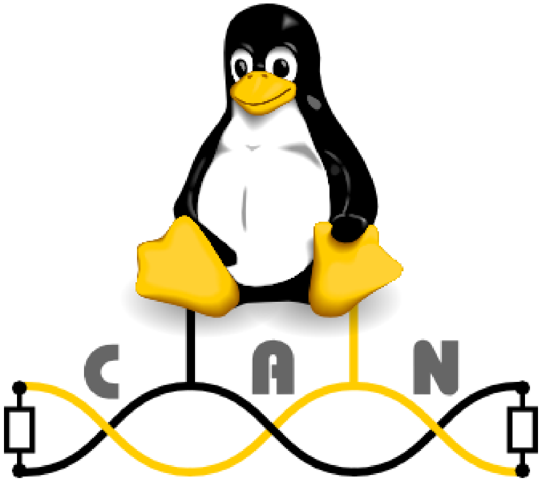
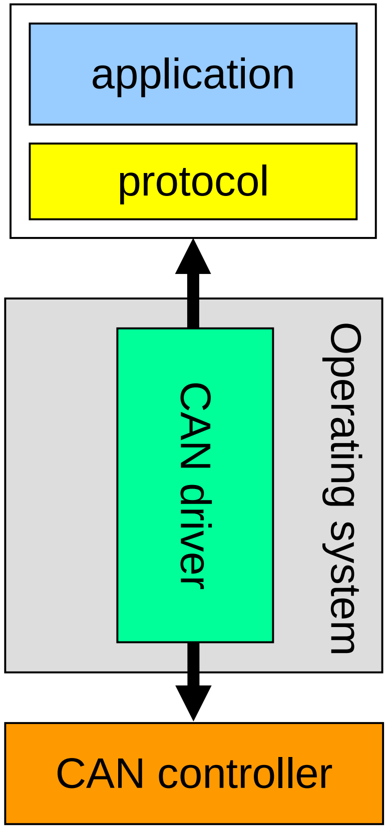


Design & separation of CAN applications

Adopting Un*x rules and network namespaces

Presentation for Automotive Grade Linux F2F, 2018-04-12, Microchip (Karlsruhe)

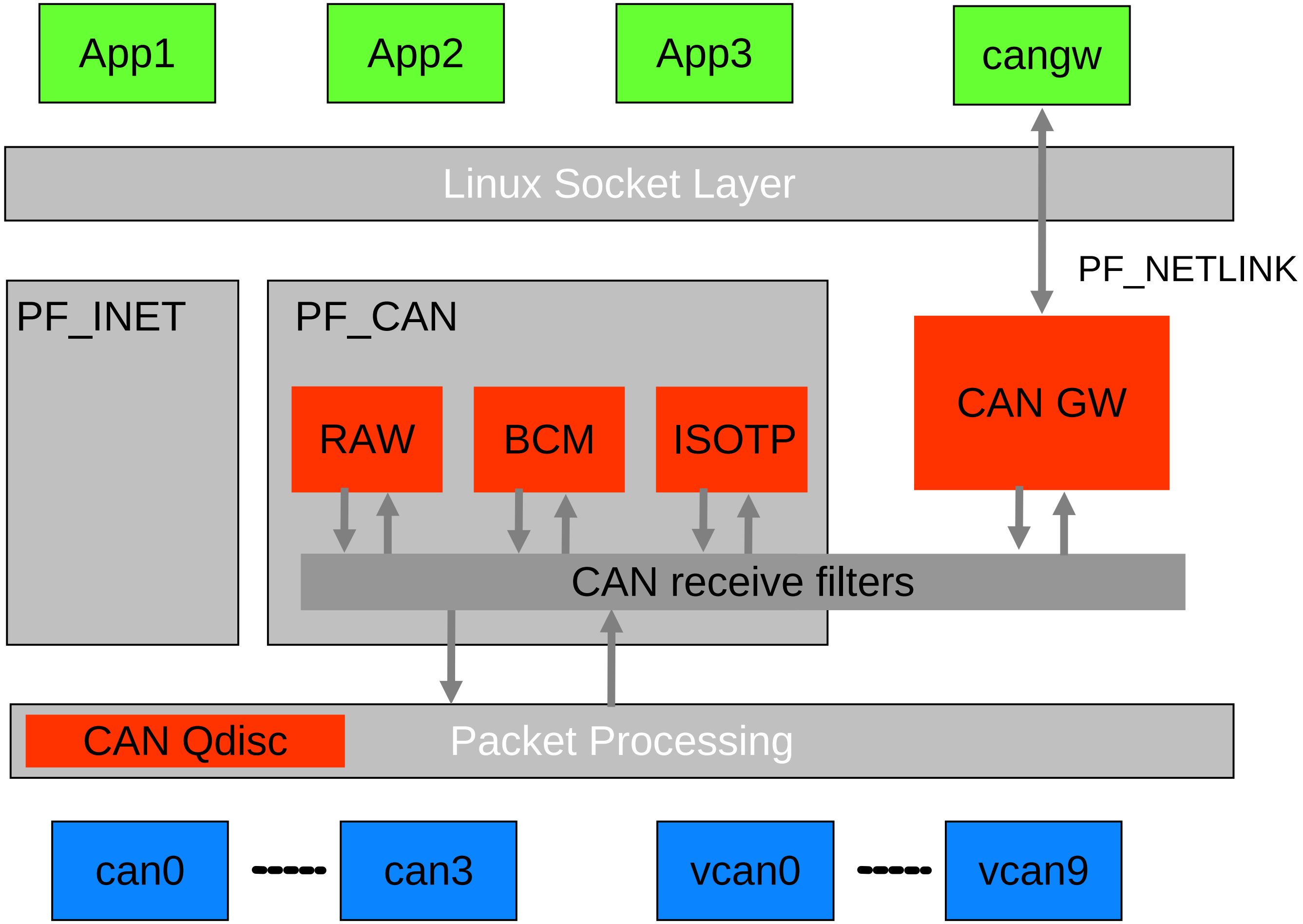
The former concepts for CAN access – recap from 2017 slides*

