

QEMU

Florent Glück - florent.gluck@hesge.ch

March 09, 2025

ISC - HEPIA

Introduction to QEMU

What is QEMU?

- QEMU (Quick EMUlator - <https://qemu.org>) is an open source **machine emulator and hosted VMM**
- Feature full virtualization of the CPU through Dynamic Binary Translation
- Can use **hardware-assisted virtualization** such as kvm, xen, whpx¹, hvf², etc.
 - they all require Intel VT-x or AMD-V
- Emulate many hardware platforms and devices, real and virtual ones
- Emulate user-level processes → allow applications compiled for one architecture to run on another

¹Windows Hypervisor Platform

²Apple Hypervisor Framework

Why QEMU?

To quote a great hacker, Drew DeVault¹:

QEMU is fast, portable, better supported by guests, and has more features than Hollywood.

There's nothing other hypervisors can do that QEMU can't, and there's plenty QEMU can that they cannot.

¹Drew DeVault's blog: <https://drewdevault.com>

A bit of history

- QEMU started in 2003 by jedi master programmer Fabrice Bellard
 - author of FFMPEG, JSLinux and many other projects¹
- Origin of QEMU: portable Just In Time translation engine for cross architecture emulation
- QEMU quickly grew to system emulation
- QEMU started with PC hardware, but now support many more: ARM, RISC-V, MIPS, PowerPC, Alpha, Sparc, SH4, etc.

¹More here: <https://bellard.org>

Where is QEMU being used?

- Cloud computing:
 - everything OpenStack
 - along KVM and Xen guests
- Cross-compilation development environments
- Android Emulator (part of SDK) (fork)
- Almost every embedded SDK out there

What can QEMU do?

- Run OS for a given architecture (i386, AMD64, ARM, RISC-V, Sparc, MIPS, etc.) on any **other** architecture
- Can run any OS as a **user application**
 - complete with graphics, sound, and network support
 - don't need to be root!
- Decent emulation performance for real world OS
 - orders of magnitude faster than Wind River Simics (simulator)
- QEMU can interact with guest OS

QEMU principle

When creating and running a VM with QEMU:

- Hardware **configuration** is specified on the **command line** (by default)¹
- **Hard disks** are represented as **files** (disk images)

¹by opposition to VirtualBox and VMWare which store configuration in a file

QEMU usage

- Binary for AMD64 (Intel/AMD 64-bit architecture) is **qemu-system-x86_64**
- On Debian/Ubuntu, install **qemu-system-x86** and **qemu-system-gui** packages
- Typical use:
 1. create an image disk with **qemu-img** (once)
 2. run **qemu-system-x86_64** which configures the VM's hardware and run it:

```
# QEMU usage example
qemu-system-x86_64 -smp cpus=1 -m 4G -hda disk.qcow -cdrom
debian-12.1.0-amd64-netinst.iso
```

- QEMU manual and options with:

```
man qemu-system-x86_64          # or: qemu-system-x86_64 --help
```

QEMU nested virtualization

- QEMU can expose the host's CPU with all supported features, notably hardware virtualization instructions, with:

```
-cpu host
```

- `-cpu host` provides **nested virtualization**
 - it allows to run an hypervisor **inside** an hypervisor (a VM inside a VM)!

QEMU with KVM

Can a host run KVM? (1/2)

- QEMU can use **hardware-assisted virtualization** provided by the underlying hypervisor: kvm, xen, whpx, etc.
- To check for hardware virtualization support (Intel or AMD) on Linux:

```
$ lscpu | grep Flags | grep "vmx\|svm"  
Flags:  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat  
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb  
rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology  
nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx  
est tm2 ssse3 sdbg fma cx16 xtpr pdcm ...
```

- Should see **vmx** (Intel) or **svm** (AMD) if hardware virtualization is present

Can a host run KVM? (1/2)

- Check that both **kvm** and either **kvm_intel** or **kvm_amd** modules are loaded into the kernel:

```
$ lsmod | grep kvm
kvm_intel          487424  4
kvm                1437696  3 kvm_intel
irqbypass         12288   1 kvm
```

- To load a module, use the **modprobe** command (as root)
- For instance, to load the **kvm** module (use **-r** to unload it):

```
sudo modprobe kvm
```

