# VM-based Containers

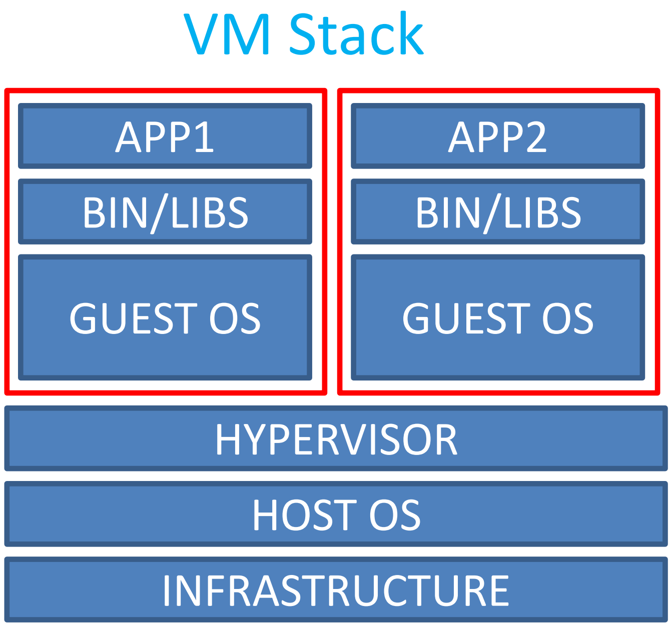
Wei Zhang
zhangwei555@huawei.com

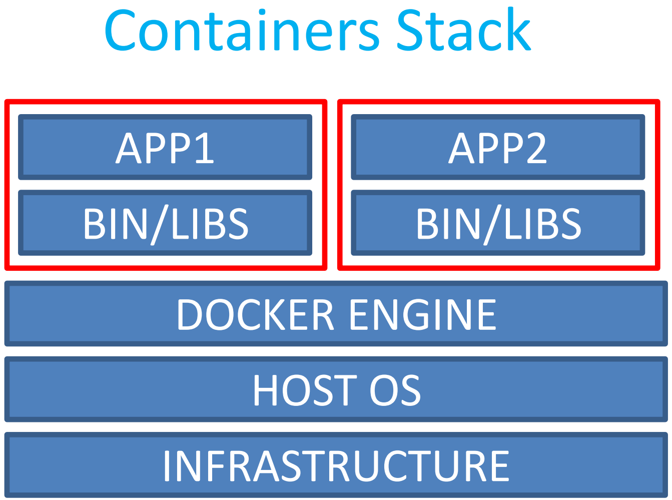
Claudio Fontana
claudio.fontana@huawei.com

# Who we are

- Wei Zhang – Beijing Huawei R&D office, working in the Containers team. In this project dealing mainly with the implementation of the VM-based container design and docker tooling integration.
- Claudio Fontana – Munich R&D office, working in the OS and virtualization team. In this project dealing mainly with the virtualization support to the project.

# Traditional comparison of Containers vs VMs




- Ease of Development/Deployment
- High performance, low overhead
- Huge ecosystem of ready-to-use components

# Problem: native containers and third party code

- Running third party code on infrastructure will introduce security concerns
- Example: Public Cloud, Telecom use cases

Need for strong isolation and security

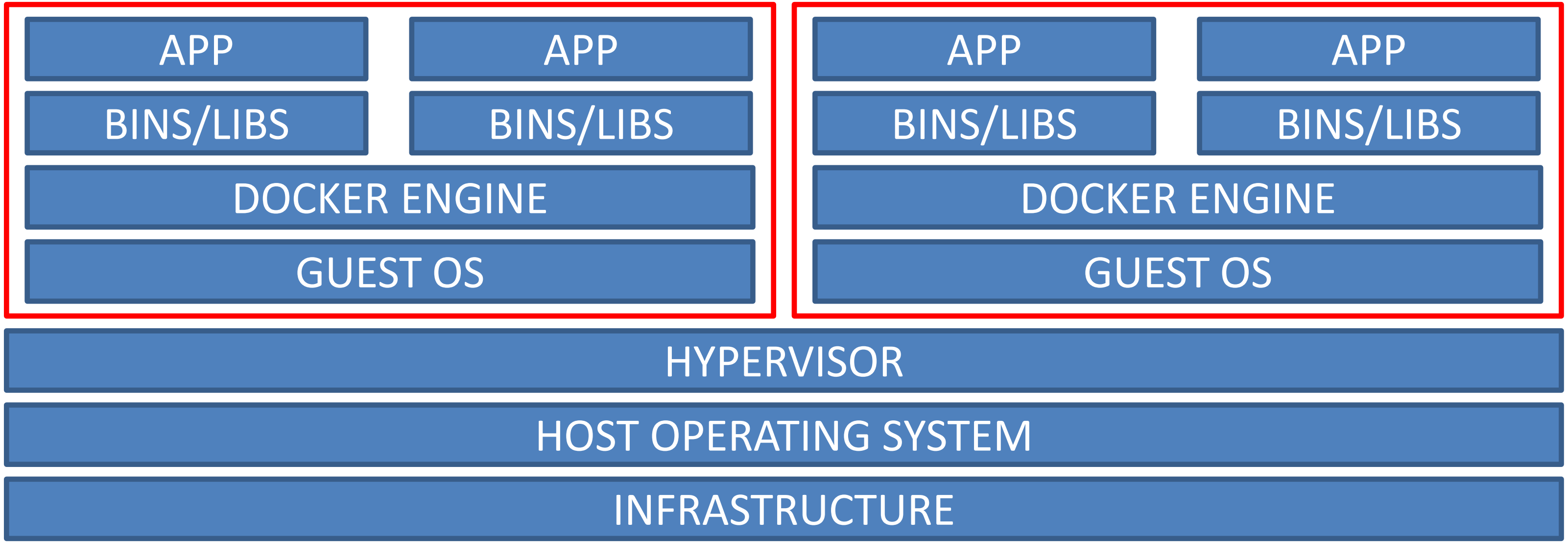
# Security features supported by docker with native containers

- Shrink attack surface:
  - Capability: restrict capabilities of process in container
  - Seccomp: filter access to syscall, forbid dangerous/unnecessary syscall inside containers
  - SElinux: customize privileges for processes, users and files.
  - User namespace: map root user in container to non-root user on host, limit privileges of users in containers
- Isolation enhancements:
  - Fuse: isolate “/proc”, useful for container resource monitoring system.

