1. Comparing Containers

Containers type

1. cloud
2. HPC
3. IoT

Containers alternative

4. unikernel

Cloud

Containers power microservices architecture that have their benefits for application services. Any application built with microservices-based methodology tends to have a chunk of small services that can be packaged in containers. All those services maintain their life cycle along with service-specific requirements of independent development, granular scaling and patching and fault remediation.

Containers for Cloud

- **consistent environment**
- **run anywhere**
- **isolation**
- **low overhead**
- **small startup time**
- **service based**
- ...

HPC

HPC leverages distributed compute and storage resources to solve complex issues with large data volumes. HPC clusters are commonly known as Supercomputers. Complex algorithms are used on large data sets to generate insights. HPC systems use a large set of CPU or GPU in parallel architecture that creates enough of a computing resource pool to execute complex mathematical algorithms.

HPC are mainly used for scientific research and big data analysis.

Containers for HPC

- **modularity**: HPC applications requires software dependencies that are numerous, complex, unusual, differently configured, or simply newer/older than what the center provides
- **scaling**: react to situations when there is a spike in the data processing, scaling the number of container to tackle such spikes

- **portability**: run the application on workstations and other test and development system not managed by the center
- **consistency**: environments can be easily, reliabily, and verifiably reproduced in the future
- **security**: exploit Linux user namespaces to run containers with no privileged operations in a multi-tenant system
- **native performance**: exploit dedicated hardware with less possible overhead
- **support for GPU**
- **support for parallel filesystems and diskless computing nodes**
- **support for workload manager and job scheduler**

Docker and HPC issues

- Docker emulates a virtual machine in many aspects, users can escalate to root
- Docker uses a root owned daemon that users can control by means of a writable socket
- can not limit access to local file systems
- no native support for GPU
- no native support for MPI
- incompatibilities with existing scheduling and resource manager paradigms
- HPC is not the appropriate use-case or interest for the Docker community

Parallelism with HPC containers

Jobs are dispatched to containers using a workload manager / resource manager (such as Slurm, SGE). Some container technologies provide native support for MPI.

IoT

IoT, refers to network of physical devices collecting and sharing data.

Containers for IoT

- **limited network access**: use less bandwidth, downloading only the necessary part
- **heterogeneous devices environments**: support for wide variety of chipset architectures. It also simplify the software configuration
- **minimal hardware resources**: use the resources (memory and computing power) of the device more conservatively. Configuration processing is minimal.
- **use smaller binaries**: reduce the resource consumption

Docker and IoT issues

- not suitable for all IoT software deployment
- not compatible with all of the hardware used by IoT
- heavyweight container runtime not suitable for IoT devices with few hardware resources
- not designed to reduce the network bandwidth consumption

Unikernel

A unikernel is a type of microservices environment that contains absolutely everything required to run a particular piece of software—including not just the software code itself, but also the operating system code necessary to host it. Plus, everything that is not strictly necessary for hosting the app is stripped out of the unikernel.

Unikernel Advantages

- self-hosted, portable and minimalist
- very small overhead
- suitable for cloud application: unikernel as an alternative to VMs
- suitable for IoT: unikernel provides everything needed to deploy the software for an IoT device
- suitable for device drivers: drivers can be supplied on an as-needed basis inside portable environments
- suitable for on-demand computing: fast to boot

References

- Containers and HPC: Mutually Beneficial
- Intel HPC developer conference
- https://sylabs.io/
- Singularity on HPC
- https://hpc.github.io/charliecloud/index.html
- Why the Internet of Things Needs Docker Containers
- Unikernels Use Cases: IoT, the Cloud and More
- user namespaces

2.1 Comparing Solutions - HPC

Comparison

	Open Source	First Release	Last Release	Stars	Test	Issues (open/closed)	Link
Singularity	Yes	Oct. 2015	Feb. 2020	1.6k	Ok	370 / 2250	https://github.com/sylabs/singularity
Shifter	Yes	Mar. 2015	Jan. 2020	280	Not Ok	33 / 61	https://github.com/NERSC/shifter
CharlieCloud	Yes	Jun. 2015	Mar. 2020	171	Ok	117 / 306	https://github.com/hpc/charliecloud

	Open Source	First Release	Last Release	Stars	Test	Issues (open/closed)	Link
Sarus	Yes	Nov. 2018	Feb. 2020	32	Ok	5 / 1	https://github.com/eth-cscs/sarus

2.1.1 Singularity

- https://sylabs.io/
- **Report**

Introduction

Singularity is an open source container platform designed to be simple, fast, and secure. Singularity is optimized for compute focused enterprise and HPC workloads, allowing untrusted users to run untrusted containers in a trusted way.