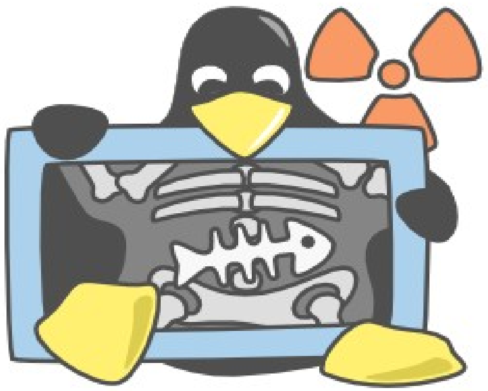


- Only one application can use the CAN bus at a time
- There was no standard Linux CAN driver model
  - Every CAN hardware vendor sells his own driver bundled to his CAN hardware
- CAN application protocols and intelligent content filters need to be implemented in userspace
- **People still think in this out-dated design pattern! :-(**

* https://wiki.automotivelinux.org/_media/agl-distro/agl2017-socketcan-print.pdf

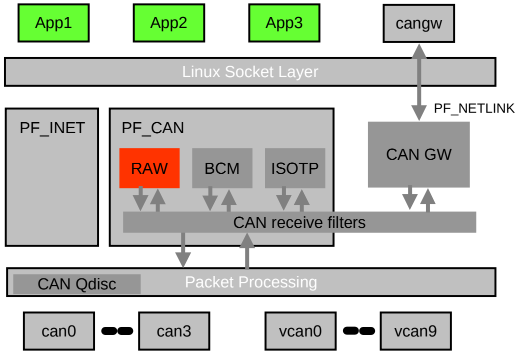
CAN network layer protocols and frame processing (recap)





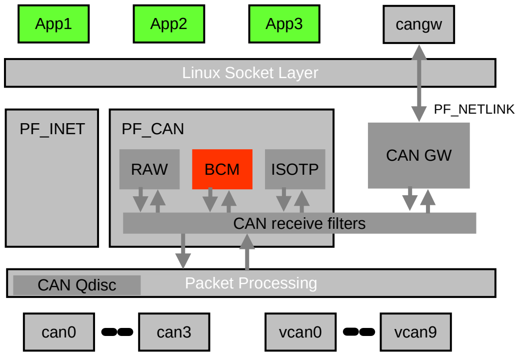
CAN_RAW – Reading and writing of raw CAN frames (recap)

- Similar to known programming interfaces
  - **A socket feels like a private CAN interface**
  - **per-socket CAN identifier receive filtersets**
  - Linux timestamps in nano second resolution
  - Easy migration of existing CAN software



- Multiple applications can run independently
  - **Network transparency through local echo of sent frames**
  - **Functions can (should!) be split into different processes**

CAN_BCM – timer support and filters for cyclic messages

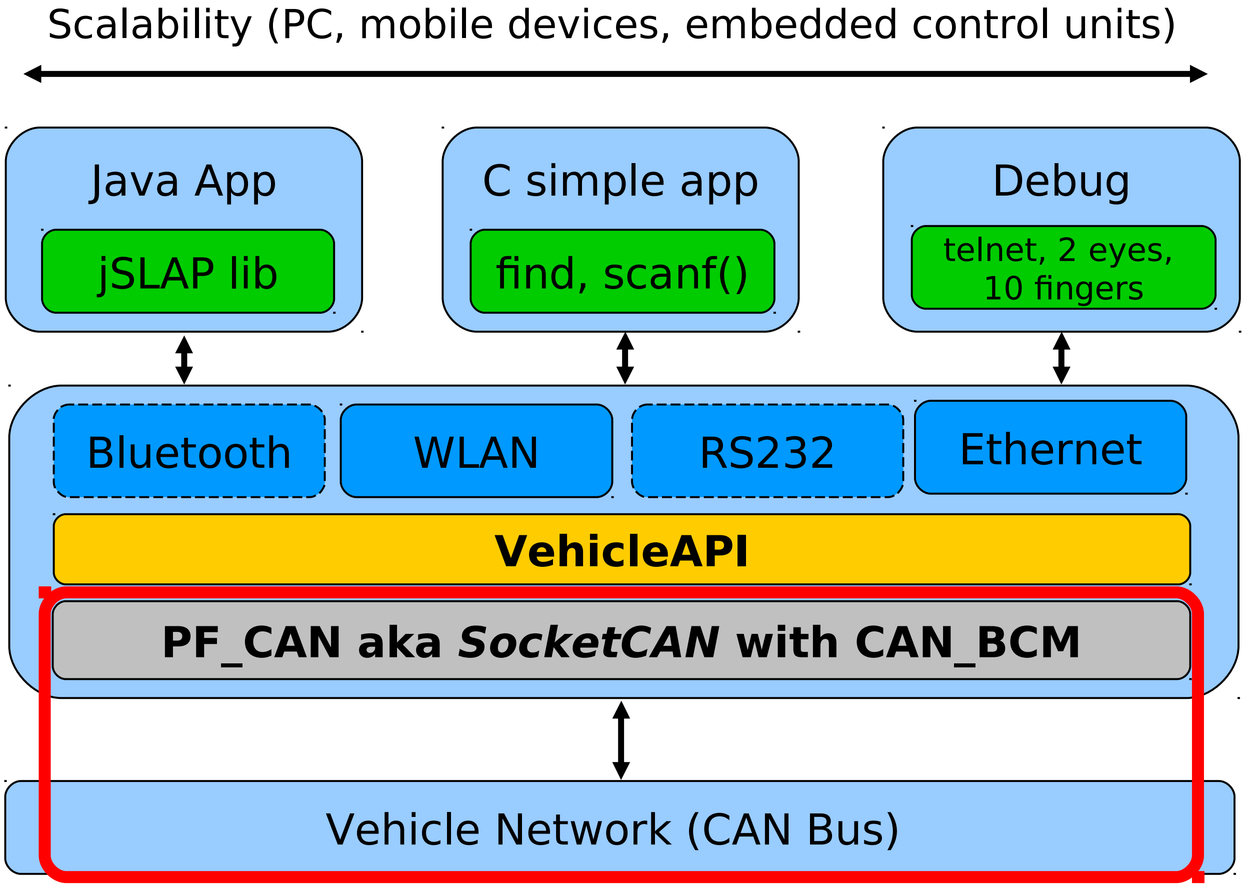


- Executes in operating system context
- **Programmable by BCM socket commands**

- CAN receive path functions
  - **Filter bit-wise content in CAN frame payload**
  - Throttle update rate for changed received data
  - Detect timeouts of cyclic messages (deadline monitoring)