# Need for more secure architecture

- Attack surface is still too large
- A single bug in the kernel can allow escape to the host

# Actual Container use for third party code




This stack again adds overheads and sacrifices ease of deployment for the sake of security

# What If a VM would…

- Boot almost as fast as native containers
- Consume fewer hardware resources
- Be invisible to the user

and at the same time…

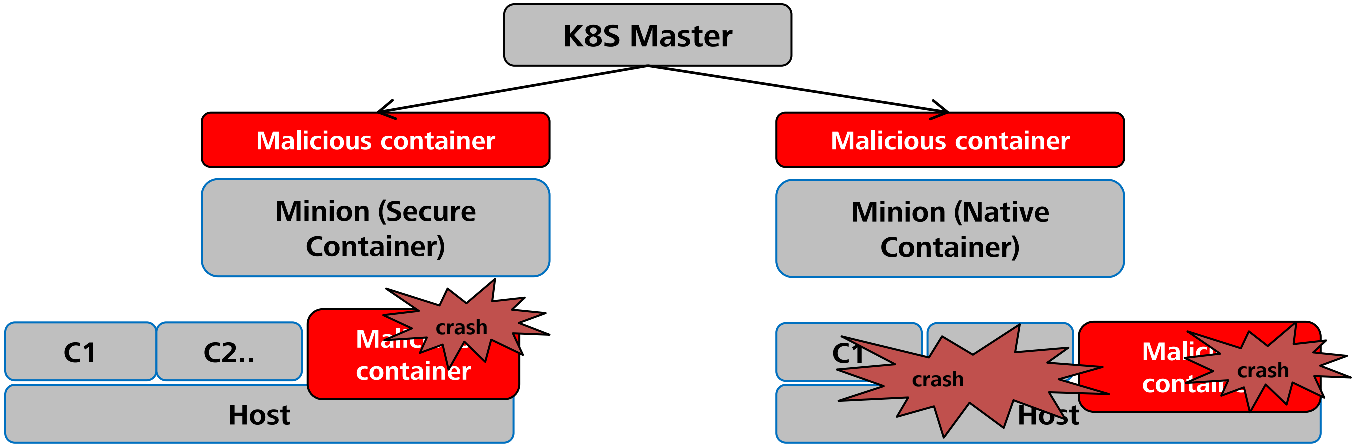
- run sandboxed containers using the normal docker tools
- be compatible with docker API and prebuilt container images
- interact with all high level tools from the container ecosystem (K8S, mesos …) without additional modifications

# What we have created

A container solution based on lightweight VMs called uVM (**microVM**) designed to be controlled by frameworks.

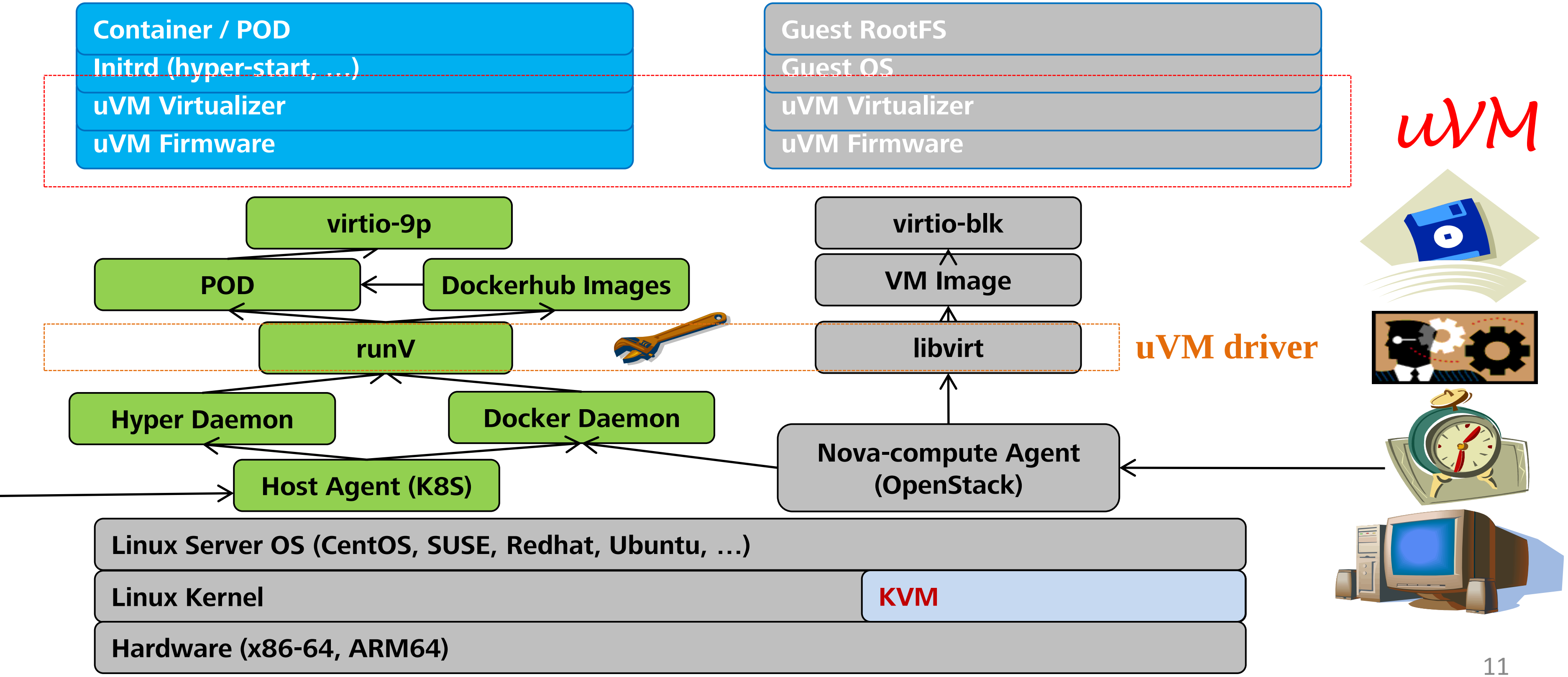
- Integration with docker based on “runV” – OCI compatible runtime created by Hyper
- Integration with lightweight QEMU VM

