QEMU

Florent Glück - florent.gluck@hesge.ch

March 24, 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, whpx[1], hvf[2], etc.
  - they all require Intel VT-x or AMD-V
- Emulate many hardware platforms and devices, real and virtual ones
- Emulate user-level processes $\longrightarrow$ allow applications compiled for one architecture to run on another

---

[1] Windows Hypervisor Platform

[2] Apple Hypervisor Framework

Why QEMU?

To quote a great hacker, Drew DeVault[1]:

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.

[1] Drew 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 projects[1]
- 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.

[1] More 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!
- Descent 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)

[1] by 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 controlers, 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

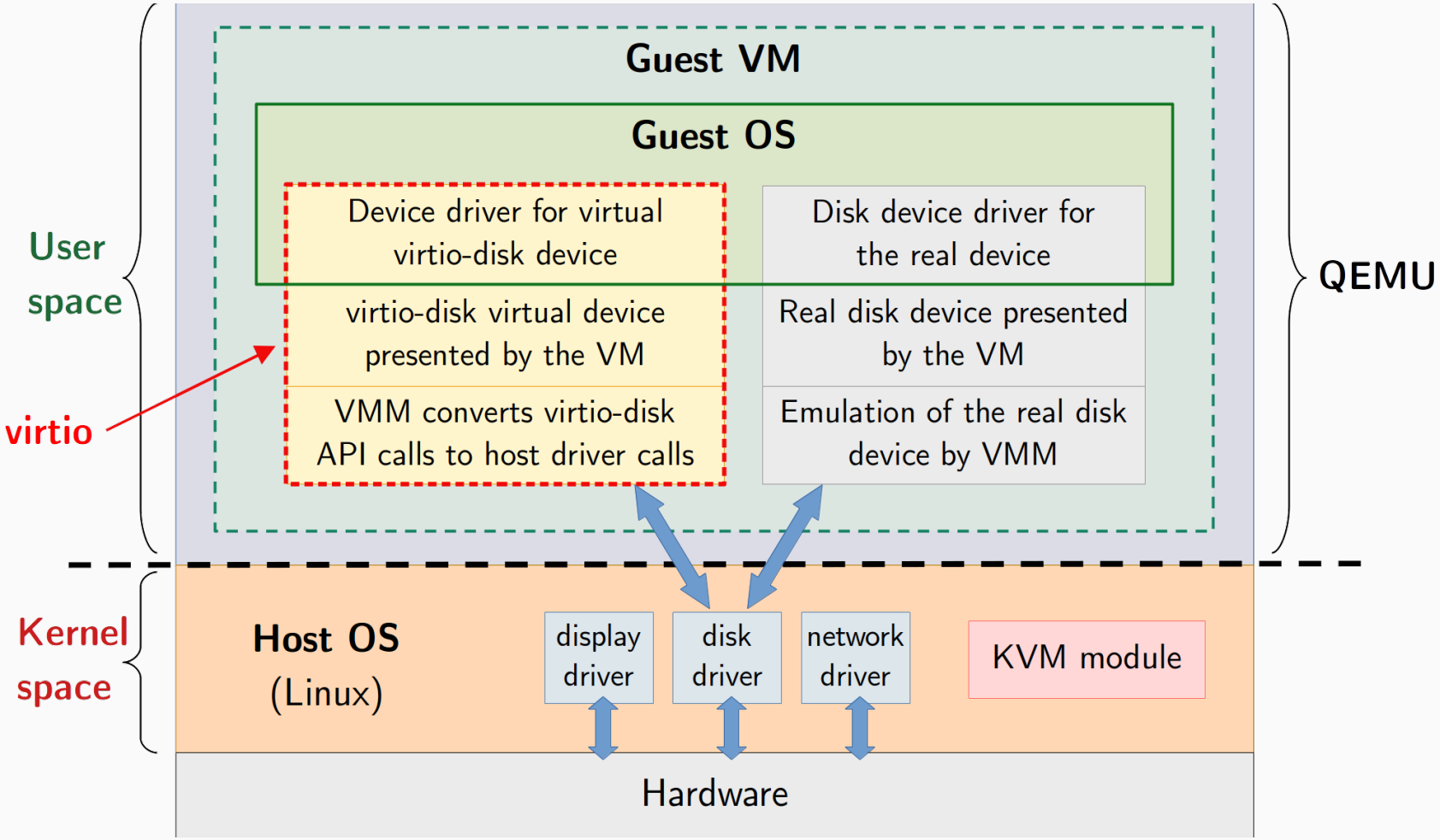
Passthrough: via VFIO

- 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 FUSE[1] and `libguestfs`

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

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

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

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

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

Read the `man` for usage examples!

