Embedded Container:

Sicher abgepackt, stets portierbar

(Bild: weerasak saeku | Shutterstock)

Container sorgen für eine sichere Trennung verschiedener Software-Teile, ohne den vollen Overhead der Virtualisierung zu verursachen. Zusätzlich zu Sicherheit und Portierbarkeit hat das noch weitere positive Nebeneffekte

Von Thomas Brinker

Schon seit einigen Jahren werden Container auf großen Server-Farmen zur Virtualisierung eingesetzt. Docker ist dabei die mit Abstand bekannteste Software-Suite. Software-Container trennen effizient konkrete Hardware und darin laufende Betriebssystem- und Applikations-Software, ohne den Overhead einer Voll-Virtualisierung. Dadurch vereinfachen Container das Verteilen sowie das Management von komplexen Software-Systemen.

Aktuell erobern sie auch die Embedded-Welt. Immer größere Speicher bilden für das Vordringen von Container-basierten Architekturen die technische Basis. Immerhin werden mittlerweile auf medizintechnischen, industriellen und auch Consumer-Geräten Web-Server, multimediale GUI-Applikationen und komplexe Middleware-Lösungen sowie hardwarenahe Backend-Dienste orchestriert. Isolation und Reduktion der Programmierschnittstellen sind hierbei zwei besonders hilfreiche Aspekte von Containern. Sie bilden auch die Grundlage für die wirtschaftlichen Nutzeffekte Container-basierter Systeme.

Software-Reproduktion mit Containern

Container sind nicht nur als Framework auf Embedded-Systemen ein interessantes Tool, sondern auch für das automatisierte und reproduzierbare Bauen von Embedded-Linux-Systemen von Interesse. Wird der Quellcode des Betriebssystems in Containern compiliert und werden im Anschluss Images darin erzeugt, so kann ausgeschlossen werden, dass andere nicht kontrollierte Dateien Einfluss auf das Ergebnis der Compilation nehmen. Für den Prozess einer validierbaren und reproduzierbaren Zusammenstellung einer Betriebssystem-Plattform für ein Industrieprodukt ist das eine zentrale Voraussetzung. Als Build Automation System nutzt beispielsweise e2factory (www.e2factory.org) schon seit 2005 Container, um produktspezifische Embedded-Linux-Systeme nachvollziehbar und wartbar zu entwickeln.

Vorteil 1: Isolation

Container sind voreinander isoliert und können nur über wenige definierte Schnittstellen miteinander kommunizieren. Die Isolation von Containern wird über die Trennung der Verzeichnisbäume, Netzwerk-Schnittstellen und weiterer Interfaces erreicht. Alle Container zusammen benutzen jedoch den gleichen Kernel. Eine Virtualisierung von Hardware findet nicht statt.

Als Schnittstelle zur Kommunikation zwischen Containern wird dabei meist auf IP-Netzwerke zurückgegriffen (**Bild 1**). Diese lassen sich mit Paketfiltern sehr genau steuern. Die Programmierung von Netzwerkschnittstellen ist heutzutage Standard und beispielsweise mit RESTful Services und/oder MQTT gut abstrahiert.

Die Isolation von Applikationen in Containern erlaubt zum Beispiel den Einsatz externer Software, die durch die Isolation kaum Schaden anrichten kann. So können beispielsweise ressourcenhungringe Java-Anwendungen in einem Container mit den exakt passenden Libraries betrieben werden. Systemüberlastungen durch hohe CPU-Last oder großen Speicherverbrauch bleiben dabei auf den Container begrenzt. Tools und Libraries, die im Container benötigt werden, können nicht zu Angriffen auf andere Container oder das Host-System genutzt werden. Die Orchestrierung dieser Tools kann pro Container durch ein jeweils eigenes BuildSystem erfolgen.

Vorteil 2: Einfache Migration

Die Linux-Systemcalls (z. B. etwa 330) sind die einzigen Funktionsaufrufe, die es einer Software im Container erlauben, diesen zu verlassen. Systemcalls sind Funktionen des Kernels, die extrem konservativ entwickelt und über geschützte Wege aufgerufen werden. Funktionen in Libraries oder externe Tools, von denen im Container Gebrauch gemacht werden soll, müssen im Dateisystem des Containers vorhanden sein. Auch sie können nur über die Systemcalls den Container verlassen. Es ist also möglich, Container mit jeweils unterschiedlichen Versionen einer Library auf einer CPU zu betreiben.

Die durch die Systemcalls stark reduzierte Außenschnittstelle eines Containers erlaubt eine einfache Migration von einem Linux-System zum anderen.