# Guest OS creates a sandbox for Containers to run in



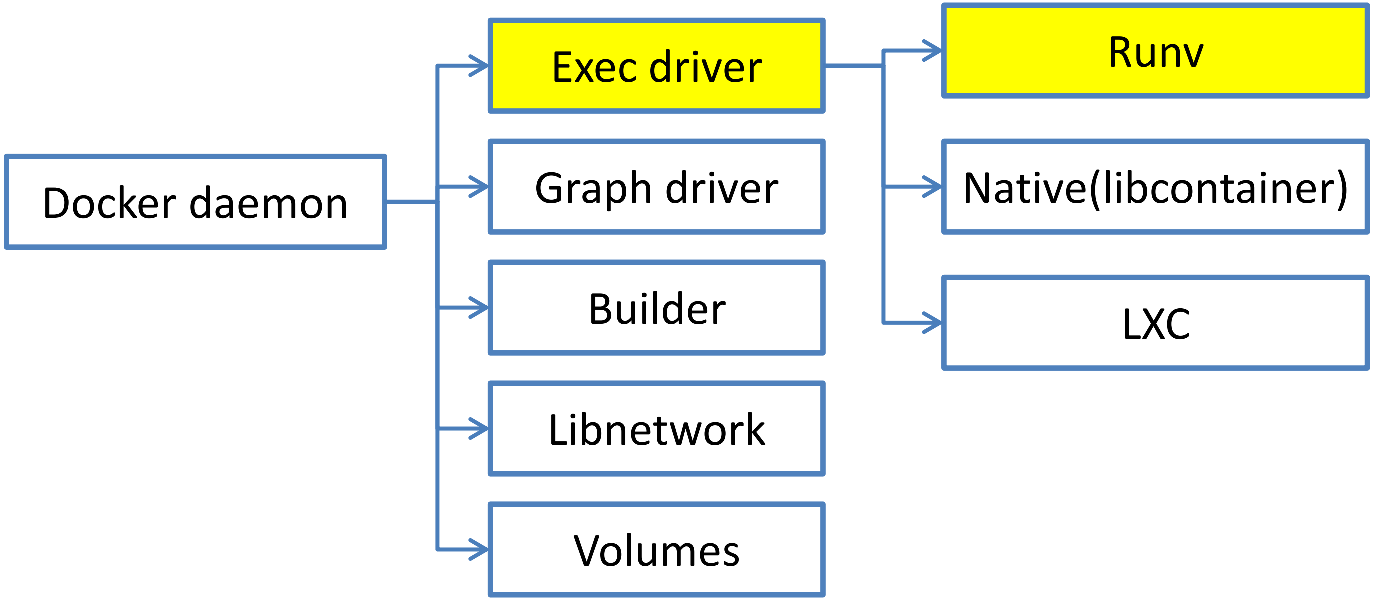
# Architecture

*for Docker Containers*



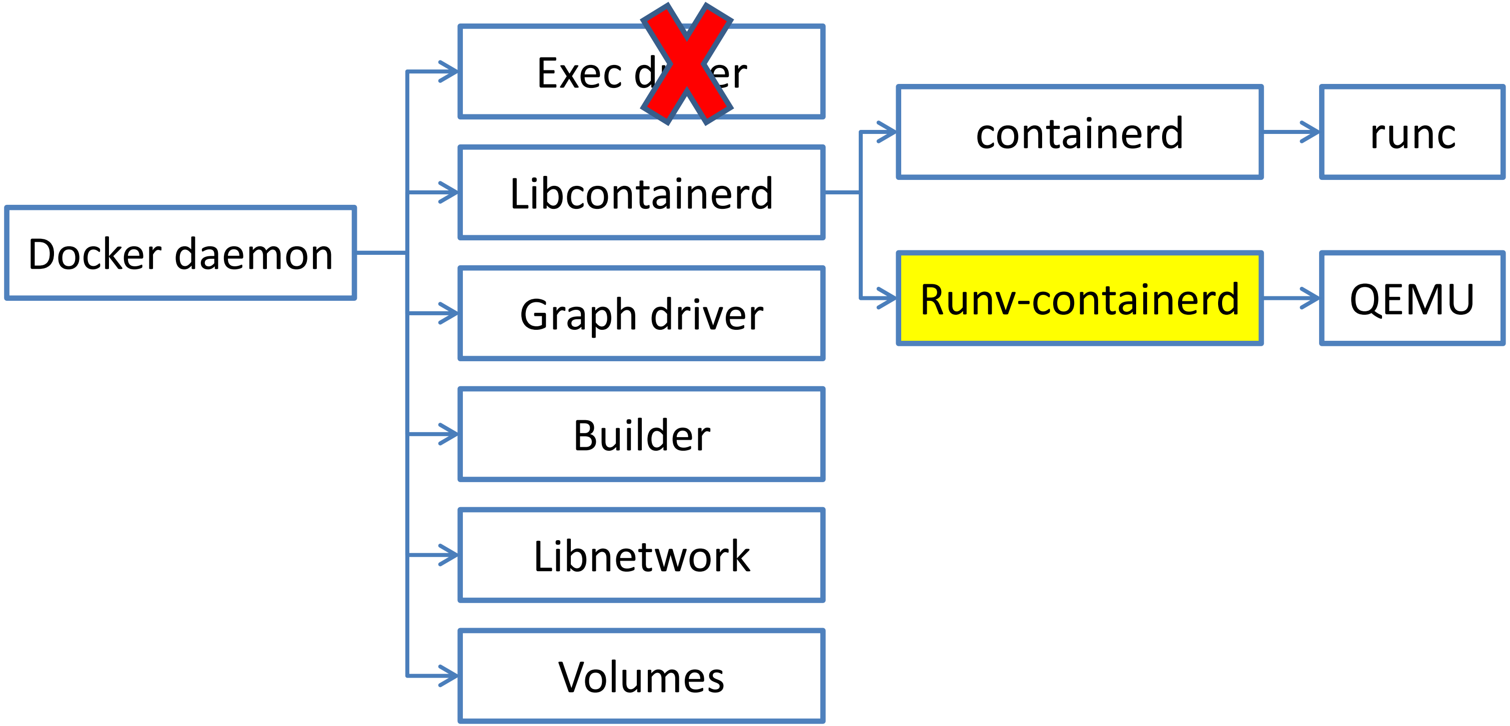
# Secure Container Evolution

- Before docker 1.11.0 (2016-04-13)



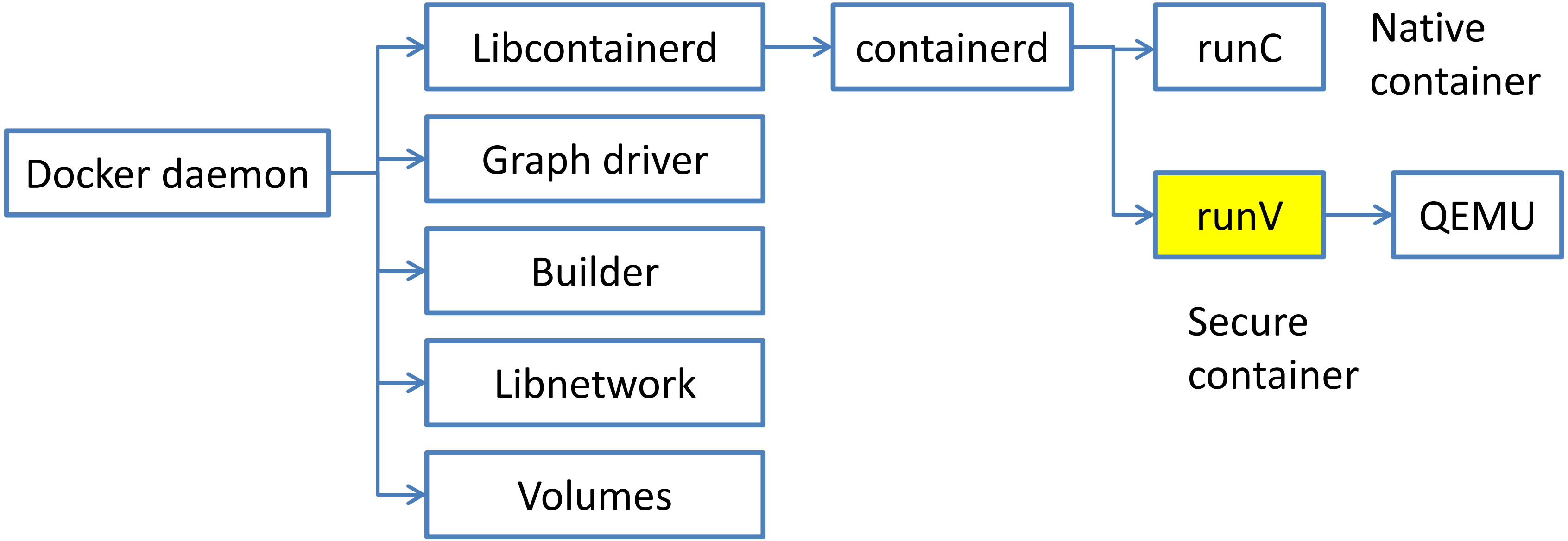
# Secure Container Evolution

- After containerd/runc introduced



# Secure Container Evolution

- Next step...