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

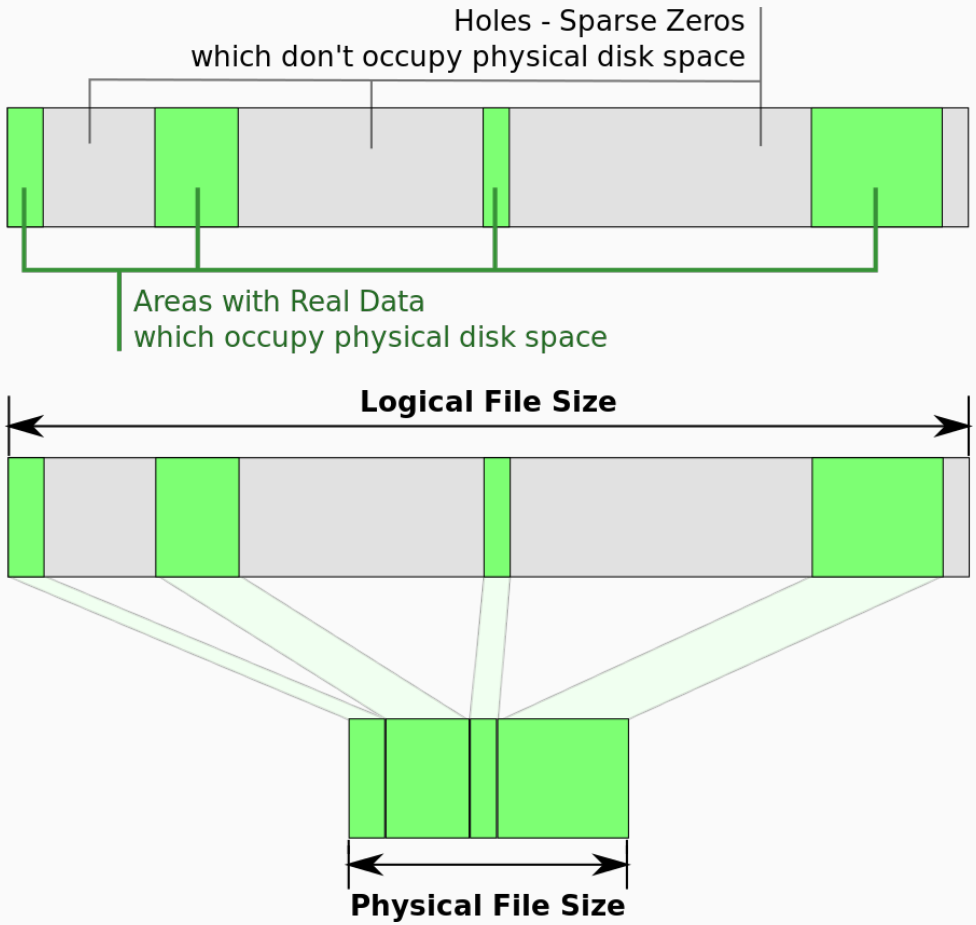
- Tools from previous slides use `libguestfs` from the `guestfs-tools` Debian package
- **These tools must not be run as root!**
- However, on Ubuntu `/boot/vmlinuz*` has the wrong_permissions preventing them from being used as non-root user
  - permissions must be changed with:

```
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
- Supported by most modern filesystems:
  - ext4, xfs, ntfs, btrfs do
- ⚠ However, not always supported ⚠
  - FAT filesystems, scp, Dropbox, etc.
- Proper sparse files support 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
```

- **Transfering 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 guest[1]:

```
-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
```