Accessing KVM

- To access the KVM device, /dev/kvm, one **must** either (depending on the Linux distribution):
 - be in the kvm **group**¹
 - have the proper **ACL permissions**²
- Typical examples of KVM API use:
 - **QEMU when launched with -machine accel=kvm**
 - any other Linux-based hypervisor using KVM
 - any application using /dev/kvm, typically a custom hypervisor

¹<https://linuxize.com/post/how-to-add-user-to-group-in-linux/>

²<https://www.redhat.com/sysadmin/linux-access-control-lists>

Devices in QEMU

QEMU devices

- QEMU supports a very large number of devices, including CPU architectures:
 - **emulated** devices, seen as real devices by the guest OS
 - **paravirtualized** devices, seen as virtual devices by the guest OS
- To list all supported devices:

```
qemu-system-x86_64 -device help
```

- To list supported options for a specific device, use this syntax:

```
qemu-system-x86_64 -device rtl8139,help  
qemu-system-x86_64 -device virtio-net-pci,help
```


Device types (from QEMU point of view)

Emulated: IDE, SATA, SCSI disk controllers, network cards, etc.

- Good compatibility (drivers usually present in guest OS)
- Low performance

Paravirtualized: virtio devices

- Good performance
- Require dedicated paravirtualized drivers in guest OS
 - drivers usually present when using Linux as guest OS