# Next step: use with docker

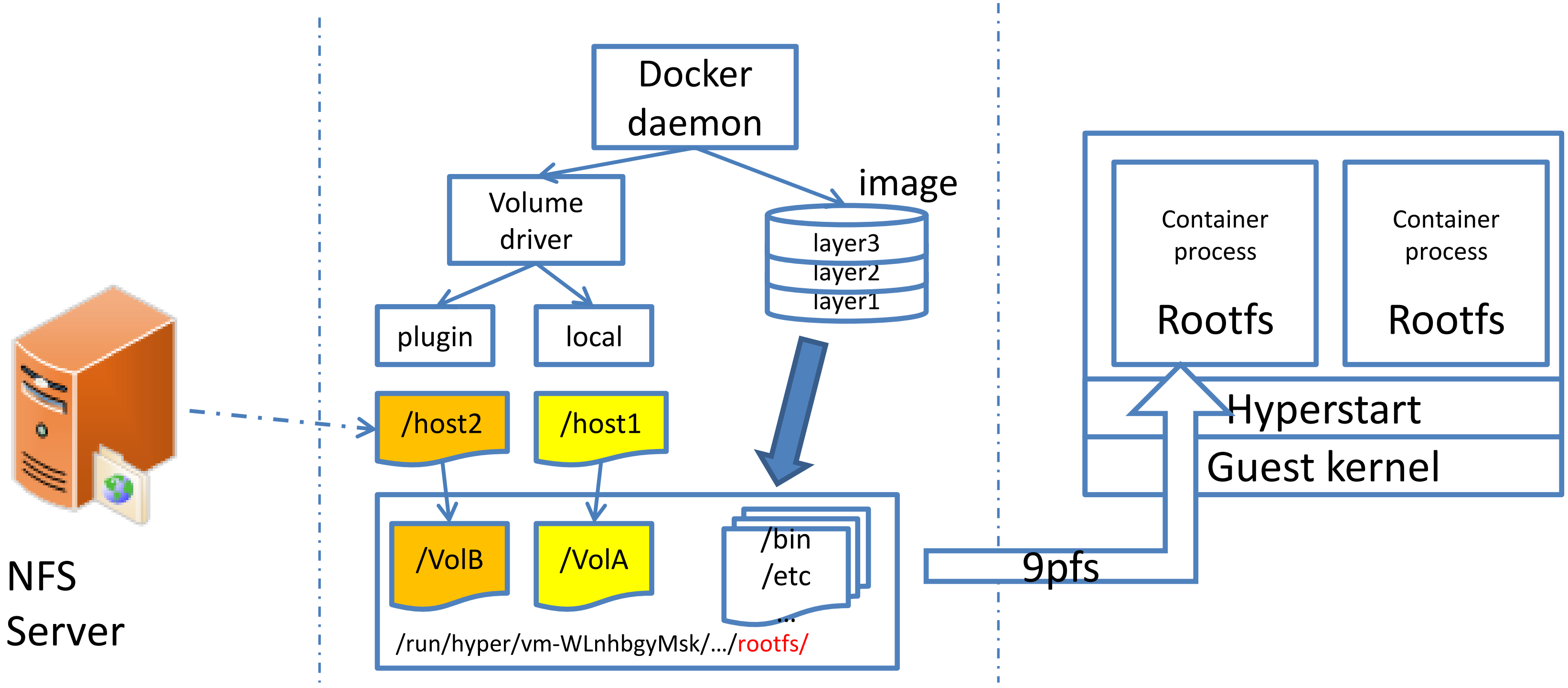
- Example usage:
  - # dockerd --add-runtime “runv” –runtime-args “--debug” …
  - # docker run --runtime “runv” -ti busybox top

- still needs better integration with K8S!
- Docker 1.12+ only

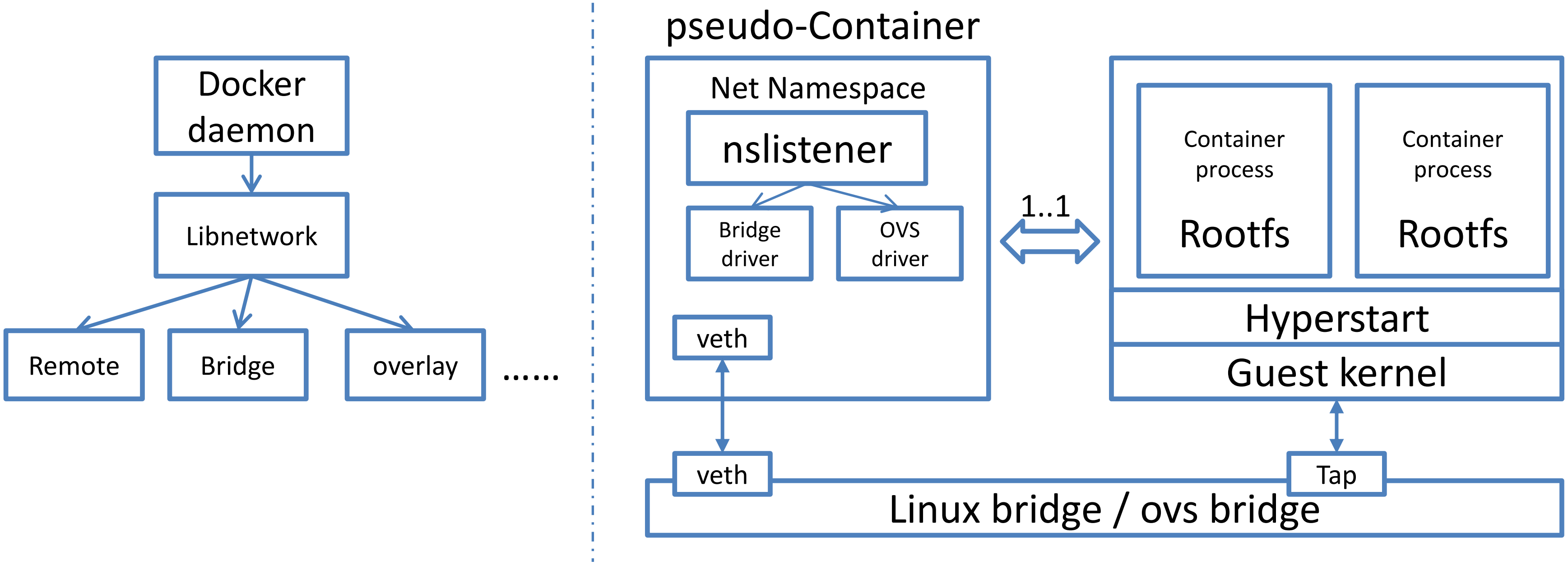
# Runtime integration Pros and Cons

- Pros:
  - Match perfectly docker's current architecture and roadmap.
  - Following OCI standard makes runV easily accepted.
- Cons:
  - RunV has to follow runC's command line API closely.
  - Standard is lagging behind runC, which is still changing quickly.
  - No path for backward compatibility until more mature standards are available.