[1] The 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 additionnal 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
```

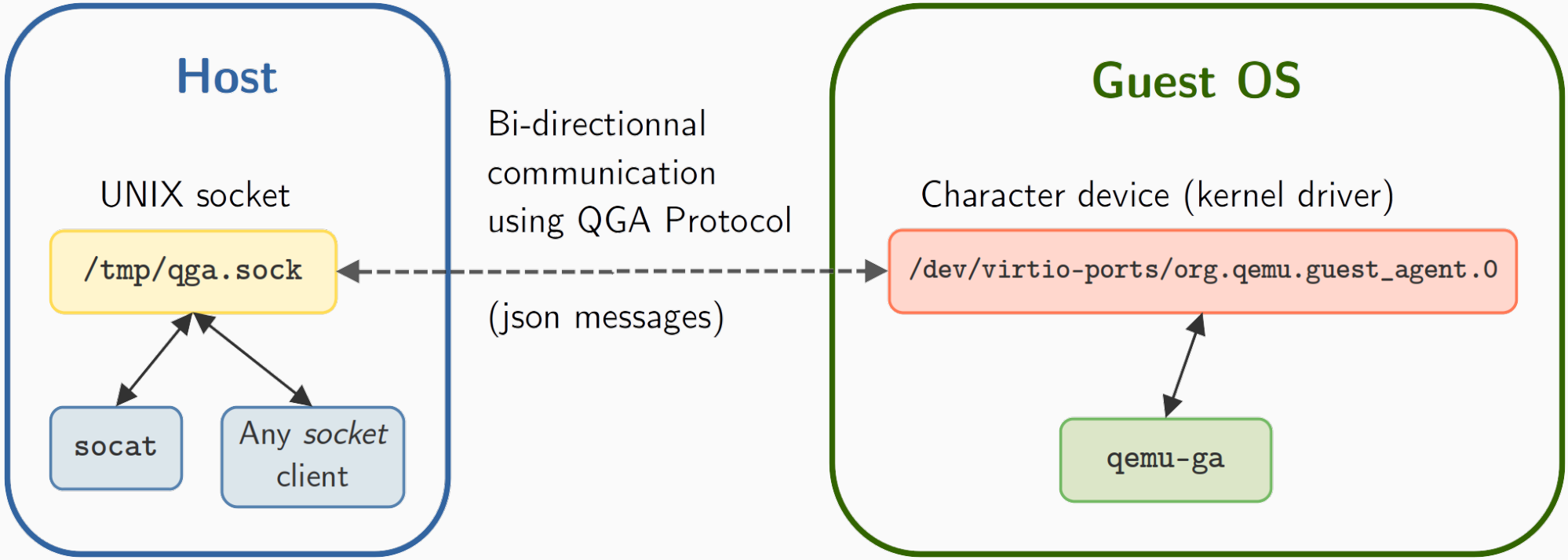
  - requires `9p`, `9pnet`, and `9pnet_virtio` kernel modules
    - `mount` loads them automatically

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

Snapshots

What is a snapshot?

- A snapshot is a “view” of a VM at a given point in time
- **Benefits**:
  - able to revert to a known specific state
  - rapidly instantiate thin-provisoned or disposable VMs

QEMU snapshot scenarios

1. Save/restore a VM' state - disk only
   - disk snapshot
2. Save/restore a VM' state - disk + RAM + devices
   - VM snapshot
3. Instantiate a thin-provisoned or disposable VM
   - disk snapshot

⚠ Most QEMU snapshots require qcow2 images!

QEMU snapshots storage policy

Internal snapshots

- All snapshots are stored **inside** the **same** qcow2 file

External snapshots

- **Each** snapshot is stored in a **different** qcow2 file
  - **chain** of qcow2 files

QEMU disk snapshots

(1) Save/restore a VM' state - disk only (internal snapshot)

First method, using **internal snapshots**

- Use **`qemu-img`** to manage the internal disk snapshots:

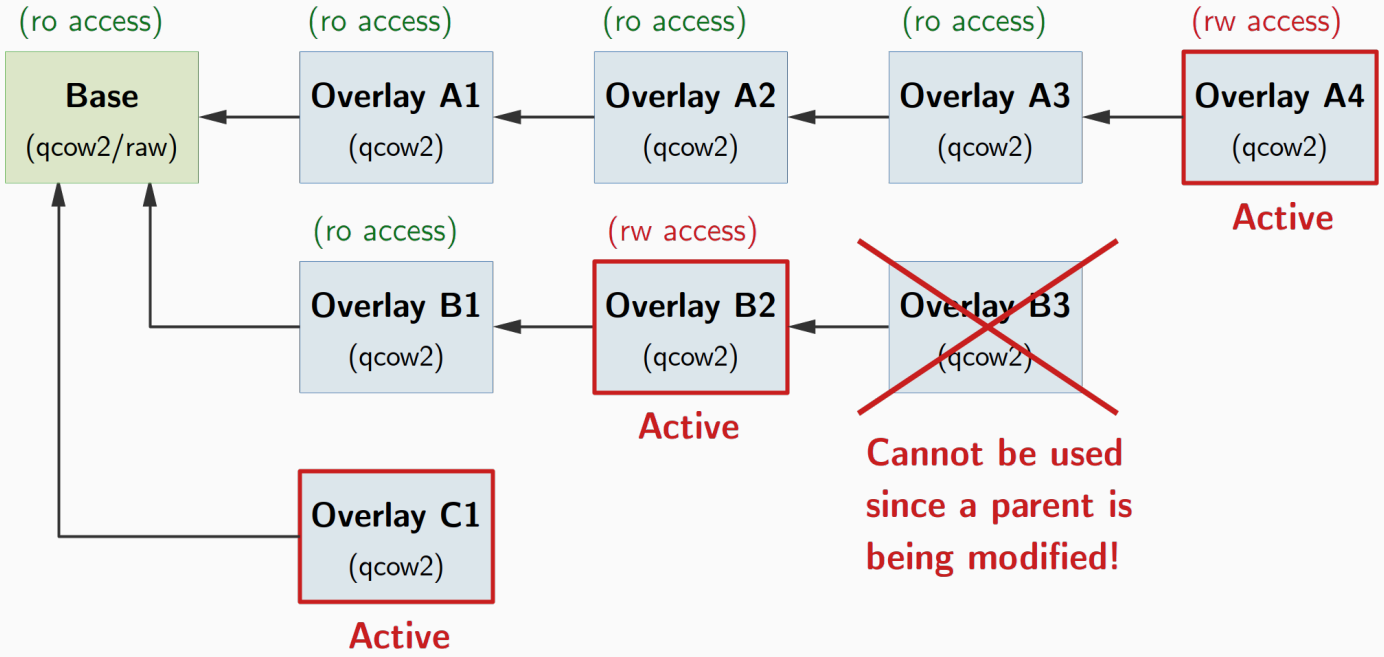
`qemu-img snapshot -c <name> <img>`	create an internal disk snapshot
`qemu-img snapshot -d <name> <img>`	delete an internal disk snapshot
`qemu-img snapshot -a <name> <img>`	apply an internal disk snapshot (revert disk to saved state)
`qemu-img snapshot -l <img>`	list all internal snapshots in the image (disk and VM)

⚠ Deleting internal snapshots does not reduce the image file size!

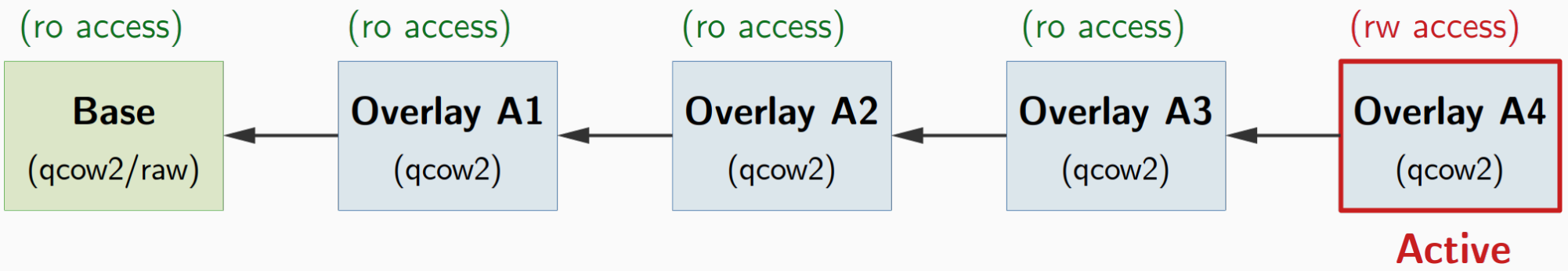
(1) Save/restore a VM' state - disk only (external snapshot)

Second method, using **external snapshots**

- **Concept**: a disk image is composed of a **base** image plus a chain of **differences** (overlays)

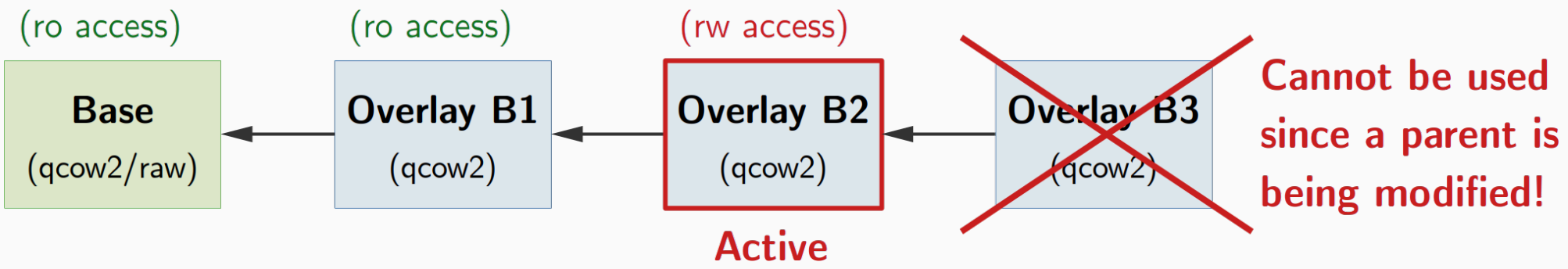


(1) Save/restore a VM' state - disk only (external snapshot)



- If QEMU runs on Overlay A4, then the VM's disk state is:
  - Base + A1 + A2 + A3 + A4
- Base + A1 + A2 + A3 are all accessed in read-only
- A4 is accessed in read-write as it is the **active** disk image
- Running QEMU on a parent image **invalidates all** children images!

(1) Save/restore a VM' state - disk only (external snapshot)

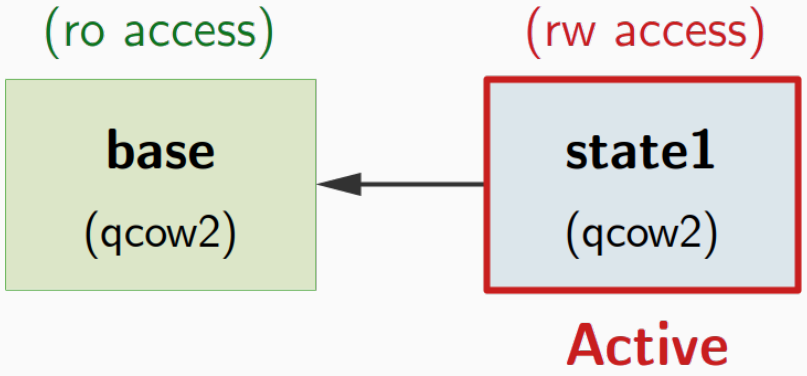


- If QEMU runs on Overlay A2, then the VM's disk state is:
  - Base + B1 + B2
- Base + B1 are accessed in read-only
- B2 is accessed in read-write as it is the **active** disk image
- Since B2 is the active image, B3 becomes **invalid!**

Create an external disk snapshot - example

To create the `state1.qcow` external disk snapshot (*overlay*) that will store the **differences** from the base (*backing*) image file `base.qcow`:

