

Running GUI Applications in Docker



Fraser Adams

Running GUI Applications in Docker

Fraser Adams

Running GUI Applications in Docker

By Fraser Adams

Copyright © 2020 Fraser Adams. All rights reserved.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk.

If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Contents

1. [Introduction](#)
 - 1.1. [The Purpose of this Book](#)
 - 1.2. [What is Docker?](#)
 - 1.3. [Why Run GUI Applications in Docker?](#)
2. [Docker Basics](#)
 - 2.1. [Enabling sudo](#)
 - 2.2. [Installing Docker](#)
 - 2.2.1. [Installing Docker on Ubuntu](#)
 - 2.2.2. [Installing Docker on Linux Mint](#)
 - 2.2.3. [Installing Docker on Debian](#)
 - 2.2.4. [Installing Docker on CentOS](#)
 - 2.2.5. [Installing Docker on Fedora](#)
 - 2.3. [The docker Group and sudo](#)
 - 2.4. [Hello Docker](#)
 - 2.5. [Container Lifecycle Management](#)
 - 2.6. [Image Lifecycle Management](#)
 - 2.7. [Managing Containers with Portainer](#)
 - 2.8. [Image Optimisation](#)
3. [X11 Overview](#)
 - 3.1. [Introduction](#)
 - 3.2. [X11 Architecture](#)
 - 3.2.1. [The X Server](#)
 - 3.2.2. [X Clients](#)

3.2.3. [The Window Manager](#)

3.2.4. [Network and Interprocess Communication](#)

3.3. [Wayland and Weston](#)

4. [Local Applications](#)

4.1. [Simple X11 Applications](#)

4.1.1. [x11-apps](#)

4.1.2. [xterm](#)

4.2. [GNOME, D-bus and AppArmor](#)

4.2.1. [gnome-calculator](#)

4.2.2. [gedit](#)

4.2.3. [evince](#)

4.3. [3D Accelerated Applications](#)

4.3.1. [Prerequisites](#)

4.3.2. [glxgears](#)

4.3.3. [glmark](#)

4.3.4. [kcalc](#)

4.3.5. [gtk3-examples](#)

4.4. [Adding Audio](#)

4.4.1. [PulseAudio](#)

4.4.2. [ALSA](#)

4.4.3. [JACK](#)

4.4.4. [OSS](#)

4.5. [Scanner Support](#)

4.5.1. [simple-scan](#)

4.5.2. [Network Scanning and saned](#)

4.6. [Webcam Support](#)

4.6.1. [camorama](#)

4.7. [Bluetooth Applications](#)

4.7.1. [blueman](#)

4.7.2. [bluetoothd](#)

4.7.3. [Bluetooth and PulseAudio](#)

4.8. [Wayland and Weston](#)

4.8.1. [gnome-calculator](#)

4.8.2. [weston](#)

5. [More Complex Applications](#)

5.1. [Modular Scripting](#)

5.2. [Browsers](#)

5.2.1. [firefox](#)

5.2.2. [firefox-flash](#)

5.2.3. [chrome](#)

5.2.4. [chromium](#)

5.2.5. [opera](#)

5.2.6. [epiphany](#)

5.3. [Compute Acceleration](#)

5.3.1. [nvidia-smi](#)

5.3.2. [cuda-nbody](#)

5.3.3. [cuda-smoke](#)

5.3.4. [clinfo](#)

5.3.5. [opencl-nbody](#)

5.3.6. [opencl-particles](#)

5.4. [Video and Graphics](#)

5.4.1. [gimp](#)

5.4.2. [inkscape](#)

5.4.3. [guvcview](#)

5.4.4. [vlc](#)

5.4.5. [smplayer](#)

5.4.6. [kodi](#)

5.4.7. [Blu-ray](#)

5.4.8. [frameworks graphics samples](#)

5.4.9. [blender](#)

5.5. [Office](#)

5.5.1. [thunderbird](#)

5.5.2. [libreoffice](#)

5.5.3. [calibre](#)

5.6. [Audio](#)

5.6.1. [qjackctl](#)

5.6.2. [qsynth](#)

5.6.3. [jack-keyboard](#)

5.6.4. [ardour](#)

5.7. [Games](#)

5.7.1. [extremetuxracer](#)

- 5.7.2. [supertuxkart](#)
- 5.7.3. [sauerbraten](#)
- 5.7.4. [dolphin-emu](#)
- 6. [Remote Applications](#)
 - 6.1. [X11 Forwarding](#)
 - 6.1.1. [x11-apps](#)
 - 6.2. [Remote Applications and D-bus](#)
 - 6.2.1. [gnome-calculator](#)
 - 6.2.2. [firefox](#)
 - 6.3. [Remote Audio](#)
 - 6.3.1. [pulseaudio-utils](#)
 - 6.4. [Exporting the Framebuffer](#)
 - 6.4.1. [vinagre](#)
 - 6.4.2. [remmina](#)
 - 6.5. [VNC](#)
 - 6.5.1. [tightvnc](#)
 - 6.5.2. [x11vnc](#)
 - 6.5.3. [tigervnc](#)
 - 6.5.4. [turbovnc](#)
 - 6.6. [RDP](#)
 - 6.6.1. [xrdp](#)
 - 6.6.2. [xrdp-native-audio](#)
 - 6.6.3. [xfreerdp](#)
 - 6.7. [Xpra](#)

6.8. NX

6.8.1. [x2goserver](#)

6.8.2. [x2goclient](#)

6.8.3. [nomachine](#)

6.9. SPICE

6.9.1. [xserver-xspice](#)

6.9.2. [remote-viewer](#)

6.9.3. [spicy](#)

7. Remote 3D Accelerated Applications

7.1. [virtualgl](#)

7.2. [glxgears-vgl](#)

7.3. [firefox-vgl](#)

7.4. [cuda-smoke-vgl](#)

7.5. [frameworks-graphics-samples-vgl](#)

7.6. [blender-vgl](#)

7.7. [extremetuxracer-vgl](#)

7.8. [dolphin-emu-vgl](#)

8. Web Applications

8.1. GTK 3 Broadway

8.1.1. [broadwayd-gnome-calculator](#)

8.1.2. [broadwayd](#)

8.2. Guacamole

8.3. [noVNC](#)

8.4. Xpra

8.5. SPICE

8.5.1. [xserver-xspice-html5](#)

8.5.2. [xserver-xspice-eyeos](#)

9. Virtual Desktops

9.1. [Xephyr](#)

9.2. [debian-buster-lxde](#)

9.3. [ubuntu-18.04-gnome](#)

9.4. [linuxmint-19.3-cinnamon](#)

9.5. [centos-7-gnome](#)

9.6. [fedora-32-kde](#)

10. [3D Accelerated Virtual Desktops](#)

10.1. [ubuntu-18.04-gnome](#)

10.2. [linuxmint-19.3-cinnamon](#)

10.3. [centos-7-gnome](#)

10.4. [fedora-32-kde](#)

11. [Remote Virtual Desktops](#)

11.1. [VNC](#)

11.1.1. [debian-buster-lxde-tightvnc](#)

11.1.2. [ubuntu-18.04-gnome-tigervnc](#)

11.1.3. [linuxmint-19.3-cinnamon-turbovnc](#)

11.1.4. [fedora-32-kde-turbovnc](#)

11.2. [RDP](#)

11.2.1. [ubuntu-18.04-gnome-xrdp](#)

11.2.2. [linuxmint-19.3-cinnamon-xrdp](#)

11.2.3. [centos-7-gnome-xrdp](#)

11.3. [NX](#)

11.3.1. [ubuntu-18.04-gnome-x2go](#)

11.3.2. [linuxmint-19.3-cinnamon-x2go](#)

11.3.3. [centos-7-gnome-x2go](#)

11.3.4. [fedora-32-kde-x2go](#)

11.3.5. [ubuntu-18.04-gnome-nomachine](#)

11.3.6. [linuxmint-19.3-cinnamon-nomachine](#)

11.3.7. [centos-7-gnome-nomachine](#)

11.3.8. [fedora-32-kde-nomachine](#)

11.4. [SPICE](#)

11.4.1. [ubuntu-18.04-gnome-spice](#)

11.4.2. [linuxmint-19.3-cinnamon-spice](#)

11.4.3. [fedora-32-kde-spice](#)

1 Introduction

1-1 The Purpose of this Book

This book provides a comprehensive step-by-step guide to packaging and running GUI applications as Docker containers. The book sets the scene with some simple and well-known X11 applications hosted locally, before tackling more advanced topics such as enabling GPU acceleration in containers to support OpenGL applications. The book also covers several less-frequently discussed topics such as remote accelerated rendering via VirtualGL to enable Cloud hosting of high-end GUI applications, D-bus integration, AppArmor, real-time scheduling, and running full virtual desktops in containers.

Although this book covers many advanced Docker concepts **it is not** a tutorial on Docker and *some* familiarity with Docker would be useful, but not essential, as all of the examples come complete with detailed explanation and the command line instructions necessary to run them.

As the point of the book is to illustrate the concepts *by example* the source code is, of course, freely available and Apache 2.0 Licensed on GitHub.

To run the examples a reasonably modern Linux distribution capable of running Docker and an X11 capable desktop are required. The OpenGL applications require either Nvidia or Open Source Mesa drivers although some OpenGL 2.1 applications should work with the VirtualBox virtual GPU.

1-2 What is Docker

Docker is a technology that provides a form of Operating System level virtualisation known as **containerisation**. Docker uses the resource isolation features of the Linux kernel such as [cgroups](#), [namespaces](#), and [union mount filesystems](#) to allow multiple isolated user space instances or **containers** to run within a single physical (or virtual) Linux instance with lower overhead than full virtual machines.

Containers may also usefully be thought of as an *application* abstraction that allows application code, dependencies and configuration to be packaged together providing a clean contract with the underlying operating system, offering maximum portability between execution environments.

Multiple containers may be run on the same physical or virtual machine, each sharing the Operating System kernel with other containers and each running as isolated processes in user space. By packaging code, configuration and other dependencies containers also enable improved efficiencies and consistency in the code delivery pipeline from development to production, improving productivity and reliability.

