Technology Corner

Containerization
–
An Alternative to Virtualization in Embedded Systems

Rudolf J Streif
CTO, ibeeto
September 2019

Abstract

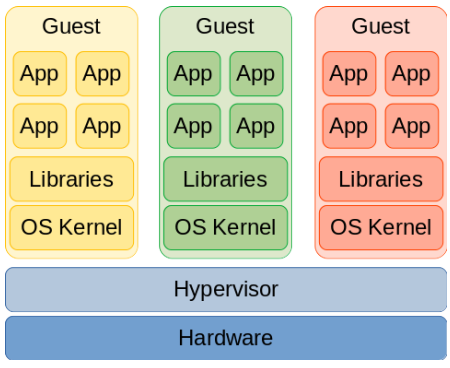
Virtualization is the foundation of cloud computing. It enables efficient deployment of independent systems across a computing infrastructure sharing resources for better utilization and eventually cost savings. With more and more powerful system-on-chip (SoC) available to embedded developers the idea of combining multiple discrete embedded systems into one using virtual machines is the logical consequence. In particular in modern automobiles where over 100 electronic control units (ECU) are deployed throughout a vehicle the concept of combining them into fewer units promises the reduction of complexity and wiring and the realization of cost savings.

While the principle idea and goals of containerization are essentially the same as for virtualization, there are significant differences between the two technologies. In a nutshell, virtualization is abstraction of hardware while containerization is abstraction of operating systems.

This ibeeto Technology Corner article explains the commonality and differences between virtualization and containerization and what new possibilities container systems can open for embedded systems.

What is Virtualization?

Virtualization is the abstraction of hardware. A software layer, commonly referred to as the *hypervisor*, provides one or more virtual machine in which full operating system stacks run, commonly referred to as *guests*. These virtual machines have virtual CPUs, memory and devices for exclusive use of the guest operating system. Typically, a guest operating system is not even aware of that it is running on a virtual machine rather than directly on hardware, affectionately dubbed as *bare metal*. The hypervisor is responsible for mapping the virtual resources provided to the virtual machines to the actual hardware, managing the virtual machine life cycle and arbitrating the concurrent use of the hardware by the different virtual machines. Hypervisors can be distinguished into two categories: Type 1 and Type 2.



Type 1 hypervisors run directly and exclusively on the hardware and provide virtual machine environments to the guest operating systems. Type 1 hypervisors are commonly used for cloud computing infrastructure such as Amazon's Elastic Compute Cloud (EC2) and

Microsoft's Azure Cloud. A Type 1 hypervisor is also what engineers would use for virtualization on embedded systems.

Type 2 hypervisors run as an application program inside another operating system, the *host*, and hence share the host resources with other applications running next to them. Type 2 hypervisors are often used in desktop environments. Examples are Oracle VirtualBox or VMWare.

The virtual CPUs provided by the hypervisor to the guest operating systems is the same architecture and instruction set than that of the underlying hardware CPU. The concept of providing virtual CPUs with a different architecture and/or instruction set is referred to as *emulation*. Embedded software developers commonly use emulators to test their programs as the CPU architecture of the *target* system, the system the software is developed for, in many cases is different from the CPU architecture of the *development* or *host* system the developer is doing the work on. An example is a developer writing an application on a PC (x86 or x86-64 architecture) for a mobile phone (ARM architecture). The most significant drawback of emulation is performance: in most cases an application runs slower in an emulator than on hardware (there are exceptions to that as it all depends on what hardware is used). Therefore emulation is not commonly used in production systems.