```
qemu-img create -F qcow2 -b base.qcow -f qcow2 state1.qcow
```



- Here, QEMU is launched on `state1.qcow`
- At any point, a new snapshot can be added to a chain of snapshots

Chain of external disk snapshots

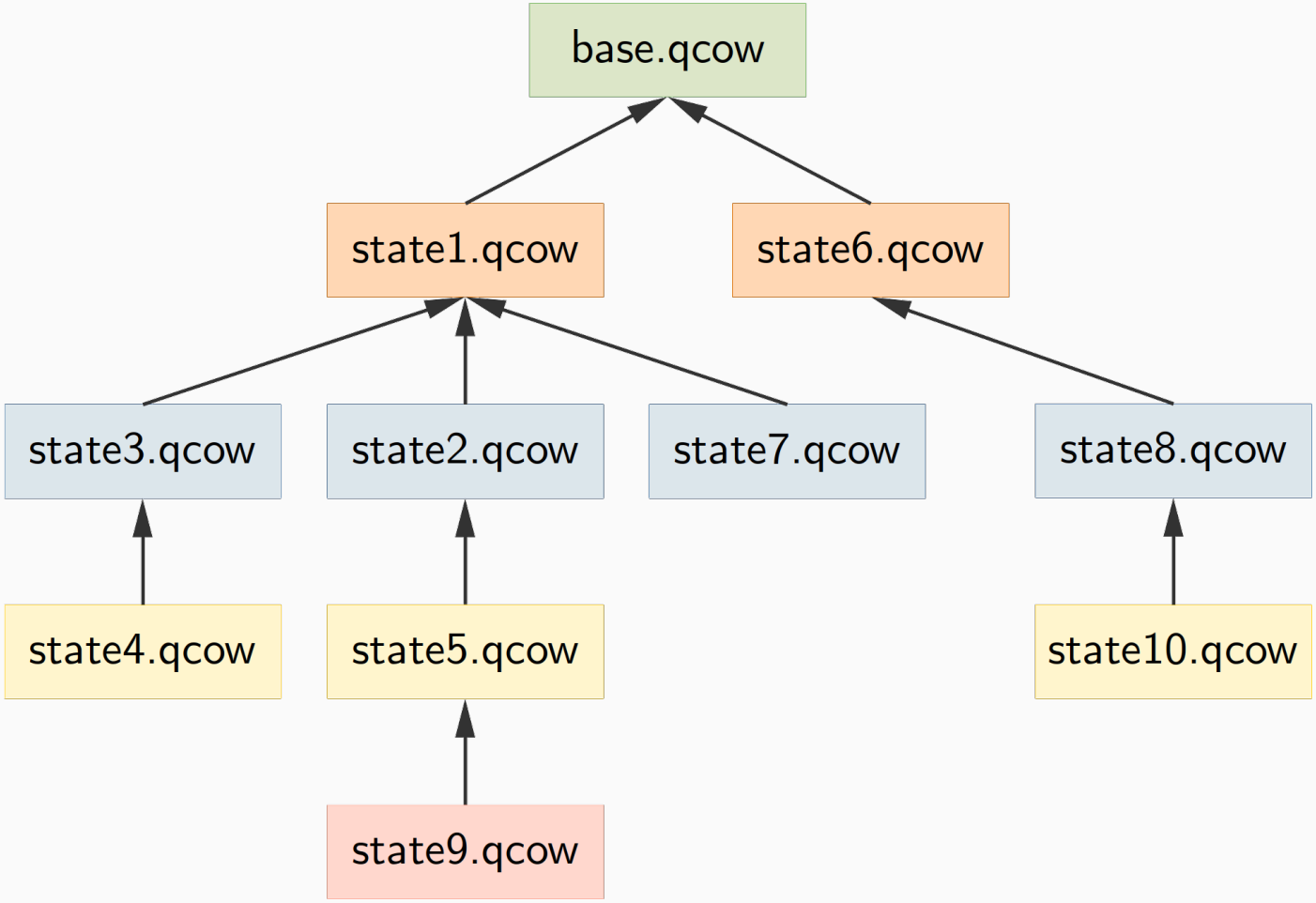
To display the chain of snapshots up to some state (here `stateC1.qcow`):