Because containers typically impose little or no overhead using Docker allows for far greater resource utilisation than full virtual machines allowing high levels of server consolidation and multi-tenancy. The low overhead of containers facilitates microservice architectures and elastic scale-out, paving the way for PaaS and Cloud migration.

1-3 Why Run GUI Applications in Docker

As the popularity of Docker explodes and it becomes one of the leading *application deployment platforms*, more and more people are recognising the benefits that the powerful application packaging features of containers can bring. Typically however, the vast majority of Docker examples on the web tend to focus on containerising **server** applications and relatively few people currently seem to be considering use-cases beyond packaging servers and services.

It is actually perfectly possible though to run *full GUI applications* in Docker containers, that is to say to run *native* graphical or video applications as opposed to applications that simply expose an HTML web interface, so perhaps the question to ask is “why bother.”

That question is fair enough, many people see Docker in the context of large data centre deployments or software development and DevOps, and on the desktop things like cluster environments and application portability might not *appear* particularly important. On the other hand Docker can help to solve many problems, for example: packaging applications in a platform agnostic way such that they “just work” irrespective of the Linux distribution they are being deployed to; allowing multiple different versions of an application to be deployed simultaneously; enabling untrusted applications to be tested more safely and **cleanly** removed; enabling licences to be more effectively managed; enabling better utilisation of high-end GPU rendering resources, etc.

2 Docker Basics

2-1 Enabling sudo

Throughout this book many instructions that require root privileges have been run using sudo, which is relatively common practice among Linux/Unix System Administrators.

For Ubuntu based distributions sudo is the default way to access root privileges, but for many other distributions including Debian, CentOS and Fedora sudo may need to be enabled. The following commands should be executed as root.

Debian

Install sudo.

```
apt-get install -y sudo
```

Add the required user to the sudo group.

```
usermod -aG sudo <username>
```

Edit the /etc/sudoers file *the safe way* .

```
visudo
```

Edit to allow users of group sudo to execute any command.

```
%sudo    ALL=(ALL:ALL) ALL
```

CentOS/Fedora

With CentOS or Fedora if a non-root user was enabled as an administrator at install time then sudo *should* already be enabled, if not:

Install sudo (CentOS).

```
yum install -y sudo
```

Install sudo (Fedora).

```
dnf install -y sudo
```

Add the required user to the wheel group.

```
usermod -aG wheel <username>
```

Edit the /etc/sudoers file *the safe way*.

```
visudo
```

Edit to allow users of group wheel to execute any command.

```
%wheel    ALL=(ALL:ALL) ALL
```

Alternatively, It is possible to allow a user that is in neither sudo nor wheel user groups to execute the sudo command by simply adding them to /etc/sudoers as follows:

```
<username> ALL=(ALL) ALL
```

2-2 Installing Docker

Before getting started, it is worth being aware that there have historically been a range of Docker related package names, including docker, docker-engine and docker.io. To add to the potential confusion, unfortunately there is even a docker package in Ubuntu that has nothing to do with containers and is actually a system tray for KDE3 or GNOME 2 docklet applications. As of Docker version 17.03.0 the official package names are **docker-ce** for Docker Community Edition and **docker-ee** for Docker Enterprise Edition.

Docker Community Edition is the new name for the free Docker products and includes the full Docker platform. According to Docker Inc. the intended audience for Docker Community Edition is developers and small teams looking to get started with Docker and experiment with container-based applications. Docker Enterprise Edition on the other hand is the commercial version supported by Docker Inc. and intended for business-critical deployments.

In order to install Docker the most *obvious* starting point is to visit the official Docker website <https://www.docker.com/>. The main website is very polished, however it is quite “corporate” in style and can be somewhat difficult and frustrating to navigate quickly from the home page. To find more useful technical information the best place to *actually* begin looking is <https://docs.docker.com/install/>.

For air-gapped systems with no Internet access, or systems with poor bandwidth, it is possible to download Docker packages and install manually from the following locations:

- **Ubuntu** : <https://download.docker.com/linux/ubuntu/>
- **Debian** : <https://download.docker.com/linux/debian/>
- **Raspbian** : <https://download.docker.com/linux/raspbian/>
- **CentOS** : <https://download.docker.com/linux/centos/>
- **Fedora** : <https://download.docker.com/linux/fedora/>

For .deb based systems, navigate to **dists** to select the required OS version, then to either **pool/stable/** or **pool/edge**, then to the required architecture, e.g. **amd64** , **arm64** , **armhf** , **s390x** , **ppc64el** to find the package.

For .rpm based systems navigate to the required OS version, then to either **x86_64/stable/Packages/** or **x86_64/edge/Packages/** to find the package.

The *recommended* installation approach however is to set up Docker's package repository for the required distribution and install from there, which provides a relatively seamless installation and upgrade experience.

The following instructions describe how to install from Docker's package repository for a range of distributions. Note that these instructions refer to the **docker-ce** URL path

[**https://download.docker.com/linux**](https://download.docker.com/linux) , for **docker-ee** go to [**https://store.docker.com/my-content**](https://store.docker.com/my-content) and for each subscription click **setup** to obtain the URL path to use in place of [**https://download.docker.com/linux**](https://download.docker.com/linux) .

2-2-1 Installing Docker on Ubuntu

Uninstall old versions

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

Set up the repository

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates \
    curl gnupg-agent software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
```

Verify that the fingerprint of the GPG key is: **9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88**

```
sudo apt-key fingerprint 0EBFCD88
```

Install the repository

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
```

Or replace “bionic stable” with e.g. “artful edge” etc.

Install Docker

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

2-2-2 Installing Docker on Linux Mint

As Linux Mint is derived from Ubuntu the instructions for installing Docker on Ubuntu largely apply for Linux Mint too, however in the *Install the Repository* step it is necessary to map the versions as follows:

For Linux Mint versions:

- Qiana, Rebecca, Rafaela, Rosa

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu trusty stable"
```

For Linux Mint versions:

- Sarah, Serena, Sonya, Sylvia

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu xenial stable"
```

For Linux Mint versions:

- Tara, Tessa, Tina, Tricia

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
```

2-2-3 Installing Docker on Debian

Uninstall old versions

```
sudo apt-get remove docker docker-engine docker.io containerd r  
unc
```

Set up the repository

```
sudo apt-get update  
sudo apt-get install apt-transport-https ca-certificates \  
curl gnupg2 software-properties-common  
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo  
apt-key add -
```

Verify that the fingerprint of the GPG key is: **9DC8 5822 9FC7
DD38 854A E2D8 8D81 803C 0EBF CD88**

```
sudo apt-key fingerprint 0EBFCD88
```

Install the repository

```
sudo add-apt-repository "deb [arch=amd64] https://download.dock  
er.com/linux/debian stretch stable"
```

Or replace “stretch stable” with e.g. “buster edge” etc.

Install Docker

```
sudo apt-get update  
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

2-2-4 Installing Docker on CentOS

Uninstall old versions

```
sudo yum remove docker docker-client docker-client-latest docke  
r-common docker-latest docker-latest-logrotate docker-logrotate  
docker-engine
```

Set up the repository

```
sudo yum install -y yum-utils lvm2 \  
device-mapper-persistent-data
```

Install the repository

```
sudo yum-config-manager --add-repo \  
https://download.docker.com/linux/centos/docker-ce.repo
```

Install Docker

```
sudo yum install docker-ce docker-ce-cli containerd.io
```

If installing from a recently added repository, you will be prompted to accept the GPG key, and the key's fingerprint will be shown. The fingerprint should match **060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35**.

Start Docker

```
sudo systemctl start docker  
sudo systemctl enable docker
```

2-2-5 Installing Docker on Fedora

Uninstall old versions

```
sudo dnf remove docker docker-client docker-client-latest docke  
r-common docker-latest docker-latest-logrotate docker-logrotate  
docker-selinux docker-engine selinux docker-engine
```

Set up the repository

```
sudo dnf -y install dnf-plugins-core
```

Install the repository

```
sudo dnf config-manager --add-repo \  
https://download.docker.com/linux/fedora/docker-ce.repo
```

Install Docker

```
sudo dnf install docker-ce docker-ce-cli containerd.io
```

If installing from a recently added repository, you will be prompted to accept the GPG key, and the key's fingerprint will be shown. The fingerprint should match **060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35**.

Start Docker

```
sudo systemctl start docker  
sudo systemctl enable docker
```

2-3 The docker Group and sudo

An important caveat to bear in mind for the following section is that **for production systems it is, in general, best to sacrifice convenience for security**. For “playing around” however, especially in disposable VMs, it is sometimes useful to be able to avoid a bit of typing.

For simplicity and convenience throughout the rest of this book most of the example Docker commands, such as `docker run` etc., are shown **without** the sudo prefix. In order to run most Docker *management* commands however, it is currently necessary for the user to have *root privileges* to connect to the Docker daemon socket, so just using `docker run` without using the sudo prefix doesn’t *generally* work out of the box.

If you don’t want to use sudo when you use the docker command, create a Unix group called “docker” (usually created by the Docker package installation) and add the required users to it. When the docker daemon starts, it makes the Docker daemon socket read/writable by the docker group.

Add the docker group if it doesn’t already exist:

```
sudo groupadd docker
```

Add the required user <username> to the docker group:

```
sudo usermod -aG docker <username>
```

Then either run `newgrp docker`, or log out and back in to activate the changes to groups.

Note again that the docker group is root equivalent. The *Docker daemon attack surface* document from the Docker documentation website illustrates why **only trusted users** should be allowed to control your Docker daemon <https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface>, readers are advised to familiarise themselves with the risks.

Although many of the examples illustrated in this book omit the sudo prefix for brevity, the scripts in the GitHub repository at <https://github.com/fadams/docker-gui> which accompany the book actually incorporate a test to check if the user running them is a member of the docker group, if not the scripts automatically apply the sudo prefix as follows:

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi
```

It is also important to note that the scripts themselves shouldn't be run as sudo, only the `docker run` command within them as, in general, many of the scripts try to reduce the privileges of the containers being launched to those of the user running the script.

2-4 Hello Docker

In order to verify that Docker has been installed correctly, simply type `docker` at the command prompt, which will return the usage of the docker command. For brevity `...` indicates that some of the more verbose usage information has been omitted.

