sockets**, allows filtering by Ethertype (in network byte order)
  - socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL)) : All raw frames
  - socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP)) : All IPv4 frames
  - socket(AF_PACKET, SOCK_RAW, htons(ETH_P_8021Q)) : All Vlan frames

binding a socket

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- The `bind()` call allows associating a **local address** to a socket, see `man 2 bind`
- For connection-oriented, necessary before being able to **accept** connections
- The socket's address format is represented by the generic `struct sockaddr`.

```
struct sockaddr {
        sa_family_t sa_family;
        char        sa_data[14];
}
```

- The sockaddr must be subclassed by **family-specific** addresses :

```
struct sockaddr_in {
        sa_family_t    sin_family; /* address family: AF_INET */
        in_port_t      sin_port;   /* TCP/UDP port in network byte order */
        struct in_addr sin_addr;   /* IPv4 address (uint32_t) */
};
```

listen(), connect() and accept()

`int listen(int sockfd, int backlog)`

- Set the socket as listening for up to `backlog` connections, `man 2 listen`

`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`

- Accepts a remote connection request on a **listening** socket, `man 2 accept`
- The peer's address is filled into the `addr` parameter
- Returns a **new socket file descriptor** for that connection

`int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`

- Connect to a remote listening socket, `man 2 connect`
- For connection-less protcols, it simply sets the destination address for datagrams

socket options

- The `socket()` syscall doesn't allow fine-tuned configuration
- Sockets are configured through `setsockopt`
  - `int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen)`
  - `man 2 setsockopt`
- The options can be used to configure the socket itself :
  - `setsockopt(fd, SOL_SOCKET, ...);`
  - `SO_ATTACH_BPF` : attach BPF programs to sockets
  - `SO_BINDTODEVICE` : bind the socket to an interface
  - see `man 7 socket`
- We can also configure the underlying protocol's behaviour :
  - `setsockopt(fd, proto_num, ...);`
  - The protocol number can be retrieved from `/etc/protocols`
  - See `man 7 ip`, `man 7 tcp`, `man 7 udp`, etc.

Socket queues

- All sockets are created with 2 queues : A **Receive** queue and a **Transmit** queue
- Queues size is the same for every socket at creation time, but can be adjusted
  - With the `SO_RCVBUF` socket option, see `man 7 socket`
  - Using the `net.core.rmem_default` sysctl
- Packets that can't be queued because the queue is full are dropped
- `netstat` shows the current queue usage of every open socket

read() and write()

- Generic syscalls, acting on any kind of file descriptors
- Does not allow passing any extra flags

```
ssize_t read(int fd, void *buf, size_t count)
```

- Reads up to `count` bytes from the socket.
- May block until data arrives, unless the socket is non-blocking

```
ssize_t write(int fd, const void *buf, size_t count)
```

- Only works on **connected** sockets
- Recipient's address is part of the socket's connection information
- For **datagrams**, `count` can't exceed the datagram size
- Not possible to know if the recipient actually received the message

send() and recv()

- Socket-only, very similar to read() and write()
- Also only works with **connected** sockets
- Accepts `MSG_xxx` bitwise flags

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags)
```

- Similar to `read()`
- `MSG_PEEK` : Receives a message without consuming it from the socket queue
- `MSG_TRUNC` : Returns the real size, even if `count` is too small
- `MSG_DONTWAIT` : Per-message non-blocking operation

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags)
```

- Accepts a remote connection request on a **listening** socket
- `MSG_MORE` : More data is yet to be sent, as a single datagram or TCP message

sendto() and recvfrom()

- Socket-only, specifies the peer address per-message
- Allows using the same socket with multiple peers on UDP

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
 struct sockaddr *src_addr, socklen_t *addrlen)
```

- Get the address of the peer that sent the message along with the message
- `src_addr` and `addrlen` may be null, equivalent to `recv()`

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
 const struct sockaddr *dest_addr, socklen_t addrlen
```

- Send data to the the peer at the specified address
- On connection-oriented sockets (e.g. TCP), `dest_addr` is ignored
- On Datagram sockets, the address overrides the `connect()` address.

sendmsg() and recvmsg()

- Allows passing ancilliary data alongside the buffers
- Ancilliary data is following the `cmsg` format
- Also allows scatter-gather buffers

```
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags)
```

- Grabs the peer address, like in `recvfrom()`
- Allows reading from the socket *error queue*
  - The error queue contains the original packet content and associated errors
  - Also used for **timestamping**

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags)
```

- Sends single or scatter-gather buffers to a designed peer
- Also accepts ancilliary data

Summary - Server

1. Create the socket

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

2. Bind to the local IP address and port

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(80);
inet_aton("87.98.181.233", &addr.in_saddr);
bind(sockfd, &addr, sizeof(addr));
```

3. Listen for new inbound connections

```
listen(sockfd, 10);
```

4. Wait and accept a new connection

```
conn_fd = accept(sockfd, &peer_addr, &peer_addr_len);
```

5. Receive data from the client

```
recv(conn_fd, buf, 128);
```

Summary - Client

1. Create the socket

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

2. Connect to the server

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(80);
inet_aton("87.98.181.233", &addr.sin_addr);
connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));
```

3. Send data to the server

```
recv(conn_fd, buf, 128);
```

Waiting for data

- The standard file descriptor polling methods also work on sockets
- `select()` and `poll()` can be used to wait for incoming packets
- The `epoll` API is becoming the preferred method nowadays
- `epoll_create()` allows creating **epoll instances** to listen on **interest lists**
- `epoll_ctl()` is used to add, modify or remove descriptors to an instance
- `epoll_wait()` is then used to wait and process **events**
- This mechanism interacts directly with **NAPI** instances for events
- New features such as *IRQ suspension* rely on epoll

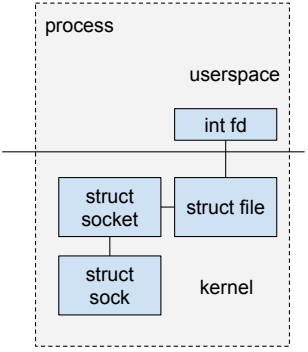
Timestamping

- Timestamping traffic is useful for **debugging** and **time synchronisation** (PTP)
- The `SO_TIMESTAMP` **sockopt** causes timestamp creation for **ingress datagrams**
- The newer `SO_TIMESTAMPING` allows configuring the timestamp source :
  - Hardware timestamp generation (configurable through `ethtool`)
  - Software timestamp generation, in the driver
  - TX sched timestamping, can help measure the queueing delay
  - TX ACK for TCP, when the acknowledgement was received
  - TX completion timestamping, when the packet finished being sent
- Timestamping can also be configured **per-packet** with `sendmsg()` and `recvmsg()`
- Timestamps are received through `recvmsg` ancilliary data in RX
- **TX timestamps** are accessible through the socket's **error queue**
  - Packets are looped-back through the error queue with an associated timestamp

io_uring

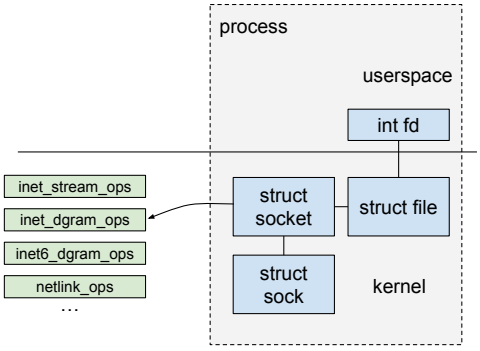
- `io_uring` is an alternative to **socket** programming
- It is an asynchronous API, originally developped for the Storage subsystem
- Aims at reducing the amount of `syscalls` such as `read` and `write`
- Recently, `io_uring` gained network support
  - Applications have to crate special ring-buffers shared with the kernel
  - Transfers are queued in a TX ring-buffer by userspace
  - A completion ring-buffer is used to know when data has been sent
  - A similar mecanism exists for RX
- Still new and gaining features, see this introduction post

Sockets in the kernel



- Sockets have a **file descriptor**
- It is handled internally with a **pseudo file**
- `struct socket` is the generic representation
  - stores the SOCK_xxx types
  - holds the `struct proto_ops` pointer
  - interfaces with the syscall API
- `struct sock` is what the network stack manipulates
  - more internal representation
  - for use mostly by the network stack
  - maintains the queues, locks, and internal state

struct proto_ops



- `struct proto_ops` implement the protocol-specific operations
- Selected at socket creation based on the **family**
- Very close to the syscall interface

```
struct proto_ops{
...
int (*bind) (struct socket *sock,
             struct sockaddr *myaddr,
             int sockaddr_len);
int (*sendmsg) (struct socket *sock,
                struct msghdr *m,
                size_t total_len);
...
};
```

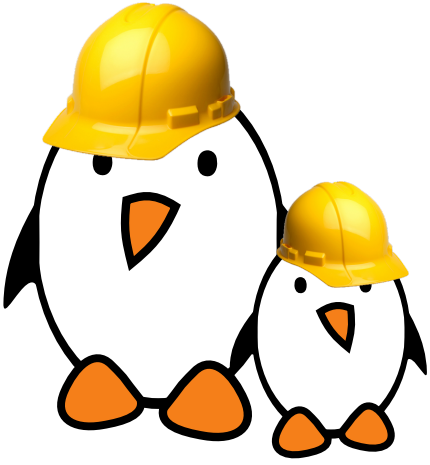
sending through a socket

1. Userspace program calls `write()`, `send()`, `sendto()` or `sendmsg()`
2. The corresponding `syscall` is invoked
3. All above syscalls end-up calling `__sock_sendmsg()`
4. `sock->ops->sendmsg()` is called (`inet6_sendmsg()`, `inet_sendmsg()`, etc.)
5. `sock->sk_prot->sendmsg()` is called (`tcp_sendmsg()`, `udpv6_sendmsg()`, etc.)
6. `skb` chain gets created through `ip_make_skb()` or `ip6_make_skb()`
7. `skb` is then sent with e.g.`udp_send_skb()`, which calls `ip_send_skb()`
8. The target `struct net_device` is retrieved with `ip_route_output_key_hash()`
   - This is cached in `struct sock`
9. `dst_output()` is eventually called, handing over from L4 to L3

L3 processing

- In `ip_finish_output()` or `ip6_finish_output()`
- The Layer 2 **MTU** is looked-up (`skb->dev->mtu`)
- The `skb` is **fragmented** if needed
- Once the **routing** information is found, the **neighbour** is looked up
- This is usually done by looking the gateway in the **ARP** table
  - ARP tables can be dumped with `ip neigh` (or `arp`)

Practical lab - Experimenting with Sockets



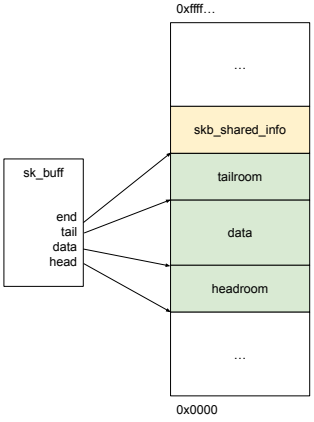
- Create a simple TCP client
- Use TCPDump
- Create a simple traffic monitoring tool

Socket Buffers

struct sk_buff (1)

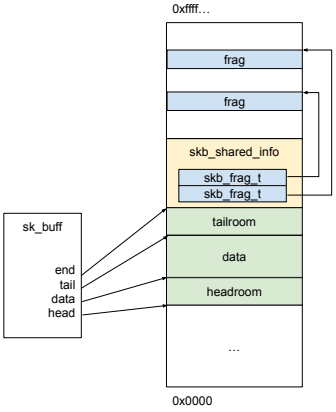
- Object that represents a packet through the stack : **soc**k**e**t **buff**er
  - `struct sk_buff` defined in `include/linux/skbuff.h`
- Created when user writes data into a socket
- Created by drivers upon receiving a packet
- Core object of the Networking Stack, often named `skb`
- It contains **meta-data** about the packet :
  - Origin/destination `struct sock` (`skb->sk`)
  - Origin/destination `struct net_device` (`skb->dev`)
  - Arrival timestamp, priority, etc.
- Also contains a lot of specific flags :
  - `wifi_acked` : Was the packet ack'd, wifi-specific
  - `decrypted` : Does the packet needs decryption ?
  - `redirected` : Was the skb redirected ?

skb payload



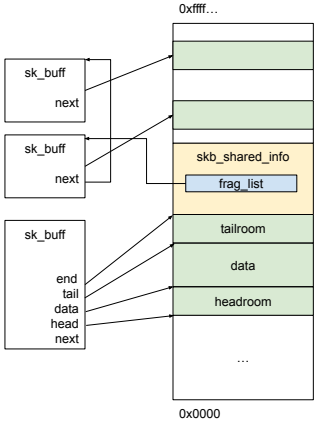
- `skb` maintains positions to the data buffer
- The **data** section is the current **payload**
- The payload boundaries (`data` and `tail`) depend on the current Layer
- `skb->len` identifies the current length of data
- `skb->head` : Start of the allocated buffer
- `skb->data` : Start of the **payload section** of the **current layer**
- `skb->tail` : End of the payload section
- `skb->end` : End of the buffer
- These pointers are **moved** when the skb traverses the stack

skb geometry : Paged skb



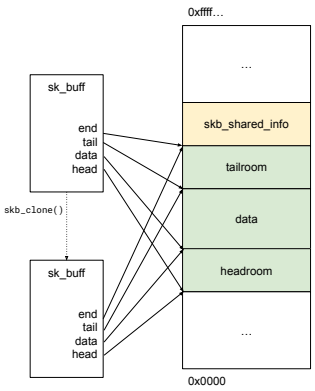
- The data section of an `skb` may be non-contiguous
- We talk about **non-linear** or **paged** skb.
- Sections of the payload are stored in the `skb_shared_info`
- Each part of the buffer is stored in an array of `skb_frag_t`
- This happens when transmitting **scatter-gather** (SG) buffers
- `skb_linearize()` will convert it to a single-buffer skb.
  - Useful for drivers that don't support SG.

skb geometry : Fragmented skb



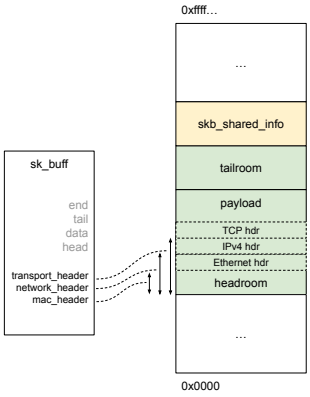
- Buffers bigger than the **MTU** needs to be fragmented
- The original `skb` gets split into multiple parts
- Each fragment is its **own skb**
- Fragments are chained together through :
  - `skb_shared_info->frag_list` for the **first skb**
  - `skb->next` for the other fragments

skb cloning and duplication



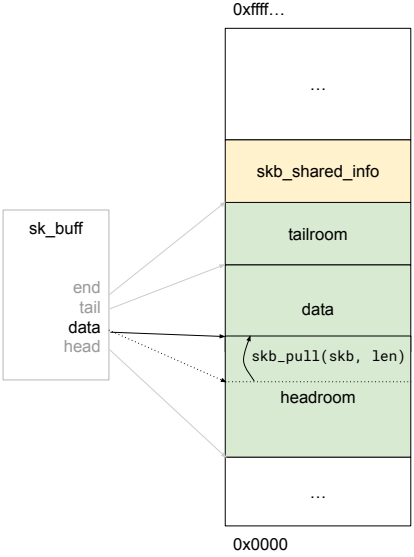
- `skb_clone()` allocates a new `struct sk_buff` pointing to an existing buffer
- Useful when the `skb` needs to be delivered multiple times
  - For Multicast, `AF_PACKET`, capturing, etc.
- The fragments are also cloned
- The buffer memory is **refcounted**
- Destroy the clone with `consume_skb()` or `kfree_skb()`
- `skb_copy()` duplicates the `skb` and all its associated memory
- `pskb_copy()` duplicates the `skb` and the **header** but clones the payload
- `skb` can also be shared, tracked with `skb->shared`

skb layer offsets



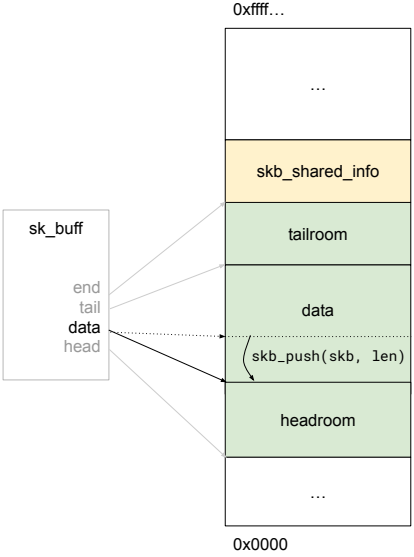
- `struct sk_buff` maintains **layer offsets** starting from `skb->head`
- Set and modified by each encapsulation or decapsulation step
- When processing a packet, each layer moves `skb->data`
- `skb_reset_xxx_header()` sets the given header **where** `skb->data` **currently is**
  - `skb_reset_mac_header()` : Called by **drivers**
  - `skb_reset_network_header()` : Called after MAC processing
  - `skb_reset_transport_header()` : Called in L3 (IP) processing

skb_pull()



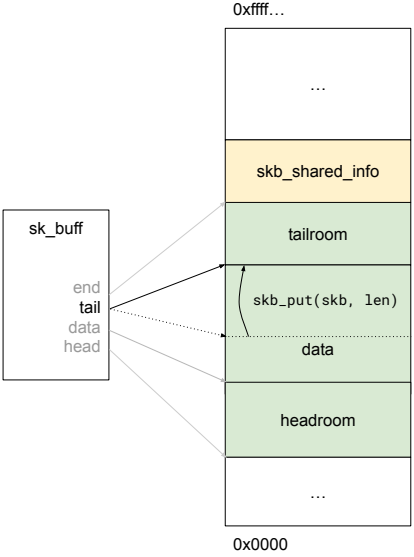
- Pulls header data, used during **decapsulation**
- Decreases `skb->len`
- Usually followed by a layer offset readjustment
- Returns the new `skb->data` pointer
- May fail if `skb->len` is too short
- May require a **checksum recompute**
  - `skb_pull_rcsum()` will update checksums

skb_push()



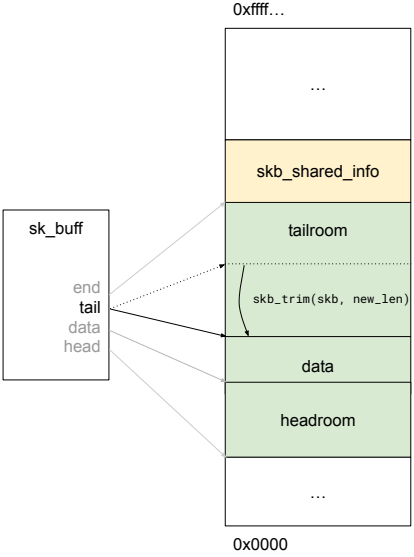
- Pushes the `skb->data` into the headroom
- Increases `skb->len`
- Used during **encapsulation**, when creating the headers
- May fail if the headroom is too short
- May require a **checksum recompute**
  - `skb_push_rcsum()` will update checksums

skb_put()



- Expands the payload section into the tailroom
- Increases `skb->len`
- Used in drivers to set the **full packet size**
- Also used by some **DSA taggers**

skb_trim()



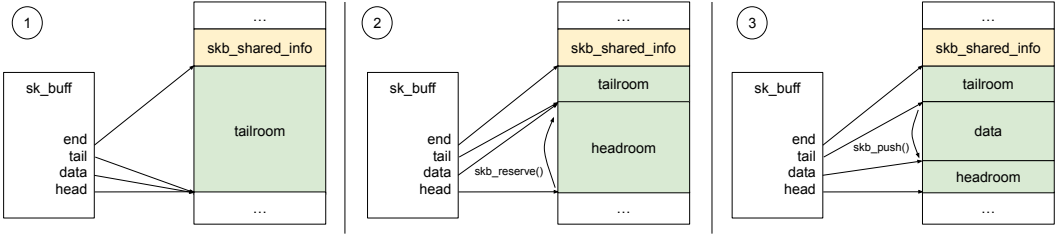
- Shrinks down the payload from its end
- Decreases `skb->len`
- Only works on **linear skb**
- Used to remove padding
- Also useful to decapsulate protocols that insert a trailer
  - e.g. PRP

pskb helpers

- **p**otentially fragmented **skb** helpers manipulate non-linear skb
- Useful if you don't know and don't mind if the skb is paged
- Not all helper have a matching pskb equivalent, not always relevant
- `skb_put()` => `pskb_put()`
- `skb_pull()` => `pskb_pull()`
- `skb_trim()` => `pskb_trim()`
- `pskb_may_pull()` indicates if a `pskb_pull` operation will succedd
  - `pskb_may_pull_reason()` returns a **drop reason** if it will fail

skb allocation

- `build_skb()` allocates a new `skb` around an existing buffer
- A new linear `skb` is allocated with `alloc_skb()`. It also allocates its data buffer.
- A paged `skb` can be allocated with `alloc_skb_with_frags()`
- A newly-allocated `skb` is empty : `skb->data == skb->head == skb->tail`
- `skb_reserve()` grows the headroom, then `skb_push()` to prepares the data section



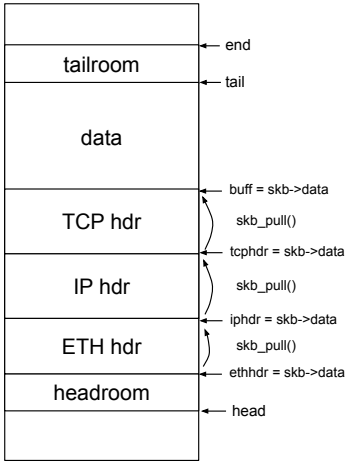
Dropping packets

- At any point, we may decide to **discard** an `skb`, it is **dropped**
- an `skb` is dropped with :