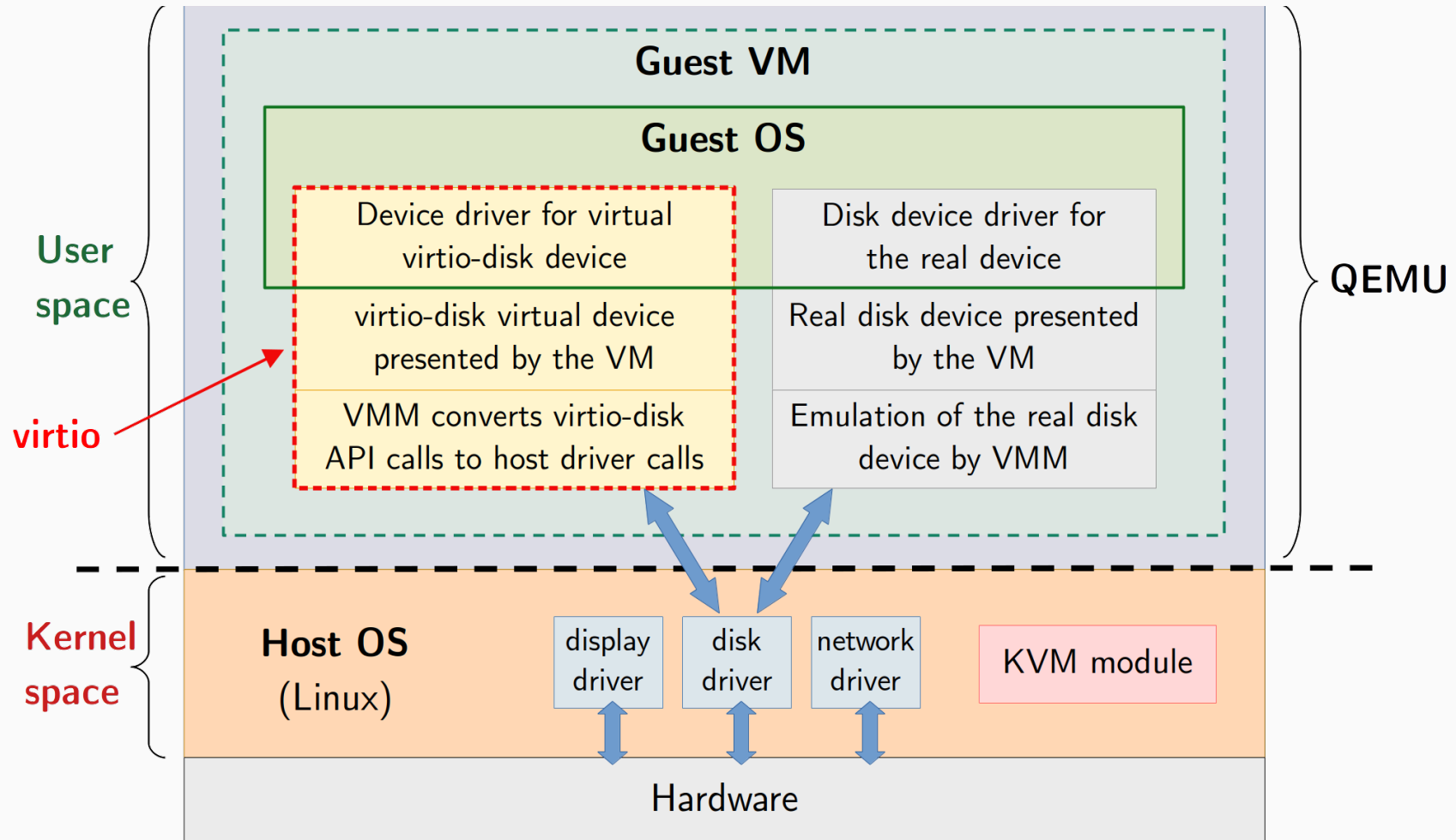
Passthrough: via VFIO

- Best, near native performance
- Limited number of PCI devices supported
- *Tricky* live migration
- Requires VT-d hardware extension

Virtio framework

- **Specification** for **paravirtualized** device (I/O) virtualization
- Abstraction layer over the hardware (devices)
 - common API for all paravirtualized devices
- Use shared memory (ring buffer) between QEMU ↔ guest OS
- Virtio provides:
 - device detection mechanism at boot (probing)
 - classes of virtual devices (network, block, memory, etc.)
 - common I/O registers
 - virtual queues (shared memory)

Virtio vs emulated driver



Virtio architecture

Frontend driver (in guest OS)

- Kernel module (driver) in guest OS
 - usually contains `virtio` in its name
- Accepts I/O requests from user process
- Transfer I/O requests to backend driver using hypercalls

Backend driver (in VMM, e.g. QEMU)

- A virtio virtual (paravirtualized) device in QEMU (e.g. `virtio-disk`)
- Accepts I/O requests from frontend driver
- Perform I/O operations via physical device through the host OS

Optimized devices for Linux guest

- Uses hardware assisted virtualization with KVM:

```
-machine accel=kvm
```

- Uses 2 virtual CPUs (vCPUs) and 4GB of RAM:

```
-smp cpus=2 -m 4096
```

- Uses a paravirtualized graphics card:

```
-vga virtio
```

- Uses a paravirtualized NIC and provides Internet access to the guest:

```
-nic user,model=virtio
```

- Uses a paravirtualized disk controller and use `disk.qcow` as a disk:

```
-drive file=disk.qcow,index=0,media=disk,format=qcow2,if=virtio
```

Storage in QEMU

Storage devices in hypervisors

- Storage devices are anything that can store content: hard drive, flash drive (SDD, USB key, SD card, etc.), DVD-ROM, CD-ROM, etc.
- Guest OSes manipulate and access storage devices **as if they were physically present** on the system
- However, on VMM side, **physical storage devices are simply... files!**
- VMM presents these files as if they were physical storage devices to the guest
- These files are called **disk images**

QEMU disk images

- QEMU supports many **disk image formats**:
 - qcow2, qed, vmdk, vhd, vdi, raw, rbd, nbd, tftp, ftp, vvfat, ftps, dmg, iscsi, parallels, bochs, quorum, etc.
- Use **qemu-img** tool to manipulate images:
 - create images
 - convert among image formats
 - resize images
 - manage disk snapshots
 - etc.

Recommended disk image formats

- **qcow2**: native QEMU image format
 - most **versatile** and **flexible**
 - many features: thin provisioning, encryption, compression, snapshots, etc.
 - **only stores used blocks (regardless of underlying filesystem)**
 - **does not require** a filesystem supporting **sparse files**
- **raw**: raw disk image format
 - image of a physical hard disk (simply a series of sectors)
 - **simple** and **very portable** (exportable to other VMMs)
 - best portability and performance, but few features
 - only stores used blocks **if sparse files supported** by the filesystem
 - otherwise zeroes are stored on disk 😞

Inspecting/modifying VM disk image files (1/3)

guestfish

- Shell and command-line tool for examining and modifying a VM disk image's filesystem

```
guestfish --ro -a disk.qcow -i ls /home/zorglub/  
guestfish --ro -a disk.qcow -i cat /etc/group
```

guestmount

- Mount a disk image's filesystem on the host using FUSE¹ and libguestfs

```
guestmount -a disk.qcow -m /dev/vda1 my_mount_dir
```