- CAN transmit path functions
  - **Autonomous timer based sending of CAN frames**
  - Multiplex CAN messages and instant data updates

CAN_BCM – Vehicle data access prototyping technology



Scalability (PC, mobile devices, embedded control units)

Java App

jSLAP lib

C simple app

find, scanf()

Debug

telnet, 2 eyes, 10 fingers

Bluetooth

WLAN

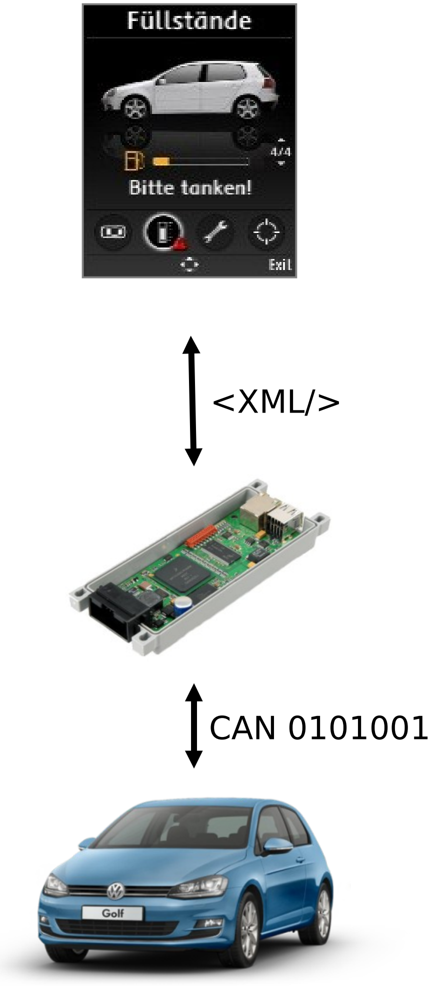
RS232

Ethernet

VehicleAPI

PF_CAN aka *SocketCAN* with CAN_BCM

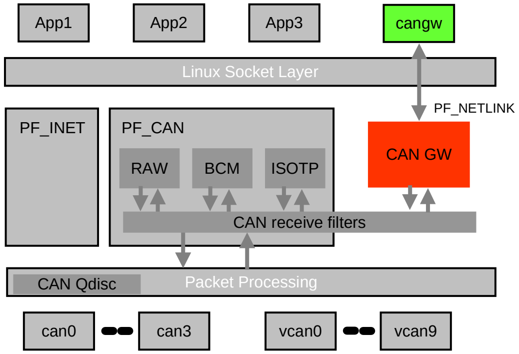
Vehicle Network (CAN Bus)



<XML/>

CAN 0101001

CAN_GW – Linux kernel based CAN frame routing (recap)



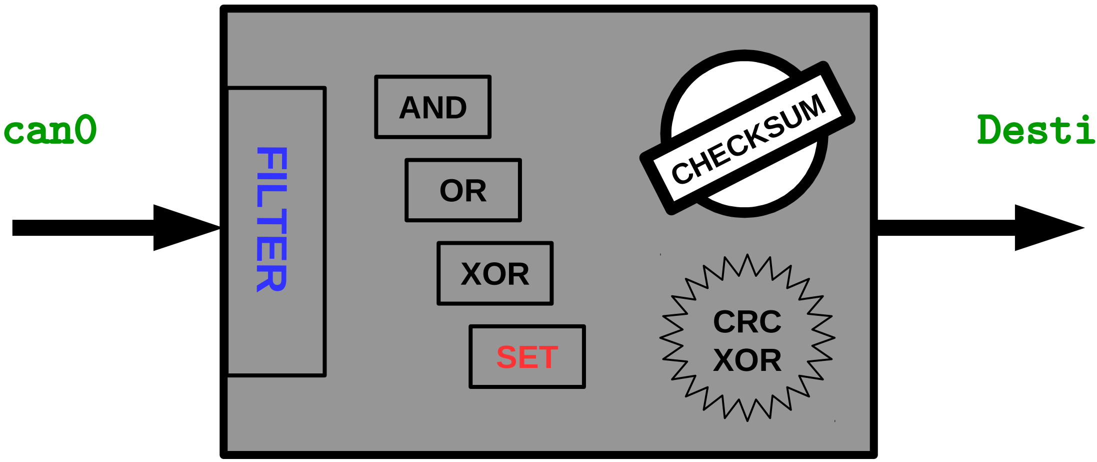
- **Efficient CAN frame routing in OS context**
- Re-use of Linux networking technology
  - **PF_CAN receive filter capabilities**
  - Linux packet processing NET_RX softirq
  - PF_NETLINK based configuration interface
    (known from Linux network routing configuration like 'iptables')
- Optional CAN frame modifications on the fly
  - **Modify CAN identifier, data length code,** payload data with AND/OR/XOR/SET operations
  - Calculate XOR and CRC8 checksums after modification
  - Support of different CRC8 profiles (1U8, 16U8, SFFID_XOR)

