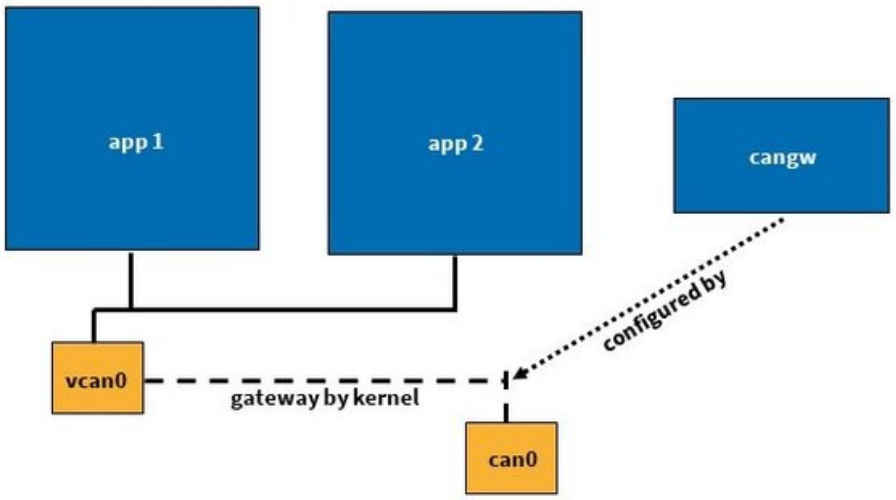
SocketCAN + Docker = The solution

Friday, 18. June 2021 | Themen



SocketCAN + Docker = The Solution

To understand all the points in this blog, a basic knowledge of Docker and SocketCAN is necessary. Basic knowledge will not be discussed in the rest of this article. The article offers a solution approach to combine SocketCAN and Docker in a stable and convenient way.

When developing highly complex embedded devices, we are often faced with the challenge of testing several CAN applications using different runtime environments in one system at the same time. Up to now, only insufficient documentation or information can be found on this very special topic. We at SYS TEC electronic have taken this as an opportunity to deal with the topic ourselves.

This article will therefore focus on finding a solution for this challenge using

SocketCAN and Docker.
Why would you want to use SocketCAN and Docker together at all? SocketCAN is a driver collection that allows the use of CAN interfaces under Linux. However, it also provides many other services in the process.

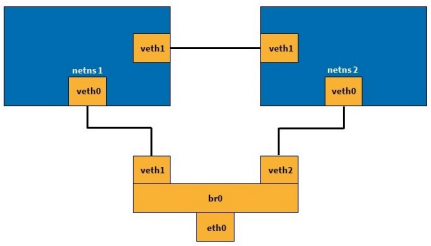
Docker, on the other hand, allows aspects of a system to be encapsulated from each other, for example to avoid unwanted cross-effects between programs or applications. It also offers a certain degree of protection for the overall system, since no unwanted access to the "outside" is possible within the Docker containers. It is also great for creating the same application environment on a variety of machines/devices. This simplifies deliveries of new and updated software.

Our goal is to install multiple Docker containers on a device, which can all be accessed via the CAN interface of the host system.

Our first consideration is to pass the interface itself directly into Docker. Typically on PCs, the CAN interface is connected via USB. Unfortunately, this solution does not work because Docker containers run purely in user space. This means the kernel is not virtualized along with it, as opposed to full hardware virtualization (typically a classic virtual machine). Furthermore, the SocketCAN drivers are not based on libusb, so there is no way to use a USB CAN interface inside the Docker container.