- Secure, single-file based container format
- Support for data-intensive workloads
- Extreme mobility
- Compatibility
- Simplicity
- Security
- User groups

Reference

- https://github.com/sylabs/singularity

Requirements and Permissions

- go
- Singularity requires ~140MiB disk space once compiled and installed.
- 2GB of RAM is recommended when building from source Full functionality of Singularity requires that the kernel supports:
- overlayFS mounts (minimum kernel >=3.18): required for full flexiblity in bind mounts to containers, and to support persistent overlays for writable containers
- unprivileged user namespaces (minimum kernel >=3.8, >=3.18 recommended): required to run containers without root or setuid privilege

Reference

- system requirements

Standards

- OCI compliant from 3.1.0
- OpenPGP standard to create and manage SIF signatures

Reference

- Singularity 3.1.0 brings in Full OCI Compliance

Images

Based on SIF: single file based, compressed, cryptographically signed, trusted, and immutable

- can be pulled from:
  - Singularity Library
  - Docker Hub
  - Singularity Hub
  - local
  - directory
  - definition file

Some comparisons between images from DockerHub and Singularity Library:

Image	Version	Arch	Singularity	DockerHub
Alpine	3.7	amd64	1.99 MB	2.01 MB
Ubuntu	18.04	amd64	35.38 MB	25.49 MB
CentOS	7	amd64	79.08 MB	72.27 MB

Reference

- SIF
- Build a Container

Performance

// TODO

Security

Singularity uses a number of strategies to provide safety and ease-of-use on both single-user and shared systems. 1. the user inside a container is the same as the user who ran the container. This means access to files and devices from the container is easily controlled with standard POSIX permissions

> The Singularity Runtime enforces a unique security model that makes it appropriate for untrusted users to run untrusted containers safely on multi-tenant resources. When you run a container, the processes in the container will run as your user account. Singularity dynamically writes UID and GID information to the appropriate files within the container, and the user remains the same inside and outside the container, i.e., if you're an unprivileged user while entering the container you'll remain an unprivileged user inside the container.

2. container filesystems are mounted nosuid and container applications run with the PR_NO_NEW_PRIVS flag set.

   Additional blocks are in place to prevent users from escalating privileges once they are inside of a container. The container file system is mounted using the nosuid option, and processes are started with the PR_NO_NEW_PRIVS flag set. This means that even if you run sudo inside your container, you won't be able to change to another user, or gain root priveleges by other means. This approach provides a secure way for users to run containers and greatly simplifies things like reading and writing data to the host system with appropriate ownership.

3. the Singularity Image Format (SIF) supports encryption of containers, as well as cryptographic signing and verification of their content

   You generally do not need admin/sudo to use Singularity containers but you do however need admin/root access to install Singularity and for some container build functions (for example, building from a recipe, or a writable image). This then defines the work-flow to some extent. If you have a container (whether Singularity or Docker) ready to go, you can run/shell/import without root access. If you want to build a new Singularity container image from scratch it must be built and configured on a host where you have root access (this can be a physical system or on a VM). And of course once the container image has been configured it can be used on a system where you do not have root access as long as Singularity has been installed there.

4. SIF containers are immutable and their payload is run directly, without extraction to disk. This means that the container can always be verified, even at runtime, and encrypted content is not exposed on disk. Restrictions can be configured to limit the ownership, location, and cryptographic signatures of containers that are permitted to be run

   A SIF file is an immutable container image that packages the container environment into a single file. SIF supports security and integrity through the ability to cryptographically sign a container, creating a signature block within the SIF file which can guarantee immutability and provide accountability as to who signed it. Singularity follows the OpenPGP standard to create and manage these signatures, and the keys used to create them

Reference

- Security in Singularity
- Singularity Security

Available Tools

- Support for Docker and OCI
- MPI
- GPU support
- plugins
- cloud library
- Singularity CRI: consists of two separate services: runtime and image, each of which implements K8s RuntimeService and ImageService respectively
- Limiting container resources with cgroups

Limits

// TODO

2.1.2 Shifter

- https://github.com/NERSC/shifter
- **Report**

Introduction

Shifter enables container images for HPC. Shifter allows an HPC system to efficiently and safely allow end-users to run a docker image. Shifter consists of a few moving parts

1. a utility that typically runs on the compute node that creates the run time environment for the application
2. an image gateway service that pulls images from a registry and repacks it in a format suitable for the HPC system (typically squashfs)
3. example scripts/plugins to integrate Shifter with various batch scheduler systems

These components are:

1. Shifter Runtime: Instantiates images securely on compute resources
2. Shifter Image Gateway: Imports and converts images from DockerHub and Private Registries
3. Work Load Manager Integration: Integrates Shifter with WLM

Design Goals:

- User independence: require no administrator assistance to launch an application inside an image
- Shared resource availability (e.g., file systems and network interfaces)
- Leverages or integrates with public image repos (i.e. DockerHub)
- Seamless user experience
- Robust and secure implementation