```
$ qemu-img info --backing-chain stateC1.qcow
```

```
image: stateC1.qcow
file format: qcow2
virtual size: 100 GiB (107374182400 bytes)
disk size: 196 KiB
cluster_size: 65536
backing file: base.qcow
backing file format: qcow2
Format specific information:
    compat: 1.1
    compression type: zlib
    lazy refcounts: false
    refcount bits: 16
    corrupt: false
    extended l2: false
Child node '/file':
    filename: stateC1.qcow
    protocol type: file
    file length: 194 KiB (198656 bytes)
    disk size: 196 KiB
```

```
image: base.qcow
file format: qcow2
virtual size: 100 GiB (107374182400 bytes)
disk size: 196 KiB
cluster_size: 65536
Format specific information:
    compat: 1.1
    compression type: zlib
    lazy refcounts: false
    refcount bits: 16
    corrupt: false
    extended l2: false
Child node '/file':
    filename: base.qcow
    protocol type: file
    file length: 194 KiB (198656 bytes)
    disk size: 196 KiB
```

External disk snapshots can form a tree hierarchy



Temporary external disk snapshots

- QEMU supports temporary disk snapshots via the `-snapshot` argument
- No need to explicitly create a separate snapshot file to run QEMU on
- Changes made to the VM while running are transparently written to a **temporary** file **deleted** when QEMU exits
- No changes are saved to the original disk image file
- Example:

```
qemu-system-x86_64 -smp cpus=2 -m 4G -hda disk.qcow -snapshot
```

Merging disk snapshots

Merging external disk snapshots (1/2)

- External disk snapshots in a chain can be **merged** together
  - offline, when the VM is not running, using `qemu-img`
  - online, when the VM is running, using QEMU Machine Protocol (QMP) commands

Merging external disk snapshots (2/2)

- Two types of merges:
  - **commit**: merge a chain of snapshots into a backing image:
    - merge children images into a parent image
    - snapshot files are **not** removed by QEMU
    - intermediate snapshots become **invalid**: no more snapshots must be created based on them
    - advisable to delete intermediate snapshots!
  - **stream**: merge a chain of backing images into a snapshot
    - merge parent images into a child image
    - parent images not needed by child image anymore

Merging: commit operations

Example of chain (`[A]` = base image, `[D]` = active snapshot):

```
[A] <-- [B] <-- [C] <-- [D]
```

- Case 1, merge `[B]` into `[A]`:

  ```
  [A] <-- [C] <-- [D]
  ```

- Case 2, merge `[B]` and `[C]` into [A]:

  ```
  [A] <-- [D]
  ```

- Case 3, merge `[B]`, `[C]` and `[D]` into [A]:

  ```
  [A]
  ```

Merging: commit operations

Example of chain (`[A]` = base image, `[D]` = active snapshot):

```
[A] <-- [B] <-- [C] <-- [D]
```

- Case 4, merge `[C]` into `[B]`:

  ```
  [A] <-- [B] <-- [D]
  ```

- Case 5, merge `[C]` and `[D]` into `[B]`:

  ```
  [A] <-- [B]
  ```

Merging: stream operations

Example of chain (`[A]` = base image, `[D]` = active snapshot):

```
[A] <-- [B] <-- [C] <-- [D]
```

- Case 1, merge everything into `[D]`:

```
[D]
```

- Case 2, merge `[B]` and `[C]` into `[D]`:

```
[A] <-- [D]
```

- Case 3, merge `[B]` into `[C]`:

```
[A] <-- [C] <-- [D]
```

Commit operations with qemu-img

- The **combined** state up to a given snapshot can be merged back into a previous backing image in the chain
- `qemu-img commit` can be used to perform a merge "commit"
- Example, where `[A]` is the base image:

```
[A] <-- [B] <-- [C] <-- [D]
```

  - merge combined states from `[B]` to `[D]` into image `[B]`:

```
qemu-img commit -f qcow2 -b B.qcow D.qcow
```

  - advisable to delete `[C]` and `[D]` as they are not valid anymore!

VM snapshots

QEMU VM snapshots

- Unlike disk snapshots, VM snapshots save the **full state** of the VM
- VM snapshots save 3 states: **disk**, **RAM**, **devices**
- Snapshots are stored as **internal snapshots**
  - all stored in the same qcow file

⚠ Given the full machine state is saved, the VM's hardware **cannot** change!

VM snapshots

- VM snapshots are managed using the QEMU monitor[1]:

`savevm <tag>`	creates (save) a VM snapshot
`delvm <tag>`	deletes a VM snapshot
`loadvm <tag>`	applies (restore) a VM snapshot
`info snapshots`	lists all snapshots (disk and VM)

- A specific snapshot can be restored when starting QEMU:
  - argument `-loadvm <tag>` starts the VM using the specified snapshot

[1] Cf. next section

QEMU monitor

QEMU monitor

The **QEMU monitor** is a console used to interact with QEMU at runtime (during its execution) to:

- Control various aspects of the VM
- Inspect the VM (registers, devices, etc.)
- Inspect the running guest OS
- Change removable media and USB devices
- Take snapshots, screenshots, audio grabs
- Perform VM live migration
- etc.

Accessing the QEMU monitor

- From QEMU's GUI: `View` → `compatmonitor0` (or similar)
- By starting QEMU with the argument: `-monitor stdio`
  - monitor accessed in the shell QEMU was started in
- By pressing `[Ctrl-Alt-2]`
  - `[Ctrl-Alt-1]` switches back to the guest OS
- Through a **telnet** server embedded and started by QEMU
  - start QEMU with: `-monitor telnet::1234,server,nowait`
  - on the "client" side: `telnet ip_qemu_server 1234`

Useful QEMU commands and arguments

`man qemu-system`	Exhaustive help on QEMU
`-monitor stdio`	Redirect the monitor to the console
`-machine accel=kvm`	Use KVM to provide hardware assisted virtualization
`-smp cpus=<n>`	Set the number of CPUs to
`-m <mem>`	Set the ammount or RAM to
`-drive ...`	Define a new drive
`-device ...`	Add a device driver
`-nic ...`	Shortcut for configuring both the guest NIC and the host network backend
`-spice ...`	Enable a Spice server
`-vga ...`	Select the type of display card to emulate
`-snapshot`	Run on a temporary disk snapshot

Virtual Desktop Infrastructure (VDI)

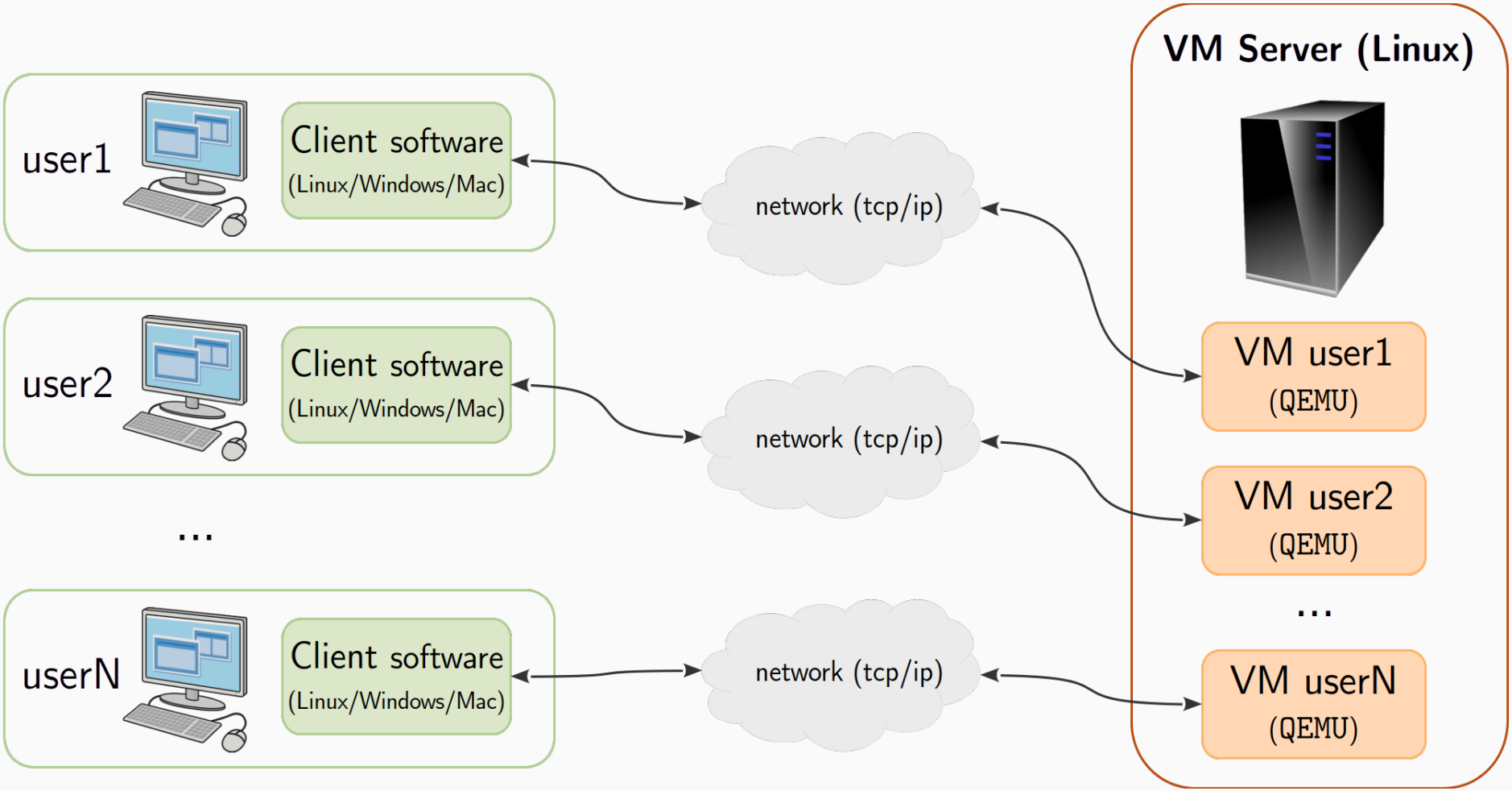
Desktop virtualization

- **Server** virtualization is commonplace and offered everywhere
  - manage virtual machines: CPU, RAM, storage, network, etc.
  - administrator access: text mode (ssh), low end graphics (VNC)
- **Desktop** virtualization **needs more**:
  - desktop integration (copy/paste, shared directory, dynamic display resize, etc.)
  - USB forwarding
  - sound forwarding
  - video stream support
  - better graphics (3D, multihead, etc.)

Desktop virtualization with QEMU

- QEMU supports two remote graphical desktop technologies: VNC and Spice
- Both technologies require:
  - a server component (VNC or Spice), embedded and running in QEMU
  - a client software used by the user to interact with the VM's desktop

Virtual Desktop Infrastructure (VDI) with QEMU



Virtual Network Computing (VNC)

- **VNC** = **V**irtual **N**etwork **C**omputing
- Very popular and available in most VMM, but offers fairly high latency
- Uses the Remote Frame Buffer (RFB) protocol to remotely control another computer
- Transmits keyboard and mouse input from one computer to another, relaying graphical-screen updates over the network
- RFB is very simple: transmit graphic primitives from server to client and event messages from client to server
- Remote desktop solution for both, virtualized, and physical infrastructures

Simple Protocol for Independent Computing Environments

- **SPICE** = **S**imple **P**rotocol for **I**ndependent **C**omputing **E**nvironments[1]
- **Goal**: provide **low latency** and **full-featured** desktop virtualization
- Spice is less known than VNC, only available in QEMU, but offers much lower latency and more features than VNC
  - e.g. USB redirection over the network = ability to use USB devices from the client
- Only available for virtualized infrastructures

[1] https://www.spice-space.org/

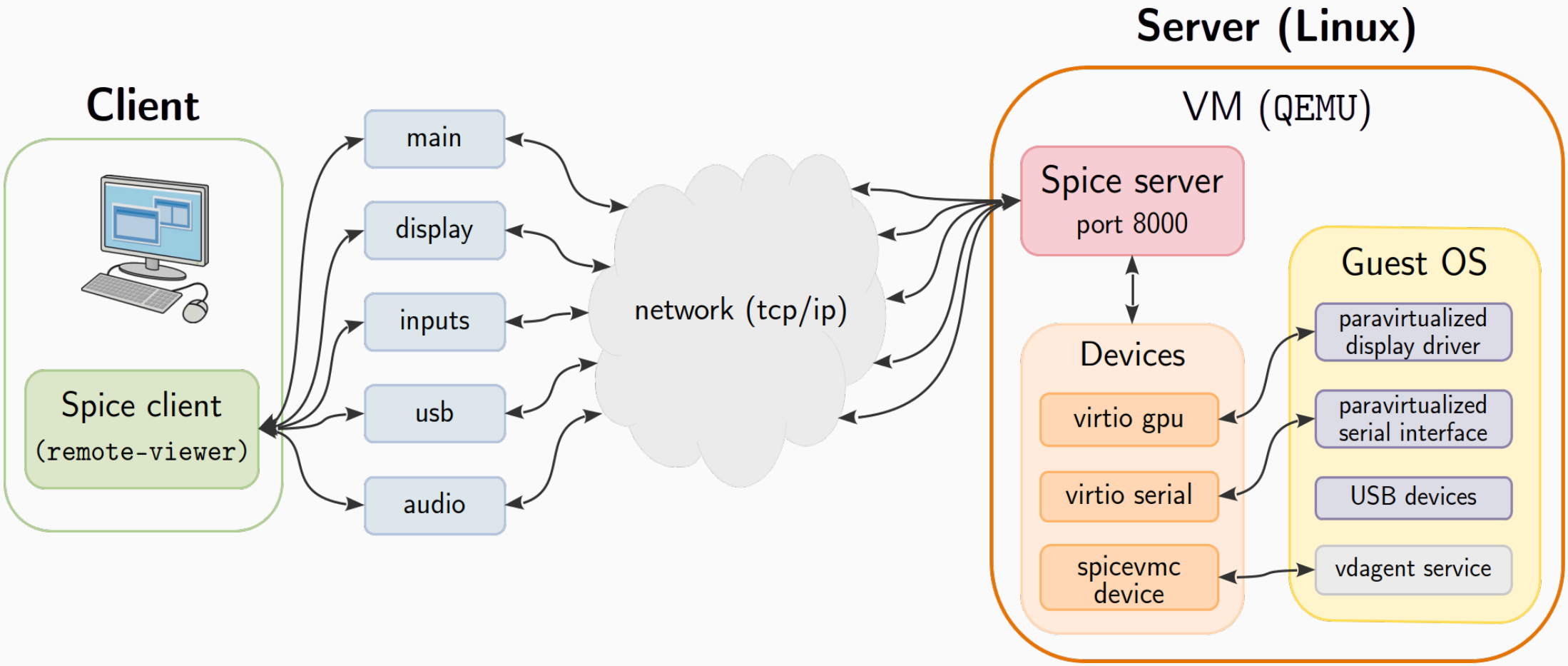
Spice overview

- Provide virtual desktop infrastructure through:
  - Spice network protocol
  - paravirtualized graphics card in QEMU
  - Spice server embedded and running in QEMU
  - `spice-vdagentd` service running in guest OS
  - Spice client software (e.g. `remote-viewer`)
- VM accessed remotely via a dedicated port on the host (on which the QEMU Spice server runs)
  - one port per VM

Spice features

- **Copy/paste** host ↔ guest OS
- **USB redirection** over the network
- **Files transfer** from host to guest OS (drag and drop)
- Much **lower latency** than VNC
- More bandwidth efficient than VNC at equivalent latency
- Multiple channels: main, display, inputs, usb, audio
  - encryption via TLS
- Access can be password protected
- Shared directory over the network
- Image compression
- OpenGL (3D graphics) acceleration

Spice architecture



Spice components

- **Server**: library used by the VMM to share the VM
  - QEMU running a Spice server exposing paravirtualized devices for Spice
- **Client**: send data to server so that user can interact with the VM
  - Example: `remote-viewer`
- **Guest**: software running in the guest OS to make Spice fully functional
  - paravirtualized graphics driver, Spice `vdagent` service
- **Protocol**: the Spice network protocol

Spice basic usage - server side (1/2)

Server side: QEMU

- Require `-vga virtio` (recommended) or `-vga qxl` graphics drivers
  - `virtio` most robust driver, but uses slightly more bandwidth
  - `qxl`: less bandwidth usage, but less stable (prone to crashes)
- Arguments to start a Spice server on port 8000 in the VM (without authentication):