CAN_GW – Routing & modification configuration entity

Routing & modification element

Source device: can0

Original content



FILTER

AND

OR

XOR

SET

CHECKSUM

CRC
XOR

Destination device: can1

Modified content

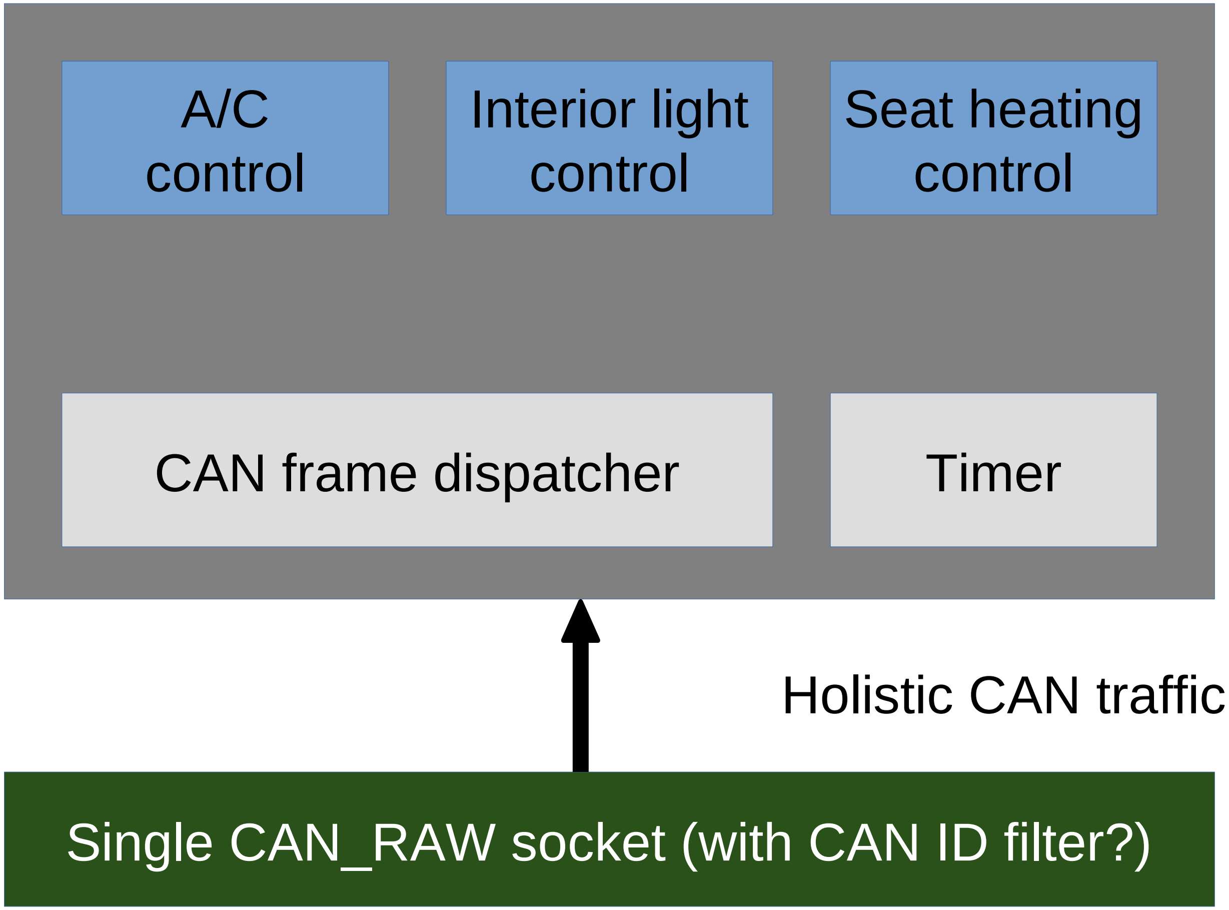
```
cangw -A -s can0 -d can1 -e -f 123:C00007FF -m SET:IL:333.4.1122334455667788
```

Some best practices on design patterns and separation

- Write programs that do **one** thing and do it well.
- … if you don’t trust a CAN application
- … if you *really* don’t trust a CAN application
- … if you *only* trust your CAN application
- Btw. why wouldn’t you trust an Open Source CAN application?

Write programs that do one thing and do it well.

(https://en.wikipedia.org/wiki/Unix_philosophy)



No!