Reference

- Index

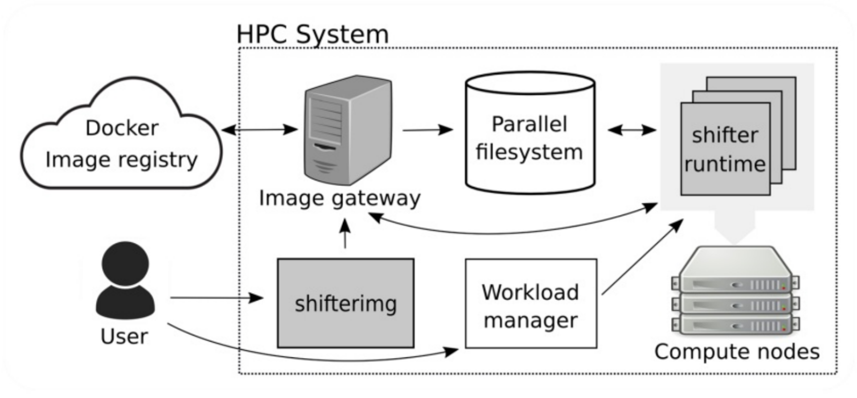

Figure 1: Shifter Architecture

Requirements and Permissions

- requires the installation of the Image Manager and the Shifter Runtime. The image manager doesn't really do any real work on its own, and the image worker uses only user-space tools to construct images (in the default configuration) so they do not need to run with root privilege

Reference

- Installation

Standards

- Docker-compatible container platform

Images

- Docker Images
- uses squashfs

Shifter works by converting Docker images to a common format that can then be efficiently distributed and launched on HPC systems. The user interface to shifter enables a user to select an image from their DockerHub account or the NERSC private registry and then submit jobs which run entirely within the container.

Shifter works by enabling users to pull images from a DockerHub or private docker registry. An image manager at NERSC then automatically converts the image to a flattened format that can be directly mounted on the compute nodes. This image is copied to the Lustre scratch filesystem (in a system area). The user can then submit jobs specifying which image to use. Private images are only accessible by the user that authenticated and pulled them, not by the larger community. In the job the user has the ability to either run a custom batch script to perform any given command supported by the image, or if a Docker entry-point is defined, can simply execute the entry-point

Reference

- Building Shifter Images

Performance

// TODO

Security

Shifter use a great deal of root privilege to setup the container environment. The "shifter" executable is setuid-root, and when run with batch integration the setupRoot/unsetupRoot utilities must be run as root. We are working to reduce the privilege profile of starting Shifter containers to reduce risk as much as possible.

Once a process is launched into the container, processes are stripped of all privilege, and should not be able to re-escalate afterwards.

Shifter enables User Defined Image environment containers. To do this while optimizing I/O on the compute nodes it does require performing several privileged operations on the execution node, both privilege to mount filesystems and rewrite the user space, and privilege to manipulate devices on the system.

Furthermore, because the environment is *user defined*, it is possible that a user could include software which could generate security vulnerabilities if a privileged process accesses such software or resources.

Reference

- Security

Available Tools

- native GPU support: automatic import of host's CUDA driver and devices
- native MPI support:
  - transparently swap container's MPI libraries with the host's at runtime
  - enables full performance from vendor-specific implementations
  - relies on MPICH ABI compatibility
- Shifter is distributed with a SPANK plugin for SLURM.

Reference

- shifter docker containers for hpc

Limits

// TODO

2.1.3 Charliecloud

- https://hpc.github.io/charliecloud/index.html
- **Report**

Introduction

Charliecloud provides user-defined software stacks (UDSS) for high-performance computing (HPC) centers. This "bring your own software stack" functionality addresses needs such as:

- software dependencies that are numerous, complex, unusual, differently configured, or simply newer/older than what the center provides;
- build-time requirements unavailable within the center, such as relatively unfettered internet access;
- validated software stacks and configuration to meet the standards of a particular field of inquiry;
- portability of environments between resources, including workstations and other test and development system not managed by the center;
- consistent environments, even archivally so, that can be easily, reliabily, and verifiably reproduced in the future; and/or
- usability and comprehensibility.

Reference

- What is Charliecloud?

Requirements and Permissions

- running the Docker daemon and executing Docker commands require privileged access, this happens on user-managed resources;
- nothing in the Charliecloud workflow requires privileged or trusted processes or daemons on center-managed resources. All privileged steps take place on user systems, and the scripts escalate with sudo as needed

Reference

- paper

Standards

- OCI compliant is in beta with ch-run-oci

Reference

- ch-run-oci

Images

- Container images can be built using Docker or anything else that can generate a standard Linux filesystem tree

Performance

- Charliecloud containers impose minimal performance penalty, because the guest is using the same kernel and devices as the host
- Performance is the same as native in tests because minimal isolation yields direct access to all resources

Reference

- paper
- paper

Security

Charliecloud uses Linux user namespaces to run containers with no privileged operations or daemons and minimal configuration changes on center resources. This simple approach avoids most security risks while maintaining access to the performance and functionality already on offer.