```
-device virtio-serial-pci
-spice port=8000,disable-ticketing=on
-device virtserialport,chardev=spicechannel0,name=com.redhat.spice.0
-chardev spicevmc,id=spicechannel0,name=vdagent
```

Spice basic usage - server side (2/2)

Server side: in guest OS

- Require Spice Guest Agent daemon running (`spice-vdagentd`)
- The Spice agent adds the following features:
  - client mouse mode (no mouse lag)
  - automatic adjustment of virtual monitor resolution
  - copy and paste
  - file transfers
- Usually already installed in most GNU Linux distributions

Spice basic usage - client side

Client side

- A specific Spice client software is required
- Most popular client is `remote-viewer` (part of `virt-viewer` project[1])
- To install `remote-viewer` on Debian/Ubuntu:

```
apt-get install virt-viewer
```

- Example, to connect to a VM running on port 8000 of server at ip 10.9.52.118:

```
remote-viewer "spice://10.9.52.118?port=8000"
```

[1] https://gitlab.com/virt-viewer/virt-viewer

Spice server: QEMU arguments explained

- `-device virtio-serial-pci`
  - add a virtio serial device
- `-spice port=8000,disable-ticketing=on`
  - create a Spice server listening on port 8000 for client connection without authentication
- `-device virtserialport,chardev=spicechannel0,name=com.redhat.spice.0`
  - open a port for `spice-vdagent` in the virtio serial device
  - the name `spicechannel0` above must match the `id=` option below
  - port name must be `com.redhat.spice.0` because it's the namespace `spice-vdagent` is looking for in the guest OS
- `-chardev spicevmc,id=spicechannel0,name=vdagent`
  - add a spicevmc device for the port above
  - the `name=vdagent` option tells Spice what the channel is for

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/