```
docker

Usage: docker COMMAND

A self-sufficient runtime for containers

Options:
  ...
    -v, --version          Print version information and quit

Management Commands:
  ...

Commands:
  ...

Run 'docker COMMAND --help' for more information on a command.
```

For example, in order to display the version of the current Docker installation:

```
docker -v
Docker version 18.06.1-ce, build e68fc7a
```

To check the Docker installation is running correctly, run the *hello-world* example from Docker Hub as follows:

```
docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:445b2fe9afea8b4aa0b2f27fe49dd6ad130dfe7a8fd0832b
e5de99625dad47cd
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://cloud.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/engine/userguide/>

2-5 Container Lifecycle Management

In the preceding example the *hello-world* image was pulled from the Docker Hub registry. Once pulled, Docker creates a container from that image and executes the **/hello** command.

Listing Containers

The list of *running* containers may be found as follows:

```
docker ps
```

Because the *hello-world* example simply runs and terminates, once it has displayed its output the container has done its job and it is stopped. The container is, however, not totally gone and may be seen by using the `docker ps -a` command:

CONTAINER ID	IMAGE	COMMAND	CREATED	PORTS
NAMES	STATUS			
a02d24a1ef95	hello-world	"/hello"	About an hour ago	Above
wonderful_knuth	Exited (0)	About an hour ago		

Adding the `-q` (quiet) option to the `ps` command allows us to list only the numeric IDs of the containers, which is useful if we wish to apply other commands to multiple containers:

```
docker ps -qa  
a02d24a1ef95
```

Removing Containers

To remove a container we can use `docker rm`, specifying either the container's ID or its name, e.g.

```
docker rm a02d24a1ef95
```

or

```
docker rm wonderful_knuth
```

Although it is straightforward to remove containers with `docker rm`, it should be clear that a usage pattern comprising many short lived “command” containers could easily result in a large number of *garbage* containers.

One simple way to tidy up garbage containers is to combine `docker rm` with the `docker ps -qa` command mentioned earlier.

```
docker rm $(docker ps -qa)
```

To avoid creating unnecessary garbage in the first place it is also possible to automatically remove the container when it exits by using `docker run` with the `--rm` option e.g.

```
docker run --rm hello-world
```

In this case when we run `docker ps -a` again we can see that the container has been cleanly removed leaving the system in a far tidier state.

Creating and Starting Containers

We saw previously how `docker run` combines several actions such as pulling an image from a registry (if necessary), creating a container from the pulled image, then starting it.

Given the *hello-world* image pulled by the previous examples we can perform these steps *separately* as follows:

```
docker create hello-world
```

Creates a container from the image, the ID or name of which must be obtained via `docker ps -a` as before.

```
docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
ATED                NAMES
23641b51e05e        hello-world        "/hello"           4 minutes ago     Exited (0) 46 seconds ago
inutes ago          stoic_shirley
```

Alternatively the `--name` option may be used.

```
docker create --name hello hello-world
```

Given the ID or name of the container it may be started thus:

```
docker start -a stoic_shirley
```

In this example the `-a` (or `--attach`) option is important as that attaches `STDOUT` and `STDERR` to the container, which is clearly necessary if it is to display its output.

interactive containers

Running *hello-world* is a useful way to begin illustrating container lifecycle management, but running a full terminal session within a container is better. To run a Debian container that executes /bin/bash you will need to use the -t (allocate a pseudo-TTY) and -i (keep STDIN open even if not attached) options to get a proper interactive session.

```
docker run --rm -it debian:stretch-slim /bin/bash
Unable to find image 'debian:stretch-slim' locally
stretch-slim: Pulling from library/debian
e7bb522d92ff: Pull complete
Digest: sha256:50a283f43b55f68914da6282263bf81f1e440b6d10565072
ef5abd3673df29e8
Status: Downloaded newer image for debian:stretch-slim
root@62a2ec960eac:/#
```

With this container running we have a full interactive bash session, which may be exited simply by using `exit` in the shell, and as we have specified `-rm` the container will be removed when we exit.

It is also possible to stop the container using `kill` (which will send `SIGKILL`) or `stop` (which will send `SIGTERM`) e.g.

```
docker kill 62a2ec960eac
```

or

```
docker stop 62a2ec960eac
```

2-6 Image Lifecycle Management

Docker containers are built from *images*, which comprise an ordered collection of root filesystem changes and their associated execution parameters known as *layers*. An image is **immutable** and typically contains a stacked union of layered filesystems.

An image needs to be passed to the docker run or create commands, and in the preceding examples the images used were *hello-world* and *debian:stretch-slim*. Docker does not hold these images locally and pulls them from the Docker Hub registry <https://hub.docker.com/>, a catalogue from which the Docker client can download pre-built Docker images.

When a container is created a new writable “container layer” is stacked on top of the underlying image layers. All changes made to the running container such as writing, modifying and deleting files, are written to this thin writable layer. The main difference between a container and image is the top writable layer, all writes to the container that modify or add data are stored there. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because all changes are stored in the writable container layer a single underlying image may be shared by multiple containers, each having their own state. Docker employs a copy-on-write strategy so any files the container does not change do not get copied to the writable layer, which means that the writable layer is kept as small as possible.

Listing Images

The list of available images may be found as follows:

docker images				
REPOSITORY	TAG	IMAGE ID	CRE	
ATED	SIZE			
debian	stretch-slim	9a091084b515	2 w	
eeks ago	55.3MB			
hello-world	latest	f2a91732366c	5 w	
eeks ago	1.85kB			

Alternatively, as of Docker 1.13 the *image* Management Command may be used to accomplish the same thing.

docker image ls				
REPOSITORY	TAG	IMAGE ID	CRE	
ATED	SIZE			
debian	stretch-slim	9a091084b515	2 w	
eeks ago	55.3MB			
hello-world	latest	f2a91732366c	5 w	
eeks ago	1.85kB			

Removing Images

To remove an image we can use `docker rmi` or alternatively `docker image rm`, specifying either the image's ID or its name, e.g.

```
docker rmi f2a91732366c
```

or alternatively:

```
docker image rm hello-world
```

Creating Images from a Container

Given the *debian:stretch-slim* image pulled by the previous examples we can create a new *container* from the image as follows, again using the *-i* and *-t* options as we want to start this container as an interactive session:

```
docker create -it debian:stretch-slim
```

We can check that it is not yet running by using `docker ps` and we can obtain the name and ID of the container as before.

CONTAINER ID	IMAGE	COMMAND	C
CREATED	STATUS	PORTS	
NAMES			
04d64284e59e	debian:stretch-slim "bash"		4
0 minutes ago	Exited (0) 3 minutes ago		
sharp_khorana			

The container can be run as follows, noting the *-i* (attach container's STDIN) and *-a* (attach STDOUT/STDERR and forward signals) options needed to start an interactive session.

```
docker start -ia sharp_khorana
```

Note that the the above procedure for creating a new container and starting an interactive session could be also accomplished by simply doing the following:

```
docker run -it debian:stretch-slim
```

As mentioned previously, all changes made to a running container, such as writing, modifying and deleting files, are written to a thin writable layer stacked on top of the underlying image layers. Docker lets you start a container using a base image, interactively make changes to it and then keep those changes by committing them into a new image.

To explore this we first go back to the interactive bash session that was created earlier and update the packages in the container, we then install the `procps` package to add the `ps` command to the container.

```
root@04d64284e59e:/# apt-get update && apt-get install -y procps
Get:3 http://security.debian.org stretch/updates InRelease [63.0 kB]
...
The following additional packages will be installed:
  libgpm2 libncurses5 libprocps6 psmisc
Suggested packages:
  gpm
The following NEW packages will be installed:
  libgpm2 libncurses5 libprocps6 procps psmisc
...
Setting up libprocps6:amd64 (2:3.3.12-3) ...
Setting up procps (2:3.3.12-3) ...
update-alternatives: using /usr/bin/w.procps to provide /usr/bin/w (w) in auto mode
update-alternatives: warning: skip creation of /usr/share/man/man1/w.1.gz because associated file /usr/share/man/man1/w.procps.1.gz (of link group w) doesn't exist
Processing triggers for libc-bin (2.24-11+deb9u1) ...
```

Now that we've made changes to the container we'd like to be able to keep them. To do this we first exit the container, which will still be available until it is removed by `docker rm`. We then commit the changes we have made to the container to create a new image called, in this case, `debian:ps`.

```
docker commit 04d64284e59e debian:ps
```

We can check the image exists as follows:

docker images			
REPOSITORY	TAG	IMAGE ID	CRE
ATED			
debian	ps	449a1512a581	15
seconds ago	73.1MB		
debian	stretch-slim	9a091084b515	2 w
eeks ago	55.3MB		

At this point we can remove the stopped container.

```
docker rm 04d64284e59e
```

We can check that the new image is as we expect with:

```
docker run --rm debian:ps ps
```

We can also create a new image with different capabilities, for example we can change the entrypoint to run `ps` directly thus:

```
docker commit --change='ENTRYPOINT ["ps"]' 04d64284e59e ps  
docker run --rm ps
```

Creating Images from a Dockerfile

Although making changes to a container manually then committing them to a new image can be useful, it is not a readily reproducible nor automated approach for creating Docker images.

A better solution is to create a *Dockerfile*, which is a recipe of instructions that will let Docker build an image *automatically and repeatably*.

A simple Dockerfile for the previous example looks like this:

```
FROM debian:stretch-slim

# Install procps
RUN apt-get update && apt-get install -y procps

ENTRYPOINT ["ps"]
```

The convention is for Dockerfiles to be called simply “Dockerfile”, though other names may be used.