```
void kfree_skb_reason(struct sk_buff *skb, enum skb_drop_reason reason);
```

- The `reason` allows reporting to users the cause of the drop
- Around 120 different reasons currently exist
  - see `include/net/dropreason-core.h`
- Drop reasons are not part of the **userspace API**, but can be retrieved with :
  - **ftrace**, via the `skb:kfree_skb` and `skb:consume_skb`tracepoints : `trace-cmd record -e skb:kfree_skb <cmd>`
  - **dropwatch** : Uses the kernel's `dropmon` mechanism through `netlink`
  - **retis** : `eBPF`-based, uses the `BTF` information to display reasons

SKB decapsulation



- When **ingress** packets traverse the stack, they are decapsulated
- Each header usually have a field indicating the nature of the upper layer
  - Ethernet header : `Ethertype` (2 bytes)
  - IPv4 header : `Protocol` (1 byte)
  - IPv6 header : `Next Header` (1 byte)
- The `ptype` list maps `Ethertypes` to `packet handlers`
- The `proto` list maps `Protocols` to `Transport handlers`
- Each stage **parses** its header, and **moves** `skb->data`
- In the last stage, `skb->data` points to the final payload

struct packet_type

- L2 protocols such as 802.3 and 802.11 usually include an **Ethertype**
- 2-byte value indicating the higher-level protocol :
  - 0x0800 for IPv4, 0x0806 for ARP
  - 0x86dd for IPv6, 0x8100 for 802.1Q (vlan)
  - See include/uapi/linux/if_ether.h
- We can associate struct packet_type with Ethertypes :

```
struct packet_type {
    __be16 type;
    struct net_device *dev;
    int (*func) (struct sk_buff *skb,
                 struct net_device *dev,
                 struct packet_type *ptype,
                 struct net_device *orig_dev);
    /* ... truncated */
};
```

dev_add_pack()

- `dev_add_pack()` registers a `struct packet_type` (`ptype`)
- if `ptype->dev` is `NULL`, the handler is registered system-wide
  - e.g. IPv4, IPv6, ARP
- otherwise, the `ptype` will only be handled on `ptype->dev`.
  - e.g. `AF_PACKET` sockets bound to an interface
- Upon match of the Ethertype, `ptype->func()` is called with the `skb`
- `ptype->list_func` can be implemented to handle multiple `skb`

IPv4 example

```
static struct packet_type ip_packet_type __read_mostly = {
        .type = cpu_to_be16(ETH_P_IP),
        .func = ip_rcv,
        .list_func = ip_list_rcv,
};

static int __init inet_init(void) /* truncated */
{
        /* For TX : Used by the socket's sendmsg */
        proto_register(&tcp_prot, 1);
        proto_register(&udp_prot, 1);
        proto_register(&ping_prot, 1);

        /* For RX : Handle the IP Ethertype */
        dev_add_pack(&ip_packet_type);
}
```

Exception : Vlan

- VLANs (802.1Q and 802.1AD) have a dedicated Ethertype, but no `struct packet_type`
- VLANS are handled directly in the receive path
- Some hardware can strip the VLAN tag themselves
  - The tag is reported out-of-band, such as in the DMA descriptors
  - The VLAN information is set in `skb->vlan_proto` and `skb->vlan_tci`
- This also allows optimizing speed by avoiding indirect branches
- Some hardware may also perform Vlan filtering

RX handlers

- Protocol information alone may not always be sufficient for custom processing
- e.g. **MACVlan** has no dedicated Ethertype
- We can attach a callback function to a netdev, executed before protocol handling

```
rx_handler_result_t rx_handler_func_t(struct sk_buff **pskb);
```

- Attached with `netdev_rx_handler_register()` (One handler per netdev)
- Handler may change the `skb`, including `skb->dev`, and return :
  - `RX_HANDLER_CONSUMED` : `skb`'s processing stops here
  - `RX_HANDLER_PASS` : continue as if the handler didn't exist
  - `RX_HANDLER_EXACT` : `PASS` to protocol only if `ptype->dev == skb->dev`
  - `RX_HANDLER_ANOTHER` : Re-process the `skb` as if it came from `skb->dev`

struct net_protocol

- Layer 3 protocols include an 8-bit identifier describing the L4 layer
  - 6 for TCP, 17 for UDP
  - 1 for ICMP, 41 for IPv6-in-IPv4
  - see include/uapi/linux/in.h
- A **transport protocol handler** is represented by struct net_protocol
  - For IPv6, it is represented by struct inet6_protocol

```
struct net_protocol {
        int (*handler)(struct sk_buff *skb);
        int (*err_handler)(struct sk_buff *skb, u32 info);
        ...
};
```

inet_add_protocol() and inet6_add_protocol()

- Transport protocols are registered in each L3 stack
- `int inet_add_protocol(struct net_protocol *prot, u8 num);`
- `int inet6_add_protocol(struct inet6_protocol *prot, u8 num);`
- Associate protocols with their respective identifiers
- Upon matching the `num` identifier, `prot->handler()` is called

struct net_offload

- Some Layer 4 protocols may be associated with a `struct net_offload`
- Used to offload **segmentation** : Let the hardware or driver do it
  - Segmentation and re-assembly is protocol-specific
  - Each protocol can register a `struct net_offload`
  - ```
    int inet_add_offload(const struct net_offload *prot, unsigned char num);
    int inet6_add_offload(const struct net_offload *prot, unsigned char num);
    ```
- skbs bigger than the **MTU** are passed to the driver
- The driver or the hardware handles splitting the packet
  - L2 and L3 headers are added, and a shorter L4 header
- On the receive side, the hardware or driver re-assembles the packets
  - Intermediate headers are stripped and the packet is re-assembled
    ```

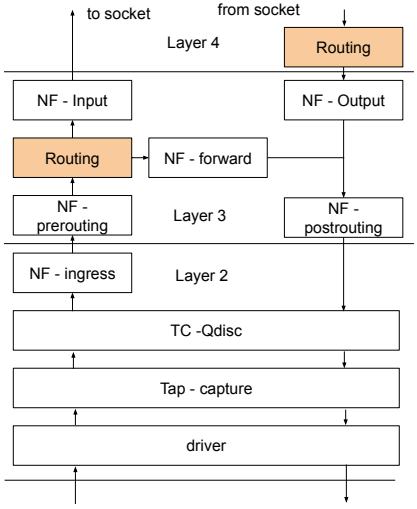
Generic Receive Offload

- GRO may be used if the driver or the hardware doesn't handle re-assembly
  - May be toggled with `ethtool -K <iface> gro off|on`
- It is **generic**, works with any Layer 4 protocol
- This still requires driver support :
  - Upon receiving packets, fragmented or not, call `napi_gro_receive()`
  - Drivers that do not support it call `netif_receive_skb()`
- GRO accumulates skbs and asks L3 and L4 to act upon it
  - `inet_gro_receive()` : e.g. Checks if Don't Fragment flag is set
  - `tcp_gro_receive()` : e.g. Flush if we exceed the TCP MSS
- GRO-held skbs are merged and eventually flushed to the regular receive path
- Can be problematic for latency or throughput in **router** mode

Generic Segmentation Offload

- Perform the segmentation either in hardware, or just before passing to the driver
- Avoids having all the segments traverse the stack
  - Routing, filtering, scheduling decisions are the same for all segments
- Pure software implementation, but can be offloaded to hardware :
  - TCP Segmentation Offload (TSO)
  - Hardware will split the TCP data based on the MSS
  - Support for partial checksum offload is required

routing

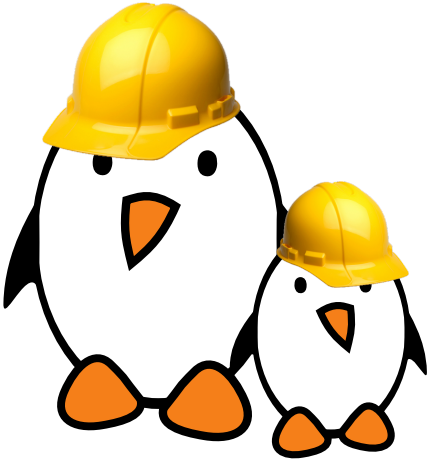


- Routing happens on **ingress** and **egress**
- Done by looking-up the **F**orwarding **I**nformation **B**ase
- Decision taken in `fib_lookup()`
- The table can be shown with **ip route**

Flow tables

- Allows a slow-path and fast-path for routing and bridging
- The first packet of a given flow goes through the whole stack
- The final routing or bridging decision is cached
- The next packet from the same flow will go through the fastpath
- These decisions may be offloaded to hardware

Practical lab - Manipulating socket buffers



- Implement a simple custom Layer 2 protocol
- Manipulate SKBs for encapsulation and decapsulation

Traffic Filtering

current solutions

- The Networking stack can filter **egress** and **ingress** traffic
  - Necessary for **firewalling**
- Filters can also identify **packets of interest** for on-the-fly modification
  - e.g. NAT : The destination **IPv4** address is re-written
- Historically, multiple solutions have been implemented in the Linux Kernel :
  - `iptables` and `ip6tables` for Layer 3
  - `arptables` and `ebtables` for Layers 2 and 3
  - These solutions have been replaced by **netfilter** and its **nftables**
- Alternative solutions exist :**eBPF**, **P4** and **TC**

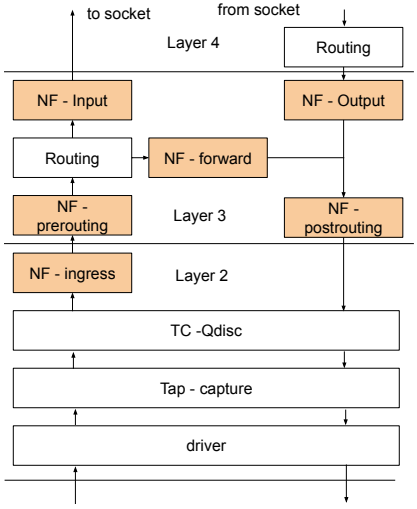
Legacy filtering solutions

- There used to be multiple traffic filtering, each for a different layer
- `iptables` and `ip6tables` : IP-level filtering and mangling
  - `net/ipv4/netfilter/ip_tables.c` and `net/ipv6/netfilter/ip6_tables.c`
- `ebtables` : Filtering based on Layer 2 information
  - Filter and forward VLANs and bridging operations
  - ARP filtering

nftables

- Originates from the Netfilter project
- More modern approach, with a centralised filtering table and multiple **hooks**
- Rules expressed in a low-level language
- Users attach **chains** to **hooks** to express rules
- Chains are stored within **tables** created by users
- See the project provided examples