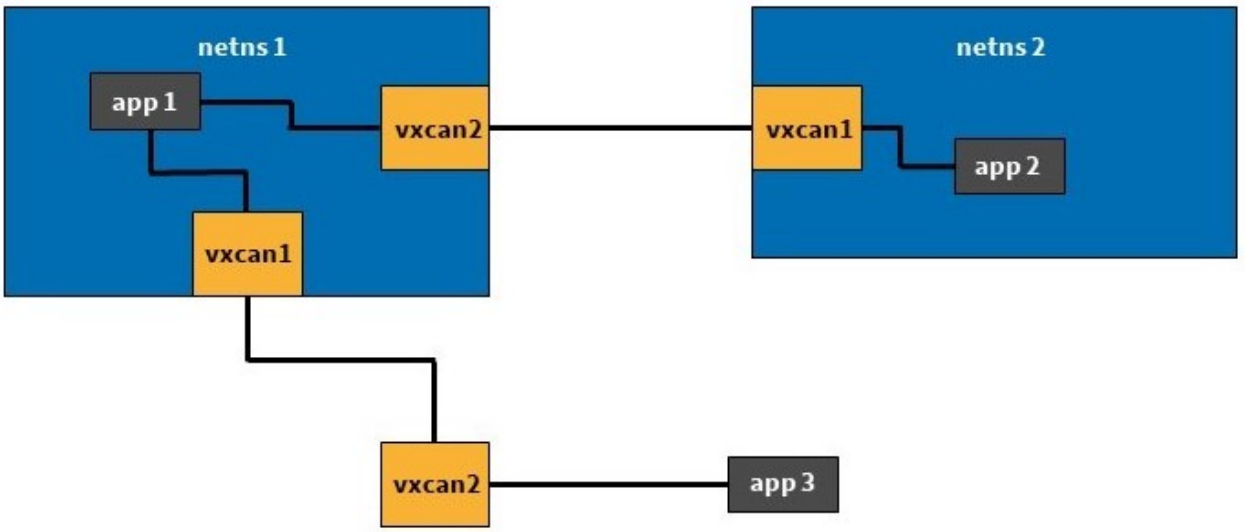
A second solution approach is to use host networking mode or privileged mode with the Docker container. This would ensure that the container can access the CAN interface of the host. However, this still results in some undesirable side effects: for this, extended (means root) privileges are needed to set up the networking. This is not a reasonable alternative, because the container isolation is lost. This means that the applications running in the container behave exactly as if they were working on the host system. However, this completely loses the key advantages of Docker.

The third and, in our opinion, best implementation option is to use virtual interfaces for the Docker containers. These are already used for TCP/IP communication between Docker containers and the host system.



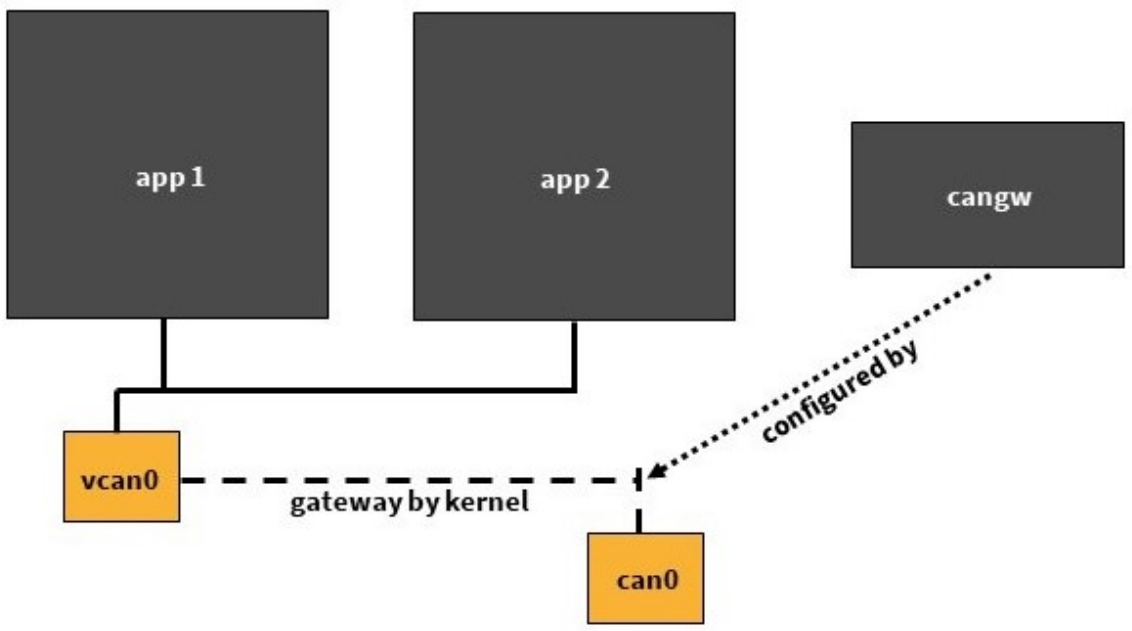
[DeepL] Figure 1: Example virtual network connection

