emlix
embedded linux systems

Embedded containers: securely packed, easily portable

by Thomas Brinker

Embedded containers: securely packed, easily portable

by Thomas Brinker

Containers ensure a secure separation of different software components without the full virtualization overhead. In addition to the security and the portability, there are other positive secondary effects.

Containers have been used for virtualization on large server farms for some years now. Docker is by far the best-known software suite. Software containers efficiently separate hardware from the running operating system and application software without the overhead of a full virtualization. In this way, containers make it easier to distribute and manage complex software systems.

Currently, containers are also being used in embedded systems. The ever-expanding memory space has made the increased use of container-based architectures technically possible. Web servers, multimedia GUI applications, complex middleware solutions and hardware-related back-end services are being orchestrated on medical-technical, industrial and consumer devices. The isolation and reduction of the programming interfaces are particularly helpful aspects of containers. These aspects form the basis for the economic benefits of container-based systems.

Advantage 1: Isolation

Containers are being kept isolated from each other and can only communicate with each other via just a few defined interfaces. Containers are isolated by separating the directory trees, network interfaces and other interfaces. All containers are using the same kernel though. There is no virtualization of hardware.

IP networks are being used as an interface to communicate between containers (image 1). These can be controlled very precisely with packet filters. Programming network interfaces is standard nowadays and well

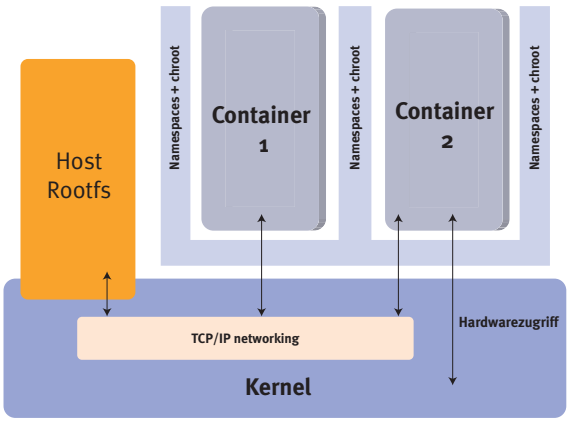

Image 1: All containers use the same kernel, but different file systems and namespaces.

abstracted with RESTful services and/or MQTT.

The isolation of applications in containers ensures that the use of external software can barely cause any damage. For example, resource-hungry Java applications can run in a container with perfectly matching libraries. Systems overloads caused by a high CPU load or memory consumption are being limited to one container. Tools and libraries that are required in the container cannot be used for attacks on other containers or the host system. The orchestration of these tools can be realized in each container with their own build system.

Software reproduction with containers

Containers are not only an interesting framework in embedded systems, but are also important for the automatized and reproducible build of embedded Linux systems. If code-compilation and image generation occur in containers, it can be excluded that other uncontrollable files influence the result of the compilation. This is a crucial requirement for the process of creating a validated and reproducible configuration of an operating system platform for an industrial product. For example, e2factory (www.e2factory.org) has been using containers as a build automation system since 2005 in order to make product-specific embedded Linux systems transparent and maintainable.

Advantage 2: Easy migration

The Linux system calls (approximately 330 in total) are the only function calls that allow software to „leave" a container. System calls are the functions of a kernel which have been developed very conservatively and can be accessed via secure paths. Functions in libraries or external tools which are being used in the container have to be in the file system of the container. They can only leave the container via system calls. In this way, it is also possible to run containers with different versions of a library on a CPU.

The greatly reduced external interface makes a simple migration from one Linux system to another possible. The system call interface is being developed in a very conservative and defensive way. Once a system call has been introduced, the developer community will not change it anymore. If a system call becomes redundant, a deactivation only occurs after a thorough consideration and a long transition period.

This means that a container is also usable on other – younger or older – Linux systems with high probability. This aspect is one of the interesting features of containers for web development. It is possible to develop a container in-house and thereafter, transfer it to a data center.

Advantage 3: Load distribution

Containers leading to little computational load can run in larger numbers on a computer. When a container needs more capacity, it can be moved to another server. There is no need to worry about libraries, tools and configurations needed on that server, because the container contains all the prerequisites. This makes it possible to interlink software development, installation and testing more closely, as it usually is the case in DevOps.

In the embedded world, this feature is of utmost interest. Once a specific application is integrated in a container, it does not matter whether manufacturer A delivers a board with Yocto-Linux and manufacturer B a board with an older build root. It is only important that it is possible to run containers.

The same is true when looking into the future. The question often arises: ,,Will the new CPU / the new board with its Linux-BSP contain all of the libraries and tools that I need?" A consistent container design makes it possible to be completely independent: One only needs to copy the container from the old to the new system and the software can continue running.

Namespaces as container borders

Namespaces create containers in Linux. The clone() system call gets all the corresponding parameters. The available namespaces are:

- mount – All mounted file systems are being integrated in the namespaces. However, new mounts remain in the namespace in which they were entered.
- pid – The process IDs newly start with 1 in new namespaces.
- net – New network interfaces are available in the the container and can be connected via bridge or routing outside of the container.
- ipc – Classic inter-process communication is being disconnected within each container in this way.
- uts – Domain name / host are being seperated.
- User ID – User IDs are being devided in this way. The ID 0 (usually ,,root") can be used in the container and as well its role within the container. Outside of the container, this ID can be mapped differently.