Netfilter hooks



- **ingress** : Filter as soon as the packet is received
- **pre-routing** : Filter before taking a routing decision
- **input** : Filter packets going to sockets
- **forward** : Filter packets forwarded to the outside
- **output** : Filter local outgoing packets
- **post-routing** : Filter all outgoing packets

netfilter in userspace

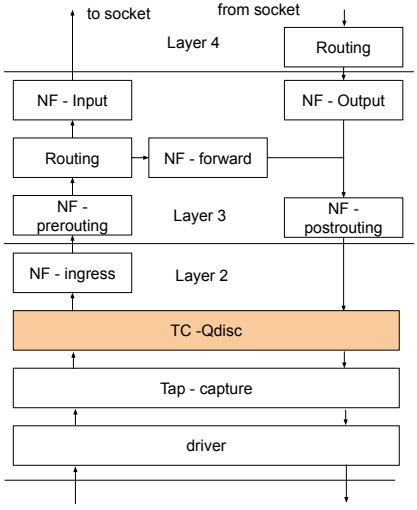
- Netfilter was integrated as another backed for existing tools like **iptables**
- Dedicated tool is called **nft**, see `man 8 nft`

Traffic Control

Packet Scheduling

- On complex systems, thousands of applications may use the same interface
- The scheduling of egress traffic needs to be configurable and predicatble
- Queueing strategies can be tunes for **throughput** and **latency**
- **tc** is the main component that deals with traffic scheduling

Packet Scheduling in the stack



- The scheduling decision occurs between **routing** and the **driver**
- On **egress**, decides which packet to enqueue
- On **ingress**, may decide to drop or redirect

TC : Traffic Control

- `tc` is a subsystem in charge of traffic control operations, namely :
  - **Traffic Shaping** : Control the transmission rate for traffic classes
  - **Traffic Scheduling** : Control the ordering and burst behaviour of outgoing traffic
  - **Traffic Policing** : Control the reception rate for traffic classes
  - **Drop control** : Control discard conditions for egress and ingress traffic
  - **Classification** : Identify packets of interest for further actions

TC use-cases

- `tc mqprio` - Assign priorities to the Network Controller's queues
- `tc taprio` - Time-aware queue priorisation, for TSN
- `tc flower` - Flow-based actions, can be offloaded to hardware
- `tc ingress` - Attach TC actions to ingress traffic

TC QDisc : Queueing Disciplines

- Controls how traffic is enqueued, in the **tx** direction
- Allows shaping the traffic very precisely on a per flow basis
- **Flows** can be assigned different **Qdisc** to define how to schedule transmission

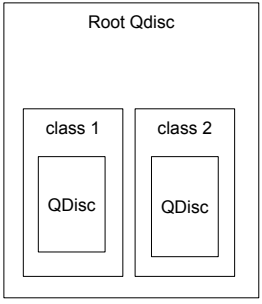
Queues

- Queue management is crucial for Lantency and Throughput
- Long queues allows absorbing network instabilities...
- ... but may cause to latencies, leading to **bufferbloat**
- The Network Interface's queues are exposed to TC
- **qdisc** algorithms select which queue is used for a given **flow**

Traffic flows

- Packets with same **addressing parameters** are part of the same **flow**
- A Layer 4 flow is defined by 4 parameters : It's a **4-tuple**
  - Source and Destination **ports**
  - Source and Destination **IP address**
- A Layer 3 flow is defined by 2 parameters : It's a **2-tuple**
  - Source and Destination **IP address**
- The **vlan id**, if applicable, may be included in the flow definition
  - FLows may thefore be **3-tuple** or **5-tuple**
- When acting on a packet, we need to identify the **flow** it belongs to :
- This is called **n-tuple classification**. The `n-tuple` is extracted, and its **hash** is computed
- Extracting the `n-tuple` value from a packet is called **bissection**
- The hash is used for the subsequent lookup operations, and may be computed in hardware.

TC example : QDisc



- Queueing Disciplines, or **qdisc**, allow configuring the queue policy
- Multiple qdisc can co-exist, separated in diffent **classes**
- **classes** are used to split traffic, and enforce policing
- Traffic is assigned to classes through **classification**

TC example : Classification

- Match traffic with priority 0 or 4, and assign it to class "1:20"

```
tc filter add dev eth0 parent 1: basic match 'meta(priority eq 0)' \
or 'meta(priority eq 4)' classid 1:20
```

- In **ingress**, classification is usually done to assign traffic to queues
- It can also be used for early filtering :

```
tc qdisc add dev eth0 ingress
tc filter add dev eth0 protocol ip parent ffff: flower \
ip_proto tcp dst_port 80 \
action drop
```

- `tc-flower` can also be offloaded to hardware, see `man 8 tc-flower`

TC example : shaping

- Traffic Shaping, consists in limiting the egress rate of a flow
- Multiple strategies exist :
  - Add Jitter on purpose : `tc qdisc add dev eth0 root netem delay 10ms 5ms`
  - Use a Token Bucket filter :
    `tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 256kbit`
- This can be combined with classification :

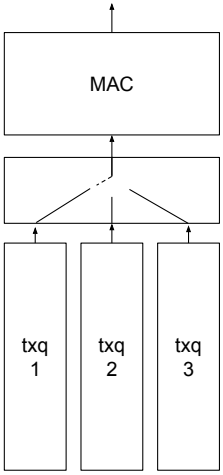
```
tc qdisc add dev eth0 root handle 1: prio
tc qdisc add dev eth0 parent 1:3 handle 30: tbf rate 250kbit
tc filter add dev eth0 protocol ip parent 1:0 prio 3 u32 match ip \
dst 192.168.42.2/32 flowid 1:3
```

TC example : editing

- TC also allows editing traffic or metadata on-the-fly
- This is done with the `skbedit` action
- This can be used to change the `skb->priority` field
- Can also control which Hardware **tx queue** will be used

```
tc filter add dev eth0 parent 1: protocol ip prio 1 u32 \
match ip dst 192.168.0.3 \
action skbedit priority 6
```

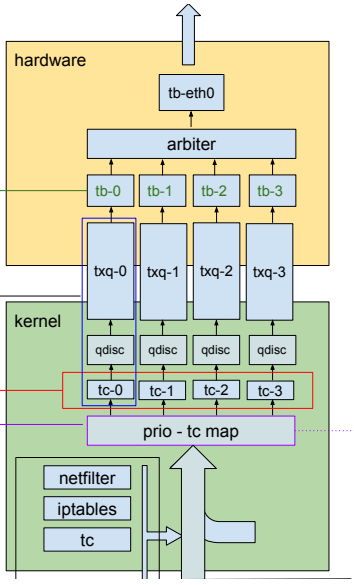
TC mqprio



- Most Network Controllers today have mutilple queues in `tx` and `rx`
- They implement in hardware a **policing** algorithmm to select the next `tx` queue to use
  - It can be a simple **weighted round robin** algorithm
  - Alternatively a **strict priority** selection
  - Some controllers also implement Time-aware scheduling for queue selection

TC mqprio

```
tc qdisc add dev eth0 \
parent root handle 100 \
mqprio num_tc 4 \
map 3 1 0 2 \
queues 1@0 1@1 1@2 1@3 \
hw 1 mode channel shaper \
bw_rlimit \
min_rate 0 0 0 0 \
max_rate 10Mbit 20Mbit 50Mbit 100Mbit
```

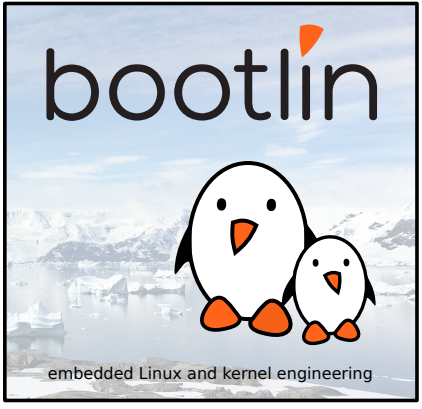


prio	tc
0	3
1	1
2	0
3	2

TC offloads

- Some TC operations can be offloaded to the Ethernet Controler, if supported
- `tc mqprio` - The Hardware will implement the queue-selectin algorithm
- `tc taprio` - For TSN-enabled hardware
- `tc flower` - Classication in **ingress** is done by hardware

eBPF

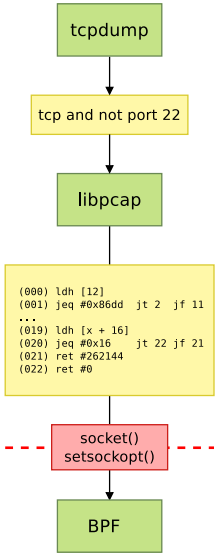


The ancestor: Berkeley Packet filter

- BPF stands for Berkeley Packet Filter and was initially used for network packet filtering
- BPF is implemented and used in Linux to perform Linux Socket Filtering (see `networking/filter`)
- tcpdump and Wireshark heavily rely on BPF (through libpcap) for packet capture

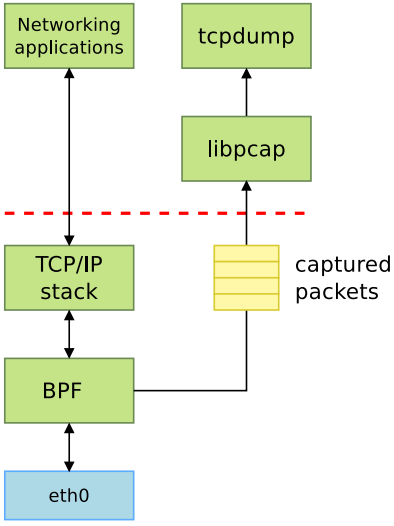
BPF in libpcap: setup

- tcpdump passes the capture filter string from the user to libpcap
- libpcap translates the capture filter into a binary program
  - This program uses the instruction set of an abstract machine (the “BPF instruction set”)
- libpcap sends the binary program to the kernel via the `setsockopt()` syscall



BPF in libpcap: capture

- The kernel implements the BPF “virtual machine”
- The BPF virtual machine executes the program for every packet
- The program inspects the packet data and returns a non-zero value if the packet must be captured
- If the return value is non-zero, the packet is captured in addition to regular packet processing



eBPF (1/2)

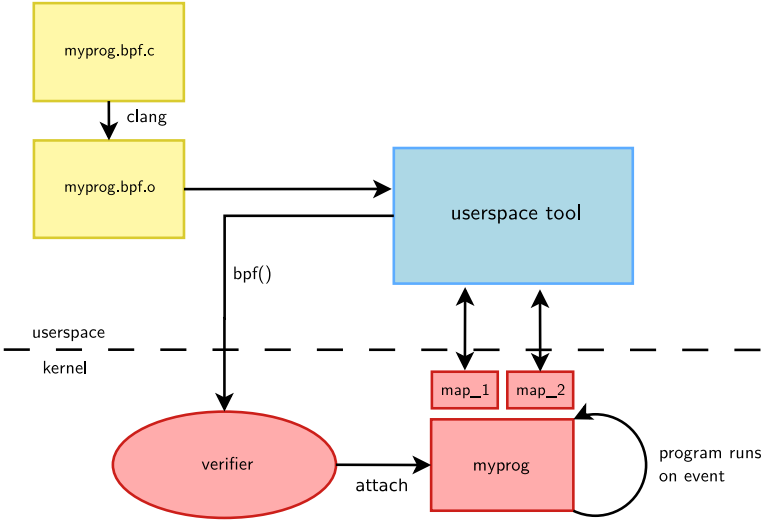
- eBPF is a new framework allowing to run small user programs directly in the kernel, in a safe and efficient way. It has been added in kernel 3.18 but it is still evolving and receiving updates frequently.
- eBPF programs can capture and expose kernel data to userspace, and also alter kernel behavior based on some user-defined rules.
- eBPF is event-driven: an eBPF program is triggered and executed on a specific kernel event
- A major benefit from eBPF is the possibility to reprogram the kernel behavior, without performing kernel development:
  - no risk of crashing the kernel because of bugs
  - faster development cycles to get a new feature ready

Image credits: https://ebpf.io/

eBPF (2/2)

- The most notable eBPF features are:
  - A new instruction set, interpreter and verifier
  - A wide variety of "attach" locations, allowing to hook programs almost anywhere in the kernel
  - dedicated data structures called "maps", to exchange data between multiple eBPF programs or between programs and userspace
  - A dedicated `bpf()` syscall to manipulate eBPF programs and data
  - plenty of (kernel) helper functions accessible from eBPF programs.

eBPF program lifecycle



Kernel configuration for eBPF

- `CONFIG_NET` to enable eBPF subsystem
- `CONFIG_BPF_SYSCALL` to enable the `bpf()` syscall
- `CONFIG_BPF_JIT` to enable JIT on programs and so increase performance
- `CONFIG_BPF_JIT_ALWAYS_ON` to force JIT
- `CONFIG_BPF_UNPRIV_DEFAULT_OFF=n` in **development** to allow eBPF usage without root
- You may then want to enable more general features to "unlock" specific hooking locations:
  - `CONFIG_KPROBES` to allow hooking programs on kprobes
  - `CONFIG_TRACING` to allow hooking programs on kernel tracepoints
  - `CONFIG_NET_CLS_BPF` to write packets classifiers
  - `CONFIG_CGROUP_BPF` to attach programs on cgroups hooks

eBPF ISA

- eBPF is a "virtual" ISA, defining its own set of instructions: load and store instruction, arithmetic instructions, jump instructions,etc
- It also defines a set of 10 64-bits wide registers as well as a calling convention:
  - `R0`: return value from functions and BPF program
  - `R1, R2, R3, R4, R5`: function arguments
  - `R6, R7, R8, R9`: callee-saved registers
  - `R10`: stack pointer

```
; bpf_printk("Hello %s\n", "World");
      0:  r1 = 0x0 ll
      2:  r2 = 0xa
      3:  r3 = 0x0 ll
      5:  call 0x6
; return 0;
      6:  r0 = 0x0
      7:  exit
```

The eBPF verifier

- When loaded into the kernel, a program must first be validated by the eBPF verifier.
- The verifier is a complex piece of software which checks eBPF programs against a set of rules to ensure that running those may not compromise the whole kernel. For example:
  - a program must always return and so not contain paths which could make them "infinite" (e.g: no infinite loop)
  - a program must make sure that a pointer is valid before dereferencing it
  - a program can not access arbitrary memory addresses, it must use passed context and available helpers
- If a program violates one of the verifier rules, it will be rejected.
- Despite the presence of the verifier, you still need to be careful when writing programs! eBPF programs run with preemption enabled (but CPU migration disabled), so they can still suffer from concurrency issues
  - There are mechanisms and helpers to avoid those issues, like per-CPU maps types.

Program types and attach points

- There are different categories of hooks to which a program can be attached:
  - an arbitrary kprobe
  - a kernel-defined static tracepoint
  - a specific perf event
  - throughout the network stack
  - and a lot more, see `bpf_attach_type`
- A specific attach-point type can only be hooked with a set of specific program types, see `bpf_prog_type` and `bpf/libbpf/program_types`.
- The program type then defines the data passed to an eBPF program as input when it is invoked. For example:
  - A `BPF_PROG_TYPE_TRACEPOINT` program will receive a structure containing all data returned to userspace by the targeted tracepoint.
  - A `BPF_PROG_TYPE_SCHED_CLS` program (used to implement packets classifiers) will receive a `struct __sk_buff`, the kernel representation of a socket buffer.
  - You can learn about the context passed to any program type by checking `include/linux/bpf_types.h`

eBPF maps

- eBPF programs exchange data with userspace or other programs through maps of different nature:
  - `BPF_MAP_TYPE_ARRAY`: generic array storage. Can be differentiated per CPU
  - `BPF_MAP_TYPE_HASH`: a storage composed of key-value pairs. Keys can be of different types: `__u32`, a device type, an IP address...
  - `BPF_MAP_TYPE_QUEUE`: a FIFO-type queue
  - `BPF_MAP_TYPE_CGROUP_STORAGE`: a specific hash map keyed by a cgroup id. There are other types of maps specific to other object types (inodes, tasks, sockets, etc)
  - etc...
- For basic data, it is easier and more efficient to directly use eBPF global variables (no syscalls involved, contrary to maps)

The bpf() syscall

- The kernel exposes a bpf() syscall to allow interacting with the eBPF subsystem
- The syscall takes a set of subcommands, and depending on the subcommand, some specific data:
  - BPF_PROG_LOAD to load a bpf program
  - BPF_MAP_CREATE to allocate maps to be used by a program
  - BPF_MAP_LOOKUP_ELEM to search for an entry in a map
  - BPF_MAP_UPDATE_ELEM to update an entry in a map
  - etc
- The syscall works with file descriptors pointing to eBPF resources. Those resources (program, maps, links, etc) remain valid while there is at least one program holding a valid file descriptor to it. Those are automatically cleaned once there are no user left.
- For more details, see man 2 bpf

Writing eBPF programs

- eBPF programs can either be written directly in raw eBPF assembly or in higher level languages (e.g: C or rust), and are compiled using the clang compiler.
- The kernel provides some helpers that can be called from an eBPF program:
  - `bpf_trace_printk` Emits a log to the trace buffer
  - `bpf_map_{lookup,update,delete}_elem` Manipulates maps
  - `bpf_probe_{read,write}[_user]` Safely read/write data from/to kernel or userspace
  - `bpf_get_current_pid_tgid` Returns current Process ID and Thread group ID
  - `bpf_get_current_uid_gid` Returns current User ID and Group ID
  - `bpf_get_current_comm` Returns the name of the executable running in the current task
  - `bpf_get_current_task` Returns the current `struct task_struct`
  - Many other helpers are available, see `man 7 bpf-helpers`
- Kernel also exposes kfuncs (see `bpf/kfuncs`), but contrary to bpf-helpers, those do not belong to the kernel stable interface.