# Volume Management



# Networking


Docker daemon
Libnetwork
Remote
Bridge
overlay
......
pseudo-Container
Net Namespace
nslistener
Bridge driver
OVS driver
veth
1..1
Container process
Rootfs
Container process
Rootfs
Hyperstart
Guest kernel
veth
Tap
Linux bridge / ovs bridge

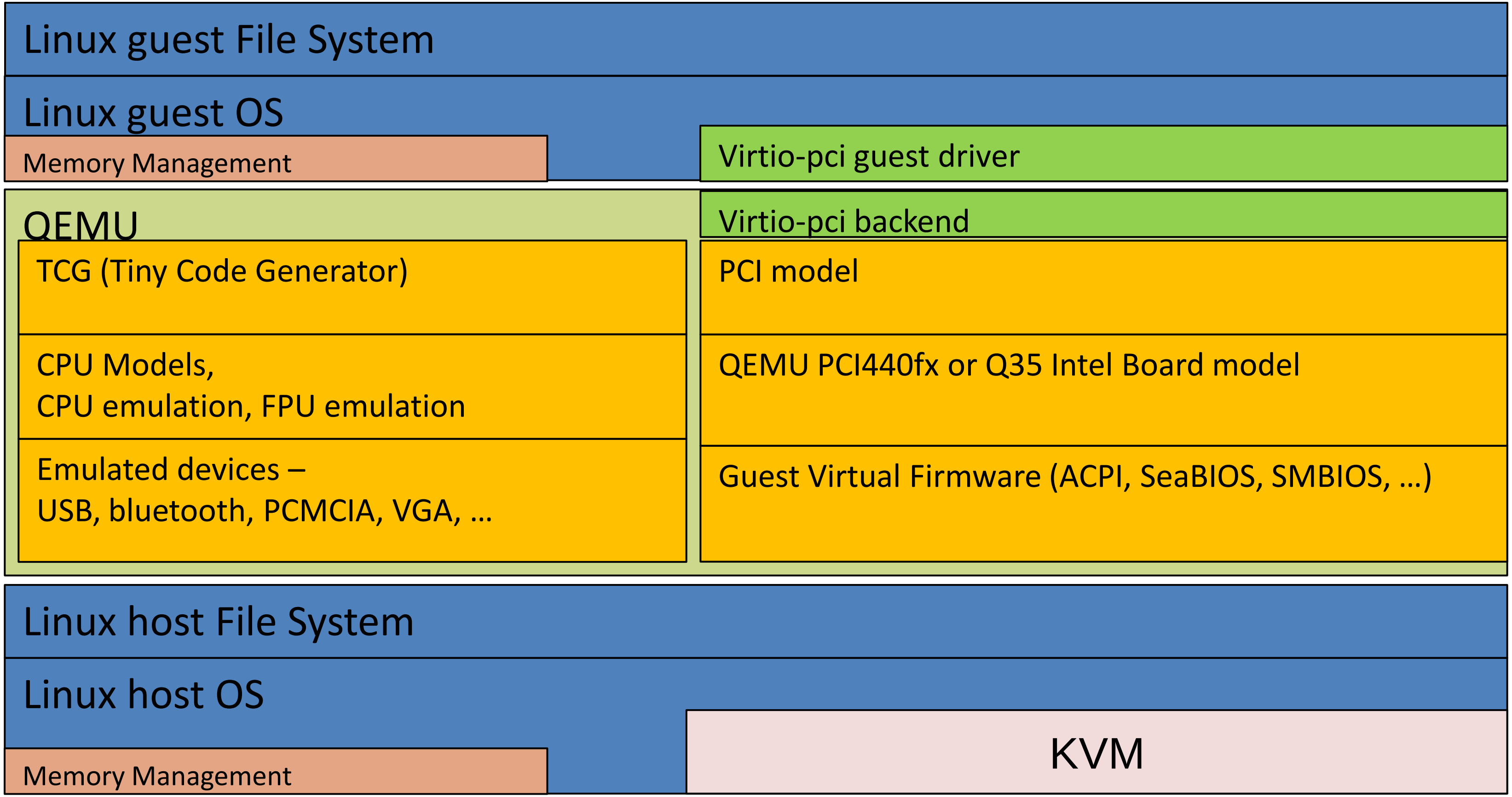
# More features

- Use a custom guest kernel
- Resource QoS throttling [cpu, memory, storage, network]
  - VM level Resource QoS (with qemu)
  - Container level Resource QoS (with cgroups, tc, …)
- Status, monitoring …

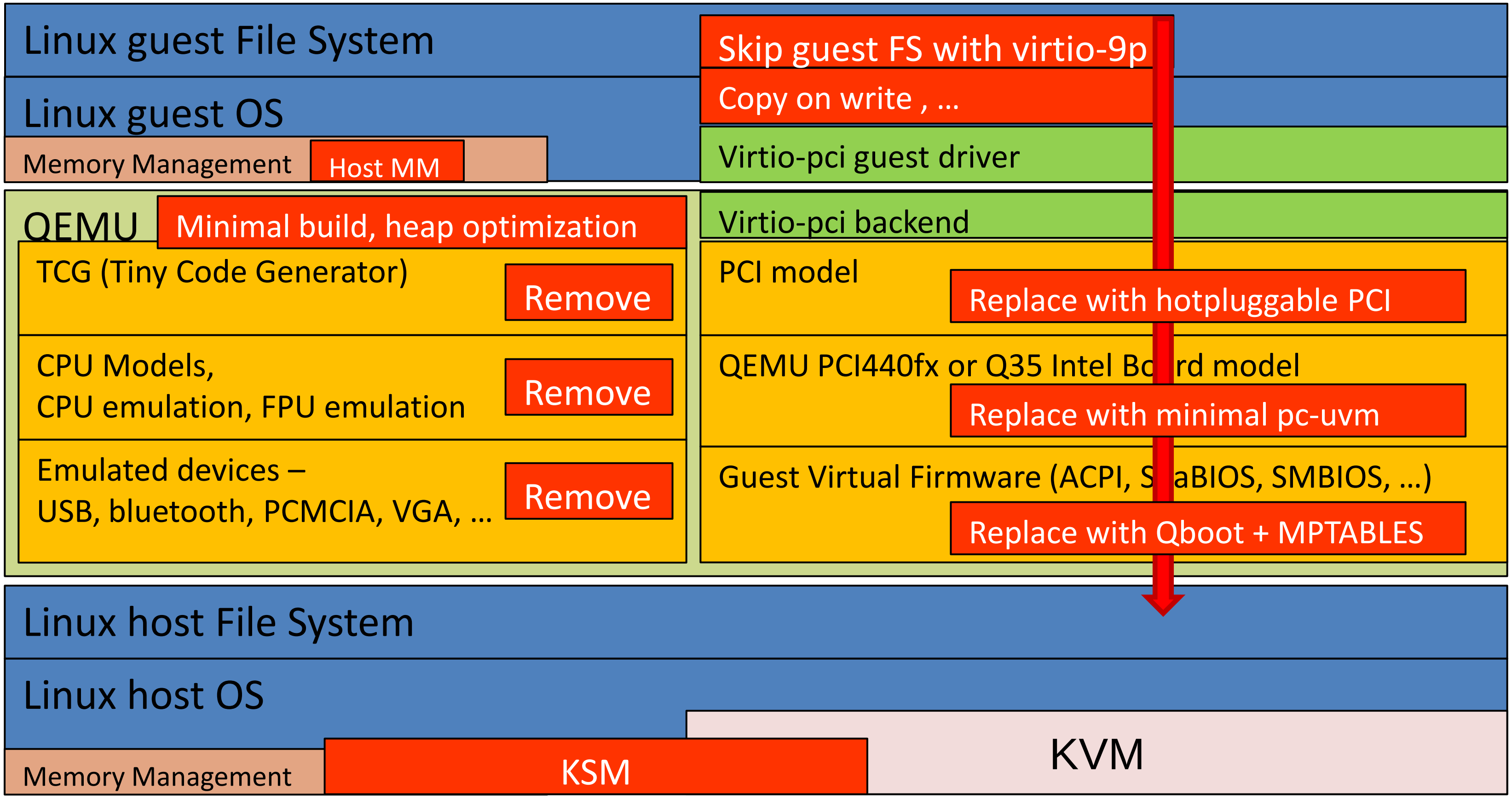
# Virtualization support (“uVM”)