Specifically, we use vxcan for this. This is also included in the Linux kernel. Vxcan allows to create virtual CAN interfaces and to set up tunnels between them. This makes it relatively easy to communicate across network namespaces. For example, between two or more Docker containers. The following figure visualizes this procedure. The two netns boxes represent the various network namespaces, which could be the Docker containers in question, for example. Via vxcan, they can communicate with each other and externally with a virtual interface.



[DeepL] Figure 2: Use vxcan between networks

However, this is only the first step of the solution. Now we have virtual interfaces for each Docker, but no physical connection to the CAN interface of the device. The last piece of the puzzle for this is CAN_GW. This allows physical and virtual CAN interfaces to be connected or bridged (compare with a network bridge). This ultimately allows one or more virtual CAN interfaces to be connected to the interface physically present on the device.



[DeepL] Figure 3: Tunnel with CAN_GW

This allows the CAN bus to be "connected" to the Docker containers via the interface on the device. Each Docker is assigned its own virtual CAN interface using vxcan. Using CAN_GW, the interface physically present on the device can then be connected to each virtual interface previously set up. This way, the CAN messages arriving at the device reach each individual Docker container independently.

For testing our demo application, two terminal windows are needed. As CAN interface, one of oursysWORXX USB CAN modules can be used on the PC. The necessary commands, are the following:

1. terminal

docker run --rm -it --name cantest ubuntu:20.04

apt-get update && apt-get install -y can-utils

2. terminal

DOCKERPID=$(docker inspect -f '{{ .State.Pid }}' cantest)

sudo ip link add vxcan0 type vxcan peer name vxcan1 netns $DOCKERPID

sudo modprobe can-gw

```
sudo cangw -A -s can0 -d vxcan0 -e

sudo cangw -A -s vxcan0 -d can0 -e

sudo ip link set vxcan0 up

sudo ip link set can0 type can bitrate 125000

sudo ip link set can0 up

sudo nsenter -t $DOCKERPID -n ip link set vxcan1 up
```

1. terminal

```
candump vxcan1
```

2. terminal

```
cansend can0 123#1122
```

Which advantages do you get by this, at first sight complex procedure?

- All systems and virtual devices are separated from each other. This reduces the "pollution" and the influence of the container stops at its borders. At the same time, this provides some protection for the host system.
- Each Docker container has its own separate IP address, just as a real application would if they were each running on a separate device.
- Reproducibility is very high. By setting up a Dockerfile, the image and container can be integrated as many times as needed and into a wide variety of systems.
- This approach is much more lightweight than creating an entire system as a virtual machine. At the same time, each VM would need its own physically present CAN interface.
- It is equally possible to use this solution on embedded devices, such as our sysWORXX CTR-700.

However, there are a few drawbacks to this solution:

- It requires some manual intervention to set up the overall system.
- The use of the CAN_GW and the resulting connection of the individual networks results in an increased message overhead, but this is of little importance in current systems.

A more detailed demonstration with further explanations and information was performed by our system architect Daniel Krüger during the Chemnitzer Linux-Tage. The video, including presentation materials, can be found here.

Further information:

- *Daniel Krüger, SocketCAN with Docker under Linux*
- *CANopen demo project*
- *Forwarding CAN Bus traffic to a Docker container using vxcan on Raspberry Pi*
- *Oliver Hartkopp, Design & separation of CAN applications*
- *Christian Gagneraud, can4docker*
- *Daniel Krüger, SocketCAN - CAN driver interface under Linux*
- *Christian Sandberg, CANopen for Python*
- *Martin Willi, Kernelpatch Move device back to init netns on owning netns delete*

SYS TEC electronic AG
Am Windrad 2
D-08468 Heinsdorfergrund

info@systec-electronic.com
Telefon: +49 (0) 3765 / 386000
Telefax: +49 (0) 3765 / 386004100