Charliecloud relies on two things to maintain security:

1. the Linux kernel to enforce access control and other aspects of security
2. the extension that user namespaces will ensure that guest UIDs are an illusion

Reference

- Interacting with the host
- paper

Available Tools

- native MPI support
- inject GPU libraries to a container with ch-fromhost

Limits

// TODO

2.1.4 Sarus

- https://github.com/eth-cscs/sarus
- **Report**

Introduction

Sarus is a software to run Linux containers on High Performance Computing environments. Its development has been driven by the specific requirements of HPC systems, while leveraging open standards and technologies to encourage vendor and community involvement.

Key features:

- Spawning of isolated software environments (containers), built by users to fit the deployment of a specific application
- Security oriented to HPC systems
- Extensible runtime by means of OCI hooks to allow current and future support of custom hardware while achieving native performance
- Creation of container filesystems tailored for diskless nodes and parallel filesystems
- Compatibility with the presence of a workload manager
- Compatibility with the Open Container Initiative (OCI) standards:
  - Can pull images from registries adopting the OCI Distribution Specification or the Docker Registry HTTP API V2 protocol
  - Can import and convert images adopting the OCI Image Format
  - Sets up a container bundle complying to the OCI Runtime Specification
  - Uses an OCI-compliant runtime to spawn the container process

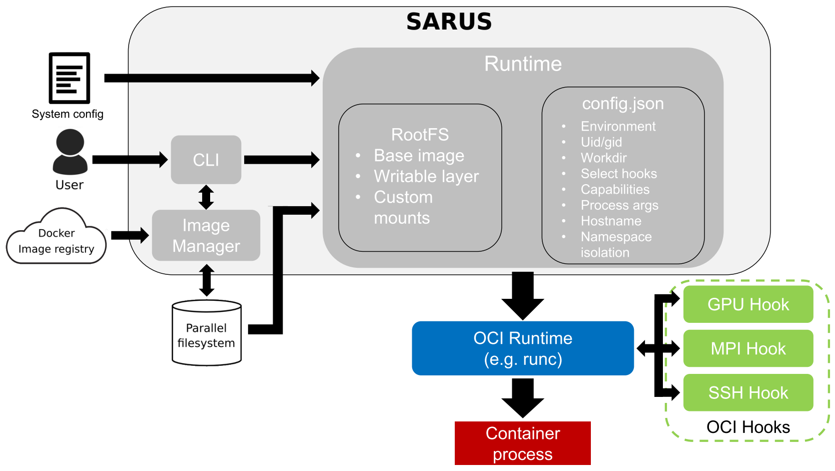

Figure 2: Sarus architecture

Reference

- Sarus - An OCI-compatible container engine for HPC

Requirements and Permissions

- Sarus is installed on the HPC and it requires root in order to set Sarus as a root-owned SUID program

Standards

- designed around the specifications of the OCI

Images

- Docker Images
- uses squashfs

Performance

- the presented results show no performance degradation when comparing the natively compiled versions with their containerized counterparts

Reference

- Sarus: Highly Scalable Docker Containers for HPC Systems

Security

- Sarus must run as a root-owned SUID executable and be able to achieve full root privileges to perform mounts and create namespaces
- Write/read permissions to the Sarus's centralized repository. The system administrator can configure the repository's location through the centralizedRepositoryDir entry in sarus.json
- Write/read permissions to the users' local image repositories. The system administrator can configure the repositories location through the localRepositoryBaseDir entry in sarus.json
- The configuration script needs to run with root privileges in order to set Sarus as a root-owned SUID program

Reference

- Review permissions required by Sarus

Available Tools

OCI Hooks:

- Native MPICH-Based MPI Support
- NVIDIA GPU Support
- Slurm Scheduler Synchronization

Reference

- Sarus: Highly Scalable Docker Containers for HPC Systems

Limits

// TODO

2.2 Conclusion

Docker was developed to answer the particular needs of web service applications, which feature substantially different workloads from the ones typically found in HPC. For this reason, deploying Docker in a production HPC environment encounters significant technical challenges, like missing integration with workload managers, missing support for diskless nodes, no support for kernel-bypassing devices (e.g. accelerators and NICs), no adequate parallel storage driver, and a security model unfit for multi-tenant systems.

A number of container platforms have thus been created to fill this gap and provide Linux containers according to the demands of HPC practitioners, retaining a varying degree of compatibility with Docker.