To support the Secure Container use case we need changes in the Virtualization stack!

# Current KVM stack for x86 Linux Server Virtualization



# Current KVM stack for x86 Linux Server Virtualization



# Result: a VM built for Containers

1. Boot time on a spinning disk with Xeon platform is around 0.1s from uVM start of QEMU process to guest application – Enough for now

2. 20MB directly cut from the memory overhead of QEMU, plus proportional improvements per VM (PSS), KSM for long term saves with minimal cpu investment.

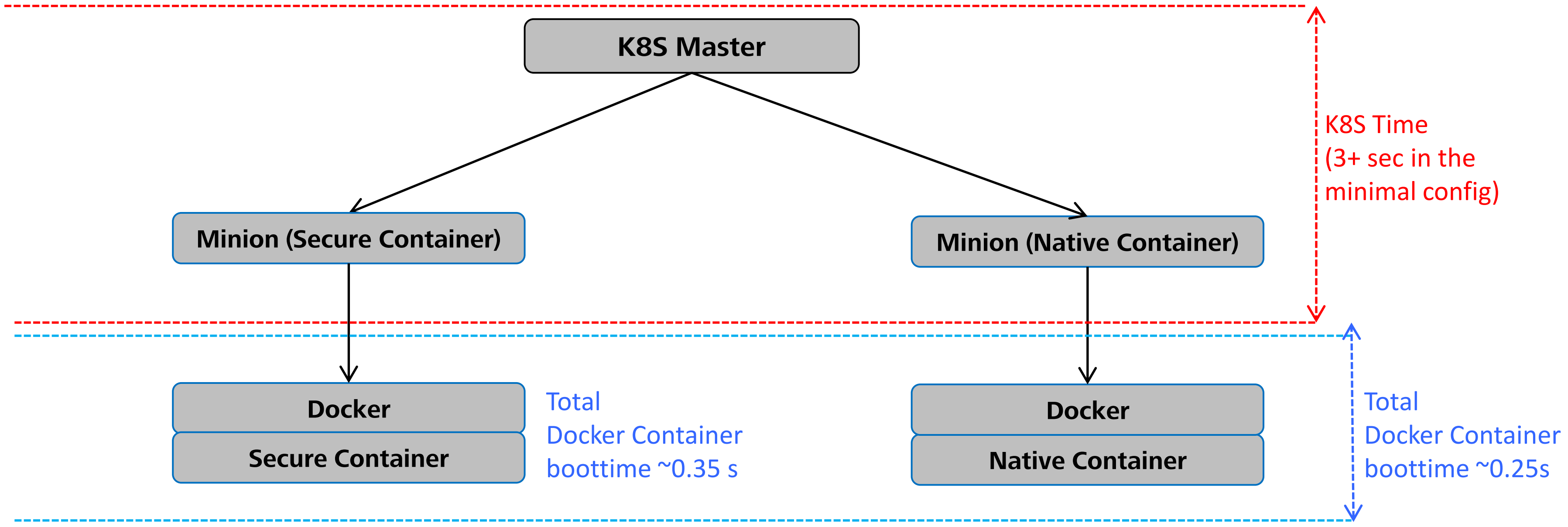
   Working on Copy on Write kernel and initrd (X86 and ARM, no ACPI)
   Exploring QEMU process data segments copy on write, [...]

3. Cpu and memory performance benchmarks show no negative impact of the changes.

4. Virtio 9p performance improvement: 3x speed improvement on both large and small blocks operations.

# Container Boottime costs

Kubernetes, Docker, Virtualization impact on boottime.
Probably need to look at the Orchestration now!
3+ seconds even in the minimal config until the container is scheduled to run.



# Specialization tradeoffs

These results are possible also because some of the software components of a KVM stack are actually unused for running modern Container services.

Part of the reason is also historical: the QEMU virtualizer has been actually designed originally for software modeling, with the goal to model physical hardware in software.

- QEMU board model, emulated devices
- Firmware

**Accurate modeling of the physical hardware, run any possible OS, QEMU is self contained**

**VS**

**Running workloads controlled by frameworks as efficiently as possible**

# uVM components summary

- **uVM Firmware:** uses Paolo's Qboot + simple MPTABLE patch for SMP. Easier to use and modify than SeaBIOS.
  Qboot, kernel, hyperstart-initrd built together as a "firmware".