Manipulating eBPF program

- There are different ways to build, load and manipulate eBPF programs:
  - One way is to write an eBPF program, build it with clang, and then load it, attach it and read data from it with bare `bpf()` calls in a custom userspace program
  - One can also use `bpftool` on the built ebpf program to manipulate it (load, attach, read maps, etc), without writing any userspace tool
  - Or we can write our own eBPF tool thanks to some intermediate libraries which handle most of the hard work, like libbpf
  - We can also use specialized frameworks like BCC or bpftrace to really get all operations (bpf program build included) handled

BCC

- BPF Compiler Collection (BCC) is (as its name suggests) a collection of BPF based tools.
- BCC provides a large number of ready-to-use tools written in BPF.
- Also provides an interface to write, load and hook BPF programs more easily than using "raw" BPF language.
- Available on a large number of architecture (Unfortunately, not ARM32).
  - On debian, when installed, all tools are named `<tool>-bpfcc`.
- BCC requires a kernel version >= 4.1.
- BCC evolves quickly, many distributions have old versions: you may need to compile from the latest sources

Image credits:
https://github.com/iovisor/bcc

BCC tools

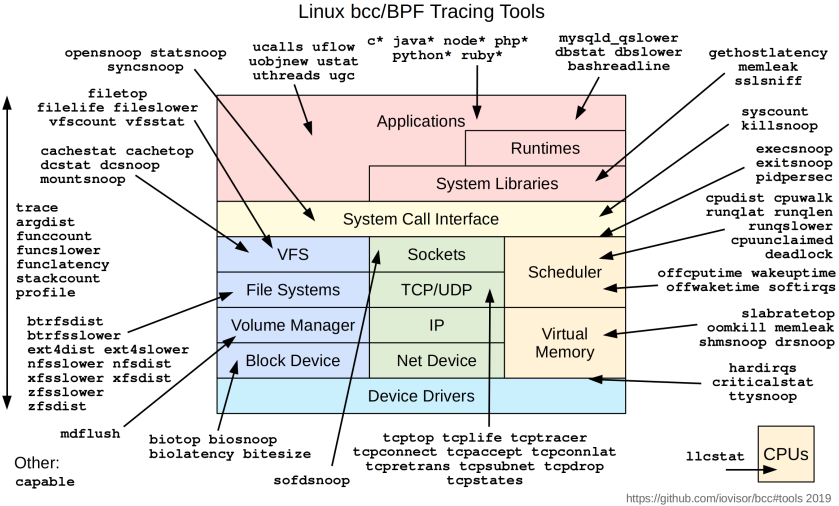


Image credits: https://www.brendangregg.com/ebpf.html

BCC Tools example

- `profile.py` is a CPU profiler allowing to capture stack traces of current execution. Its output can be used for flamegraph generation:

```
$ git clone https://github.com/brendangregg/FlameGraph.git
$ profile.py -df -F 99 10 | ./FlameGraph/flamegraph.pl > flamegraph.svg
```

- `tcpconnect.py` script displays all new TCP connection live

```
$ tcpconnect
PID    COMM         IP SADDR            DADDR            DPORT
220321 ssh          6  ::1              ::1              22
220321 ssh          4  127.0.0.1        127.0.0.1        22
17676  Chrome_Child 6  2a01:cb15:81e4:8100:37cf:d45b:d87d:d97d 2606:50c0:8003::154 443
[...]
```

- And much more to discover at https://github.com/iovisor/bcc

Using BCC with python

- BCC exposes a `bcc` module, and especially a `BPF` class
- eBPF programs are written in C and stored either in external files or directly in a python string.
- When an instance of the `BPF` class is created and fed with the program (either as string or file), it automatically builds, loads, and possibly attaches the program
- There are multiple ways to attach a program:
  - By using a proper program name prefix, depending on the targeted attach point (and so the attach step is performed automatically)
  - By explicitly calling the relevant attach method on the `BPF` instance created earlier

Using BCC with python

- Hook with a *kprobe* on the `clone()` system call and display `"Hello, World!"` each time it is called

```
#!/usr/bin/env python3

from bcc import BPF

# define BPF program
prog = """
int hello(void *ctx) {
    bpf_trace_printk("Hello, World!\\n");
    return 0;
}
"""
# load BPF program
b = BPF(text=prog)
b.attach_kprobe(event=b.get_syscall_fnname("clone"), fn_name="hello")
```

libbpf

- Instead of using a high level framework like BCC, one can use libbpf to build custom tools with a finer control on every aspect of the program.
- libbpf is a C-based library that aims to ease eBPF programming thanks to the following features:
  - userspace APIs to handle open/load/attach/teardown of bpf programs
  - userspace APIs to interact with attached programs
  - eBPF APIs to ease eBPF program writing
- Packaged in many distributions and build systems (e.g.: Buildroot)
- Learn more at https://libbpf.readthedocs.io/en/latest/

eBPF programming with libbpf (1/2)

my_prog.bpf.c

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

#define TASK_COMM_LEN 16
struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __type(key, __u32);
  __type(value, __u64);
  __uint(max_entries, 1);
} counter_map SEC(".maps");

struct sched_switch_args {
  unsigned long long pad;
  char prev_comm[TASK_COMM_LEN];
  int prev_pid;
  int prev_prio;
  long long prev_state;
  char next_comm[TASK_COMM_LEN];
  int next_pid;
  int next_prio;
};
```

- The fields to define in the `*_args` structure are obtained from the event description in `/sys/kernel/tracing/events` (see this example)

eBPF programming with libbpf (2/2)

my_prog.bpf.c

```c
SEC("tracepoint/sched/sched_switch")
int sched_tracer(struct sched_switch_args *ctx)
{
  __u32 key = 0;
  __u64 *counter;
  char *file;

  char fmt[] = "Old task was %s, new task is %s\n";
  bpf_trace_printk(fmt, sizeof(fmt), ctx->prev_comm, ctx->next_comm);

  counter = bpf_map_lookup_elem(&counter_map, &key);
  if(counter) {
          *counter += 1;
          bpf_map_update_elem(&counter_map, &key, counter, 0);
  }

  return 0;
}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

Building eBPF programs

- An eBPF program written in C can be built into a loadable object thanks to clang:

```
$ clang -target bpf -O2 -g -c my_prog.bpf.c -o my_prog.bpf.o
```

  - The `-g` option allows to add debug information as well as BTF information
- GCC can be used too with recent versions
  - the toolchain can be installed with the `gcc-bpf` package in Debian/Ubuntu
  - it exposes the `bpf-unknown-none` target
- To easily manipulate this program with a userspace program based on libbpf, we need "skeleton" APIs, which can be generated with to `bpftool`

bpftool

- `bpftool` is a command line tool allowing to interact with bpf object files and the kernel to manipulate bpf programs:
  - Load programs into the kernel
  - List loaded programs
  - Dump program instructions, either as BPF code or JIT code
  - List loaded maps
  - Dump map content
  - Attach programs to hooks (so they can run)
  - etc
- You may need to mount the bpf filesystem to be able to pin a program (needed to keep a program loaded after bpftool has finished running):

```
$ mount -t bpf none /sys/fs/bpf
```

bpftool

- List loaded programs

```
$ bpftool prog
348: tracepoint  name sched_tracer  tag 3051de4551f07909  gpl
loaded_at 2024-08-06T15:43:11+0200  uid 0
xlated 376B  jited 215B  memlock 4096B  map_ids 146,148
btf_id 545
```

- Load and attach a program

```
$ mkdir /sys/fs/bpf/myprog
$ bpftool prog loadall trace_execve.bpf.o /sys/fs/bpf/myprog autoattach
```

- Unload a program

```
$ rm -rf /sys/fs/bpf/myprog
```

- Dump a loaded program

```
$ bpftool prog dump xlated id 348
int sched_tracer(struct sched_switch_args * ctx):
; int sched_tracer(struct sched_switch_args *ctx)
  0: (bf) r4 = r1
  1: (b7) r1 = 0
; __u32 key = 0;
  2: (63) *(u32 *)(r10 -4) = r1
; char fmt[] = "Old task was %s, new task is %s\n";
  3: (73) *(u8 *)(r10 -8) = r1
  4: (18) r1 = 0xa7325207369206b
  6: (7b) *(u64 *)(r10 -16) = r1
  7: (18) r1 = 0x7361742077656e20
[...]
```

- Dump eBPF program logs

```
$ bpftool prog tracelog
kworker/u80:0-11  [013] d..41  1796.003605: bpf_trace_printk: Old task was kworker/u80:0, new task is swapper/13
<idle>-0          [013] d..41  1796.003609: bpf_trace_printk: Old task was swapper/13, new task is kworker/u80:0
sudo-18640        [010] d..41  1796.003613: bpf_trace_printk: Old task was sudo, new task is swapper/10
<idle>-0          [010] d..41  1796.003617: bpf_trace_printk: Old task was swapper/10, new task is sudo
[...]
```

bpftool

- List created maps

```
$ bpftool map
80: array  name counter_map  flags 0x0
    key 4B  value 8B  max_entries 1  memlock 256B
    btf_id 421
82: array  name .rodata.str1.1  flags 0x80
    key 4B  value 33B  max_entries 1  memlock 288B
    frozen
96: array  name libbpf_global  flags 0x0
    key 4B  value 32B  max_entries 1  memlock 280B
[...]
```

- Show a map content

```
$ sudo bpftool map dump id 80
[{
  "key": 0,
  "value": 4877514
  }
]
```

- Generate libbpf APIs to manipulate a program

```
$ bpftool gen skeleton trace_execve.bpf.o name trace_execve > trace_execve.skel.h
```

- We can then write our userspace program and benefit from high level APIs to manipulate our eBPF program:
  - instantiation of a global context object which will have references to all of our programs, maps, links, etc
  - loading/attaching/unloading of our programs
  - eBPF program directly embedded in the generated header as a byte array

Userspace code with libbpf

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "trace_sched_switch.skel.h"

int main(int argc, char *argv[])
{
    struct trace_sched_switch *skel;
    int key = 0;
    long counter = 0;

    skel = trace_sched_switch__open_and_load();
    if(!skel)
        exit(EXIT_FAILURE);
    if (trace_sched_switch__attach(skel)) {
        trace_sched_switch__destroy(skel);
        exit(EXIT_FAILURE);
    }

    while(true) {
        bpf_map__lookup_elem(skel->maps.counter_map, &key, sizeof(key), &counter, sizeof(counter), 0);
        fprintf(stderr, "Scheduling switch count: %d\n", counter);
        sleep(1);
    }

    return 0;
}
```

eBPF programs portability (1/2)

- Kernel internals, contrary to userspace APIs, do not expose stable APIs. This means that an eBPF program manipulating some kernel data may not work with another kernel version
- The CO-RE (Compile Once - Run Everywhere) approach aims to solve this issue and make programs portable between **kernel versions**. It relies on the following features:
  - your kernel must be built with `CONFIG_DEBUG_INFO_BTF=y` to have BTF data embedded. BTF is a format similar to dwarf which encodes data layout and functions signatures in an efficient way.
  - your eBPF compiler must be able to emit BTF relocations (both clang and GCC are capable of this on recent versions, with the `-g` argument)
  - you need a BPF loader capable of processing BPF programs based on BTF data and adjust accordingly data accesses: `libbpf` is the de-facto standard bpf loader
  - you then need eBPF APIs to read/write to CO-RE relocatable variables. libbpf provides such helpers, like `bpf_core_read`
- To learn more, take a look at Andrii Nakryiko's CO-RE guide

eBPF programs portability (2/2)

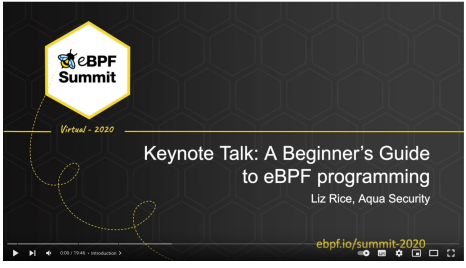
- Despite CO-RE, you may still face different constraints on different kernel versions, because of major features introduction or change, since the eBPF subsystem keeps receiving frequent updates:
  - eBPF tail calls (which allow a program to call a function) have been added in version 4.2, and allow to call another program only since version 5.10
  - eBPF spin locks have been added in version 5.1 to prevent concurrent accesses to maps shared between CPUs.
  - Different attach types keep being added, but possibly on different kernel versions when it depends on the architecture: fentry/fexit attach points have been added in kernel 5.5 for x86 but in 6.0 for arm32.
  - Any kind of loop (even bounded) was forbidden until version 5.3
  - `CAP_BPF` capability, allowing a process to perform eBPF tasks, has been added in version 5.8

eBPF for tracing/profiling

- eBPF is a very powerful framework to spy on kernel internals: thanks to the wide variety of attach point, you can expose almost any kernel code path and data.
- In the mean time, eBPF programs remain isolated from kernel code, which makes it safe (compared to kernel development) and easy to use.
- Thanks to the in-kernel interpreter and optimizations like JIT compilation, eBPF is very well suited for tracing or profiling with low overhead, even in production environments, while being very flexible.
- This is why eBPF adoption level keeps growing for debugging, tracing and profiling in the Linux ecosystem. As a few examples, we find eBPF usage in:
  - tracing frameworks like BCC and bpftrace
  - network infrastructure components, like Cilium or Calico
  - network packet tracers, like pwru or dropwatch
  - And many more, check ebpf.io for more examples

eBPF: resources

- BCC tutorial: https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md
- libbpf-bootstrap: https://github.com/libbpf/libbpf-bootstrap
- A Beginner's Guide to eBPF Programming - Liz Rice, 2020
  - Video: https://www.youtube.com/watch?v=lrSExTfS-iQ
  - Resources: https://github.com/lizrice/ebpf-beginners

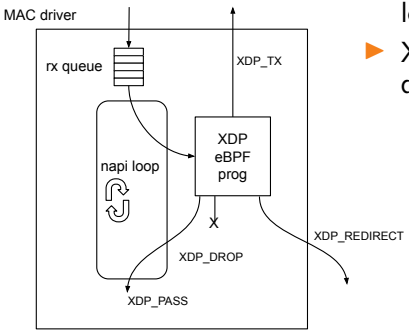


XDP

eXpress Data Path

- Run an eBPF program as close to frame reception as possible
- Support is Hardware-specific and driver-specific
- Introduced for high-performance networking, but available on embedded devices
- Take very fast decisions in the driver, with user-configurable eBPF code
- Used for fast routing, DDoS protection, firewalling, etc.
- With `AF_XDP`, offers an upstream alternative to kernel bypass

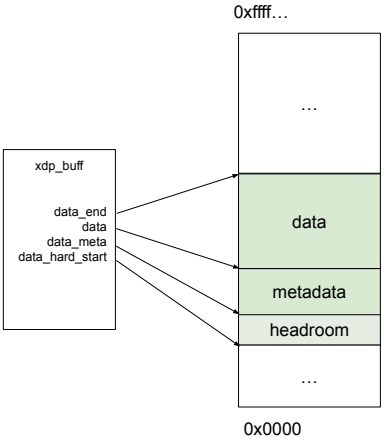
XDP program



- XDP programs are run by the MAC driver in the NAPI loop
- XDP programs may edit the received frame, and take a decision :
  - `XDP_PASS` : Packet continues to the Networking stack
  - `XDP_DROP` : Packet is immediately dropped
  - `XDP_ABORTED` : Similar `XDP_DROP` but triggers a tracepoint
  - `XDP_TX` : Packet is sent back from the same interface
  - `XDP_REDIRECT` : Packet is sent either :
    - back from another interface
    - to another CPU for processing
    - to an `AF_XDP` socket

XDP hook - driver side