¹https://en.wikipedia.org/wiki/Filesystem_in_Userspace

Inspecting/modifying VM disk image files (2/3)

guestfs-tools Debian/Ubuntu package provides many useful tools:

<code>virt-rescue</code>	run a rescue shell on a VM
<code>virt-builder</code>	build VM images quickly
<code>virt-copy-out</code>	copy files and dirs out of a VM disk image
<code>virt-copy-in</code>	copy files and dirs into a VM disk image
<code>virt-resize</code>	resize a VM disk
<code>virt-sparsify</code>	make a VM disk sparse
<code>virt-edit</code>	edit a file in a VM
<code>virt-ls</code>	list files in a VM
<code>virt-filesystems</code>	list filesystems, partitions, block devices, in a disk image

Read the man for usage examples!

Inspecting/modifying VM disk image files (3/3)

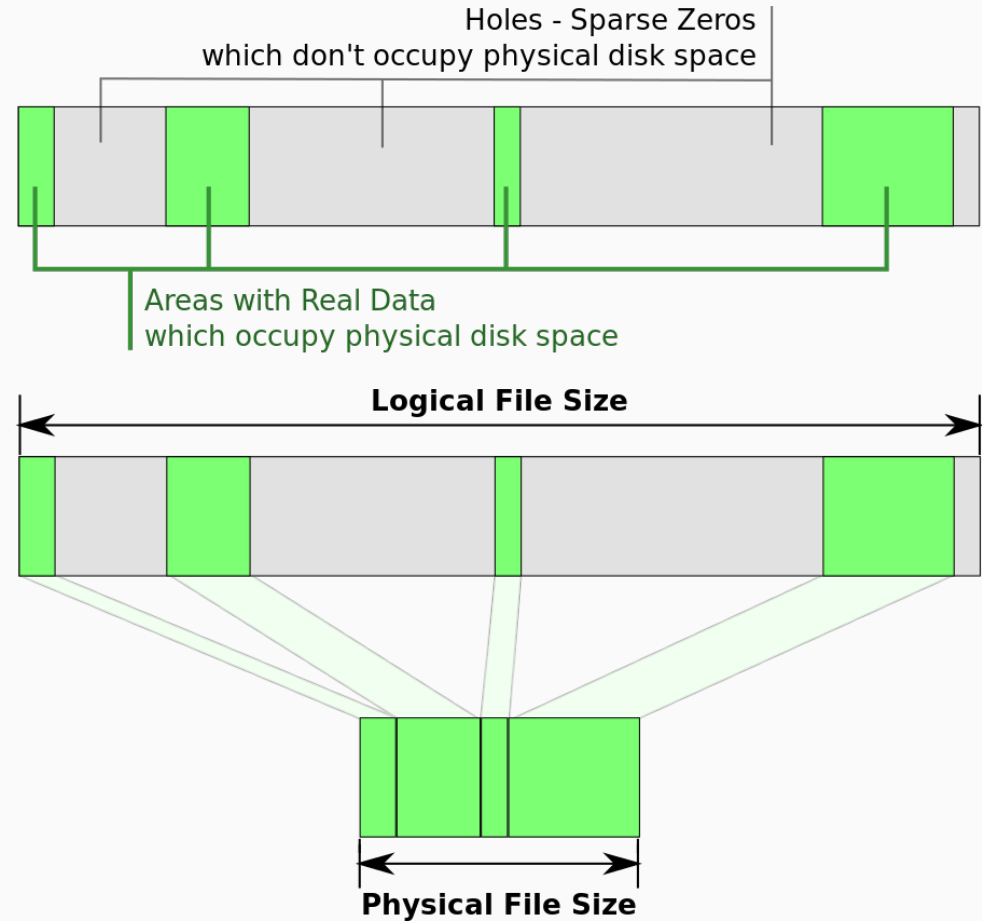
- Tools from previous slides use `libguestfs` from the `guestfs-tools` Debian package
- **These tools must not be run as root!**
- However, on Ubuntu `/boot/vmlinuz*` has the [wrong_permissions](#) preventing them from being used as non-root user
 - permissions must be changed with:

```
sudo chmod 0644 /boot/vmlinuz*
```