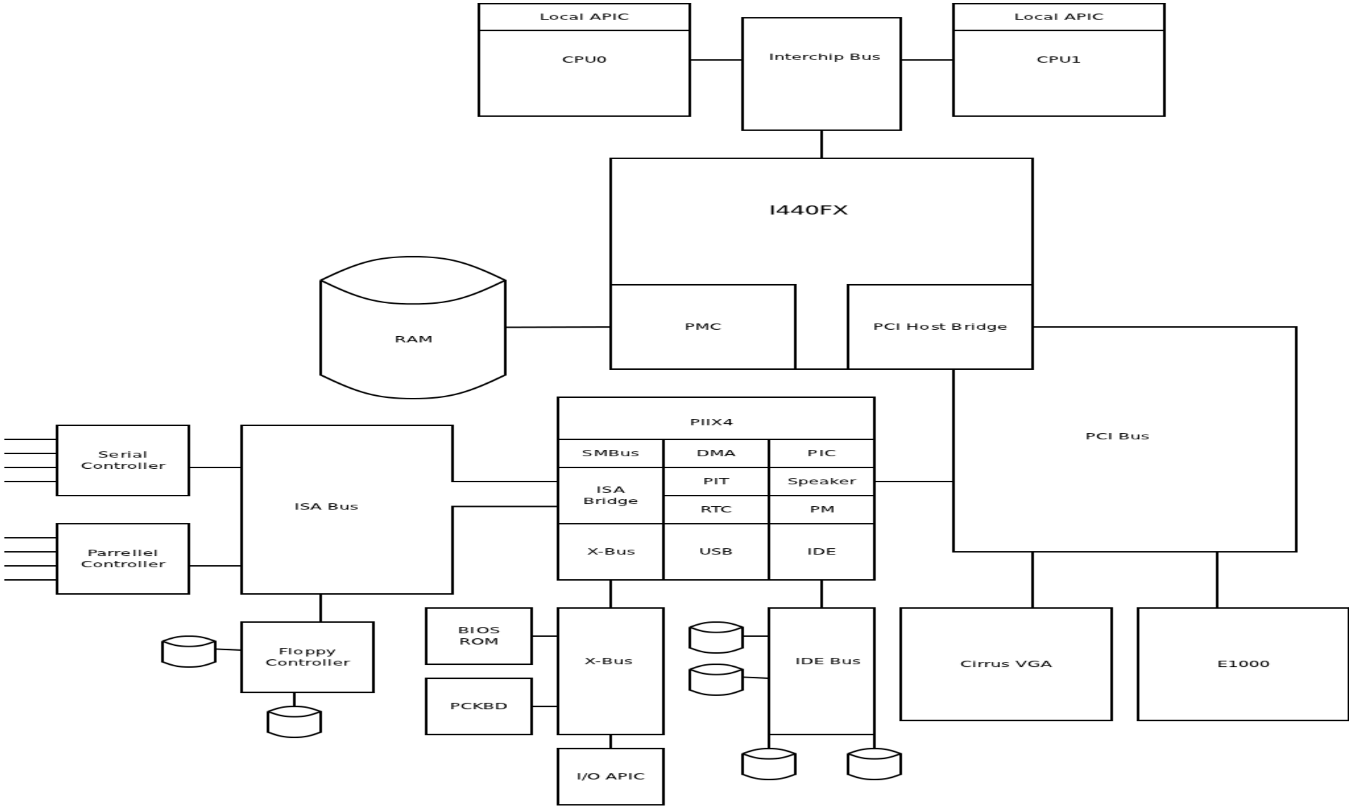
- **uVM QEMU:** implements a new board model and new features

- **uVM Linux:** guest patches
  * fastboot
  * smp
  * performance

- X86-64 and ARM-64 support

# uVM x86 Board simplification

The Intel PCI-440fx has been used as the starting point for the uvm x86 board model.



# uVM Board simplification

"Removed" many components, which means either a device config
(which is now considered for real), or an additional configure option or configure option fix.
==> minimal build: QEMU = 3 MB binary vs usual 40MB binary (*Note).

| Action | Items |
|---|---|
| Remove | •ISA-DMA and other ISA devices. Just keep 1 serial, no parallel, no VGA, no floppies, etc.<br>•Power Management, ACPI and ACPI hot-plug<br>•SMBIOS, SMRAM and PCI-PAM<br>•TCG, Replay, Disassembly, Non-KVM CPU models<br>•PCMCIA, USB, BT, I2C |
| Add | •new uVM default config<br>•pflash boot device cmdline support<br>•virtio-9p and virtio-net I/O bandwidth and iops limits, optimized memory footprint<br>•framework-controlled hotplug (Controlled by docker, RunV and guest init) |

# Linux host and guest OS

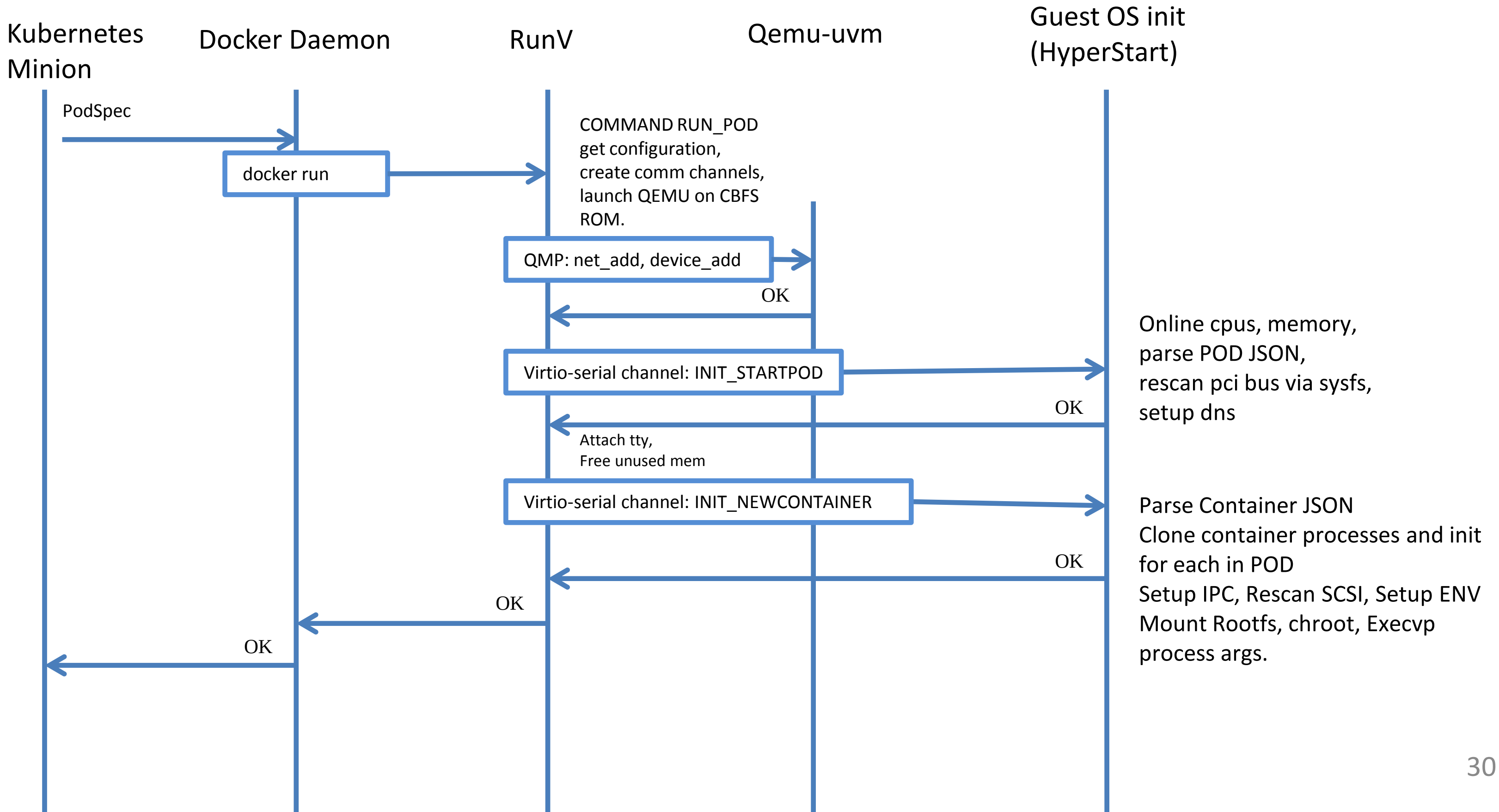
The Linux Host requirements: KVM, KSM, 4.1+

Tested with all kinds of generally available Linux-based Server OSes.