```
u32 bpf_prog_run_xdp(const struct bpf_prog *prog, struct xdp_buff *xdp);
```



- `struct bpf_prog` : The XDP program attached to the interface
- `struct xdp_buff` : A representation of the buffer
- XDP runs *before* the `skb` is even created
- a `struct xdp_buff` is a very simple and lightweight representation of a frame

XDP hook - eBPF side

example program

```
SEC("xdp")
int xdp_dummy_prog(struct xdp_md *ctx)
{
        return XDP_PASS;
}
```

struct xdp_md definition

```
struct xdp_md {
        __u32 data;
        __u32 data_end;
        __u32 data_meta;
        __u32 ingress_ifindex; /* rxq->dev->ifindex */
        __u32 rx_queue_index;  /* rxq->queue_index  */
        __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

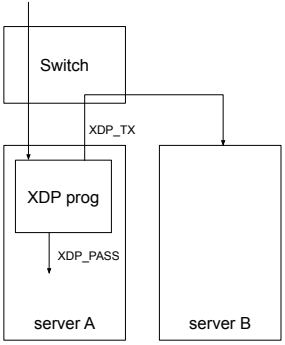
XDP_DROP

- Used for firewalling and DDoS protection
- A XDP Program returning `XDP_DROP` causes the frame to be dropped immediately
- `XDP_ABORTED` is similar, but triggers a tracepoint.
- Happens before the `struct sk_buff` is even created
- If the driver uses `page_pool`, the buffer is recycled
- Extremely efficient way of filtering

XDP_PASS

- A XDP Program returning `XDP_PASS` causes the frame to continue through the network stack
- The XDP program may modify the frame
- After `XDP_PASS`, a `struct sk_buff` will be created by the MAC driver
- The usual processing of the packet through the stack will occur

XDP_TX



- A XDP Program returning `XDP_TX` re-emits the frame on the same interface
- The frame may be modified by the program
- Main advertised use-case is to perform load-balancing

XDP_REDIRECT

- Redirects the frame towards a target identified by a **map**
- Programs don't directly return `XDP_REDIRECT`
- The special bpf helper `bpf_redirect_map()` must be used
  - See `man 7 bpf-helpers`

Example XDP redirect

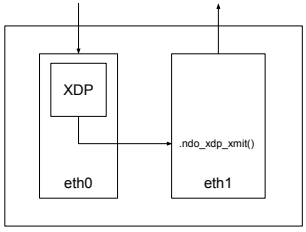
```
struct {
        __uint(type, BPF_MAP_TYPE_DEVMAP);
        __uint(max_entries, 8);
        __uint(key_size, sizeof(int));
        __uint(value_size, sizeof(int));
} tx_port SEC(".maps");

SEC("xdp")
int xdp_redirect_map_0(struct xdp_md *xdp)
{
        return bpf_redirect_map(&tx_port, 0, 0);
}
```

bpf_redirect_map

```
long bpf_redirect_map(void *map, __u64 key, __u64 flags)
```

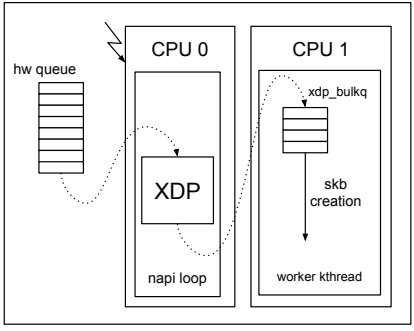
- See man 7 bpf-helpers
- eBPF helper for XDP_REDIRECT actions
- The redirection target is map[key], where map can be :
  - A BPF_MAP_TYPE_DEVMAP, value is an ifindex
  - A BPF_MAP_TYPE_CPUMAP, value is a cpu number
  - A BPF_MAP_TYPE_XSKMAP, value is a queue index

XDP_REDIRECT - To device



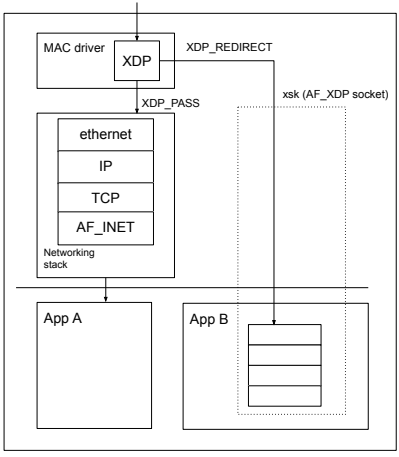
- Uses `BPF_MAP_TYPE_DEVMAP`, Documented here
- Forwards the frame to another XDP-enabled `struct net_device`
- The target device must implement `.ndo_xdp_xmit()`
- No `struct sk_buff` is created, the `struct xdp_buff` is sent directly
- `bpf_redirect()` can be used directly

XDP_REDIRECT - To CPU



- Uses BPF_MAP_TYPE_CPUMAP, Documented here
- Make the packet processing occur on another CPU core
- Useful for CPU load balancing
- Also used to circumvent hardware issues
  - Flawed hash computation in hardware for RSS
  - Wrong internal interrupt routing

XDP_REDIRECT - To Socket



- Uses `BPF_MAP_TYPE_XSKMAP`, Documented here
- Frames are forwarded directly to user memory attached to an `AF_XDP` socket (**XSK**)
- Upstream Linux's response to out-of-tree kernel bypass (e.g. DPDK)
- The driver is still in kernel, and the XDP program choses if bypass is needed for each frame
- No copy occurs, a **dedicated hardware queue** is needed
- Memory is shared with the **UMEM**, bound to a `queue_id` with `bind()`
- UMEM regions are shared ring-buffers, where user buffers are directly mapped to hw queues

XDP support in a driver

1. Implement the execution and return-code handling of the BPF XDP programs
   - Fairly straightforward, done in the main NAPI loop
2. Make sure the data handling meets the following constraints :
   - Frame must be **readable** and **writeable**
   - There must be **a headroom** big enough to fit `struct xdp_frame`
   - There must be **a tailroom** big enough to fit all `skb_shared_info`
3. XDP frags is supported since v5.16
   - Allows using XDP with non-linear frames, which used to be impossible
4. The `struct xdp_buff` layout uses `struct skb_frag` as well

Loading an XDP program

- XDP programs are built like any other eBPF program :

```
clang -O2 -g -target bpf -c xdp_prog.c -o xdp_prog.o
```

- They can be loaded with `iproute2` :

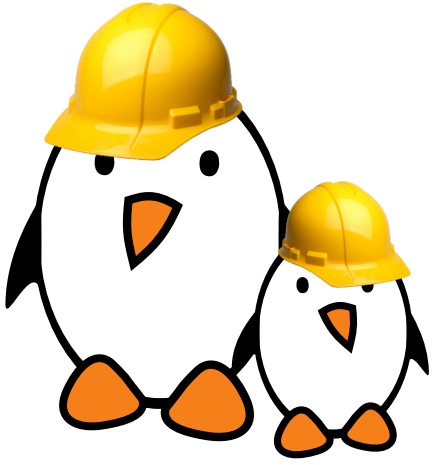
```
ip link set dev eth0 xdp obj xdp-prog.o
```

- `iproute2` xdp support is recent, `xdp-loader` from `xdp-tools` can be used :

```
xdp-loader load eth0 xdp_drop.o
```

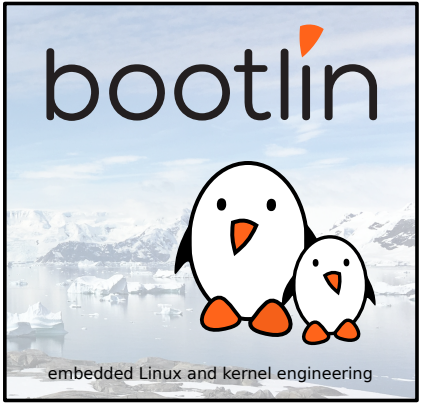
- `bpftool` can also be used to attach XDP programs
- `ethtool -S <iface>` shows the XDP statistics
- `xdp-monitor` shows detailed statistics using BPF tracing

Practical lab - Using eBPF, XDP example



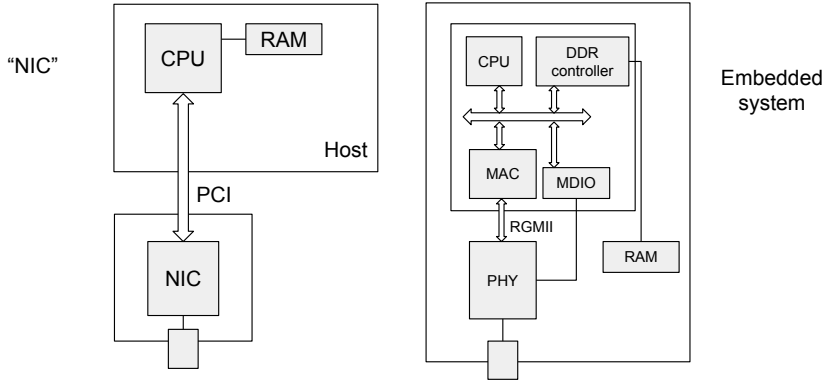
- Write a simple XDP program

Network device drivers



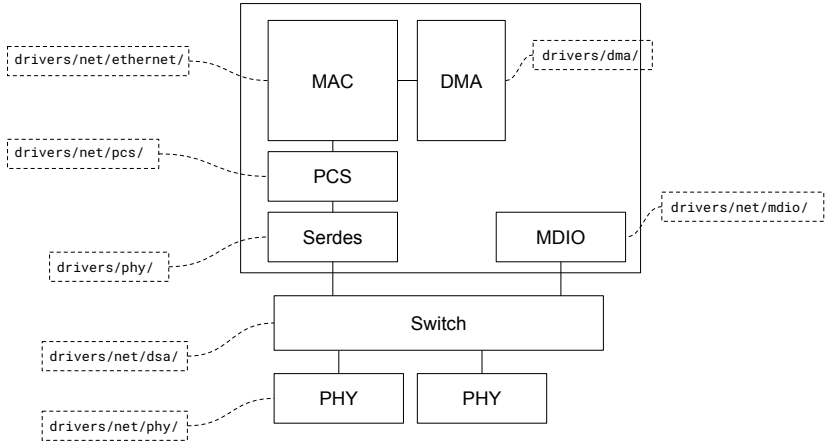
NIC and MAC

- Terms **NIC** and **MAC** are sometimes used interchangeably
- **N**etwork **I**nterface **C**ontroller, usually refers to "Network Cards"
  - MAC and PHY integrated in a single component. Usually, the PHY is transparent
- On embedded systems, we control each individual component

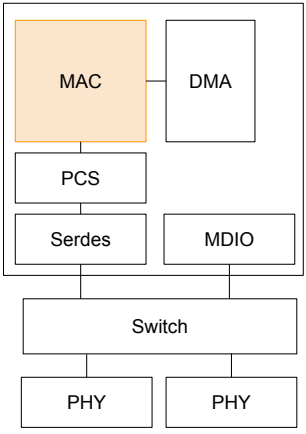


Low level networking components

- Multiple drivers are involved to configure a Network Interface
- Not all of them are required (depending on the design)
- Some MAC drivers also include DMA, Serdes, PCS and even PHY drivers

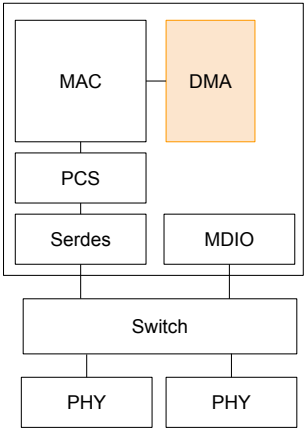


Low level networking components - MAC



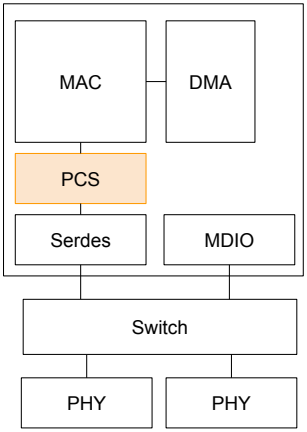
- The main component of a network interface
- Represented by `struct net_device`
- Drivers are in `drivers/net/ethernet/`
- In charge of **Sending** and **Receiving** frames
- Configures all the **Hardware offloaded** features
- Reports status and statistics
- Some devices include a PCS, Serdes, MDIO, PHY, DMA and even a Switch controller in the MAC
  - The single MAC driver handles it all

Low level networking components - DMA



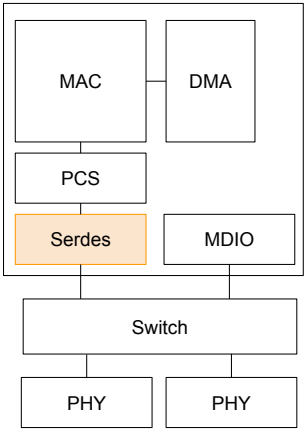
- Some MAC controllers are connected to a shared **DMA Controller**
- The controller handles DMA transfers for multiple devices
- The MAC requests `struct dma_chan` for TX and RX
  - This is done using the **dmaengine** API
- Drivers are in `drivers/dma/`
- It is not unusual to have MAC with an integrated DMA controller
  - In that case, we don't use the dmaengine API

Low level networking components - PCS



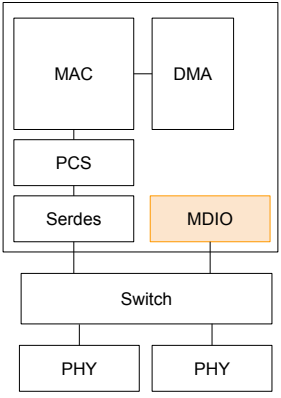
- **P**hysical **C**oding **S**ublayer
- Represented by `struct phylink_pcs`
- Drivers in `drivers/net/pcs/`
- Component in charge of Data Encoding
  - For signal integrity, bits are encoded into symbols
  - At 100Mbps : 4 bits data, 5 bits symbols (4b/5b)
  - At 1000Mbps : 8b/10b
  - At 10Gbps : 64b/66b
- Also in charge of in-band signaling
  - Link status, speed, duplex, flow-control
- Can be transparently handled by the MAC (no driver)
- The MAC driver may register its own PCS instance(s)
- Some IPs are re-used across vendors, dedicated drivers are then used

Low level networking components - Generic PHY



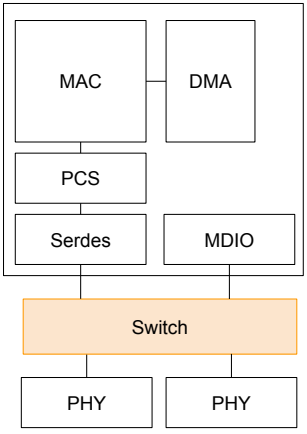
- Generic PHY, driving the physical link that come out of the MAC
- Represented by `struct phy`
- Drivers in `drivers/phy/`
  - Not to be confused with `drivers/net/phy/`
- Usually drives **SerDes** lanes if the MAC interface is serialized
- Also used by other subsystems : USB, PCI, Sata, etc.
- Controls the physical link parameters
  - Drive strength
  - Timings
  - link training, etc.
- Sometimes transparently handled by the MAC without a dedicated driver

Low level networking components - MDIO



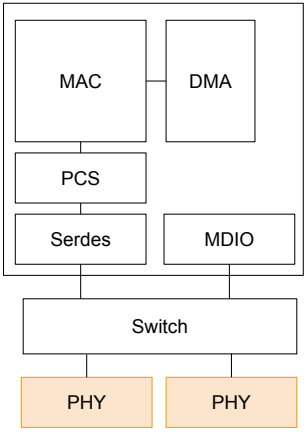
- **M**anagement **D**ata **I**nput **O**utput
  - a.k.a. **SMI** : **S**erial **M**anagement **I**nterface
  - a.k.a. **MIIM** : **M**edia **I**ndependent **I**nterface **M**anagement
- Bus controller represented by `struct mii_bus`
- Peripherals represented by `struct mdio_device`
- Drivers in `drivers/net/mdio/`
- Management bus for most Ethernet PHYs and DSA Switches
  - Only bus for PHYs
  - Some DSA switches can be controlled by SPI or I^2C
- Provides ways to access registers, physically similar to I^2C
- Often controlled by the MAC driver, but can be standalone

Low level networking components - Switch



- DSA Switches are standalone chips, with one more ports connected to the SoC's MAC
  - **D**istributed **S**witch **A**rchitecture
  - Relies on switchdev for the switching operations
- Switches can also be integrated within the SoC
  - The MAC driver implements the switchdev operations
- DSA switch represented by `struct dsa_switch`
- DSA switch port represented by `struct dsa_port`
- Switch port represented by `struct net_device` (even for DSA)
- Drivers in `drivers/net/dsa/` and `drivers/net/ethernet/`