Sparse files

Sparse files

- A sparse file is a file that does not store unused space (or *holes*)
- Data blocks containing no data (zeros) are **not stored** to disk
- Most modern filesystems support sparse files:
 - ext4, xfs, ntfs, btrfs do
 - ⚠ FAT filesystems **do not**!
- Requires support from both, filesystem **and** application



Handling sparse files

- To create a 10MB sparse file:

```
truncate -s 10M myfile  
dd if=/dev/zero of=myfile bs=10M count=1 conv=sparse
```

- To display a file's real allocated space (usually in blocks of 1024 bytes):

```
ls -s file  
du file
```

- To convert a file into a sparse file:

```
fallocate -v -d file
```


- To convert a sparse file into a non-sparse file:

```
cp file nonsparse_file --sparse=never
```

Copying sparse files

- Linux **cp** command **transparently** handles copy of sparse files
 - if unsure, use:

```
cp --sparse=always
```

- **Transferring sparse files over the network is usually not supported!**
 - files **lose** their sparse property!
 - server and client must **both** implement **support** for sparse files
 -  **scp** does not support sparse files!
 - use **rsync** over **scp** instead (using ssh key pairs)

```
rsync -P --sparse source_file destination_machine:
```


Interacting with guest OS

Port forwarding

Traffic to a port on the host can be **forwarded** to a port in the guest

- Here, we forward TCP traffic from port 8000 on the localhost interface (127.0.0.1) on the host, to port 22 in the guest¹:

```
-nic user,hostfwd=tcp:127.0.0.1:8000-:22
```

- Then, to connect to a ssh server listening on port 22 in the guest:

```
ssh janedoe@localhost -p 8000
```

¹The option `model=virtio` can be added to use a paravirtualized network card

Shared directories

- QEMU uses the [9p](#) protocol and virtio driver to share directory between host and guest OS
 - same directory can be shared by multiple guest OS
- **Host:** run QEMU with these additional arguments, where MOUNT_TAG is the share name:

```
-virtfs local,path=PATH_TO_SHARE,mount_tag=MOUNT_TAG,security_model=mapped
```

- **Guest OS:** mount the virtual filesystem, specifying the **9p** type:

```
sudo mount -t 9p MOUNT_TAG MOUNT_DIR
```

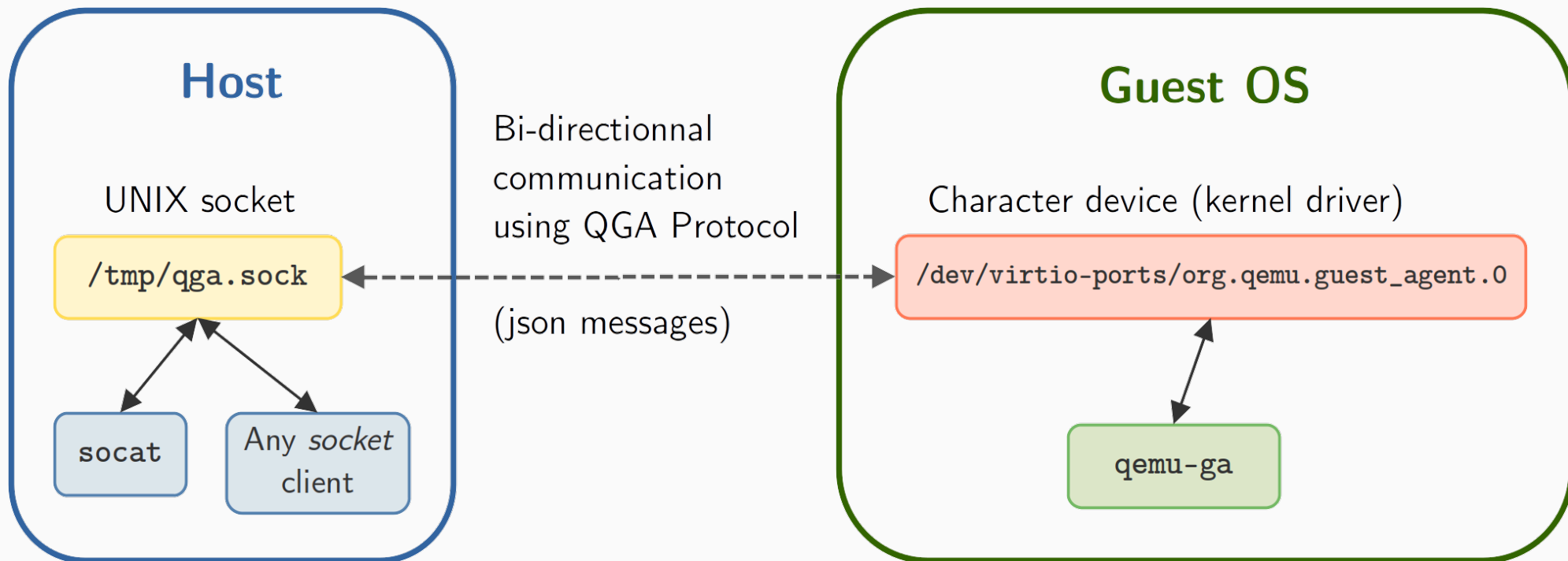
- requires 9p, 9pnet, and 9pnet_virtio kernel modules
 - mount loads them automatically