The `docker build` command is used to build the ps image from the Dockerfile above.

```
docker build -t ps .
```

As with the earlier example where we created the ps image from a container, we can create a ps container from the image and run it as follows:

```
docker run --rm ps
```

2-7 Managing Containers with Portainer

Portainer (<http://portainer.io/>) is a simple management UI for Docker (not connected in any way to the author of this book).

The source code for Portainer is available on GitHub (<https://github.com/portainer/portainer>), but Portainer is itself Dockerised and may be installed and run by simply running the following command:

```
docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock portainer/portainer
```

Running the above command pulls the [portainer/portainer](#) image from [Docker Hub](#), installs the image, then runs a container in the background in detached mode via the `-d` option. The `-p 9000:9000` option binds the container port 9000 to host port 9000 and the

`-v /var/run/docker.sock:/var/run/docker.sock` option provides the container with access to the Docker daemon socket, which allows processes running in that container to extract information about other containers.

Portainer is a really useful and intuitive way to reduce the initial barrier to entry of Docker and it is especially useful for helping to tidy up “orphaned” containers and images, which can be fairly easily missed when starting out with Docker and unfamiliar with the basic Docker Command Line Interface.

Full Portainer documentation may be found at <https://portainer.readthedocs.io/en/stable/>.

The following diagram illustrates the image management interface of Portainer showing a range of images currently installed on the host <myhost>.

The screenshot shows the Portainer interface running in Mozilla Firefox. The URL in the address bar is `myhost:9000/#/images/`. The left sidebar has a dark blue background with white text and icons. The 'Images' option is selected, indicated by a blue icon and a white background. The main content area has a light gray background. At the top, there's a 'Pull image' form with fields for 'Name' (containing 'e.g. ubuntu:trusty') and 'Registry' (with a note: 'leave empty to use default'). Below this is a table listing ten Docker images, each with a checkbox, ID, tags, size, and creation date. The table has columns for Id, Tags, Size, and Created. The first image listed is `sha256:b2ec6e5c3e...` with tags `vic-bluray:latest`, size `487.3 MB`, and created on `2018-12-24 14:32:50`.

	Id	Tags	Size	Created
<input type="checkbox"/>	sha256:b2ec6e5c3e...	vic-bluray:latest	487.3 MB	2018-12-24 14:32:50
<input type="checkbox"/>	sha256:e52b87b3fe...	virtualgl:latest	131 MB	2018-12-08 10:53:59
<input type="checkbox"/>	sha256:93fd78260b...	ubuntu:18.04	86.2 MB	2018-11-19 21:20:42
<input type="checkbox"/>	sha256:d449426bc5...	portainer/portainer:latest	9.2 MB	2017-01-31 22:18:51
<input type="checkbox"/>	sha256:df57f2312c...	firefox:latest	497.8 MB	2018-10-07 09:19:45
<input type="checkbox"/>	sha256:8638e9dfba...	firefox-flashdrm:latest	545.7 MB	2018-10-11 13:36:51
<input type="checkbox"/>	sha256:9740e52f00...	extract-libpepflashplayer:latest	89.2 MB	2018-10-11 14:25:39
<input type="checkbox"/>	sha256:184356db7d...	debian-stretch-slim	55.3 MB	2018-07-17 01:28:04
<input type="checkbox"/>	sha256:c47214d7f6...	build-libfreshwrapper:latest	74.9 MB	2018-10-11 14:38:00

2-8 Image Optimisation

The Dockerfiles presented in this book have attempted to provide a reasonable compromise between clarity and efficiency by using “slim” versions of official base images and, where possible, minimising the number of layers created by combining commands into a single RUN statement.

Image optimisation is not however the primary focus of this book and one particular point to note is that the majority of the examples are deliberately presented as self-contained Dockerfiles with minimal dependencies on the other examples. By choosing to have each example self-contained in this way the main compromise has been to lose some of the potential efficiency benefits of Docker’s layered filesystem.

To illustrate what this means in practice consider that many applications are built on libraries such as the GTK+ toolkit, thus each application will pull the required library and all its dependencies into the image. Where many applications are being built a more efficient approach might be to first create a GTK base image comprising the main libraries and dependencies and use that image as the application base.

Ultimately, the optimal set of compromises will very much depend on the deployment scenarios employed by each individual reader. For those deploying many applications, creating a layered set of base images might be the most efficient approach, whilst for others carefully optimising each image individually might well be the best approach.

3 X11 Overview

3-1 Introduction

The X Window System is a networked graphical window system that was originally developed at MIT in 1984. There have been several versions of X and the most recent version, known as X11, was first released in 1987.

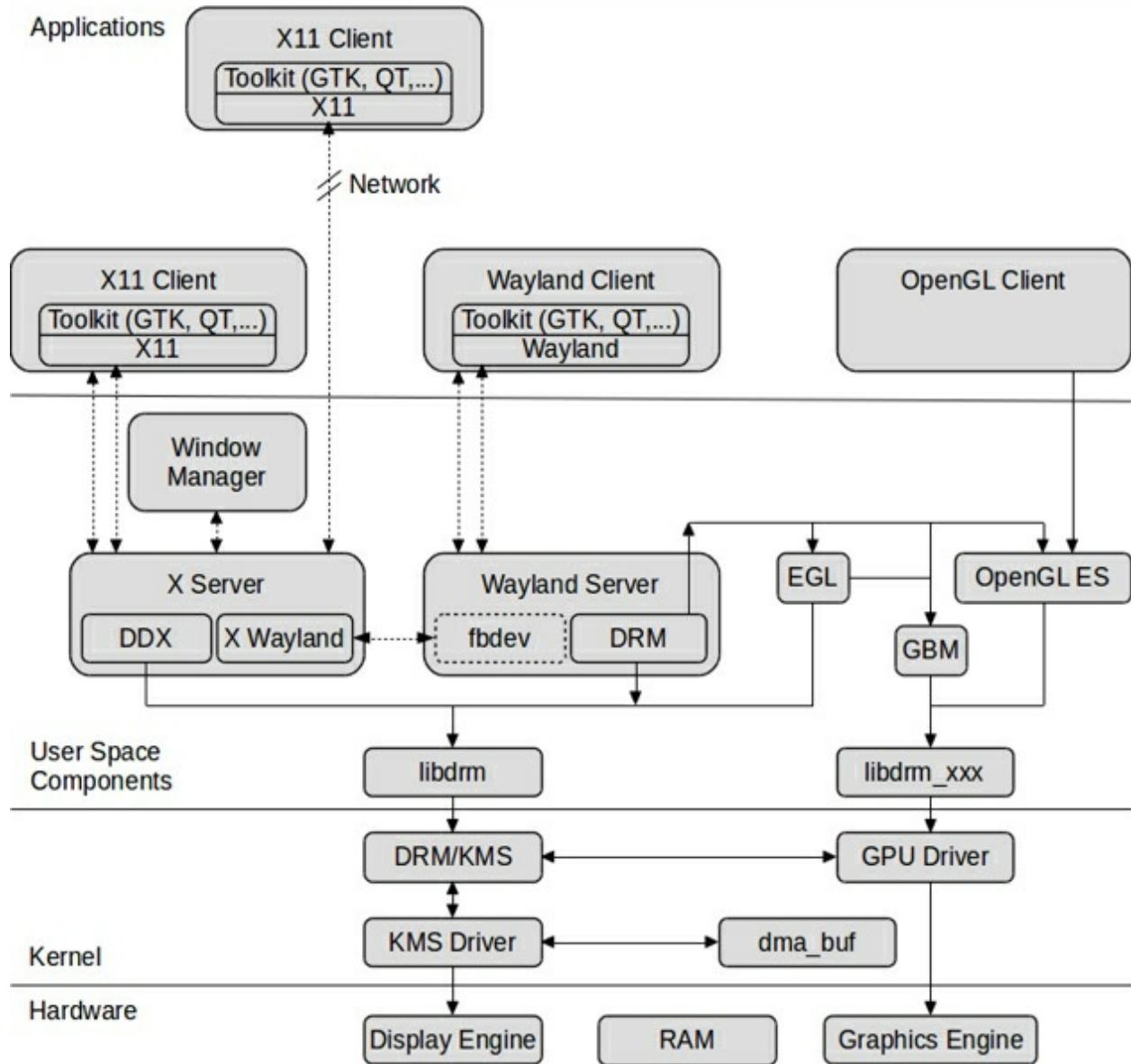
X11 has been adopted as something of a de facto standard, particularly on Unix-like systems, although the modern range of desktops, widgets and extensions would probably make it unrecognisable to the original developers.

3-2 X11 Architecture

The majority of window systems are *kernel-based* and tightly coupled to the operating system. By contrast, the X Window System was designed from the outset to be operating system agnostic and entirely comprises user space processes.

X is based on a *client-server* architecture comprising two key components: *X servers* that provide the display rendering capabilities and user input handling and *X clients*, which are the application programs. Note that with X the server is local to the user and it is the *client* that may be remote, which can often confuse newcomers to X as that is the opposite arrangement to how most people visualise clients and servers, where it is the *server* that is usually the remote entity.

The graphics architecture of a modern Linux system is illustrated below showing the X11, Wayland and OpenGL paths for completeness.



In this illustration the user space components, kernel and hardware are local to the user, as are Wayland and OpenGL Applications, but X11 Applications may be local to the user or remote due to the network based nature of the X11 protocol.

3-2-1 The X Server

The X Server tracks the output from any running clients, updating the display to reflect that output. The X Server also handles input from the keyboard, mouse, etc. forwarding events to the relevant client applications.

User input and some other types of information like inter-client communication information, window state, and region exposure are passed from server to client as *events*. An event is simply a packet of information that informs the client of something that it needs to act upon, such as mouse movement.

When a client receives an event, it responds by issuing a request to the server for some action affecting the display to be performed, for example requesting that a window be resized or moved. The server responds to requests by clients by updating the appropriate windows.