Low level networking components - Ethernet PHY



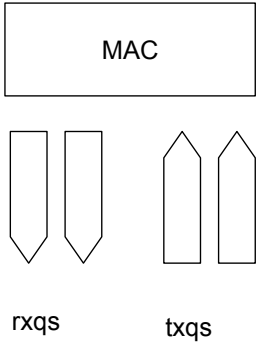
- In charge of 802.3 Layer 1 (PHY) operations
- Represented by `struct phy_device`
- Drivers in `drivers/net/phy/`
- Specific to MDIO PHYs, as per the 802.3 specification
- In charge of link management :
  - Auto-negociation of Speed and Duplex
  - Link detection
- A generic driver exists using only standard registers
- The PHY management framework is called **phylib**

Ethernet controller driver

Ethernet driver endpoints

- Ethernet controllers are represented by `struct net_device`
- Entry points are the `struct net_device_ops`
- Extra ethernet-specific callbacks implemented with `struct ethtool_ops`

```
dev->netdev_ops = &mvneta_netdev_ops;
dev->ethtool_ops = &mvneta_eth_tool_ops;
```



- Most Ethernet controllers today have multiple **transmit** and **receive** queues
- **tx** queues hold **descriptors** for packets that are yet-to-be-sent
- The NIC will dequeue une packet at a time during transmission, from one of the tx queues
- The TX de-queueing behaviour can sometimes be controlled : Weighted Round-Robin, Per-queue priorities, etc.
- **rx** queues hold descriptors for packets received that weren't yet handled by the CPU
- the RX buffer size depends on the configured **MTU**

RX filtering

- The **receive filtering** is adjusted with :

  ```
  void (*ndo_set_rx_mode)(struct net_device *dev);
  ```

- `dev->flags` contains the new filtering parameters :
  - `IFF_PROMISC` : If set, the interface must go in **promiscuous mode**
    - No hardware filtering of incoming frames must occur
    - Used by tools like `tcpdump`, or `ip link set dev eth0 promisc on`
  - `IFF_ALLMULTI` : If set, all **multicast frames** must be accepted
- The **unicast** and **multicast** address list must be updated :
  - `dev->uc` contains all the Unicast addresses the interface must accept
  - `dev->mc` contains all the Multicast addresses the interface must accept
- The address list in maintained by the Networking stack, updated through `dev_uc_add()`, `dev_uc_del()`, `dev_mc_add()`, etc.

Changing the MTU

- **M**aximum **T**ransmit **U**nit, used by the upper layers for fragmentation
  - Stored in `netdev->mtu`
- User-modifiable with `ip link set dev eth0 mtu 1500`
  - Triggers a call to the `.ndo_change_mtu` netdev ops
- Changing the MTU may require the **re-allocation** of RX buffers in the queues

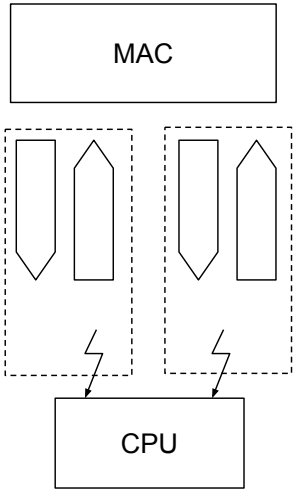
`.ndo_change_mtu()` - option 1

```
if (netif_running(ndev
        return -EBUSY; /* Can't change the MTU while the interface is UP */

WRITE_ONCE(ndev->mtu, new_mtu);
```

`.ndo_change_mtu()` - option 2

```
WRITE_ONCE(ndev->mtu, new_mtu);

foo_stop_dev(dev); /* Stop sending and receiving, empty the queues*/
foo_realloc_queues(dev); /* Re-allocate buffers for the new MTU */
foo_start_dev(dev); /* Resume */
```

Channels



- `tx` and `rx` queues will notify queueing and dequeueing through `interrupts`
- Multiple queues may share the same interrupt line
- A **channel** represents an interrupt line and its associated queues
- Channels can be added or removed, depending on hardware support

```
void (*get_channels)(struct net_device *, struct ethtoo
int (*set_channels)(struct net_device *, struct ethtool
```

Receiving data

- The receive path for a Ethernet Controller Driver must use the **NAPI** API
- The entry-point is an **interrupt handler** that will be called upon frame reception
- There may be multiple interrupts for RX (per-queue, per-cpu...)
- NAPI mandates a short **top half** that acknowledges the interrupt and **masks it**
- The handler then calls `napi_schedule()`.
- Most of the processing occurs in **softirq** context, on the **same CPU** that handled the interrupt

NAPI principle

- NAPI does **not** stand for **N**ew **API**
  - it was new in Linux v2.4, today NAPI means NAPI
- Designed to avoid interrupt interference, as traffic often occurs in bursts
- The **first packet** of a burst is handled through interrupt
- The interrupt stays disabled, each subsequent packet is pulled using **polling**
- The polling stops when the **budget** is exhausted (by default, 50 packets)
- The driver must then re-enable the interrupts for further processing
- NAPI is not a batch processing mechanism, which is achievable through **interrupt coalescing**

NAPI Loop

- Register the NAPI polling function (per-queue) : `netif_napi_add`
- Polling function runs in **softirq** context : **Cannot sleep**

1. Hardware IRQ fires upon receiving a first packet
2. IRQ handler disables interrupts for that queue, and calls `napi_schedule`
3. The NAPI system calls the polling function
   `int (*poll)(struct napi_struct *napi, int budget`
4. Each received frame counts as **1** budget item
   - If the budget is exhausted but there are still packets to process, return `budget`
   - The polling function may be called with a budget of `0`, for TX processing only
5. If the budget isn't exahusted but all packets are processed, call `napi_complete_done()` and unmask interrupts

NAPI instances

- NAPI poll handlers are registered with `netif_napi_add()`
- Multiple NAPI **instances** can be registered
  - Usually one per **channel** or per **CPU**

napi .poll()

```
static int foo_poll(struct napi_struct *napi, int budget)
{
    while(rx_done < budget) {
        buff = foo_queue_get_next_desc();
        foo_do_xdp(buff);
        skb = foo_build_skb(buff);
        napi_gro_receive(skb);
        rx_done++;
    }
}
```

Interrupt Coalescing

- **Hardware feature** where the MAC waits for multiple packets to be received before triggering the interrupt
- Users configure a number of pending packets and a timeout for RX interrupt generation
- Must be fine-tuned depending on the use-case :
  - High threshold allows batching the interrupts, suitable for high-throughput workloads
  - Low threshold allows triggering interrupts as soon as data is received, for low-latency workloads
- `ethtool -C eth0 adaptive-rx on adaptive-tx on tx-usecs-irq 50`
- Software IRQ coalescing also exists, using NAPI and a polling-rearm timer

Transmit path

- Packets are sent from a network controller through the `.ndo_start_xmit` callback
  - `netdev_tx_t .ndo_start_xmit(struct sk_buff *skb, struct net_device *dev)`
  - it is the only mandatory `net_device_ops` member !
- The `.ndo_start_xmit()` callback enqueues and sends the `skb`
- If the skb is fragmented, all fragments must be sent
- In case of an error, the `skb` is dropped, but we still return `NETDEV_TX_OK`
- The `dev->stats` and driver-specific counters must be updated

NAPI TX

- TX **completions** are handled in the NAPI loop
- The driver acknowledges that packets were correctly sent
- During a NAPI poll, drivers are free to acknowledge as many TX packets as they want
- `budget` does not apply to TX packets
  - The NAPI poll function can acknowledge as my TX packets as it wants

Buffer management

- Hardware queues contain pre-populated **dma descriptors**
- On the receive side, when receiving packets the queues must be **refilled**
- Usually done at the end of the NAPI loop
- This is driver specific, but the buffers can be kept in **pools**

Page pool

- Page pools allows fast page allocation without locking
- One `struct page_pool` must be allocated per-queue
- Getting a page from `page_pool` happens without locking
- This is only used on the **receive** side
- Using `page_pool` is strongly recommended to support **XDP**
- See the page pool documentation

Timestamping

- Packet timestamping can be done in RX and TX
- Hardware timestamping is configured in the `.ndo_ioctl()`
  - Being replaced in favor of `.ndo_timestamping()`
- Timestamp is stored in the `skb_shared_info`
- On TX timestamping, the `skb` is cloned, timestamp is attached to the clone
  - The clone is sent back to the **socket error queue**

Netdev features

- `features` represents hardware offload capabilities
  - Checksumming, Scatter-gather, segmentation, filtering (mac / vlan)
  - see `ethtool -k <iface>`
  - attributes of `struct net_device`
- Drivers set `netdev.hw_features` at init, and can also set `netdev.features`
  - `features` : The current active features
  - `hw_features` : Features that can be changed (*hw != hardware*)
- Users but also the core might want to change the enabled features
  - Child devices might require some features to be disabled
- `.ndo_fix_features()` filters incompatible feature sets for the driver
- `.ndo_set_features()` applies the new feature set
- https://docs.kernel.org/networking/netdev-features.html

Offloading

- Most modern controllers can perform themselves some operations on packets
- **checksumming** offload makes so the CPU doesn't have to check or compute checksums
- **filtering** offloads makes so that the MAC drops unknown MAC addresses and VLANs
- **classification**-capable controllers may implement more powerful features :
  - **header hashing** computes a has of a specific set of fields in the header. Useful for RSS.
  - **flow steering** allows specific actions (enqueueing, drop, redirection) to be done based on specific header values
- **crypto** offload for some tunneling technologies such as MACSec
- Offloading can however make debugging harder, as decisions are taken before packets reach the CPU
  - Most controllers will expose counters, accessible over `ethtool -S <iface>`

Checksumming

- Some protocols include a checksum of the header or payload in the frame
  - Ethernet has the **FCS** that checksums the whole frame
  - IPv4 header includes a Header Checksum for the IPv4 header itself
  - TCP and UDP checksums the header and the payload
- Checksum computation and verification need to be done for every frame
- Some devices can compute and verify checksums at the hardware level
- TX checksumming involves computing the checksum of a section of the packet, and editing the checksum inline
- **caution :** Outgoing traffic will be shown with wrong checksums in captures

Receive **S**ide **S**teering

- Hardware feature on the **receive** side to spread traffic handling across queues and cores
- Requires per-queue or per-cpu interrupt support in the Ethernet controller
- Incoming packets can't be arbitrarily steered to any CPU or queue
  - Risk of out-of-order delivery, and bad caching behaviour
- The hardware parses the packet header and computes a hash of some of its fields
  - Usually CRC32 or Toeplitz
- The Hash is then used as a lookup index in a RSS table to get the RX queue
- `.get_rxfh` and `.set_rxfh`

XPS

- Associate TX queues with CPU cores
- Frames enqueued from a given core always go in the same queue
- Avoids contention on enqueueing
  - Each CPU can enqueue without cross-CPU locking
- Optimizes the completion handling
- `netif_set_xps_queue(dev, cpumask_of(queue), queue);`

Flow steering

- Advances controllers have the ability to **parse** packet headers
- This is complex, as the header fields are not at fixed offsets (presence of VLAN tags, encapsulation, etc.)
- This is usually achieved with internal **TCAM-based lookup engines**
- Traffic is classified based on specific fields : src/dst MAC, src/dst IP, src/dst Port, VLAN, etc.
- Classified traffic are assigned actions :
  - Drop (DoS mitigation)
  - Redirection
  - Enqueueing
  - Policing (rate limiting)

TC and ethtool steering

- Flow steering can be offloaded via `ethtool` :

```
ethtool -K eth0 ntuple on
ethtool -N eth0 flow-type tcp4 vlan 0xa000 m 0x1fff action 3 loc 1
```

- Steers Vlan-tagged TCP over IPv3 with priority 6 (`0xa000 & 0x1fff`) in queue 3
- Implemented in the `.set_rxnfc` ethtool op, *e.g.* `mvpp2_ethtool_set_rxnfc()`
- Can also be done with **tc flower**
  - Through `.ndo_setup_tc`
- Both APIs use the same representation : `struct flow_rule`

Multi-queue and priorisation

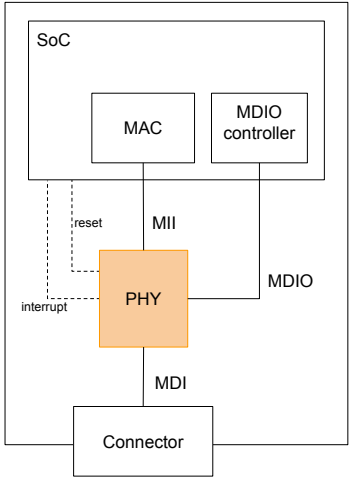
- Some Ethernet Controllers have multiple `tx` and`rx` queues
- On the **transmit** side, allows shaping and priorizing traffic
- On the **receive** side, allows load-balancing and per-queue actions
- flows can be assigned to dedicated **rx queues**
- These queues can in turn be pinned to CPUs, apps, VMs, of be rate-limited.
- Most of this is controlled through **tc** in `.ndo_setup_tc`
- Example in `mvneta_setup_mqprio()`

Other `struct ethtool_ops`

- `.get_ringparam` and `.set_ringparam`
  - Adjust the size of queues
- `.get_strings` and `get_ethtool_stats`
  - Returns hardware stats
- `.get_link_ksettings` and `set_link_ksettings`
  - Get the link speed and duplex
  - Usually the driver queries the PHY driver to get this information
- `.get_wol` and `.set_wol`
  - Configure Wake-on-Lan

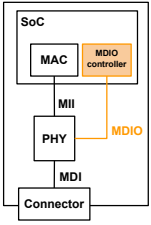
PHY driver and link management

PHY devices



- Ethernet PHYs handle Layer 1 of the OSI model
- Standardised by IEEE 802.3
- **M**edia **I**ndependent **I**nterface
  - Communication bus between MAC and PHY
- **M**edia **D**ependent **I**nterface
  - Communication medium with the **link partner**
  - Can be Cat6 cable, Fibre, Coax, backplane, etc.
- **M**anagement **D**ata **I**nput **O**utput
  - Control bus for PHY devices
  - Can be shared by multiple PHYs
  - Allows accessing PHY registers
- Optionally, PHYs can raise interrupts
  - *e.g.* to report link status change
- Optionally, PHYs can have a reset line

MDIO Bus



- Most common bus to access Ethernet PHYs
- Addressable, 32 addresses
- Physically very similar to i2c
  - An **adapter** initiates all transfers to **devices**
  - 2 physical signals : **MDC** for the clock, **MDIO** for data
- 802.3 defines 2 protocols for MDIO :
  - Clause 22 : 5 bits device address, 5 bits register address, 16 bits data
  - Clause 45 : 3-part addresses : 5 bits addresses, 5 bits **devtype**, 16 bits register address, 16 bits data
  - **devtype** allows addressing sub-components of the PHY : PCS, PMA/PMD, etc.
  - C45 is backwards compatible with C22
- Register layout is defined by **802.3**, with room for vendor-specific registers
- a gpio bitbang MDIO driver exists

MDIO driver

- MDIO controller drivers are represented by `struct mii_bus`
- Contains callback ops for `C22` and `C45` access :

```
struct mii_bus {
        const char *name;
        void *priv;
        int (*read)(struct mii_bus *bus, int addr, int regnum);
        int (*write)(struct mii_bus *bus, int addr, int regnum, u16 val);
        int (*read_c45)(struct mii_bus *bus, int addr, int devnum, int regnum);
        int (*write_c45)(struct mii_bus *bus, int addr, int devnum, int regnum, u16 val);
        int (*reset)(struct mii_bus *bus);
/* ... truncated ... */
};
```

MDIO accessors

- Raw read/write operations :

```
int mdiobus_read(struct mii_bus *bus, int addr, u32 regnum);
int mdiobus_write(struct mii_bus *bus, int addr, u32 regnum, u16 val);
int mdiobus_c45_read(struct mii_bus *bus, int addr, int devad, u32 regnum);
int mdiobus_c45_write(struct mii_bus *bus, int addr,  int devad, u32 regnum, u16 val);
```

- Wrapped by phylib for convenience : `phy_read()`, `phy_read_mmd()`, etc.
- Unlocked versions : `__mdiobus_write()`, `__phy_write()`, etc.
  - Caller implements their own locking
  - Useful for large transfers, *e.g.* loading a firmware