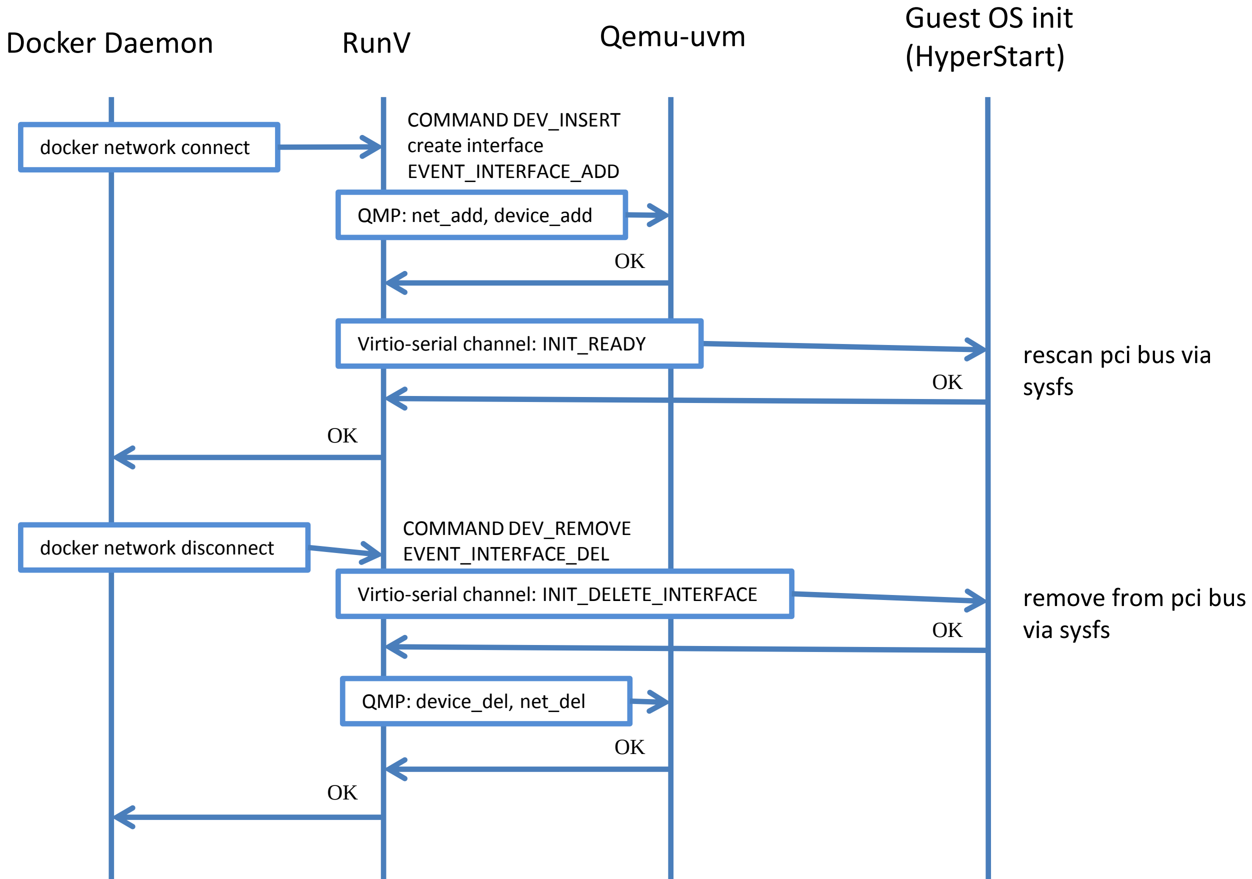
Guest OS is comprised of a optimized guest kernel and a simplified OS included in an init derived from Hyperstart, which acts as an “agent” inside the VM to do the will of the framework controlling the VM.

-9pfs optimizations for large chunks, adding layer to v9fs writeback path to minimize number of 9p messages exchanged
-Removed bottlenecks from 9pfs to solve small chunks terrible performance
-9pfs optimizations for memory overhead
-allow SMP from cmdline params (no dep on BIOS or ACPI).

# Example flow: container create



# Example flow: net hotplug



# Upstream plans

- Full solution is started as internal project
- Started evaluations for production use
- Specific features are being contributed upstream

# QEMU upstreaming

- Better QoS for I/O
  - 9p throttling
  - virtio-net throttling
- QEMU configurability
  - disable-tcg
  - more configure options
  - plain fixes
- Memory optimizations

# Linux kernel upstreaming

- 9p file system improvements
  - Performance improvements
  - Fixes
  - Benchmark comparisons and results

# RunV upstreaming

- Volume support
- Pod support
- Network support
  - Network information collection
  - Ovs support
- Integration test framework
- Customize kernel/initrd
- Bugfix
- Others…(Cgroup, … still on the way)

# References

QEMU: www.qemu.org
Development Mailing list: qemu-devel@nongnu.org
http://lists.nongnu.org/archive/html/qemu-devel/

KVM: www.linux-kvm.org
Development Mailing list: kvm@vger.kernel.org
ARM: kvmarm@lists.cs.columbia.edu

Linux kernel: www.kernel.org
Development Mailing list: linux-kernel@vger.kernel.org

Docker: www.docker.com/
Codes: https://github.com/docker/docker

Hyper: www.hyper.sh
RunV: https://github.com/hyperhq/runv
Hyperstart: https://github.com/hyperhq/hyperstart

Qboot: https://github.com/bonzini/qboot

…

# Thank you!

# Comparison: ClearContainer 2.0

| Feature | Huawei Secure Container | Intel ClearContainer 2.0 |
|---|---|---|
| Bootloader | QBoot | QEMU pc-lite custom bootloader from Pmode |
| Firmware | none | ACPI, … |
| Virtual platform | QEMU pc-uvm (based on 440fx) | QEMU pc-lite (based on Q35) |
| Rootfs | Virtio-9p | Virtio-9p |
| Guest Kernel | uVM patches | ClearLinux |
| Runtime | runV | COR |
| Guest OS | Hyperstart init (.c) | Mini-OS SystemD based guest |
| Hotplug control | via RunV and Hyperstart | Via QEMU-ACPI |
| Optimization focus | Memory overhead reduction | Bootime reduction |
| Architecture | X86-64 and ARM64 | X86-64 |