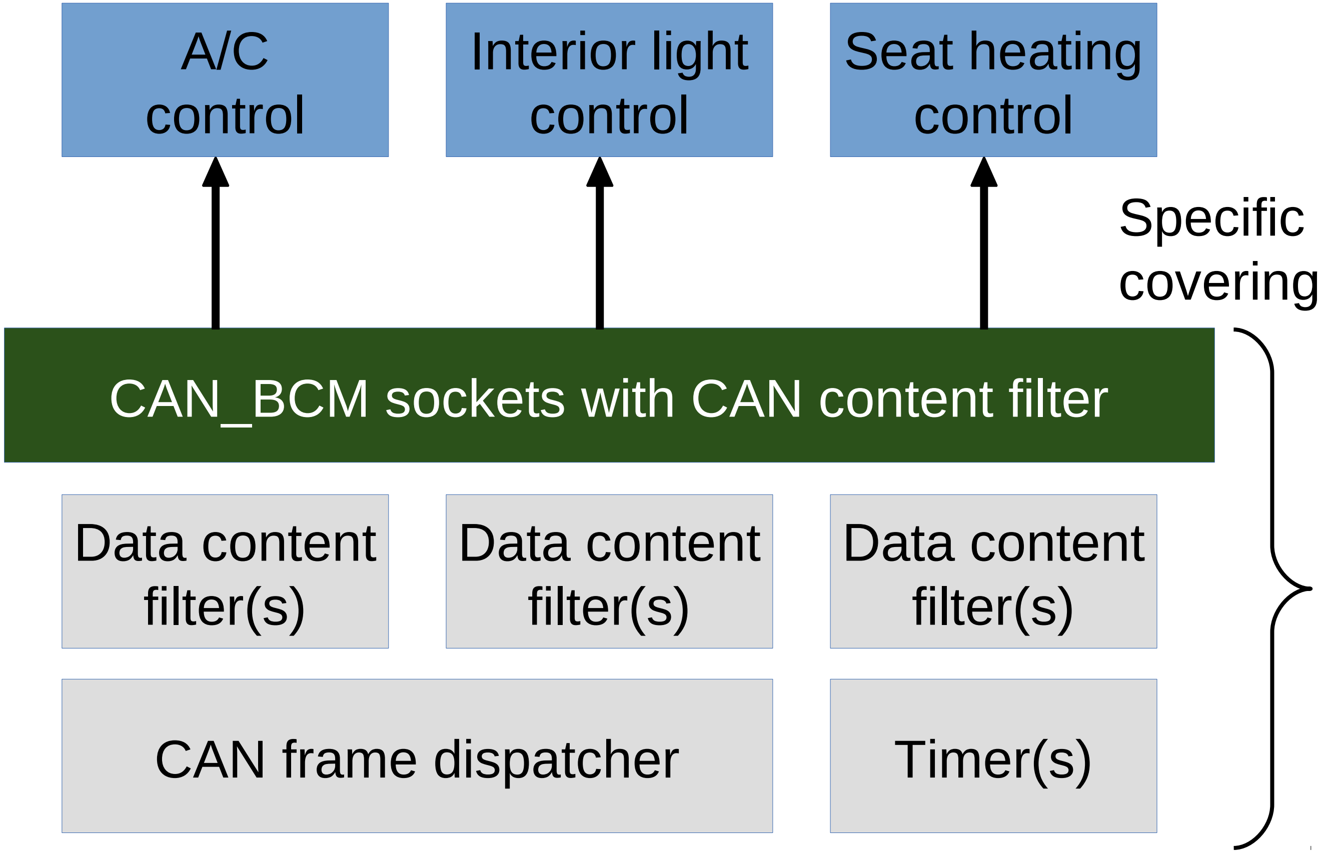
Monolithic application

Holistic CAN traffic covering all use-cases

Write programs that do one thing and do it well.

(https://en.wikipedia.org/wiki/Unix_philosophy)

Yes!



Specific CAN traffic covering **one** use-case

Different BCM sockets (instances of multiple data filters & timers)

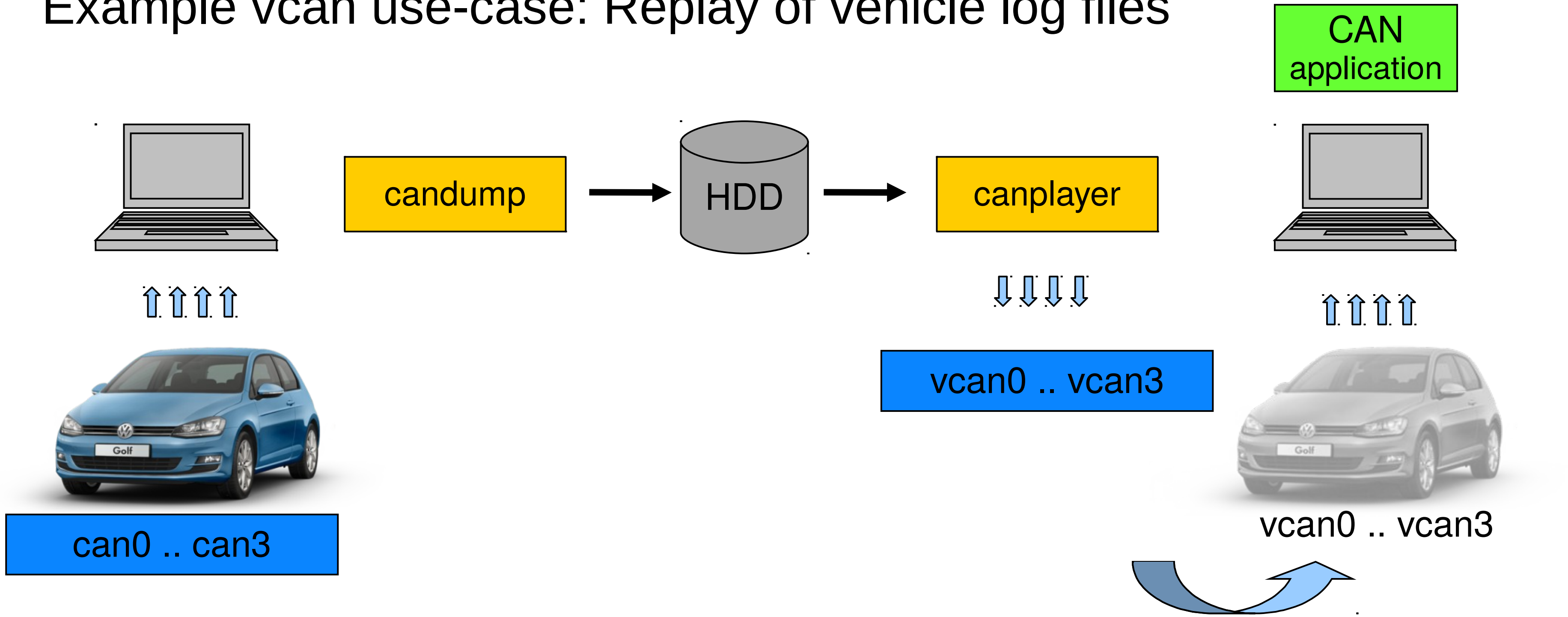
- **Separation, maintainability, minimized code/complexity/dependency, etc.**

… if you don’t trust a CAN application

- Give the application a dedicated virtual CAN bus
- Make use of CAN_GW to forward just the needed traffic

Virtual CAN network device driver (vcan) – recap from 2017

- No need for real CAN hardware
- Local echo of sent CAN frames ‘loopback device’
- **vcan instances can be created at run-time**
- Example vcan use-case: Replay of vehicle log files