MDIO access from userspace

- `ioctl` based API, limited on purpose
- Useful for debugging, but interferes with phylib
  - Phylib and drivers have no way to track user-made configuration
- Main userspace tool is `phytool`
  - `phytool read eth0/1/2` : Read register 2 from mdio device at address 1
  - `phytool read eth0/1:2/3` : Read register 3 on MMD 2 from mdio device at address 1
- Can be tedious for indirect accesses
- **mdio-tools** uses an out-of-tree module to access MDIO over Netlink

MDIO controllers in devicetree

arch/arm64/boot/dts/marvell/armada-37xx.dtsi

```
mdio: mdio@32004 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "marvell,orion-mdio";
    reg = <0x32004 0x4>;
};
```

arch/arm/boot/dts/st/stm32mp15xx-dkx.dtsi

```
&ethernet0 {
    mdio {
        compatible = "snps,dwmac-mdio";
        /* ... */
    };
};
```

- SoCs may have dedicated MDIO controllers
  - Dedicated drivers with their own `compatible`
- Some MACs and DSA switches have an integrated MDIO controller
  - `mdio` child node within the MAC controller's node

Ethernet PHYs identification

- 802.3 specifies that registers 0x2 and 0x3 are **identifiers**
  - `OUI` (24 bits) and `Model` information (10 bits)
- PHY drivers register which identifier they support
  - `phy_driver.phy_id`
- We don't need per-device `compatible` strings in devicetree
- PHY `compatible` is used to indicate :
  - The MDIO clause :
    - `ethernet-phy-ieee802.3-c22`
    - `ethernet-phy-ieee802.3-c45`
  - The PHY id, if PHY reports the wrong information
    - *e.g.* `ethernet-phy-id2000.a231`

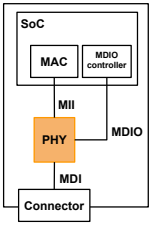
Ethernet PHYs in devicetree

arch/arm64/boot/dts/marvell/armada-8040-mcbin.dtsi

```
&cp0_mdio {
    status = "okay";

    ge_phy: ethernet-phy@0 {
        reg = <0>;
    };
};

&eth0 {
        phy-handle = <&ge_phy>;
}
```

- `reg` - mandatory
  - The PHY's address on the MDIO bus
  - Usually assigned via PCB straps
- `reset-gpios` : GPIO reset line
- `rx|tx-internal-delay-ps`
  - RGMII delays adjustments
- `leds` : LEDs driven by the PHY
- `interrupts` :
  - Status interrupt, level-triggered
- `sfp` : *phandle* to an SFP cage description

PHY devices in the kernel



- A PHY driver is represented by `struct phy_driver`
- PHY instances are represented by `struct phy_device`
  - By convention, objects are named `phydev` or `phy`
- **All** Ethernet PHY devices are `mdio` devices
  - Fixed-PHY uses an emulated bus
  - Memory-mapped PHYs can use a regmap conversion layer
- Managed by the **phylib** PHY framework
- PHY drivers mostly handle the vendor-specific aspects
- Most of the standardised logic is generic, and implemented in phylib
- A Generic driver implements only the standard logic
  - Used as a fallback when a PHY is detected with no associated driver

PHY device role

- The PHY driver reports the **link status** :
  - Updated by `phy_driver.read_status()`
    - Called upon PHY interrupt, or polled
  - `phydev.link` : Link with the partner is `UP` or `DOWN`
  - `phydev.speed` : Established link speed, in Mbps
  - `phydev.duplex` : Established duplex (half or full)
- It is in charge of configuring and performing the **Link negociation**
  - Based on what the MAC can do and user-specified parameters
  - *e.g.* `ethtool -s eth0 speed 100 duplex full autoneg on` on a 1G interface
- It may implement some **offloaded operations**
  - Some PHYs can offload **MACSec**
  - PHY timestamping is implemented by some devices

PHY device role - 2

- **Wake on Lan** can be implemented at the PHY level
  - The PHY receives the magic packet
  - It triggers an interrupt to wake the system up
  - `phy_driver.set_wol()` and `phy_driver.get_wol()`
- Some PHYs can perform **cable testing**
  - Detects cable and connector faults
  - `ethtool --cable-test eth0`
- They may report stats, useful for debugging link bringup
  - `ethtool --phy-statistics eth0`
- `BaseT1S` PHYs can configure the plca parameters

Fixed-link

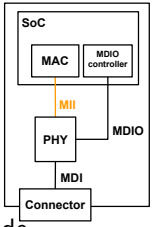
- The PHY is responsible for reporting the link state, but doesn't always exist
- *e.g.* MAC to MAC links, between a SoC and a DSA switch
- Fixed-link allows describing a link that is always UP
- In creates a virtual PHY internally that reports fixed parameters

fixed link example

```
&eth0 {
        /* ... */
        fixed-link {
                speed = <1000>;
                full-duplex;
        };
};
```

MII

- **M**edia **I**ndependent **I**nterface
- Conveys the data stream between MAC and PHY
- Specified in devicetree via `phy-mode` or `phy-connection-type`
- In some scenarios, the mode may change dynamically
  - For **serialized** modes that are physically compatible
  - Depending on the negociated link speed, the PHY may change its mode
  - example: the Marvell 88x3310 PHY
  - When link speed is negociated at 1Gbps, uses **SGMII**
  - When link speed is negociated at 2.5Gbps, uses **2500BaseX**
  - When link speed is negociated at 10Gbps, uses **10GBaseR**
- On the MAC side, may require spefific **PCS** and **Serdes** configuration



MII flavours - Parallel interfaces

- `MII` : Also describes a 8-bit, 10/100Mbps interface
- `RMII` : **R**educed **MII** : 4 bits, 10/100Mbps
  - Popular mode for 100Mbps interfaces
- `GMII` : **G**igabit **MII** : 8 bits, 10/100/1000Mbps
  - Rarely found on PCBs, mostly unsed within the SoC
- `RGMII` : **R**educed **G**igabit **MII** : 4 bits, 10/100/1000Mbps
  - Popular mode for 1Gbps interfaces
- `XGMII` : **X** (Roman Numeral 10) **G**igabit **MII** : 32 bits, 10Gbps
- `XLGMII` : **XL** (Roman Numeral 40) **G**igabit **MII** : 32 bits, 40Gbps
  - **XGMII** and **XLGMII** are on-silicon modes, not used on PCBs

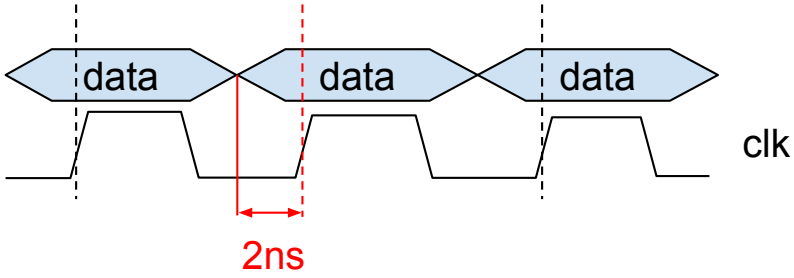
MII flavours - Serial interfaces

Differential pairs for Data (1 RX + 1 TX = 1 **lane**), clock optional, inband signaling

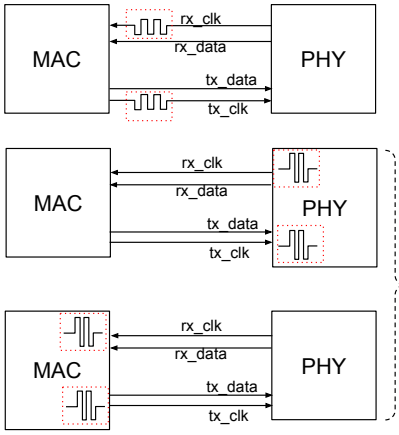
- `Cisco SGMII` : **S**erialized **G**igabit **MII**, 1 lane, 10/100/1000Mbs
  - *de facto* standard. Lane always clocked at 1.25GHz
  - Frames are repeated for 10 and 100Mbps
  - Inband signalling : Special word sent on the link to negotiate speed, duplex and flow control
- `QSGMII` (Quad SGMII) : Mux 4 different MAC to PHY links on a single 5Gbps lane
- `USXGMII` : Standard for 10Gbps link. Supports 10/100/1000Mbps, 2.5/5/10Gbps
  - Implements **rate matching** : The clock speed adjusts to follow the link speed
  - Supports multiplexing up to 8 links on the same lane
- `XAUI` and `RXAUI` : Standard, 10Gbps on 4 or 2 lanes, 10b/8b encoding.

RGMII delay

- RGMII is a popular interface on embedded sytems
- Per the specification, **clock** must have a 2ns delay from **data**
- This is to ensure data signals have settled when sampled



RGMII modes



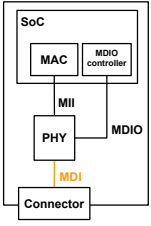
- The delays can be added using different methods :
- Longer PCB lines for the clock
  - Very rarely done
- Most PHYs and some MACs can insert delays internally
  - RGMII-**ID** modes : **I**nternal **D**elay
  - Preferred solution, delays are adjustable
- Delays may only need to be added in one direction
  - RGMII-TXID : TX delays are internal
  - RGMII-RXID : RX delays are internal
- Some MAC and PHYs have **hardwired** delays

RGMII modes in devicetree

- `phy-mode` in devicetree : Hardware representation
  - `phy-mode = "rgmii";` : No delays needs to be added
  - `phy-mode = "rgmii-id";` : delays need to be added internally
  - `phy-mode = "rgmii-txid";` : delays need to be added in TX
  - `phy-mode = "rgmii-rxid";` : delays need to be added in RX
- Internally, these mode are represented as `PHY_INTERFACE_MODE_RGMII[_ID|_TXID|_RXID]`
- The MAC driver reads the mode, and passes it to the PHY driver
- If the MAC inserts delays, it modifies the mode passed to the PHY
  1. e.g. `phy-mode = "rgmii-id";`
  2. MAC inserts delays in TX, but not in RX
  3. MAC passes `PHY_INTERFACE_MODE_RXID` to the PHY

MDI - Media Dependent Interface

- A huge number of physical protocols are defined by the 802.3 standard
- As of v6.15, 120 linkmodes are supported
- They follow a specific naming convention from IEEE 802.3
- speedBand-MediumEncodingLanes : 1000Base-T, 10GBase-KR, 10Base-T1...
- Band: BASEband, BROADband or PASSband.
- Medium
  - Base-**T**: Link over twisted-pair copper cables (Classic RJ45).
  - Base-**K**: Backplanes (PCB traces) links.
  - Base-**C**: Copper links.
  - Base-**L**, Base-**S**, Base-**F**: Fiber links.
  - Base-**H**: Plastic Fiber.
- Encoding: Describe the block encoding used by the PCS
  - Base-**X**: 10b/8b encoding.
  - Base-**R**: 66b/64b encoding.
- Lanes: Number of lanes per link (for Base-**T**, number of twisted pairs used).



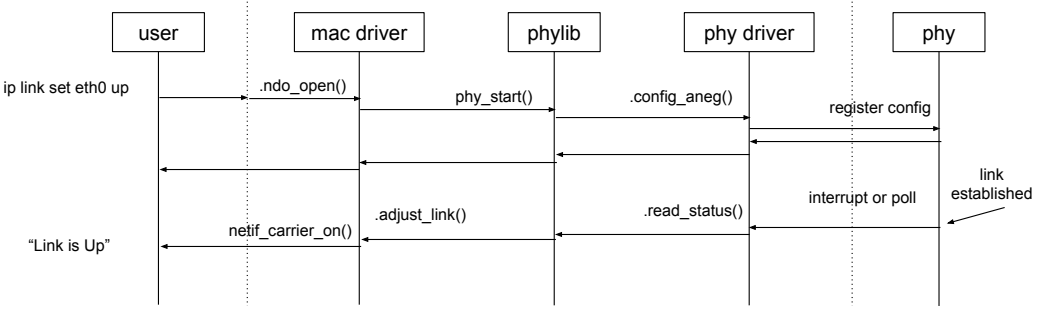
linkmodes

- In 802.3 Clauses 22 and 45, standard registers report the capabilities
- Allows dynamically building the list of MDI modes supported by the PHY
- Done in `genphy_read_abilities()` and `genphy_c45_pma_read_abilities()`
- PHY drivers can implement their own `phy_driver.get_features()`
- Get the supported linkmodes **on the interface** : `ethtool eth0`
  - Intersection between :
  - what the PHY can do : `phydev->supported`
  - what the MAC can do based on the **mac capabilities**
  - what the in-use **MII** interface can convey
- The **advertised** linkmodes take into account the user settings

Interactions between MAC and PHY drivers

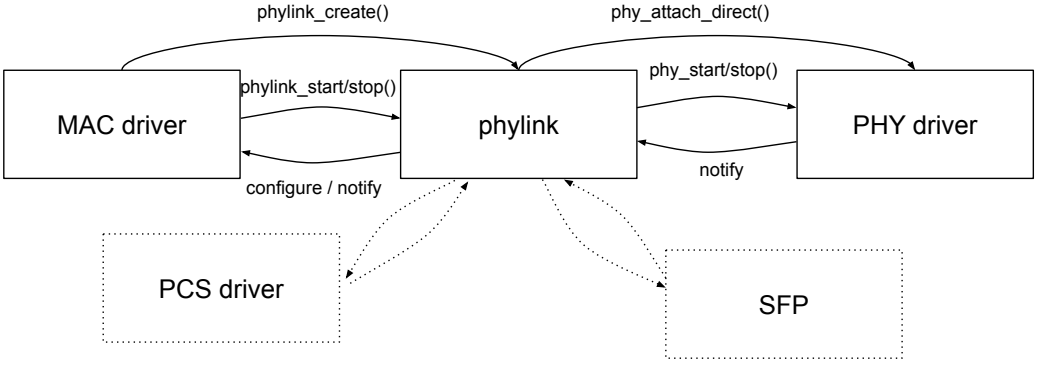
- **phylib** provides an simple API for PHY consumers
  - `phy_start()`, `phy_stop()`, `phy_connect()`
  - `phy_suspend()`, `phy_resume()` for power management
- MAC drivers may use that API, however it has some limitations :
  - It can't handle MII reconfiguration
  - It introduces some layering violations : MAC driver access `phydev->*` fields
  - It makes it difficult to support other Layer 1 technologies such as **SFP**
- phylink is a framework that sits between MAC drivers and PHY drivers
- It abstracts away the PHY from the MAC, and provides a feature-full set of callbacks for MAC configuration
- It also handles PCS configuration

- MAC drivers that use **phylib** directly call `phy_connect()` to link with the PHY
  - Their `.ndo_open()` calls `phy_start()` to establish the link
  - Their `.ndo_close()` calls `phy_stop()` to quiesce it
- They register an `.adjust_link` callback to the `phydev` for link change notification



phylink

- The phylink framework abstracts the Layer 1 configuration away
- MAC driver can transparently connect to a PHY or an SFP module
- Handles MII reconfiguration, PCS configuration, ethtool reconfiguration
- Doesn't superseeds phylib, but complements it for the MAC API



phylink - MAC ops

- MAC driver populate a set of callbacks in `struct phylink_mac_ops` registered to phylink
- `.mac_config` : Reconfigure the **MII** mode and parameters, **major reconfig**
- `.mac_link_up` : Notify that the link with the partner is established
  - Negociated speed, duplex and flow control are passed
  - The MAC should re-adjust its settings, if possible without bringing the link down
- `.mac_link_down` : Notify that the link partner is gone
- `.mac_select_pcs` : The MAC returns which `struct phylink_pcs` must be used
  - MACs may have multiple PCS, chosen based on the MII
- `.mac_enable/disable_tx_lpi` : Configures the **Low Power Idle** modes, for **E**nergy **E**fficient **E**thernet

phylink - MAC capabilities

- When creating the `struct phylink` instance, the MAC indicates its capabilities
- This is done by populating a `struct phylink_config` object
- `mac_capabilities` : indicates all Speeds and Duplex settings supported
- `supported_interfaces` : indicates all MII interfaces this MAC can output

phylink config example

```
phylink_config.mac_capabilities = MAC_ASYM_PAUSE | MAC_SYM_PAUSE | MAC_10 |
                                  MAC_100 | MAC_1000FD | MAC_2500FD;
phy_interface_set_rgmii(phylink_config.supported_interfaces);
__set_bit(PHY_INTERFACE_MODE_MII, phylink_config.supported_interfaces);
__set_bit(PHY_INTERFACE_MODE_GMII, phylink_config.supported_interfaces);
__set_bit(PHY_INTERFACE_MODE_SGMII, phylink_config.supported_interfaces);
__set_bit(PHY_INTERFACE_MODE_1000BASEX, phylink_config.supported_interfaces);
__set_bit(PHY_INTERFACE_MODE_2500BASEX, phylink_config.supported_interfaces);
```

SFP

- **S**mall **F**ormfactor **P**luggable is defined by the SFF standards
- It allows having a hot-pluggable **module** that deals with the Media side
- Useful for **Fibre** links, but also exists in Copper flavours (BaseT or DAC)
- Each module has a standardized behaviour and interface :
  - An **i2c** bus and some GPIOs are used to control the module
  - An **eeprom** is accessible on the **i2c** bus, at address `0x50`
  - Its content is standardized, indicating the capabilities, vendor, model, etc.
  - Some modules also provide Diagnostics and Montoring over i2c : Temperature, Power output, etc.