- Singularity is a container platform designed around an image format consisting of a single file. While it is able to convert Docker images through an import mechanism, it is centered on its own non-standard format and can also build its own images starting from a custom language. Support for specific features, e.g. OpenMPI and NVIDIA GPUs, is integrated into the Singularity codebase.
- Charliecloud is an extremely lightweight container solution focusing on unprivileged containers. It is able to import Docker images, requires no configuration effort from system administrator, and it is a user-space application, achieving high levels of security. Charliecloud's minimal feature set allows fine-grained control over the customization of the container. On the other hand, it requires a substantial level of proficiency from the user to set up containers for non-trivial use cases.
- Shifter is a software developed to run containers based on Docker images on HPC infrastructures. It is designed to integrate with the Docker workflow and features an image gateway service to pull images from Docker registries. Shifter has recently introduced the use of configurable software modules for improved flexibility. However, the modules and the executables have to be developed and configured specifically for Shifter.
- Sarus is a tool for HPC systems that instantiates feature-rich containers from Docker images. It fully manages the container life-cycle by: pulling images from registries, managing image storage, creating the container root filesystem, and configuring hooks and namespaces. For instantiating the containers, Sarus leverages on runc, an OCI-compliant kernel-level tool that was originally developed by Docker

Reference

- Sarus: Highly Scalable Docker Containers for HPC Systems

2.3 More information

- State of HPC containers

2.2 Comparing Solutions - IoT

Comparison

	Open Source	First Release	Last Release	Stars	Test	Issues (open/closed)	Link
Balena	Yes	Oct. 2015	Dec. 2019	471	Ok	37 / 49	https://github.com/balena-os/balena-engine

2.2.1 Balena

- https://www.balena.io/
- **Report**

Introduction

balenaEngine

balenaEngine is a new container engine purpose-built for embedded and IoT use cases and compatible with Docker containers. Based on Docker's Moby Project, balenaEngine supports container deltas for 10-70x more efficient bandwidth usage, has 3x smaller binaries, uses RAM and storage more conservatively, and focuses on atomicity and durability of container pulling.

Features:

- Small footprint: 3.5x smaller than Docker CE, packaged as a single binary
- Multi-arch support: Available for a wide variety of chipset architectures, supporting everything from tiny IoT devices to large industrial gateways
- True container deltas: Bandwidth-efficient updates with binary diffs, 10-70x smaller than pulling layers
- Minimal wear-and-tear: Extract layers as they arrive to prevent excessive writing to disk, protecting your storage from eventual corruption
- Failure-resistant pulls: Atomic and durable image pulls defend against partial container pulls in the event of power failure
- Conservative memory use: Prevents page cache thrashing during image pull, so your application runs undisturbed in low-memory situations