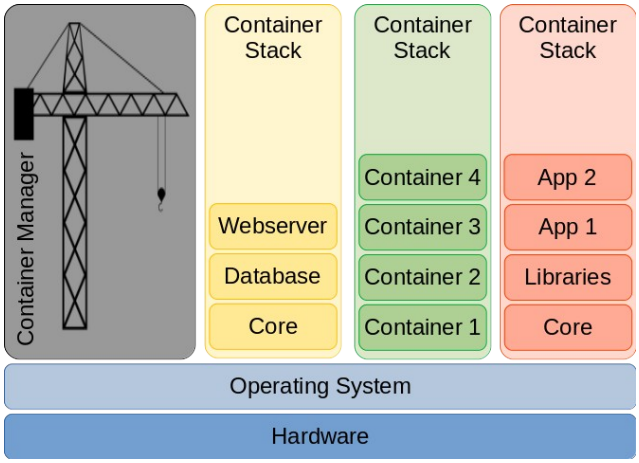
In theory virtualization looks like a straight-forward concept. The challenges are of course in the details of correctly and transparently virtualizing the hardware for the guest operating systems. This is well-understood for any hardware that is commonly used for cloud computing infrastructure, that is, server-class computers with x86-64 CPUs, RAM, hard drives and network interfaces. For these types of devices virtual drivers are standard for any hypervisor. It becomes much more challenging for embedded devices that have no standard hardware mold. Such devices have a plethora of different I/O interfaces such as I2C, SPI, CAN, etc. as well as specific hardware such as cellular modems, GPS receivers, DSRC radios to name a few common to the automotive industry. If a virtual device driver is not available then there is no other solution but to allow a single guest to directly access the hardware through its own drivers. Then of course, there is no sharing of the hardware among the guests.

Because virtualization is hardware abstraction, different guest operating systems can run inside of separate virtual machines on top of the same hypervisor and the same hardware at the same time. For example, Windows, Linux and other operating systems can run inside of virtual machines managed by VirtualBox running on top of a Linux host at the same time.

What is Containerization?

The concept of containerization is based on the separation of operating system kernel and application software (programs and libraries, also commonly referred to as *user-space*). If you

look at Linux distributions, what makes one distribution unique and distinguishable from another is how the user-space is assembled. As long as kernel and user-space are compatible you can for example run an Ubuntu user-space on top of an Archlinux kernel and vice-versa. That is essentially what containers are doing in a nutshell.



Similar to the hypervisor, a *container manager* is responsible for managing the lifecycle of the containers. The container manager runs on top of an operating system which provides the kernel and the core system infrastructure. However, the container manager is not a shim between the operating system and the containers. It does not provide the runtime environment for the containers and also does not manage the system resources for concurrently running containers. That is all done by the kernel of the underlying operating system. Once a container is started by the container manager, the container directly interacts with the operating system kernel. That is a fundamental difference to virtualization where the hypervisor is essentially a software layer between the virtual machines and the underlying OS kernel.

Another essential difference between virtualization and containerization is that the former are monolithic while the latter can be modular. While of course an entire user-space stack such as Ubuntu with all necessary libraries and applications can be run as a single container, it is much more flexible to actually let the container manager build a *container stack*. Think of a container stack as Legos™: a series of smaller containers that are assembled to provide the required functionality. For example, a web server could be assembled from a base container with the core libraries, a database container, and a finally a container with the actual web server. The container manager assembles the runtime container stack instance from container images. The same container images can be reused concurrently thus saving storage on the host system. A configuration file, commonly referred to as a *container playbook* or simply *playbook*, tells the container manager from what containers to assemble the runtime instance and how the containers are connected with each other. This is a very powerful concept, as simply by changing the playbook entirely different configurations can dynamically be created. That is not possible with virtual machines (of course, you could run containers inside of a virtual machine).

One thing you cannot do with containerization is to run different operating systems. If the underlying kernel is Linux then the container stacks are Linux user-space (of course, you could run a hypervisor inside of a container).

Why bother?

Now that we explained virtualization and containerization, the question is why and when to use the one or the other, and, more fundamentally, do we need them at all in embedded systems? The following bullet points are commonly cited when talking virtualization and containerization. We are not claiming that this list is exhaustive, your application might have other and/or additional requirements.

- **Resource Utilization and Sharing** – Both virtualization and containerization promise to provide better resource utilization as they allow multiple systems to be combined onto one hardware. Since not all applications do not make use of resources all the time this would lead to better utilization and eventually less cost for hardware. However, this is also the very hallmark of a multi-processing and multi-tasking operating system. We could just run the applications directly on top of a multi-processing and multi-tasking operating system and achieve a similar result without the overhead and complexity of virtualization or containerization. As a matter of fact, on an embedded system the use of virtualization could lead to less efficient memory usage as these systems typically cannot use virtual memory that moves unused memory blocks to storage and loads them again when needed. Consequently, a worst-case fixed amount of memory needs to be allocated to the virtual machines.
- **Isolation** – Virtualization and containerization provide isolation from other processes and applications. However, once again, the same is true for multi-tasking operating systems. They isolate processes into their own space using virtual memory pages etc. In effect, these are the same techniques a hypervisor is using to isolate the virtual machines.
- **Security** – Albeit commonly quoted as a reason for deploying virtualization or containerization, neither one of them are security concepts. If a virtual machine or a container and the applications running inside them are not hardened for security, virtualization and containerization make no difference for vulnerability. They may limit the effect of a vulnerability to the particular virtual machine or container thus possibly not compromising the entire system but that is not a given either. Since neither virtual machines or containers are islands and are typically connected to each other or other systems through a network or other means, viruses, malware etc. can also spread (this is no different to PC networks in organizations where a virus on a single infected system can easily spread across the entire organization).