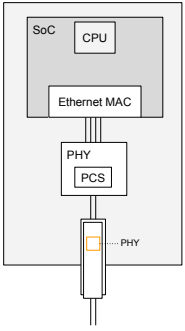
SFP

- The internals of an SFP module are a black box, but some modules may have a PHY within
- The PHY may be accessed over the i2c bus, but not always
- If accessible, the embedded PHY can be managed by the kernel
- If the SoC can't output a **serialized** interface, a **media converter** can be used

Ethtool reporting

- Userspace can retrieve information reported by the PHY drivers through `ethtool`
- Contrary to `struct net_device`, `struct phy_driver`s don't implement `ethtool ops`
- `phylib` implements the `struct ethtool_phy_ops`, and calls into `struct phy_driver`
- The netdev `ethtool_ksettings_get` and `ethtool_ksettings_set` have PHY-centric implementation :
  - They report the current link settings : Speed, duplex, linkmodes
  - Also report Link-partner information : The **advertised** linkmodes
  - See `phylink_ethtool_ksettings_set()` and `phy_ethtool_ksettings_set()`

PHY reporting with Netlink

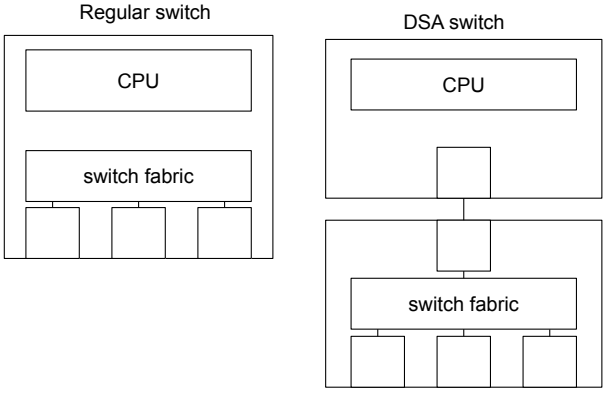


- Some hardware topologies may have **more than one PHY** attached to a MAC
  - When an SFP module is driven by a PHY, and contains a PHY itself
  - When a PHY is used as a **media converter**
- Netlink requests targetting PHY devices can now be passed a **phy index**
  - Implemented by `drivers/net/phy/phy_link_topology.c`

Switch drivers

Ethernet switches in Linux

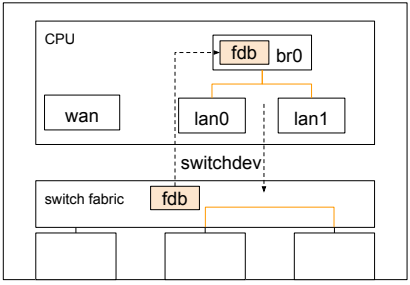
- The **switchdev** framework allows configuring the **switching fabric**
- Switch ports are represented as regular `net_device`



Ethernet switches in Linux

- By default, without any further configuration, each port is **independent**
- As each port has their `net_device`, they have a their own `net_device_ops`
- Non-DSA switches are just regular Ethernet drivers, with extra logic for switchdev
- DSA switches have their ports handled by the DSA port infrastructure
  - Implemts the `.ndo_start_xmit`

Switchdev



- **switchdev** is a framework allowing drivers to implement switching configuration ops
- Bridging, VLANs, filtering, queueing, redirection, snooping, etc.
- The hardware **must** be able to report internal reconfiguration events

Notifiers

- Drivers don't implement any kind of `switchdev_ops`
  - Switch-related events don't specifically target a single `netdev`
- Drivers instead subscribe to **kernel notifications** through `notifiers`
- Userspace bridging configuration triggers reconfiguration events
  - `NETDEV_CHANGEUPPER` : A `netdev` has a new `upper_dev`
  - `BR_STATE_FORWARDING` : A `bridge port` is set in forwarding state
- The switch reports internal events, that the driver notifies to the kernel
  - `call_switchdev_notifiers()`

Switchdev notifiers - example

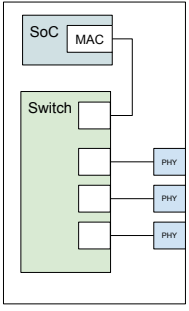
switchdev example

```
static int adin1110_switchdev_event(struct notifier_block *unused,
                                    unsigned long event, void *ptr)
{
    if (!adin1110_port_dev_check(netdev))
        return NOTIFY_DONE;

    switch (event) {
    case SWITCHDEV_FDB_ADD_TO_DEVICE:
    case SWITCHDEV_FDB_DEL_TO_DEVICE:
        /* Add item to FDB */
    }

    return NOTIFY_DONE;
}

static struct notifier_block adin1110_switchdev_notifier = {
    .notifier_call = adin1110_switchdev_event,
};

static int adin1110_setup_notifiers(void)
{
    register_switchdev_notifier(&adin1110_switchdev_notifier);
}
```

Switchdev notifications

- `NETDEV_CHANGEUPPER` : A `netdev` was added to or removed from a bridge
- `SWITCHED_FDB_ADD_TO_DEVICE` : A FDB entry was added by user
- `SWITCHED_PORT_OBJ_ADD` : Generic notifier to add an entry to a port
  - `SWITCHDEV_OBJ_ID_PORT_VLAN` : A port belongs to a VLAN
  - `SWITCHDEV_OBJ_ID_PORT_MDB` : Add a Multicast address to a port
- Drivers notify the kernel when the operation was able to be offloaded
  - *e.g.* `call_switchdev_notifiers(SWITCHDEV_FDB_OFFLOADED, ndev, &info.info, NULL);`

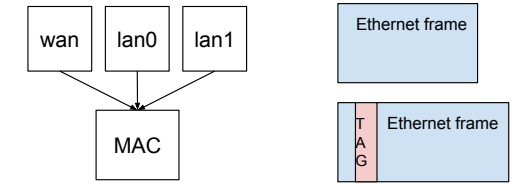
DSA

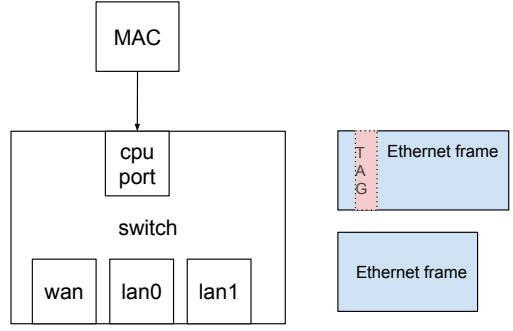


Distributed **S**witch **A**rchitecture

- Mainly used by dedicated switch chips
- One or more ports are connected to SoC interfaces
- DSA switches may be chained together
- The CPU to Switch link is called the **cpu conduit** or **cpu port**
- Switch to Switch links are called **dsa conduits**
- Other interfaces are called **user** ports
- Frames on **conduits** are often **tagged** to identify the destination port
- DSA **uses** switchdev, and does not replace it

DSA tagging





- A vendor-specific TAG is added to the frame
- It contains the identifier of the egress port
- The frame is sent from the CPU to the switch
- The switch strips the tag and sends it on the port
- The opposite happens on receive

DSA in devicetree

```
&mdio {
  switch0: ethernet-switch@1 {
    compatible = "marvell,mv88e6085";
    reg = <1>;

    dsa,member = <0 0>;

    ethernet-ports {
      #address-cells = <1>;
      #size-cells = <0>;
      [...]
    };
  };
};
```

- **reg** : Address on the MDIO bus
- **dsa,member** : Position in the **cluster**, if applicable
- **ethernet-ports** : Contains the list of ports

DSA Ports in devicetree

```
    ...
    ethernet-ports {
      switch0port0: ethernet-port@0 {
        reg = <0>;
        label = "cpu";
        ethernet = <&eth0>;
        phy-mode = "rgmii-id";
        fixed-link {
          speed = <1000>;
          full-duplex;
        };
      };

      switch0port1: ethernet-port@1 {
        reg = <1>;
        label = "wan";
        phy-handle = <&switch0phy0>;
      };

      switch0port2: ethernet-port@2 {
        reg = <2>;
^^Ilink = <&switch1port0>;
      };

      ...
    };
    ...
```

- **reg** : Port number
- **label** : Port name, will become the interface name
- **ethernet** : phandle to the CPU-side MAC interface
- **link** : phandle to another DSA switch's port, for cascading
- PHY mode and phandle

Tagging

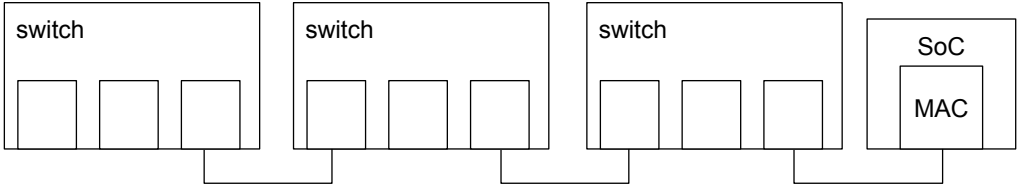
- Tagging happens outside of the switch driver, in a dedicated tagger : `net/dsa/tag_*.c`
- Some switches support multiple tagging formats
  - It can be specified in **devicetree**

DSA tagger

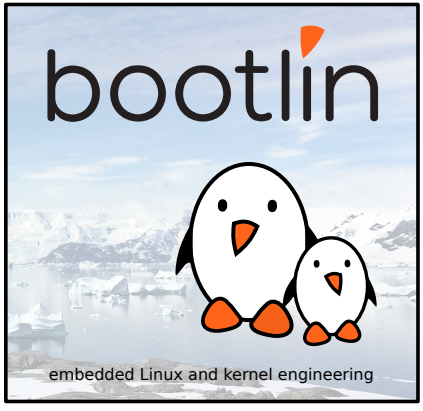
```
static const struct dsa_device_ops foo_ops = {
    .name = "foo",
    .proto = DSA_TAG_PROTO_FOO,
    .xmit = foo_tag_xmit,
    .rcv = foo_tag_rcv,
    .needed_headroom = FOO_HDR_LEN,
    .promisc_on_conduit = true,
};
```

Chaining

- Some DSA switches can be daisy-chained
- The ports that link switches together don't have associated `net_device`



Debugging and tracing the Network Stack



Challenges

- Latency issues : Can come from different locations
  - The network itself
  - Internal queueing and buffering
  - Hardware and OS-level latencies
- Throughput issues
  - May depend on the traffic type
  - May simply be a symptom : TCP retransmissions due to bad L1 link quality
- Link issues
  - May be hardware related
  - The kernel can't only tell you what it knows

Monitoring traffic and drops

- In case of large number of repeats or local drops
- dropwatch : monitor the in-kernel packet drops, see `man 1 dropwatch`

```
$ dropwatch -lkas
$ dropwatch> start
2 drops at ip6_mc_input+1a8 (0xffffffff83347ba8) [software]
```

- retis : eBPF based monitoring. monitor drops as well as `skb` lifetime
- See the official documentation

```
retis collect -c skb-drop --stack
```

Monitoring traffic with a capture

- Tools such as wireshark and tcpdump use `AF_PACKET` sockets for monitoring
- Some hardware devices may include extra information
  - *e.g.* the **radiotap** headers on 802.11 frames
- The monitoring happens between the **driver** and `tc`
- On the receive side, it happends before firewalling with netfilter
- Capture format is standardised with the `pcap` format
  - frames can be replayed, or analysed on another host

Offloading

- Offloading issues are hard to troubleshoot, as the host doesn't see them
- `wireshark` may tell you **receive checksumming** issues
- `ethtool -k eth0` shows you the **features**
- Hardware counters should be used for debugging :
  - `ethtool -S eth0`
  - `ethtool --phy-statistics eth0`
  - `ethtool -S eth0 --groups eth-mac|eth-phy|eth-ctrl|rmon`
- Some information may be available in **debugfs**
- `ip -s link show eth0` shows software counters
- `cat /proc/interrupts` indicate the hardware interrupt counters
- `cat /proc/softirq` indicate the softirq counters

Traffic generation

- iperf3 : Troughput testing, fairly simple to use
- netperf : Made by kernel developers, similar to iperf3, more featureful
- scapy : Traffic generator written in python, craft arbitrary frames
- DPDK's pktgen
  - PKTgen uses **kernel bypass**, not supported on every platform
  - Allows very fast packet crafting (10gbs)
  - Useful to test multi-flow setups, or HW offloading

iperf3

- Widely used traffic generator
- `iperf3 -s -D` : Start in server mode
- `iperf3 -c 192.168.1.1` : Start in client mode, default is TCP
- `iperf3 -c 192.168.1.1 -u -b 0` : UDP mode, with unlimited bandwidth
- `iperf3 -c 192.168.1.1 -u -b 0 -l 100` : UDP mode, small packets
- `iperf3 -c 192.168.1.1 -P 16` : Multi-flow mode

scapy

- Traffic generator written in python. See the official website
- Allows generating arbitrary traffic very easily
- Each header can be crafted, for high flexibility
- Very easily scriptable

```
# IPv4 with ToS field varying between 1 and 4
sendp(Ether()/IP(dst="1.2.3.4",tos=(1,4)), iface="eth0")

# Raw ethernet frame
sendp(Ether(dst="00:51:82:11:22:02"), iface="eth0")

# Send and wait for reply, simple ping implementation
packet = IP(dst="192.168.42.1", ttl=20)/ICMP()
```

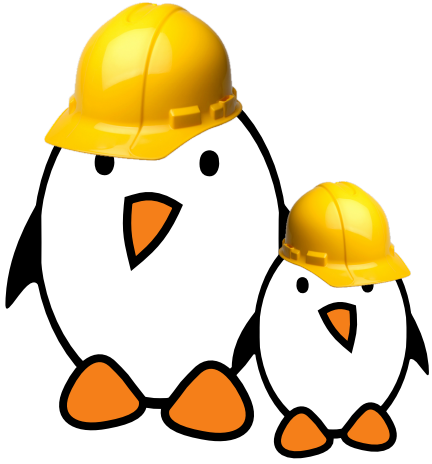
Counters

- Layer 4 counters : Maintained by the kernel.
  - `netstat -s` or `cat /proc/net/netstat`
  - More statstics in `/proc/net/stat`
- Layer 3 counters : Provided by `ip -s link show`
- Layer 2 counters : Hardware-provided
- XDP programs can be custom-written to gather specific statistics
- `xdp-monitor -s` tracks XDP statistics such as the number of drops and redirects

profiling

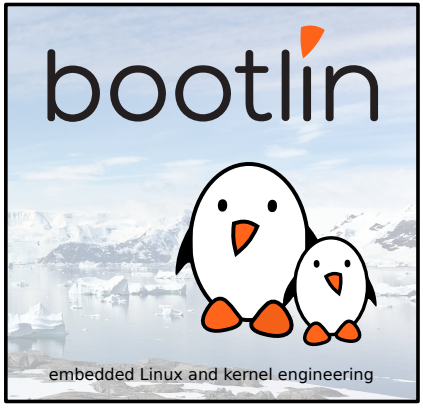
- Allows identifying the bottlnecks in software
- `perf` can be used : Rely on hardware and software counters
- `Flamegraphs` help identify software bottlenecks, in kernel and userspace
  - Covered in our debugging training
- `ftrace` will generate timelines of events, to track **latencies**
- See the ftrace documentation

Practical lab - Debugging performance issues



- Troubleshoot a performance issue
- Analyze traffic with tcpdump and wireshark
- Troubleshoot a link issue

Last slides



Thank you!

And may the Source be with you

Rights to copy

Document sources: https://github.com/bootlin/training-materials/