Namespace

The term namespace has its origin in programming. The names for objects – especially in the object-orientated programming – are organized in a sort of tree structure and uniquely addressed by a corresponding path name. Simply put, this means that within such a space, each name explicitly corresponds to an object. However, the same name can be used in another namespace for a different object. Besides, these independent namespaces can be connected with each other within a hierarchy.

(Source: Wikipedia)

- cgroups – The control groups managing the CPU, memory and I/O use of resources can also be organized in their own name spaces in each container.

„cgroups" play a key role in isolating the containers from each other and the host system. They ensure that none of the containers occupies all of the memory space or the CPU with infinite loops.

After the namespaces have been set up and further configurations (e.g. network adresses and routing) have been carried out, the container can be entered with chroot () or pivot_root() and a specific application can be started. This application can only access the namespaces and files in the container (image 2). The directory tree has to contain all of the needed files. The chroot() command blocks the access to all the other files and d irectories above the container root. As usually in namespaces, access is blocked by non-addressability.

Docker installation not needed

All of the steps described above for creating a container can be programmed manually. However, it is recommended to use a tool that integrates a lot of these things. It can be a complete and therewith heavy-weight Docker installation. In many cases, it is more useful to rely on tools that are really relevant

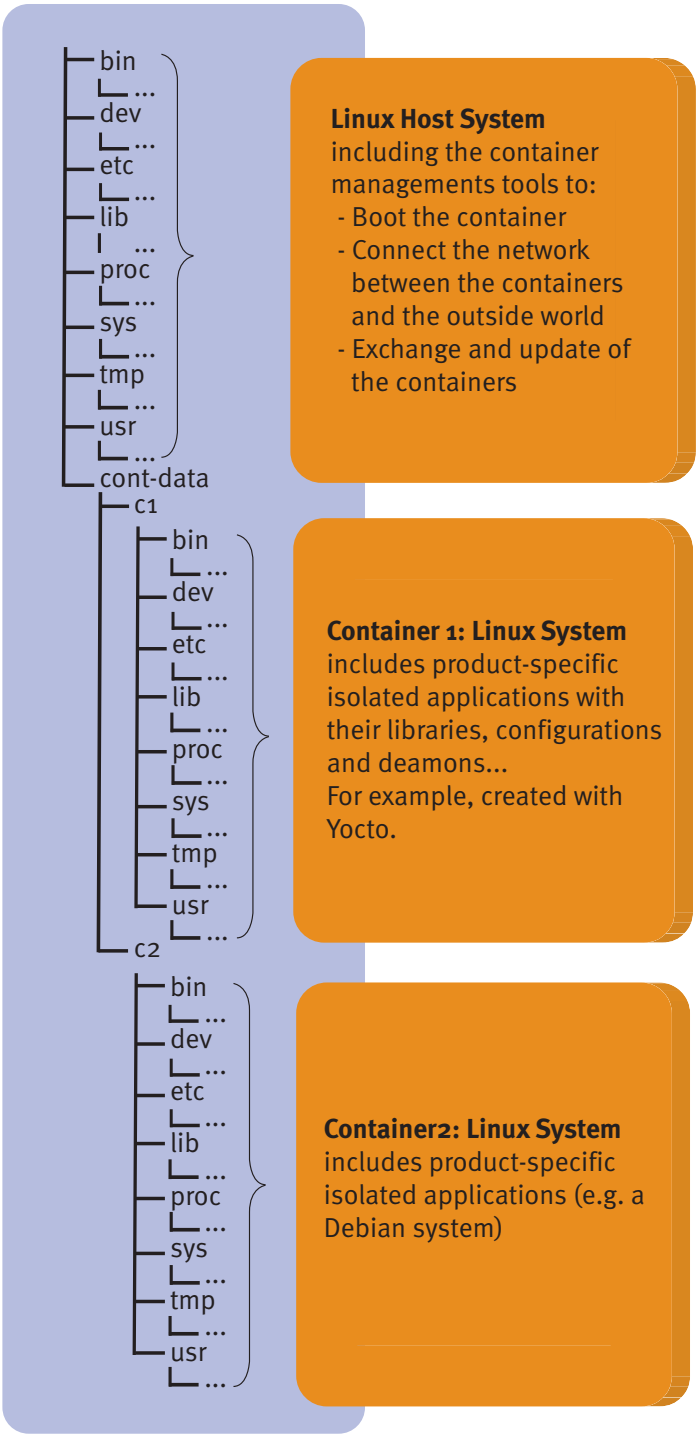

Image 2: Directory tree of a container system. Applications in the containers use c2/ or c1/ as root directory. Access to higher directories in the hosting system is not possible.

for the running container though. The following tools – all with a slightly different focus - are available:

- runc
- LXC
- rkt

rkt und runc are being written in go, whereas LXC ends up being more streamlined in C. LXC has its own config file syntax in which the desired characteristics of the namespaces are being defined. Runc is strongly orientated towards the OCI runtime standard and also constitutes its reference implementation. Rkt uses the app Container Images. The configuration in this app is similar to OCI in JSON, but also carries the root file system as an image.

Containers provide flexibility

Containers are an established and high-performing instrument to migrate software packages easily between different boards, CPUs and software environments. Nowadays, hardware and software become obsolete at a very high rate. In this context, containers ensure a long product life cycle. The isolation of containers from each other makes it also possible to integrate independent third-party software whose quality and trustworthiness is not as high.

Thomas Brinker
is a senior system engineer and project manager at emlix. He deals with the architecture and design of embedded Linux systems in medical technology, in the consumer and industrial field as well as their network integration.
thomas.brinker@emlix.com

The German version of this paper was published in Elektronik 22/18.