In a modern X Server like the X.Org Server, the server is layered into two key parts. Device Independent X (**DIX**) implements the core X protocol and interacts with clients, implementing software rendering and event delivery. Device Dependent X (**DDX**) interacts with the hardware, either directly through other user space libraries such as libdrm to interact with the Direct Rendering Manager (DRM) or indirectly via libraries like OpenGL which can provide an additional layer of hardware independence, such as in the Glamor hardware independent 2D acceleration driver, which translates X render primitives to OpenGL operations.

3-2-2 X Clients

An X client is an application program that communicates with the X Server via the X protocol. The X Window System allows multiple clients to run simultaneously, each displayed in a separate window and each potentially running on a different host due to the network transparency of the X protocol.

In addition to communicating with the server, X clients also need to be able to communicate with other clients. For example a client may wish to tell the window manager, which is itself just another X client, where to place its icon. This inter-client communication is facilitated via properties, which are pieces of information associated with a window or font and stored in the server. Storing properties in the server makes the information visible to all clients.

One common use for properties in inter-client communication is when a client wishes to tell the window manager the name of the application associated with its window. The name might typically correspond to the client's name by default, but in many cases clients may specify a different name when run, such as a text editor including the name of the file being edited as part of its name. The client application's name is stored in the server in the property called WM_NAME.

The X Window System has no standard UI/UX interface nor design guidelines, which has resulted in several significantly different interfaces from XView/OLIT to CDE & Motif, though KDE/Qt and GNOME/GTK have now come to dominate.

3-2-3 The Window Manager

In contrast to kernel-based window systems, where the way the window system operates is inherent to the operating system, control of the X Window System is largely concentrated in a user space process called the window manager, which is really just another X client. The look and feel of the desktop of a particular system is largely determined by the window manager being used.

The window manager controls the placement and appearance of windows and commonly includes features to allow the user to open, close, minimize, maximize, move, resize, and keep track of running windows, including window decorators. Many common window managers come with a range of utilities and “applets” such as docks, task bars, application launchers, desktop icons, and wallpaper.

In modern systems window managers tend to be *compositing*, which provide applications with an off-screen buffer for each window. The window manager composites the window buffers into an image representing the screen and writes the result into the display memory and since the screen is double buffered, it does not flicker during updates.

The most common window managers currently in use include Muffin used by Cinnamon, Mutter used by GNOME and KWin used by KDE, though lightweight distributions favour window managers such as Xfwm used by Xfce and JWM, a stacking window manager used in Damn Small Linux and Puppy Linux.

3-2-4 Network and Interprocess Communication

It is a fundamental design principle of the X Window System that clients and servers should operate independently and be platform agnostic.

Device independence and network transparency however come at a cost, though in practice most of the overhead is in the form of latency from network round-trip delay which can be minimised by good application design. To facilitate efficient IPC where the X Server and clients are running on the same host modern X11 implementations use Unix domain sockets and shared memory (via the MIT-SHM extension).

Although the X protocol was well designed, and somewhat ahead of its time in 1984, it has a number of issues. Perhaps the most serious issue is that the network traffic between the X Server and X clients is not encrypted making a man-in-the-middle attack fairly trivial, though this is easily resolved by using an SSH tunnel. Another issue is that X clients cannot readily be disconnected from the server then reconnected, this problem is usually resolved by taking a different approach to network transparency, such as X Persistent Remote Applications (Xpra), Virtual Network Computing (VNC) or other Remote Desktop Protocols. Finally, modern 3D accelerated applications can be an issue and although the X11 GLX extension permits indirect rendering this can be inefficient and it is often desirable to be able to use more powerful GPUs that may exist on remote hosts. This book shall explore some options for remoting in later chapters.

3-3 Wayland and Weston

[Wayland](#) is a display server protocol between a compositor (a display server that is also a compositing window manager) and its clients, as well as a library implementing the protocol.

The Wayland protocol is a client-server protocol in which clients are graphical applications that can directly write data into the framebuffer using the EGL rendering API. EGL is used to avoid dependencies on X and also enables Weston to run on GPUs which don't support the full OpenGL API.

Wayland is intended as a simpler replacement for X, but although it is technically independent of X the large number of existing X Clients mean that X will be intertwined with Wayland for compatibility reasons for the foreseeable future, as illustrated by the diagram in the [X11 Architecture](#) section.

Weston is a reference implementation of a Wayland compositor also developed by the Wayland project. Weston has a number of back-ends that enable it to run directly on Linux KMS, another Wayland server or on X11. Weston also supports X Clients via XWayland, which provides backwards compatibility to X applications in a Wayland stack.

XWayland is activated by instructing Weston to load the XWayland module, e.g. `weston --xwayland`. Weston starts listening on a new X display socket, and exports it in the environment variable DISPLAY. When the first X client connects, Weston launches a special X server as a Wayland client to handle the X client and all future X clients.

4 Local Applications

The best place to start a journey is usually at the beginning, and when learning how to run GUI applications in Docker starting out the journey by running simple containers on a local machine communicating directly with the local display server certainly has the fewest dependencies and the least number of moving parts to go wrong.

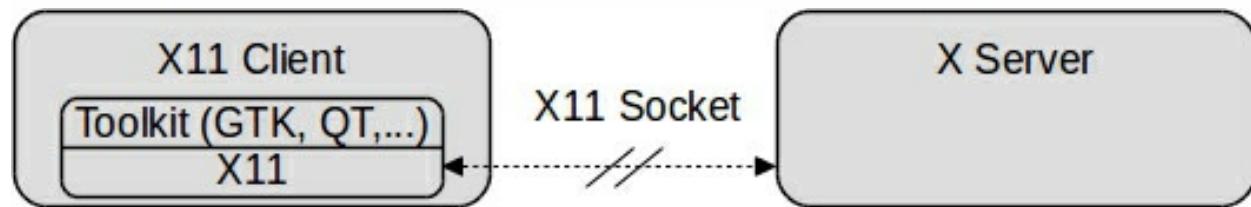
The source code for all of the examples in this book may be found on GitHub at <https://github.com/fadams/docker-gui> and can be easily installed by running:

```
git clone https://github.com/fadams/docker-gui.git
```

or by downloading <https://github.com/fadams/docker-gui/archive/master.zip> and unzipping.

4-1 Simple X11 Applications

The [X11 Architecture](#) section illustrated the Graphics Architecture of a modern Linux system. In order to identify a possible containerised architecture it is useful to consider a simplified diagram comprising just the X client and X Server.



In simple terms an X client is an application program that communicates with the X Server via the X protocol, usually using either the [Xlib](#) or [XCB](#) libraries to implement the client-side of the protocol. The X protocol is transmitted over the X11 socket to the X Server, which interprets the commands and renders them to the display.

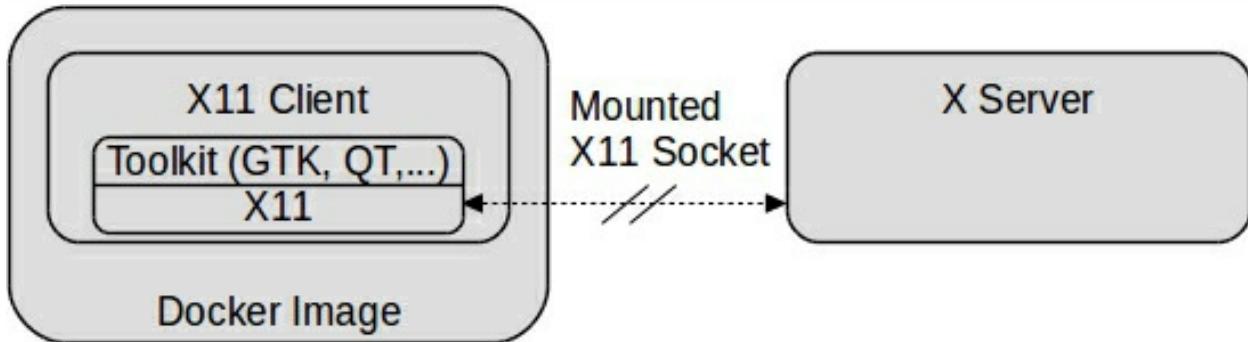
In order to create a containerised GUI application we will need to include the full X11 client stack in the container, including any toolkits such as GTK or Qt used by the application, in addition the application will need access to an X11 socket.

There are two approaches that may be used to provide a containerised client with access to an X11 socket:

- The container could share the host's X11 socket. Where the client and server are co-located the X11 socket is a Unix domain socket, so sharing it is simply a case of bind-mounting the socket's file system name as a volume into the container.
- The container could bundle a headless X Server such as [Xvfb](#) or [Xorg with a dummy device](#) and use a VNC server or similar remote desktop application to capture and export the virtual framebuffer display.

The second of these approaches is a little more complex, but potentially provides the highest level of container isolation albeit at the cost of some performance and efficiency. It will be covered later in the context of remote applications, though it may also be used for highly isolated local applications.

The first approach is conceptually very simple and extremely lightweight, comprising the following architecture:



Directly sharing the host's X11 socket with the container (and possibly `--ipc=host` too if the application makes use of the MIT-SHM extension) is the most efficient and highest performance approach for running a GUI application in a container, indeed this approach has essentially the same performance as running the application natively on the host.

Although this approach maximises performance, at face value it also breaks container isolation. In particular allowing the container to connect directly to the host's X11 socket allows a malicious application in the container to perform keylogging on any window and (potentially) allows remote host control.

It is important to note however that the X11 socket being passed to the application container does not *necessarily* have to be the *host's* X11 socket. That is to say, this simple approach could also be considered as part of a more modular solution whereby a **second** container could bundle a headless X Server, VNC server etc. and we could instead expose *that* X Server's X11 socket to the application's container rather than the host's.

Such a modular approach follows a similar pattern to bundling a headless X Server in the application container, but is arguably a more microservice pattern, this will be explored later in the book when we discuss remote applications.