How to create and name a virtual CAN network device

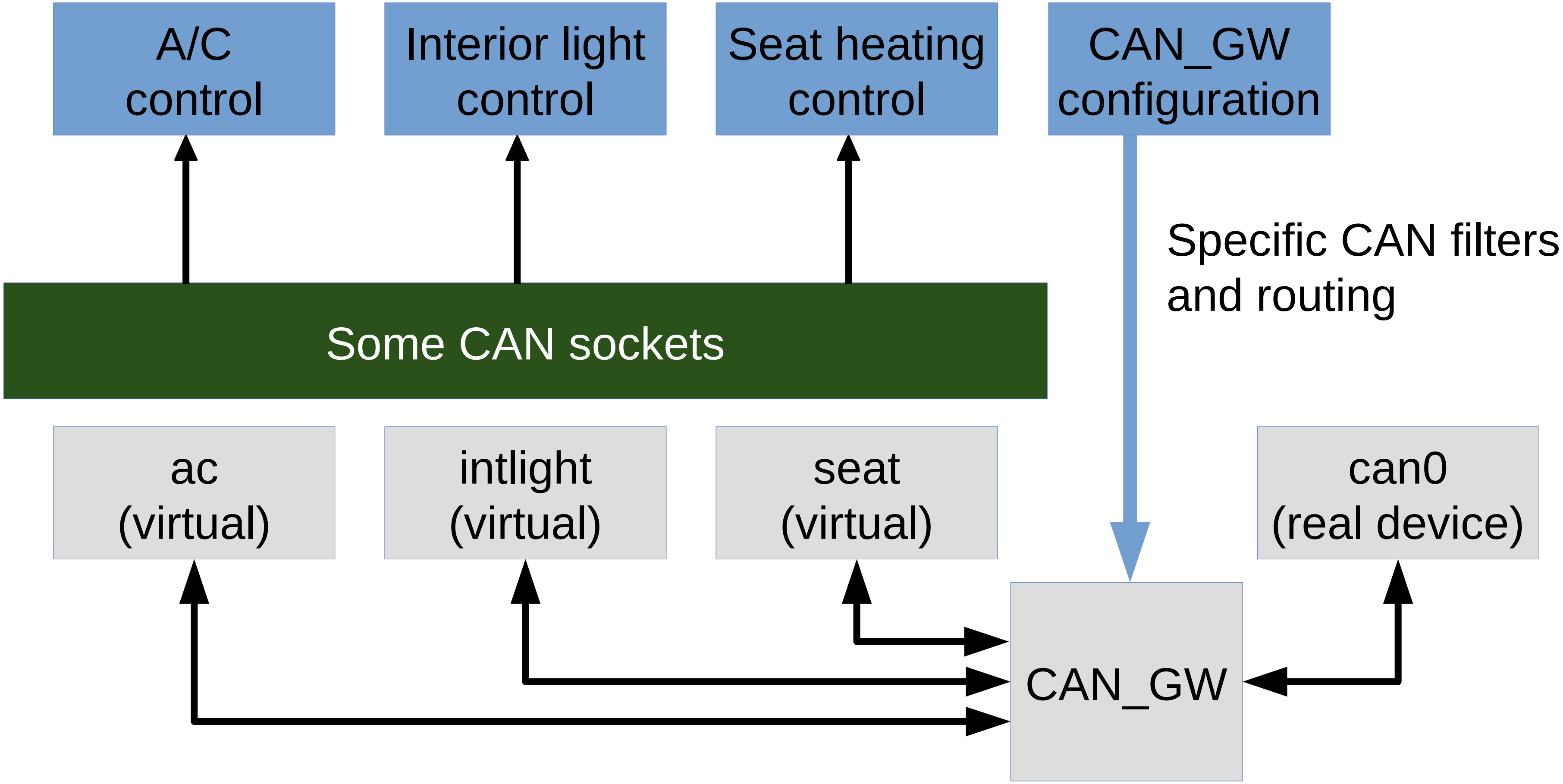
- Loading the virtual CAN driver into the Linux kernel

```
sudo modprobe vcan
```

- Create virtual CAN interfaces

```
sudo ip link add type vcan
sudo ip link add dev helga type vcan
sudo ip link set vcan0 up
sudo ip link set helga up
```