balenaOS

A bare-bones, Yocto Linux based host OS, which comes packaged with balenaEngine

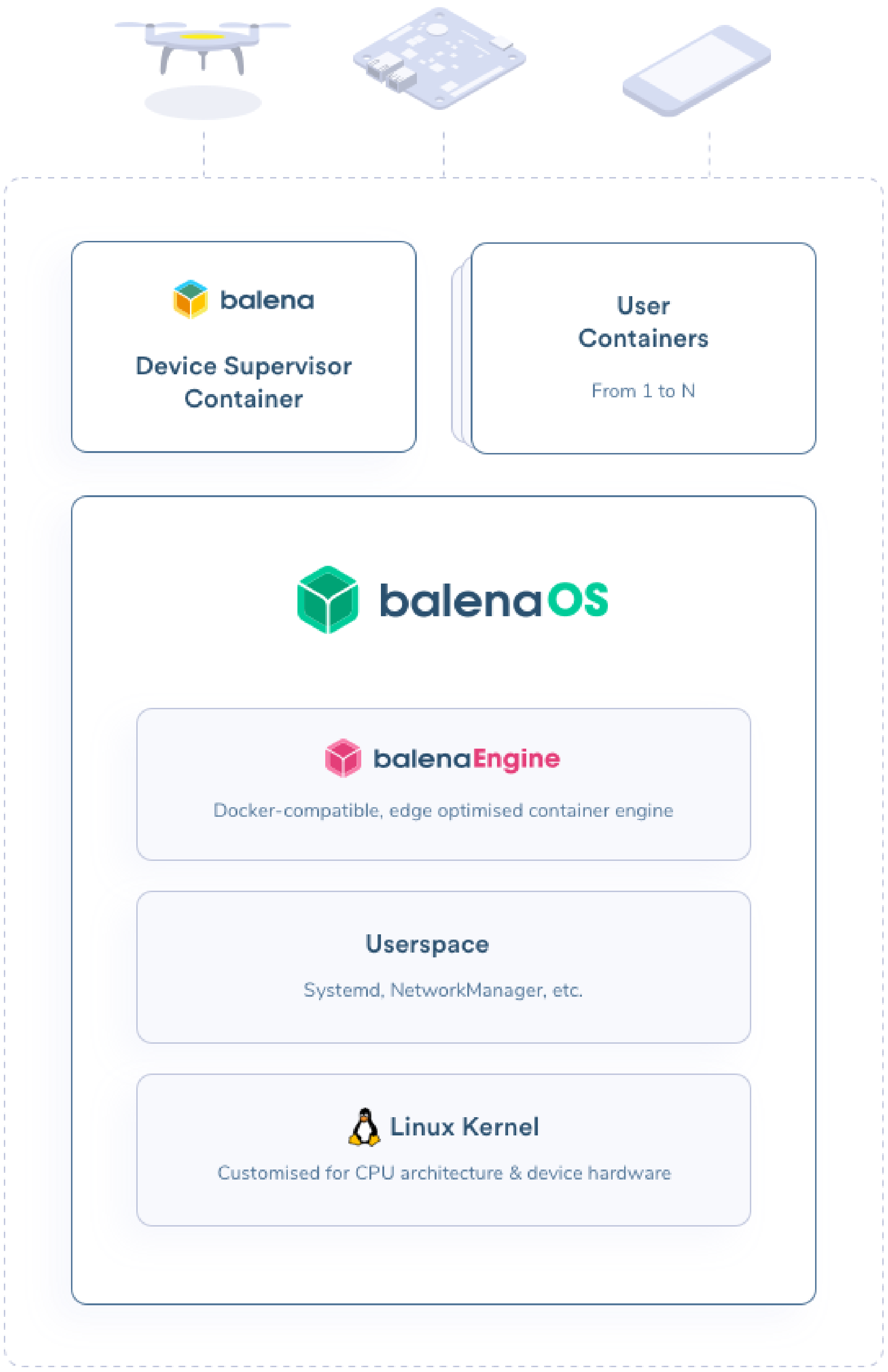
balena
Device Supervisor Container
User Containers
From 1 to N
balenaOS
balenaEngine
Docker-compatible, edge optimised container engine
Userspace
Systemd, NetworkManager, etc.
Linux Kernel
Customised for CPU architecture & device hardware

Figure 3: balena Ecosystem

- The host OS is responsible for kicking off the device supervisor, balena's agent on your device, as well as your containerized services. Within each service's container you can specify a base OS, which can come from any existing Docker base image that is compatible with your device architecture.
- The base OS shares a kernel with the host OS, but otherwise works independently. If you choose, your containers can be configured to run as privileged, access hardware directly, and even inject modules into the kernel.
- The balena device supervisor runs in its own container, which allows us to continue running and pulling new code even if your application crashes.

balenaCloud

The core balena platform encompasses device, server, and client-side software, all designed to get your code securely deployed to a fleet of devices

Reference

- balenaEngine
- What is balena

Requirements and Permissions

Compatibility:

- aarch64
- amd64
- armv5e
- armv6l
- armv7hf
- i386

Reference

- balenaEngine

Standards

- balenaEngine will be versioned with the equivalent version of Docker it's closest to

Images

- Docker Images that are compatible with the device architecture
- Container deltas: balenaEngine comes with support for container deltas, a way of computing a binary description of what changed between two images. A delta can be pushed to the standard docker registry, it has the same format as a docker image

Reference

- Container deltas

Performance

- Technical comparison
- Delta updates allow up to 70x improvement
- Binaries are on average 3.5x smaller
- page cache usage is minimal (with the downside of a slower pull)

Security

BalenaOS is a thin Linux environment that supports the balena services and user application containers. BalenaOS is built using Yocto Linux, the de facto standard for building lightweight embedded Linux environments. Using Yocto allows balena to build images that contain no unused or unnecessary code in either userspace or the running kernel, minimizing the device's available attack surface. All balena software running on devices is 100% open source and can be independently audited and verified.

- Device access: every device has an API key that can be used to read information about the device or the application the device is associated with and to allow a device to request an application update
- Runtime management: Balena uses OpenVPN to control the device state (e.g. device reboot, device shutdown, application restart, etc.). Devices only connect outbound to the VPN and all traffic over the VPN is encrypted with TLS. When the VPN is enabled, SSH access is available to the application container. Balena VPN disallows device-to-device traffic and prohibits outbound traffic to the Internet

Reference

- Balena security

Available Tools

- balenaOS
- balenaCloud

Limits

- balenaEngine provides a poor documentation while a good documentation is available for the balena ecosystem
- Some Docker features left out (most needed in cloud deployments and therefore not warranting inclusion in a lightweight IoT-focused container engine):
  - Docker Swarm
  - Cloud logging drivers
  - Plugin support

  - Overlay networking drivers
  - Non-boltdb backed stores (consul, zookeeper, etcd, etc.)