4-1-1 x11-apps

The [Dockerfile](#) and launch scripts for our first containerised GUI application are in the [x11-apps](#) directory of [docker-gui](#) under [4-local-applications/simple-X11-applications](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    x11-apps && \
    rm -rf /var/lib/apt/lists/*
CMD ["xlogo"]
```

This is a fairly trivial Dockerfile that uses debian:stretch-slim as a base image, installs the x11-apps package and sets xlogo as the default command. To build the image simply run:

```
docker build -t x11-apps .
```

And check that the image has been successfully built:

```
docker image ls
REPOSITORY          TAG      IMAGE ID      C
REATED              SIZE
x11-apps            latest   77721f2519a0   1
6 seconds ago       72.4MB
```

In order to run the application the “trick”, as discussed previously, is to mount the X11 socket as a volume which can be mounted read-only, and export the DISPLAY environment variable to the container, that is to say the `docker run` command should look like [xlogoVo.sh](#):

```
docker run --rm \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
x11-apps
```

Unfortunately however that's not quite enough, as X11 authentication is likely to prevent connections from the container, giving an error message similar to the following:

```
No protocol specified
Error: Can't open display: unix:0
```

The simplest way to resolve that is to use the [xhost](#) program, which is used to add and delete host names or user names to the list allowed to make connections to the X server.

Ideally xhost should be used with the *Server Interpreted* family name to provide *user-based* controls as follows:

```
xhost +si:localuser:root
```

However some Linux distributions, Ubuntu 14.04 based distributions in particular, don't seem to honour this command despite saying **localuser:root being added to access control list**.

The only *consistent* xhost based approach seems to be to allow all non-network local connections to the X Server via:

```
xhost +local:
```

So the script to invoke the `docker run` command should look like [xlogoV1.sh](#):

```
xhost +local: # Add non-network local connections to ACL.  
docker run --rm \  
    -e DISPLAY=unix$DISPLAY \  
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \  
    x11-apps  
xhost -local: # Remove non-network local connections.
```

All being well, this command should enable access to the local X Server for local connections, export the DISPLAY environment variable to the container and mount the X11 socket as a volume. The command should then run the container's default command xlogo, which should display a window similar to the following.



Finally, when the window is closed, access to the local X Server for local connections granted earlier will be removed.

Although using xhost to enable access to the X Server works, in general **it is a bad idea** to use xhost. In particular, when things don't work users often resort to simply trying:

```
xhost +
```

Doing that on a network connected host will **disable all X Server access control** and could inadvertently give *every host on the Internet* full access to the X Server, **don't do it!**

To be fair to xhost, as previously mentioned *in theory* it does actually support user level authentication via:

```
xhost +si:localuser:<user>
```

but in practice that doesn't seem to be supported on all Linux distributions, moreover the Server Interpreted access control family is limited to local connections, using system calls to ask the kernel what user is on the other end of a local socket.

A better approach for X Window System authorisation is to use token (cookie) based authorisation. When using MIT-MAGIC-COOKIE-1, the client sends a 128 bit cookie along with the connection setup information. If the cookie presented by the client matches one that the X server has, the connection is allowed access. The cookie is chosen so that it is hard to guess; xdm generates such cookies automatically when this form of access control is used. The user's copy of the cookie is usually stored in the .Xauthority file in the home directory, although the environment variable XAUTHORITY can be used to specify an alternative location.

The simplest approach to take for token based authentication in containers is to bind-mount the user's .Xauthority file as a volume and use the XAUTHORITY environment variable to specify its location. Typically, however, .Xauthority files are *hostname specific*. It is therefore necessary to set the container's hostname to that of the host running the X Server, but it's not ideal to run multiple containers with the same hostname though Docker will let you. Note too that this approach will not work correctly on Wayland/XWayland based systems.

The `docker run` command should look like [xlogoV2.sh](#):

```
docker run --rm \
    -h $(hostname) \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$XAUTHORITY \
    -v $XAUTHORITY:$XAUTHORITY:ro \
    x11-apps
```

To take this idea a step further, in order to avoid setting the container's hostname it is necessary to create an *additional* .Xauthority file **with a wildcard hostname** as follows:

```
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
```

Noting that the .Xauthority is recreated with a different cookie every time the X Session starts, so it is necessary to recreate the wildcarded .Xauthority then too.

These commands create an additional .Xauthority file called .Xauthority.docker (or if XAUTHORITY was used to specify an alternative name a .docker file based on that name is created).

The **xauth nlist \$DISPLAY | sed -e ‘s/^..../ffff/’ | xauth -f** line sets the first 16 bits of the nlist output, which switches the Authentication Family to “Family Wild”, allowing user based authentication from any host.

The script to invoke the `docker run` command should look like [xlogoV3.sh](#):

```
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^..../ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
docker run --rm \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
x11-apps
```

We now have a container doing token based authentication with the user's X Server, but we could do better as this approach will not work correctly on Wayland/XWayland based systems. If we run the following in another terminal we may see that the container is running with elevated privileges:

```
docker exec -it $(docker ps -q -f ancestor=x11-apps) /bin/bash
```

```
root@4c22c369d58c:/# whoami  
root
```

Now applications running as root *inside* containers are not as bad as applications running as root *natively*, but it is still *good practice* to reduce the privileges and is **necessary** on systems running Wayland, which uses uid for authentication.

The script to invoke the `docker run` command of our final version should look like [xlogoV4.sh](#):

```
# Create .Xauthority.docker file with wildcarded hostname.  
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}  
DOCKER_XAUTHORITY=${XAUTH}.docker  
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY  
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE  
R_XAUTHORITY nmerge -  
  
docker run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-e DISPLAY=unix$DISPLAY \  
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \  
-e XAUTHORITY=$DOCKER_XAUTHORITY \  
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \  
x11-apps
```

This is *very similar* to our previous attempt, however we have now reduced the privileges of the container by setting the user

`-u $(id -u):$(id -g)`. Note that we have to use the *numeric ID* rather than username because the opencontainers specification requires that the user id passed to [runc](#) in Linux is a number, and resolving user names is *left to the layers above* (see [moby/issues/22323](#)).

Using `-v /etc/passwd:/etc/passwd:ro` to mount `/etc/passwd` read-only is really what the *layers above* bit of the previous paragraph amounts to. It's not *strictly* necessary in order to run the application in the container with reduced privileges, but passing it means that the user name as well as user ID in the container will match that of the user actually running the container, we can see this by again running:

```
docker exec -it $(docker ps -q -f ancestor=x11-apps) /bin/bash
```

Where the username is now the user running the container.

On systems running Wayland with XWayland when [xlogoV4.sh](#) is run we are likely to see warnings such as:

```
cp: cannot stat '/home/<user>/.Xauthority': No such file or directory
xauth:  /home/<user>/.Xauthority.docker not writable, changes will be ignored
xauth:  file /home/<user>/.Xauthority does not exist
xauth: (argv):1: unable to read any entries from file "(stdin)"
```

This is because Wayland uses uid to authenticate and XWayland does not create a `.Xauthority` file and ignores it if it exists. To run the examples without these warnings a dummy `.Xauthority` file should be created as follows:

```
rm -rf ~/.Xauthority && rm -rf ~/.Xauthority.docker
touch ~/.Xauthority
xauth add $(hostname)/unix:0 . 5ca1ab1edecea5eddeadbea7acce55e5
```

We can now create a more generalised version of the script [x11-apps.sh](#), which allows us to run any of the applications in the x11-apps package:

```
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
docker run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
x11-apps $@
```

The simple applications in the x11-apps package may be run as follows:

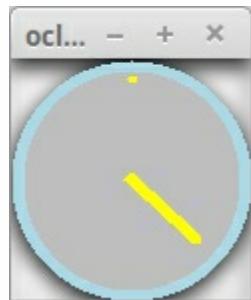
```
./x11-apps.sh xmag
```



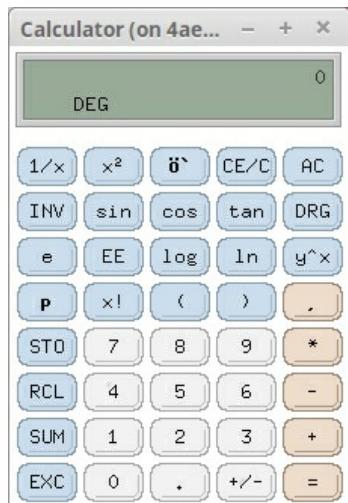
```
./x11-apps.sh xeyes
```



```
./x11-apps.sh oclock
```



```
./x11-apps.sh xcalc
```



4-1-2 xterm

Using what we have learned so far containerising [x11-apps](#) we should be able to follow a very similar pattern if we wish to containerise similar simple X11 applications such as [xterm](#).

The [Dockerfile](#) is in the [xterm](#) directory of [docker-gui](#) under [4-local-applications/simple-X11-applications](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    xterm && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["xterm"]
```

This Dockerfile simply installs the xterm package and sets the xterm executable as the entrypoint. To build the image run:

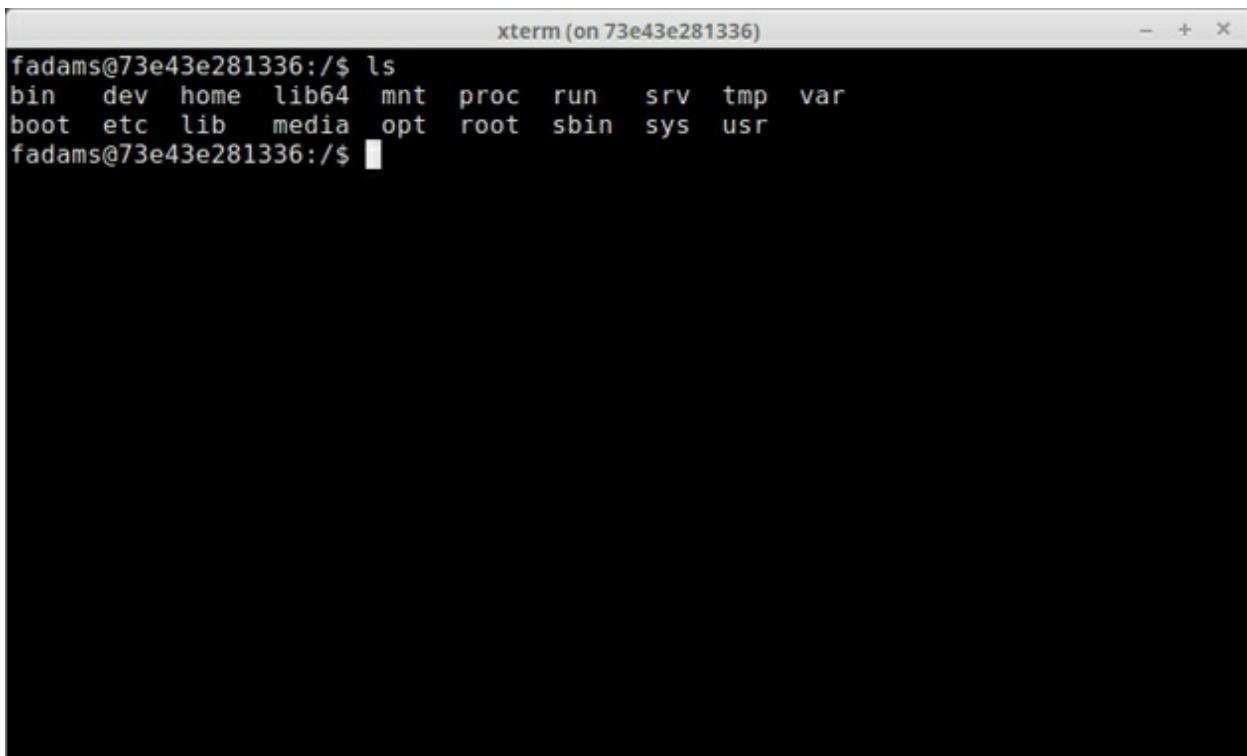
```
docker build -t xterm .
```

The [xterm.sh](#) launch script is as follows:

```
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
```

```
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
xterm $@
```

This should display a window similar to the following.



Note that xterm does not have a menu bar. To access xterm's three menus, users hold the control key and press the left, middle, or right mouse button.

4-2 GNOME, D-bus and AppArmor

4-2-1 gnome-calculator

Although the [x11-apps](#) provide a useful introduction to GUI applications they are not really representative of the sort of applications to be found on a modern Linux desktop, which are somewhat more complicated and which typically have many more system dependencies. A more representative, but still relatively simple, application is the [GNOME Calculator](#).

The [Dockerfile](#) is in the [gnome-calculator](#) directory of [docker-gui](#) under [4-local-applications/gnome-dbus-and-apparmor](#).

```
FROM debian:stretch-slim

# Install gnome-calculator
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    gnome-calculator && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["gnome-calculator"]
```

This Dockerfile installs the gnome-calculator package and sets gnome-calculator as the entrypoint. To build the image run:

```
docker build -t gnome-calculator -f Dockerfilev0 .
```

Note that in this case we supply a *named* Dockerfile, as we shall shortly discover that there is an issue with this image.

The launch script to invoke the `docker run` command should look like [`gnome-calculatorVo.sh`](#), which follows the same pattern that was used in our [`x11-apps.sh`](#).

```
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

docker run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
gnome-calculator
```

This should display a window similar to the following.



The good news is that the application clearly runs, but there are two issues: the first is that although it is possible to change the calculator mode (to Advanced, Programming etc.) configuration changes are not persisted, the second is the large number of warnings similar to the following (truncated) console output.

```
** (gnome-calculator:1): WARNING **: Couldn't connect to accessibility bus: Failed to connect to socket /tmp/dbus-HZZs6ejDt7: Connection refused

(gnome-calculator:1): GLib-GIO-CRITICAL **: g_dbus_proxy_new_sync: assertion 'G_IS_DBUS_CONNECTION (connection)' failed

(gnome-calculator:1): GLib-GIO-CRITICAL **: g_dbus_proxy_get_name_owner: assertion 'G_IS_DBUS_PROXY (proxy)' failed

(gnome-calculator:1): dconf-CRITICAL **: unable to create directory '/home/<user>/.cache/dconf': Permission denied. dconf will not work properly.

...
(gnome-calculator:1): GLib-Net-WARNING **: couldn't load TLS file database: Failed to open file '/etc/ssl/certs/ca-certificates.crt': No such file or directory

...
** (gnome-calculator:1): WARNING **: currency.vala:459: Couldn't download IMF currency rate file: Unacceptable TLS certificate

** (gnome-calculator:1): WARNING **: currency.vala:459: Couldn't download ECB currency rate file: Unacceptable TLS certificate
```

In the actual console output there are *a lot* of the Permission denied messages:

```
(gnome-calculator:1): dconf-CRITICAL **: unable to create directory '/home/<user>/.cache/dconf': Permission denied. dconf will not work properly.
```

We can get a good idea of the cause of this issue if we run:

```
docker exec -it $(docker ps -q -f ancestor=gnome-calculator) /bin/bash
```

and in the exec'ed shell run:

```
ls -al home
total 12
drwxr-xr-x 3 root root 4096 Jan  8 08:14 .
drwxr-xr-x 34 root root 4096 Jan  8 08:14 ..
drwxr-xr-x 2 root root 4096 Jan  8 08:14 <user>
```

We can see that the home directory of the user running the container is *owned by root* inside the container.

Unfortunately permission issues with mounted volumes are a well-known problem with Docker, and are probably one reason why many people leave containers running as root (if the

`-u $(id -u):$(id -g)` part of the run script is removed this error “goes away”). In our case mounting the `$DOCKER_XAUTHORITY` volume causes the file `/home/<user>/Xauthority.docker` to be created inside the container and although the `.Xauthority.docker` file itself has the correct permissions **its parent directories do not**.

Apart from having the container run as root, which we'd prefer to avoid, there are two ways around this.

The first approach is to have the Dockerfile create a non-root user with *useradd/groupadd* and switch to it with `USER <user>` as illustrated in the following Dockerfile snippet.

```
# Add user "appuser" so we can run as a non privileged
# user below. Note that the useradd and groupadd commands
# specify uid and gid of 1000, which is a fairly common
# value for the main user of desktop Linux systems.
# Replace 1000 with desired uid/gid found by running the
# command: id -u <user>
RUN groupadd -r -g 1000 appuser && useradd -u 1000 -r -g appuse
r -G audio,video appuser && \
    mkdir -p /home/appuser && \
        chown -R appuser:appuser /home/appuser
...
# Run application as non privileged user.
USER appuser
```

This approach is quite common and is good if the user being added is a “processing user”, as is likely to be the case for a server application, but if the user is a *desktop user* and we want to be able to persist application data on the host this approach is fiddly and tends to lead to non-portable images as each user in a multi-user system would have a different uid.

The second approach is to simply create the `/home/<user>` directory *on the host* and bind-mount it, that is to say we create a directory (with the correct ownership and permissions) on the host that we can use as a de facto “home directory” in the container for the user running the container.

The script to invoke the `docker run` command should look like [`gnome-calculatorV1.sh`](#), which is very similar to [`gnome-calculatorVo.sh`](#), but with the addition of a line to create the required host directory and a line to mount it:

```
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)
docker run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
gnome-calculator
```

Adding the “home directory” to the container certainly improves things, but many of the errors remain. Next to consider are the TLS related errors, like:

```
(gnome-calculator:1): GLib-Net-WARNING **: couldn't load TLS file database: Failed to open file '/etc/ssl/certs/ca-certificates.crt': No such file or directory
```

Fixing these is simply a case of installing the `ca-certificates` package using the following updated [`Dockerfile`](#):

```
FROM debian:stretch-slim

# Install gnome-calculator
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    ca-certificates libcanberra-gtk3-module \
    packagekit-gtk3-module gnome-calculator && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["gnome-calculator"]
```

and building it with:

```
docker build -t gnome-calculator .
```

The remaining errors all appear to be related to [dconf](#), which is a low-level configuration and settings management system that is part of GNOME 3 and uses [D-bus](#) as a message bus. It is also very likely that the dconf issues are the reason why changes made to the calculator mode are not persisted, as when the gnome-calculator is closed there are lots of errors of the form:

```
(gnome-calculator:1): dconf-WARNING **: failed to commit change
s to dconf: Failed to execute child process "dbus-launch" (No s
uch file or directory)
```

The author of this book doesn't claim to have any dconf/D-bus expertise, but it seems clear that there are two general approaches to solving this: one is to integrate with the host's dconf infrastructure; the other is to run a dconf/D-bus instance inside the container.

It is likely, given the author's relatively limited dconf knowledge, that the simplest initial option would be to integrate with the host's dconf infrastructure. This approach has the advantage that containerised instances of desktop UI components should work fairly seamlessly with native instances, but comes with the disadvantage that there is somewhat poorer application sandboxing, though that is likely to be no worse than that caused by exposing the X11 socket to the container.

The first place to look is the environment variable.

DBUS_SESSION_BUS_ADDRESS which, depending on the Linux distribution being used, looks something like this:

```
echo $DBUS_SESSION_BUS_ADDRESS  
unix:abstract=/tmp/dbus-Km3nUEFT0i,guid=f780f831fbf67724b0a0f04  
85a49403a
```

or this:

```
echo $DBUS_SESSION_BUS_ADDRESS  
unix:path=/run/user/1000/bus
```

The former seems to be used on Linux Mint, CentOS and Fedora, the latter on Debian and Ubuntu.

Bind-mounting the socket /tmp/dbus-Km3nUEFT0i in the former case or /run/user/1000/bus in the latter and passing the DBUS_SESSION_BUS_ADDRESS environment variable to the container, in an analogous way to how the X11 socket and DISPLAY variable have been passed, would seem to be the obvious thing to try.

Unfortunately that doesn't work for distributions that use the first case, though it does for distributions that use the second.

The issue turns out to be because in some Linux distributions D-bus uses [abstract sockets](#), which are *network-namespace specific* and are not bound to any filesystem namespace handle, so have no obvious way to be passed to a container.

The thinking behind using abstract sockets for the D-bus session sockets was most likely as a simple cleanup mechanism (abstract sockets only exist in memory), though in a modern desktop Linux environment the natural place to put the session socket is perhaps in `XDG_RUNTIME_DIR`, which is usually something like `/run/user/1000`, as this is an ephemeral location tied to the user login.