QEMU Guest Agent (QGA)

QGA is a mechanism that allows the VMM to **interact** with the guest OS

- QGA must be installed in guest OS
 - typically the **qemu-ga** service (daemon)
- Allows QEMU to perform many operations:
 - get guest OS information
 - read/write a file in guest OS
 - sync and freeze the filesystems
 - shutdown/reset/suspend guest OS
 - etc.
- Uses QEMU Guest Agent Protocol to exchange messages via a UNIX socket

QEMU ↔ OS guest communication via QGA



QGA principle

- VM must be created with a special paravirtualized QGA device
- In guest OS, this device is exposed in `/dev`
 - e.g. `/dev/virtio-ports/org.qemu.guest_agent.0`
 - possible to read/write from/to it
- On the host, this device is a socket created in the filesystem
 - e.g. `/tmp/qga.sock`
 - possible to read/write from/to it
- On the host, we write QGA commands and read values returned by guest OS
 - commands/responses are serialized as JSON objects
 - **commands are asynchronous!**

Using QEMU Guest Agent (1/2)

1. VM must be started with these additional arguments:

```
-device virtio-serial  
-device virtserialport,chardev=qga0,name=org.qemu.guest_agent.0  
-chardev socket,path=/tmp/qga.sock,server=on,wait=off,id=qga0
```

- creates /tmp/qga.sock UNIX socket on the host
- creates /dev/virtio-ports/org.qemu.guest_agent.0 device in guest OS

Using QEMU Guest Agent (2/2)

2. In guest OS, `qemu-guest-agent` must be installed:

```
sudo apt-get install qemu-guest-agent
```

- installs `/usr/sbin/qemu-ga` daemon

3. In guest OS, enable and start the service with (usually not needed):

```
sudo systemctl enable qemu-guest-agent  
sudo systemctl start qemu-guest-agent
```


QEMU Guest Agent commands: examples

- Obtain information about the guest OS:

```
{ echo '{"execute": "guest-info"}'; sleep 1; } | socat unix-connect:/tmp/qga.sock - |  
aeson-pretty
```

- Shutdown guest OS:

```
{ echo '{"execute": "guest-shutdown"}'; sleep 1; } | socat unix-connect:/tmp/qga.sock  
- | aeson-pretty
```

- Close a previously opened file on guest OS (here, handle 1000):

```
{ echo '{"execute": "guest-file-close", "arguments": {"handle": 1000}}'; sleep 1; } |  
socat unix-connect:/tmp/qga.sock - | aeson-pretty
```

The list of supported commands is [available here](#)

Resources

- QEMU documentation
<https://qemu.readthedocs.io>
- Live Block Device Operations
<https://qemu.readthedocs.io/en/master/interop/live-block-operations.html>
- QEMU shared folders with 9pfs
<https://wiki.qemu.org/Documentation/9psetup>
- Using QEMU Machine Protocol (QMP)
<https://wiki.qemu.org/Documentation/QMP>
- Introduction to VirtIO
<https://blogs.oracle.com/linux/post/introduction-to-virtio>
- VGA and other display devices in QEMU
<https://www.kraxel.org/blog/2019/09/display-devices-in-qemu/>