2.3 Comparing Solutions - Unikernel

Comparison

	Open Source	First Release	Last Release	Stars	Test	Issues (open/closed)	Link
Firecracker	Yes	Oct. 2017	Mar. 2021	11.7k	Ok	190 / 631	https://github.com/firecracker-microvm/firecracker
OSv	Yes	Dec. 2012	Set. 2019	3k	Not Ok	313 / 694	https://github.com/cloudius-systems/osv
MirageOS	Yes	Feb. 2013	Mar. 2021	1.4k	Ok	79 / 305	https://github.com/mirage/mirage
Kata Containers	Yes	Dec. 2017	Mar. 2021	421	-	40 / 69	https://github.com/kata-containers/kata-containers

2.3.1 Firecracker

- https://firecracker-microvm.github.io/
- **Report**

Introduction

Firecracker is an open source virtualization technology that is purpose-built for creating and managing secure, multi-tenant container and function-based services.

Until now, you needed to choose between containers with fast startup times and high density, or VMs with strong hardware-virtualization-based security and workload isolation. With Firecracker, you no longer have to choose. Firecracker enables you to deploy workloads in lightweight virtual machines, called microVMs, which provide enhanced security and workload isolation over traditional VMs, while enabling the speed and resource efficiency of containers. Firecracker was developed at Amazon Web Services to improve the customer experience of services like AWS Lambda and AWS Fargate.

Firecracker is a virtual machine monitor (VMM) that uses the Linux Kernel-based Virtual Machine (KVM) to create and manage microVMs. Firecracker has a minimalist design. It excludes unnecessary devices and guest functionality to reduce the memory footprint and attack surface area of each microVM. This improves security, decreases the startup time, and increases hardware utilization.

Firecracker consists of a single micro Virtual Machine Manager process that exposes an API endpoint to the host once started

Firecracker is already powering multiple high-volume AWS services including AWS Lambda and AWS Fargate.

Requirements and Permissions

- Firecracker currently supports Intel CPUs, with AMD and Arm support in developer preview
- Firecracker supports Linux host and guest operating systems with kernel versions 4.14 and above, as well as OSv guests
- KVM
- < 125 ms startup time
- < 5 MiB memory footprint per microVM

Standards

- None

Images

Firecracker launches microVMs and is not responsible for images

Performance

// TODO

Security

- Firecracker microVMs use KVM-based virtualizations that provide enhanced security over traditional VMs. This ensures that workloads from different end customers can run safely on the same machine. Firecracker also implements a minimal device model that excludes all non-essential functionality and reduces the attack surface area of the microVM
- Simple Guest Model: Firecracker guests are presented with a very simple virtualized device model in order to minimize the attack surface: a network device, a block I/O device, a Programmable Interval Timer, the KVM clock, a serial console, and a partial keyboard (just enough to allow the VM to be reset).
- Process Jail: the Firecracker process is jailed using cgroups and seccomp BPF, and has access to a small, tightly controlled list of system calls.
- Static Linking: the firecracker process is statically linked, and can be launched from a jailer to ensure that the host environment is as safe and clean as possible
- Firecracker also ships with a separate jailer used to reduce the blast radius of a compromised VMM process. The jailer isolates the VMM in a chroot, in its own namespaces, and imposes a tight seccomp filter. The filter whitelists system calls by number and optionally limits system-call arguments, such as limiting ioctl() commands to the necessary KVM calls.

Reference

- Firecracker – Lightweight Virtualization for Serverless Computing
- The Firecracker virtual machine monitor

Available Tools

- Firecracker is integrated with Kata Containers
- Firecracker is integrated with Weave FireKube (via Weave Ignite)
- Firecracker is integrated with containerd via firecracker-containerd
- It's also available within the UniK unikernel and microVM platform
- Firecracker can run Linux and OSv guests

Limits

// TODO

2.3.2 OSv

- http://osv.io/
- **Report**

Introduction

OSv is the versatile modular unikernel designed to run unmodified Linux applications securely on micro-VMs in the cloud. Built from the ground up for effortless deployment and management of micro-services and serverless apps, with superior performance.

Today's cloud-based applications run a heavyweight stack: the hypervisor, which divides the hardware resources among virtual machines; the operating system, which divides the virtual machine's resources among applications; and the application server, which divides the application's resources among the end users.

Clearly, there is a lot of duplication going on. Each layer adds its own overhead in an attempt to abstract away and hide the problems caused by the lower layer. The result is inefficient and complex.

Enter OSv - the operating system for the cloud. On the one hand, designed and optimized to run on a hypervisor. On the other hand, designed to run an application stack without getting in the way. Designed for the cloud.