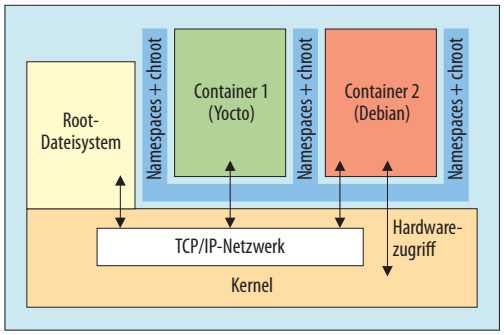

Bild 1. Alle Container nutzen den gleichen Kernel, aber unterschiedliche Dateisysteme und Namespaces. (Quelle: Emlix)

Die Systemcall-Schnittstelle wird sehr konservativ und defensiv entwickelt. Einmal eingeführte Systemcalls werden von der Entwickler-Community nicht mehr verändert. Sollte ein Systemcall überflüssig werden, wird dieser erst nach langer Abwägung und Übergangszeit deaktiviert.

Hieraus folgt, dass ein Container auch auf anderen – jüngeren oder älteren – Linux-Systemen mit sehr hoher Wahrscheinlichkeit nutzbar ist. Diese Eigenschaft ist in der Web-Entwicklung eine der interessantesten Eigenschaften von Containern, bedeutet es doch, dass man Container mit einer Web-Anwendung „zu Hause“ entwickeln und dann in ein beliebiges Rechenzentrum übertragen kann.

Vorteil 3: Lastverteilung

Verursachen die Container wenig Rechenlast, werden sie in größerer Anzahl auf einem Rechner betrieben. Braucht der Container mehr Leistung, wird er exklusiv auf einen anderen Server verschoben. Welche Libraries, Tools und Konfigurationen auf dem Server benötigt werden, ist unerheblich; der Container bringt alle Systemvoraussetzungen mit. Entwicklung, Installation und Test von Software lassen sich so erheblich enger verzahnen, so wie es beispielsweise bei DevOps üblich ist.

In der Embedded-Welt ist genau diese Eigenschaft ebenfalls von zentralem Interesse: Ist die anwendungsspezifische Applikation erst einmal in einen Container integriert, ist es egal, ob das Board von Hersteller A mit einem Yocto-Linux oder von Hersteller B mit einem älteren Buildroot kommt: Hauptsache, es kann Container betreiben.

Das gleiche gilt, wenn man in die Zukunft blickt. Es stellt sich dann oft die Frage: „Wird die neue CPU/das neue Board mit seinem Linux-BSP, alle „meine“ notwendigen Libraries und Tools mitbringen? Bei konsequentem Container-Design ist man vollständig unabhängig, kopiert einfach den Container vom alten auf das neue System und kann die Software weiter betreiben.

Namespaces als Containergrenzen

Container werden in Linux mittels Namespaces (**Kasten**) erzeugt. Dem

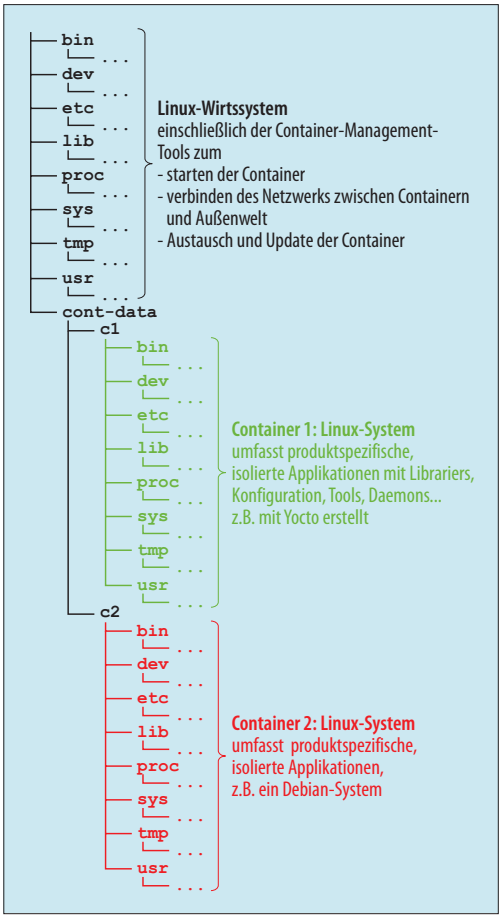

Bild 2. Verzeichnisbaum eines Containersystems. Applikationen in den Containern sehen c2/ oder c1/ als root-directory. Zugriffe oberhalb in das bewirtende System sind nicht möglich. (Quelle: Emlix)

clone()-Systemcall werden dazu entsprechende Parameter mitgegeben. Es stehen folgende Namespaces zu Auswahl:

- mount – Es werden alle bisher gemounteten Dateisysteme in den Namespaces übernommen. Neue Mounts bleiben jedoch in dem Namespace, in dem sie vorgenommen wurden.
- pid – Die Prozess-IDs beginnen im neuen Namespace mit 1.
- net – Neue Netzwerk-Interfaces stehen im Container zur Verfügung und können außerhalb des Containers per Bridge oder Routing verbunden werden.
- ipc – Klassische Interprozess-Kommunikation wird hiermit pro Container getrennt.
- uts – Domainname/host werden getrennt.
- User ID – Hiermit werden die Nutzer-IDs aufgetrennt. Die ID 0 (normalerweise „root“) kann damit im Container genutzt werden und innerhalb des Containers auch deren Rolle. Außerhalb kann die ID jedoch auf eine andere Rolle gemapped werden.
- cgroups – Die Controls-Gruppen zur Steuerung von CPU-, Speicher- und I/O-Ressourcenverbrauch können ebenfalls in eigene Namespaces pro Container geordnet werden.

Namespace

Der Namensraum (englisch namespace) ist ein Begriff aus der Programmierung. Dabei werden – vor allem bei der objektorientierten Programmierung – die Namen für Objekte in einer Art Baumstruktur angeordnet und über entsprechende Pfadnamen eindeutig angesprochen.
Vereinfacht bedeutet dies, dass innerhalb eines solchen Raumes jeder Name eindeutig ein Objekt bezeichnet. Der gleiche Name kann jedoch in einem anderen Namensraum wieder frei zur Bezeichnung eines anderen Objekts benutzt werden. Außerdem können diese unabhängigen Namensräume innerhalb einer Hierarchie verbunden werden. *(Quelle: Wikipedia)*

Bei der Abschottung von Containern voreinander und vor dem Host-System spielen die „cgroups“ eine zentrale Rolle. Sie schützen davor, dass ein Container sämtlichen Arbeitsspeicher okkupiert oder die CPU durch Endlosschleifen dauerhaft belegt.

Nachdem die Namespaces errichtet und weitere Konfigurationen, wie beispielsweise Netzwerk-Adessen und -Routing, vorgenommen sind, wird der Container mit chroot() oder pivot_root() betreten und eine definierte Anwendung gestartet. Diese Anwendung kann dann ausschließlich auf die Namespaces und Dateien im Container zugreifen. (**Bild 2**) Der Verzeichnisbaum muss alle notwendigen Dateien enthalten. Der chroot()-Befehl schließt alle anderen Dateien und Verzeichnisse oberhalb der Containerwurzel vom Zugriff aus. Wie bei Namespaces üblich, ist der Zugriff durch Nicht-Adressierbarkeit versperrt.

Docker-Installation nicht nötig

Die oben benannten Schritte zur Errichtung eines Containers können sämtlich manuell programmiert werden. Es empfiehlt sich jedoch, ein Tool zu nutzen, das vieles integriert. Das können komplette und damit schwergewichtige Docker-Installationen sein. In vielen Fällen ist es jedoch sinnvoller, nur auf die für den Container-Betrieb relevanten Tools zurückzugreifen. Folgende Tools mit jeweils etwas anderen Fokus stehen neben anderen dafür zur Auswahl:

- runc
- LXC
- rkt

Während rkt und runc in go geschrieben sind, kommt LXC schlanker in C daher. LXC hat eine eigene Config-Datei *syntax*, in der die gewünschten Eigenschaften der Namespaces definiert werden. Runc orientiert sich ganz stark am OCI-Runtime-Standard und bildet auch dessen Referenzimplementierung. Rkt verwendet App Container Images. Darin ist dann die Konfiguration ähnlich zu OCI in JSON enthalten, aber zusätzlich auch das Rootfilesystem als Image.

Container geben Flexibilität

Container sind ein etabliertes und leistungsstarkes Mittel um komplexe Software-Pakete einfach zwischen verschieden Boards, CPUs und Software-Umgebungen zu migrieren. In Zeiten von zügiger Obsoleszenz von Hard- und Software stellen sie somit ein Mittel zur Gestaltung eines langen Produktlebenszyklus‘ dar. Die Abschottung der Container voreinander erlaubt zusätzlich die Integration von Software unabhängiger Dritter, deren Qualitäts- und Vertrauensbasis geringer ist. *jk*

Thomas Brinker
ist Senior System Engineer und Project Manager der emlix GmbH. Er beschäftigt sich mit Architektur und Design von Embedded Linux-Systemen in der Medizintechnik, im Konsumenten- und im industriellen Bereich sowie deren Netzwerkintegration.
thomas.brinker@emlix.com