- **Different Runtime Environments** – This is where virtualization and containerization have a clear advantage of running applications on the same operating system. Both can provide entirely different runtime environments with different libraries, configuration etc. As already mentioned, virtualization can even provide entirely different operating systems.
- **System Lifecycle Management (SLM)** – System lifecycle management is easiest with applications running on a single operating system. Applications can be started and stopped at will, the entire system can be shut down or reset, or put into a power saving state such as suspend-to-RAM. With containerization SLM is not that much more complicated since container life cycles are handled by the container manager. And since the processes running inside a container are still simply processes running on top of the host OS suspend-to-RAM is essentially implicit. With virtualization SLM becomes much more complex as the OS running inside of a virtual machine has its own life cycle management. While virtual machines can be suspended to RAM as any process, device drivers of a guest OS directly handling hardware and interrupts can have unintended side effects on the ability of suspending an entire system to RAM.
- **System Startup and Shutdown Performance** – This is always a hot topic for any embedded device. We are used to our computers, and even our smart phones, to take their sweet time to boot up and shutdown, but our expectations from TVs, appliances etc. and, for that matter cars, are much different . These systems, we expect to be ready and operational instantaneously. For some applications that is already a problem with an OS such as Linux. Virtualization and containerization add additional time for the virtual machines and the containers to be ready respectively. Virtualization has a clear disadvantage as the virtual machine has to start up an entire operating system before being able to run any applications. Launching containers is much faster.
- **Software Updates and Software-over-the-Air (SOTA)** – Software updates for common desktop and server systems are standard. Individual software packages and even the operating system kernel to entire system upgrades can be performed mostly automatically. They have also become very stable and in case of a failure a fallback to the previous system is generally possible. In cases of a catastrophic failure resulting in an inoperable system, it can be restored from scratch albeit eventually at the expense of loss of data (hence, the backup warning before the process begins). Even for today's smart phones software updates over the air are commonplace and stable.

  Virtualization provides different avenues for software updates: individual software packages inside the virtual machine can be updated or the entire virtual machine image can be updated on the host (of course if applications inside the virtual machine

store data inside the virtual machine such data would be lost unless it is stored elsewhere). If there is sufficient storage the previous virtual machine image can be retained as a backup in case of a failure.

Containerization offers an entirely new angle to software updates. If you have worked with Ansible, Docker, Kubernetes and other container managers before you know that when the container manager initializes a container stack according to a playbook it first checks if it can find the right containers with the correct versions in its local repository. If a container is missing it can download it from a remote repository. This is a powerful concept that can be used for software updates: if a software update for a container stack is necessary a new playbook can be sent to the device. When the container manager starts the container stack the next time, it uses the new playbook. Since the new playbook contains updated containers not available in the local repository the container manager downloads it them from the remote repository before launching the container. If there is a problem with the new container stack, the container manager can simply fall back to the previous playbook. Updating the base operating system and the container manager, however, still requires an image upgrade, but that image is much smaller than an entire system.

Summary

If you are looking at deploying virtualization for your next embedded project, you should also have a closer look at containerization. Unless you have a requirement for an operating system that does not support containerization or you need to run different operating systems in virtual machines, containerization can provide an excellent alternative with less technical challenges for resource sharing and lifecycle management.

Contact Information

We would like to hear from you! If you have feedback to this ibeeto Technology Corner article please contact us:

ibeeto
Rudolf J Streif
rudolf.streif@ibeeto.com
+1.855.442.3386

ibeeto is a hardware and software consulting firm providing development and engineering services as well as a variety of training courses. We are happy to assist you with getting your next embedded project off the ground.