Dedicated virtual CAN interfaces for each application



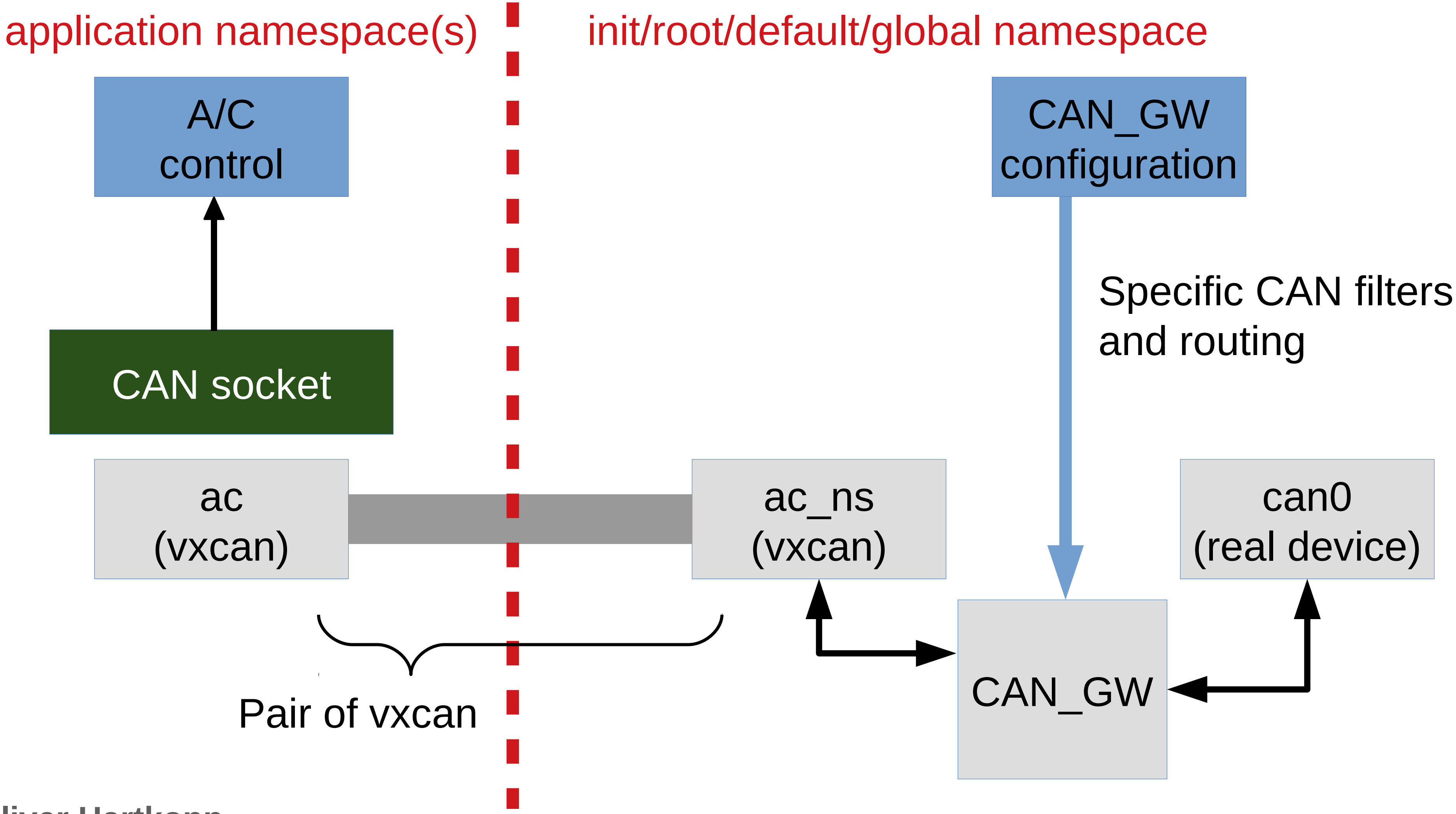
… if you don’t trust a CAN application

- Give the application a dedicated virtual CAN bus
- Make use of CAN_GW to forward just the needed traffic
- But still the application might access the ‘real CAN device’ can0
- This is not really a separation but helps with testing and may cover unintended (erroneous) sending on wrong CAN identifiers
- Maybe other Linux security measures (e.g. SELinux) can also help in this case?!? Did not check so far ...

… if you *really* don't trust a CAN application

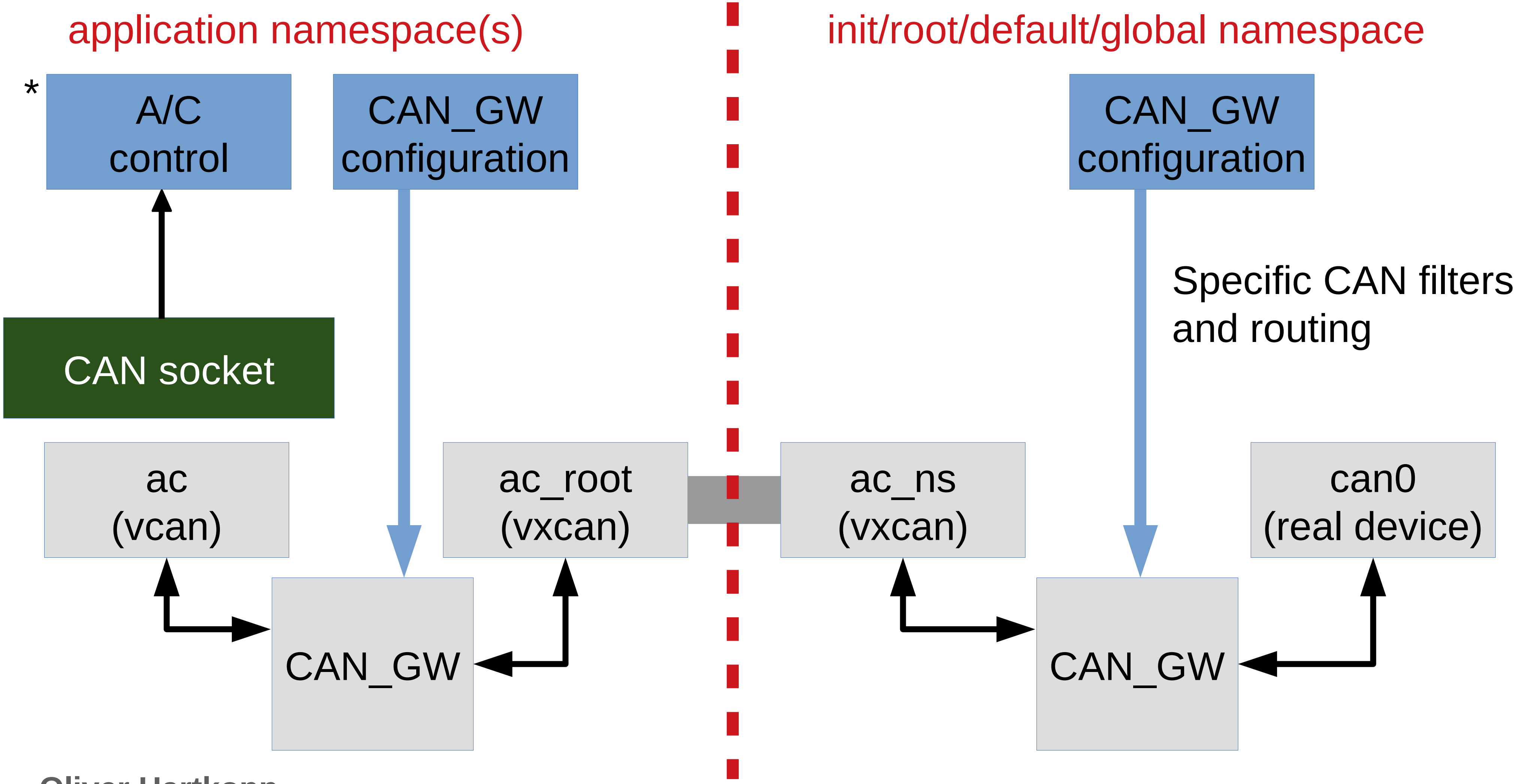
- Since Linux 4.12 the CAN subsystem supports network namespaces
- Net namespaces are required for LXC, Docker, etc.
- You can now deploy your specific containers with CAN functionality
- To connect different containers (in different network namespaces) the **veth** driver can create **a pair of** virtual ethernet devices that establish some kind of ethernet patch cable between containers
- Since Linux 4.12 a new **vxcan** driver can connect different namespaces in a similar way. The vxcan instances do not have IP addresses and only can transfer CAN frames like vcan devices.
- N.B. vxcan's do not provide the local IFF_ECHO feature!
- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit?id=a8f820a380a2a06fc4fe1a54159067958f800929

Dedicated VXCAN interface for each application in namespace



VXCAN interfaces just forward; without local echo (IFF_ECHO)!