Features:

- Single address space, ring 0
- Java Virtual Machine integration
- lock-free algorithms
- Interrupt handling in threads
- Van Jacobson style network stack

- State of the art file system

Requirements and Permissions

OSv gives you low overhead and rapid turnaround, like containers, but with the deployment flexiblity you only get from true virtualization. With the Capstan build tool, you can build and run with one command, but create a complete virtual machine that will run on your existing cloud environment. Creating a VM image adds only 6-7MB of overhead, three seconds of build time, and a few lines of configuration

OSv supports many managed language runtimes including unmodified JVM, Python 2 and 3, Node.JS, Ruby, Erlang as well as languages compiling directly to native machine code like Golang and Rust.

Limited hardware support: Xen, KVM, and Vmware.

Reference

- http://osv.io/
- http://osv.io/technology

Standards

- none

Images

ISVs who offer a packaged application as a virtual machine image can benefit in several ways from releasing on OSv. The VM images for OSv-based virtual appliances are small, often only 6-7MB larger than the application itself. And the ISV does not need to maintain and support the large set of software and configuration required by even the simplest guest images on other platforms.

Performance

Benchmark for redis and memcached available

Reference

- http://osv.io/benchmarks

Security

Unikernels reduce the attack surface due to the self-containedenvironment only enabling low-level features

Reference

- A Security Perspective on Unikernels

Available Tools

- Capstan is a command-line tool for rapidly running your application on OSv unikernel. It focuses on improving user experience when building the unikernel and attempts to support not only a variety of runtimes (C, C++, Java, Node.js etc.), but also a variety of ready-to-run applications (Hadoop HDFS, MySQL, SimpleFOAM etc.).
- available on Amazon EC2 and Google GCE

Limits

// TODO

2.3.3 MirageOS

- https://mirage.io/
- **Report**

Introduction

MirageOS is a library operating system that constructs unikernels for secure, high-performance network applications across a variety of cloud computing and mobile platforms. Code can be developed on a normal OS such as Linux or MacOS X, and then compiled into a fully-standalone, specialised unikernel that runs under a Xen or KVM hypervisor.

This lets your services run more efficiently, securely and with finer control than with a full conventional software stack.

MirageOS uses the OCaml language, with libraries that provide networking, storage and concurrency support that work under Unix during development, but become operating system drivers when being compiled for production deployment. The framework is fully event-driven, with no support for preemptive threading.

Reference

- https://mirage.io/wiki/technical-background

Requirements and Permissions

- a working OCaml compiler (4.05.0 or higher).
- the OPAM source package manager (2.0.0 or higher).
- an x86_64 or armel Linux host to compile Xen kernels

Reference

- https://github.com/mirage/mirage

Standards

Images

MirageOS is a 'library operating system' for constructing secure, high-performance network applications across a variety of cloud computing and mobile platforms. It works by treating the Xen hypervisor as a stable hardware platform, allowing us to focus on high-performance protocol implementations without worrying about having to support the thousands of device drivers found in a traditional OS.

Performance

Security

- the current release contains clean-slate libraries for TLS, TCP/IP, DNS, Xen network and storage device drivers], HTTP, and other common Internet protocols, but all written in a completely type-safe fashion so that they are resistant to attacks such as buffer overflows

Reference

- faq

Available Tools

- mirage cli

Limits

- the unikernel creation process is quite complex
- MirageOS is based around the OCaml language

2.3.4 Kata Containers

- https://katacontainers.io/

Introduction

Kata Containers is an open source community working to build a secure container runtime with lightweight virtual machines that feel and perform like containers, but provide stronger workload isolation using hardware virtualization technology as a second layer of defense.

This is because the traditional OCI runtime such as(runC) relies on Linux kernel features, such as cgroups and namespaces to provide isolation when spawning containers. As a result, containers share the same kernel which is usually considered less secure than using traditional virtualisation. In order to deal with the aforementioned challenges the Kata Containers project has been founded.

Reference

- What is Kata Containers

Requirements and Permissions

- Kata Containers currently works on systems supporting the following technologies:
  - Intel VT-x technology.
  - ARM Hyp mode (virtualization extension).
  - IBM Power Systems.
  - IBM Z mainframes.
- root permission is needed to check if the system is capable of running Kata containers
- the runtime has a built-in command to determine if your host system is capable of running and creating a Kata Container

Standards

- OCI container format
- Kubernetes CRI interface
- legacy virtualization technologies

Images

kata-runtime creates a QEMU/KVM virtual machine for each container or pod, the Docker engine or kubelet (Kubernetes) creates respectively

Performance

Delivers consistent performance as standard Linux containers; increased isolation without the performance tax of standard virtual machines.

Security

Runs in a dedicated kernel, providing isolation of network, I/O and memory and can utilize hardware-enforced isolation with virtualization VT extensions.

Available Tools

- GPU support
- Docker support
- Kubernetes support

Limits

List of limitations available here