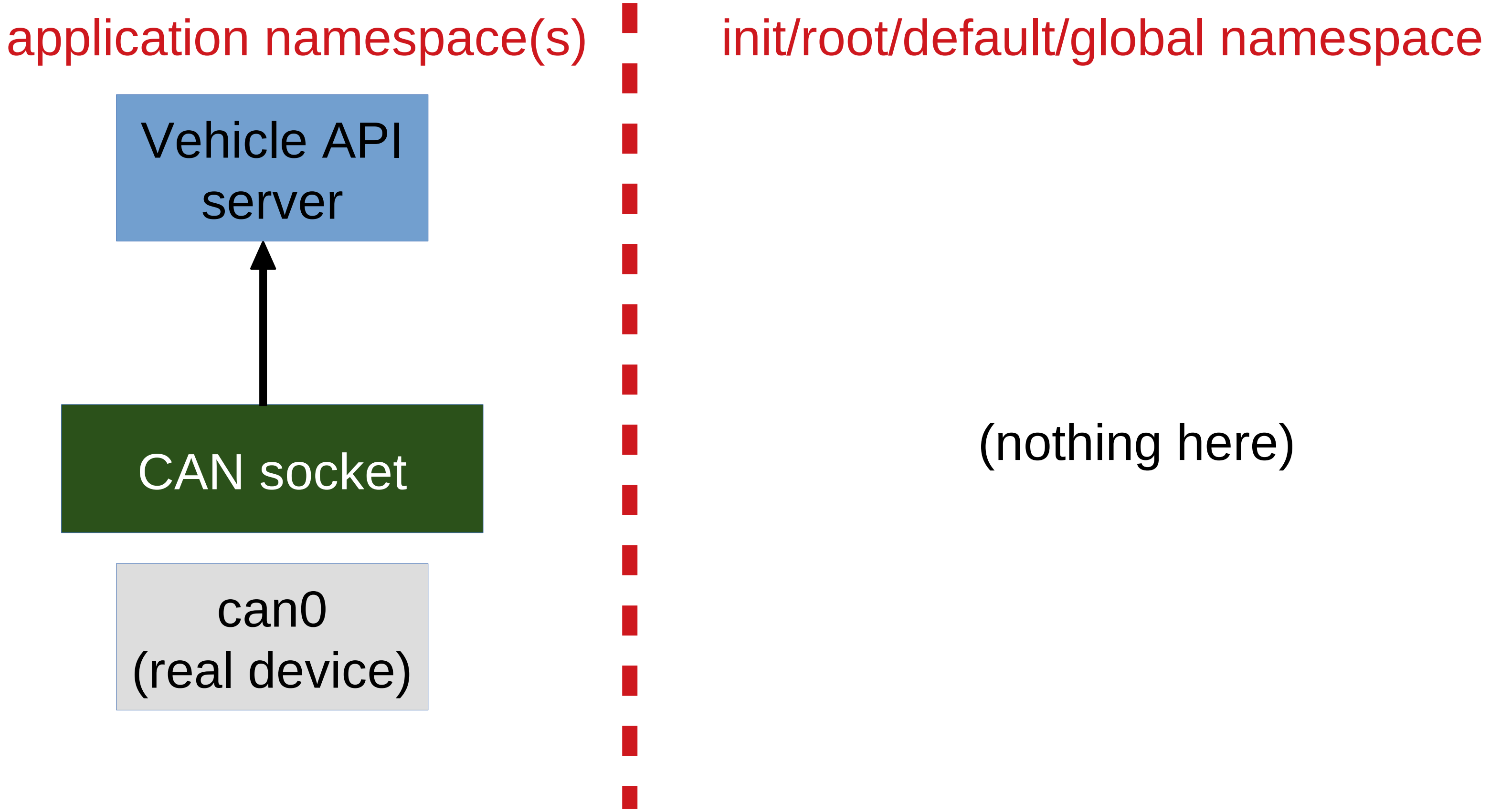
To support multiple* applications in a namespace use **vcan** via CAN_GW there



… if you *only* trust your CAN application

- Move the real(!) CAN interface into the namespace where **only** your trusted application(s) can access the CAN bus
- The real CAN interface is **not accessible** in the default namespace anymore
- Can make sense when you have a single container managing the vehicle interfaces or vehicle abtraction services

The real(!) CAN interface is moved into the namespace



- **Excellent setup to run a Vehicle API which provides abstract data objects through a TCP/IP service to different namespaces via veth/IP**

Btw. why wouldn’t you trust an Open Source CAN application?

- Separation via CAN_GW and network namespaces is fun and enables the setup and distribution of easy-to-use containers
- Btw. the best approach is still having a proper design (‘do **one** thing and do it well’) with minimized code using all of the fancy functionality that SocketCAN provides out-of-the-box and transparency/use/testing through the Open Source community

- Some references to namespace documentations:
- https://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/
- https://blogs.igalia.com/dpino/2016/04/10/network-namespaces/
- http://www.opencloudblog.com/?p=66
- https://marc.info/?l=linux-can&m=149046502301622&w=2

Many thanks!

```
$> cat linux/MAINTAINERS | grep -B 2 -A 14 Hartkopp

CAN NETWORK LAYER
M:        Oliver Hartkopp <socketcan@hartkopp.net>
M:        Marc Kleine-Budde <mkl@pengutronix.de>
L:        linux-can@vger.kernel.org
W:        https://github.com/linux-can
T:        git git://git.kernel.org/pub/scm/linux/kernel/gut/mkl/linux-can.git
T:        git git://git.kernel.org/pub/scm/linux/kernel/gut/mkl/linux-can-next.git
S:        Maintained
F:        Documentation/networking/can.rst
F:        net/can/
F:        include/linux/can/core.h
F:        include/uapi/linux/can.h
F:        include/uapi/linux/can/bcm.h
F:        include/uapi/linux/can/raw.h
F:        include/uapi/linux/can/gw.h

$> _
```