Possible solutions for dealing with abstract sockets include:

- Setting `--net=host` in the `docker run` command, which will use the host's network namespace and thus make the abstract socket visible to the container (at the expense of some sandboxing).
- Running a proxy (which has to be a host application) that proxies between a filesystem namespace socket, which can then be mounted, and the abstract socket.
- Modifying `/etc/dbus-1/session.conf` (or `/usr/share/dbus-1/session.conf`) either to add an additional listener or to modify the existing listener to use the *runtime* path instead of the *tmpdir* path.

Enabling Abstract Sockets with `-net=host`

The first option is the simplest and the launch script should look like [gnome-calculatorV2.sh](#):

```
if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)
docker run --rm \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
    gnome-calculator
```

this gets rid of all the error messages for Debian 9, CentOS 7 and Linux Mint 17, but for Ubuntu 17 and Linux Mint 18 we are now bombarded by lots of [AppArmor](#) related errors.

Resolving AppArmor Errors

On systems with AppArmor enabled by default, such as Ubuntu 17 and Linux Mint 18, we see a lot of errors similar to the following if we attempt to use D-bus from a Docker container:

```
(gnome-calculator:1): dconf-WARNING **: failed to commit changes to dconf: GDBus.Error:org.freedesktop.DBus.Error.AccessDenied : An AppArmor policy prevents this sender from sending this message to this recipient; type="method_call", sender="(null)" (inactive) interface="org.freedesktop.DBus" member="Hello" error name="(unset)" requested_reply="0" destination="org.freedesktop.DBus" (bus)
```

...

It turns out that if AppArmor is enabled on the host system then *by default* Docker containers are protected by an *automatically generated* default profile named **docker-default**, as described in the Docker documentation

<https://docs.docker.com/engine/security/apparmor/>

Unfortunately the docker-default profile, although fairly liberal, denies access to D-bus, hence all the error messages, so we either have to disable AppArmor using:

```
--security-opt apparmor=unconfined
```

which is clearly not ideal, or else supply another profile which actually does enable D-bus access.

An AppArmor profile [docker-dbus](#), based on information about the docker-default profile from the [Docker documentation](#), has been provided to accompany these examples and is hosted in the [bin](#) directory of the docker-gui installation.

```
#include <tunables/global>

profile docker-dbus flags=(attach_disconnected, mediate_deleted)
{
    #include <abstractions/dbus-session>
    #include <abstractions/dbus-accessibility>

    network,
    capability,
    file,
    umount,

    deny @{PROC}/* w,    # deny write for all files directly in /proc (not in a subdir)
        # deny write to files not in /proc/<number>/** or /proc/sys/*
    *
    deny @{PROC}/{{[1-9],[^1-9][^0-9],[^1-9s][^0-9y][^0-9s],[^1-9]
    }[^0-9][^0-9][^0-9]*/** w,
    deny @{PROC}/sys/[^k]** w,    # deny /proc/sys except /proc/sys/k* (effectively /proc/sys/kernel)
    deny @{PROC}/sys/kernel/{?,??,[^s][^h][^m]**} w,    # deny everything except shm* in /proc/sys/kernel/
    deny @{PROC}/sysrq-trigger rwk1x,
    deny @{PROC}/kcore rwk1x,
    deny mount,
    deny /sys/[^f]/** wklx,
    deny /sys/f[^s]/** wklx,
    deny /sys/fs/[^c]/** wklx,
    deny /sys/fs/c[^g]/** wklx,
    deny /sys/fs/cg[^r]/** wklx,
    deny /sys/firmware/** rwk1x,
    deny /sys/kernel/security/** rwk1x,
}
```

To install the docker-dbus profile temporarily we can run:

```
sudo apparmor_parser -r -w /path/to/docker-gui/bin/docker-dbus
```

To remove the docker-dbus profile we can run:

```
sudo /etc/init.d/apparmor stop
sudo apparmor_parser -R /path/to/docker-gui/bin/docker-dbus
sudo /etc/init.d/apparmor start
```

For the [docker-dbus](#) AppArmor profile to be permanently installed it should be copied into /etc/apparmor.d/docker-dbus and the system rebooted.

To check if the profile has been installed correctly:

```
sudo apparmor_status | grep docker
    docker-dbus
    docker-default
```

The link <https://help.ubuntu.com/community/AppArmor> provides a useful introduction to AppArmor.

It is also worth pointing out that it seems to be quite dependent on the Linux distribution being run as to whether or not AppArmor is enabled by default. It seems to be on Ubuntu 17 and Linux Mint 18, but not for CentOS 7 or Debian 9, so it's possible you may not see some of these issues, the author somewhat stumbled into the AppArmor D-bus issues whilst testing the launch scripts on various distributions in VMs in an effort to maximise portability.

[gnome-calculatorV3.sh](#) adds the AppArmor updates, defaulting to **unconfined** if the docker-dbus profile is not installed:

```
if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro"
fi

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)
docker run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
    gnome-calculator
```

Unfortunately, despite all of our work to connect to dconf/D-bus our containerised gnome-calculator *still* doesn't persist changes to configuration across restarts. Well that's actually not *quite* true, if we were to open the desktop's **native** gnome-calculator we should see that it is set to the same mode as the mode we were in the last time we exited our containerised gnome-calculator.

Referring again to the [dconf overview](#) : dconf has a *partial* client/server architecture. It uses D-Bus, but the server is *only involved in writes* , reads are performed *by direct access* (via mmap) to the on-disk database. A final step is therefore to bind-mount \$HOME/.config/dconf/user, giving a launch script that looks like [gnome-calculatorV4.sh](#) :

```
if [[ $DBUS_SESSION_BUS_ADDRESS == **"abstract"** ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY nmerge -
```

```
# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un) /.config/dconf
docker run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
gnome-calculator
```

Enabling Abstract Sockets with a Proxy

For systems where D-bus listens on abstract sockets our examples so far have set `--net=host` to make the abstract socket visible to the container, which is undesirable.

An alternative approach is to run a proxy **on the host** that proxies between a filesystem namespace socket, which can then be mounted, and the abstract socket.

The simplest way to create such a proxy is to use the [socat](#) tool, which should be available in application repositories for most Linux distributions.

```
rm $XDG_RUNTIME_DIR/bus
socat UNIX-LISTEN:$XDG_RUNTIME_DIR/bus,fork ABSTRACT-CONNECT:$(
    echo $DBUS_SESSION_BUS_ADDRESS | sed -e 's/[^\=]*=\([^\,]+\),.*/\
\1/' )
```

In essence this command tells socat to listen on a Unix domain socket with the filesystem path \$XDG_RUNTIME_DIR/bus, which is likely to be something like /run/user/1000/bus, and relay the connection to the specified abstract socket, which is obtained via the RegEx parse of the contents of \$ DBUS_SESSION_BUS_ADDRESS.

The RegEx looks for any number of non = characters

([^=]*) followed by =, it then matches and saves as many non-commas as it can find (\([^\,]\+\)) , followed by a comma and the rest of the line (, . *) . This means that it will replace everything up to and including the first = and after the first comma with whatever non-comma characters it finds after the first = on the line.

The [gnome-calculator-using-dbus-proxy.sh](#) launch script illustrates the changes required to run our containerised GNOME calculator with the D-bus proxy:

```
if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    if test -S "$XDG_RUNTIME_DIR/bus"; then
        echo "Using D-bus proxy."
        DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bu
s:ro -e NO_AT_BRIDGE=1 -e DBUS_SESSION_BUS_ADDRESS=unix:path=$X
DG_RUNTIME_DIR/bus"
    else
        echo "D-bus proxy socket is not bound."
    fi
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1 -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS
_ADDRESS"
fi
```

```
if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
docker run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
    gnome-calculator
```

Modify D-bus to use Filesystem Sockets

Although using a proxy to map to abstract sockets works, and may actually be the best approach if we wish to add additional bespoke filters, it has several issues. The most important issue is that lifecycle management of the proxy is important and fairly complex, for example the proxy must obviously be running before applications can use it, and we need to ensure the socket handle is cleared up after use and prevent multiple proxies trying to bind to the same socket, etc. etc. These issues were somewhat glossed over in the previous chapter.

Where we don't wish to add any extra D-bus filtering a more robust option is, perhaps, to configure D-bus to use filesystem sockets rather than abstract sockets. This is *actually* the default in some distributions such as Debian 9 and Ubuntu 17. To enable it in other distributions we have to modify either /etc/dbus-1/session.conf or /usr/share/dbus-1/session.conf depending on the distribution.

In session.conf there is a line that looks like:

```
<listen>unix:tmpdir=/tmp</listen>
```

if this is changed to:

```
<listen>unix:runtime=yes;unix:tmpdir=/tmp</listen>
```

Then the D-bus session bus will listen on \$XDG_RUNTIME_DIR/bus as described in the [D-bus Specification](#) in the Server Address Format section.

4-2-2 gedit

During our exploration of [gnome-calculator](#) we discovered that even *seemingly simple* desktop applications often end up requiring far more integration hooks than one might imagine.

We shall now try to apply the knowledge that we've gained getting GNOME Calculator working to another GNOME application [gedit](#). Note that the approaches employed for solving the dconf/D-bus and AppArmor issues shall not be discussed again in detail in this chapter, so if you are interested in them you should read the [gnome-calculator](#) chapter first.

The [Dockerfile](#) is in the [gedit](#) directory of [docker-gui](#) under [4-local-applications/gnome-dbus-and-apparmor](#).

```
FROM debian:stretch-slim

# Install gedit
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    libcanberra-gtk3-module packagekit-gtk3-module \
    gedit gedit-common && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["gedit"]
```

To build the image simply run:

```
docker build -t gedit .
```

The launch script to invoke the `docker run` command should look like [geditVo.sh](#) below, which is more or less identical to [gnome-calculatorV4.sh](#) and should work well with no additional tweaking required.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == **"abstract"** ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

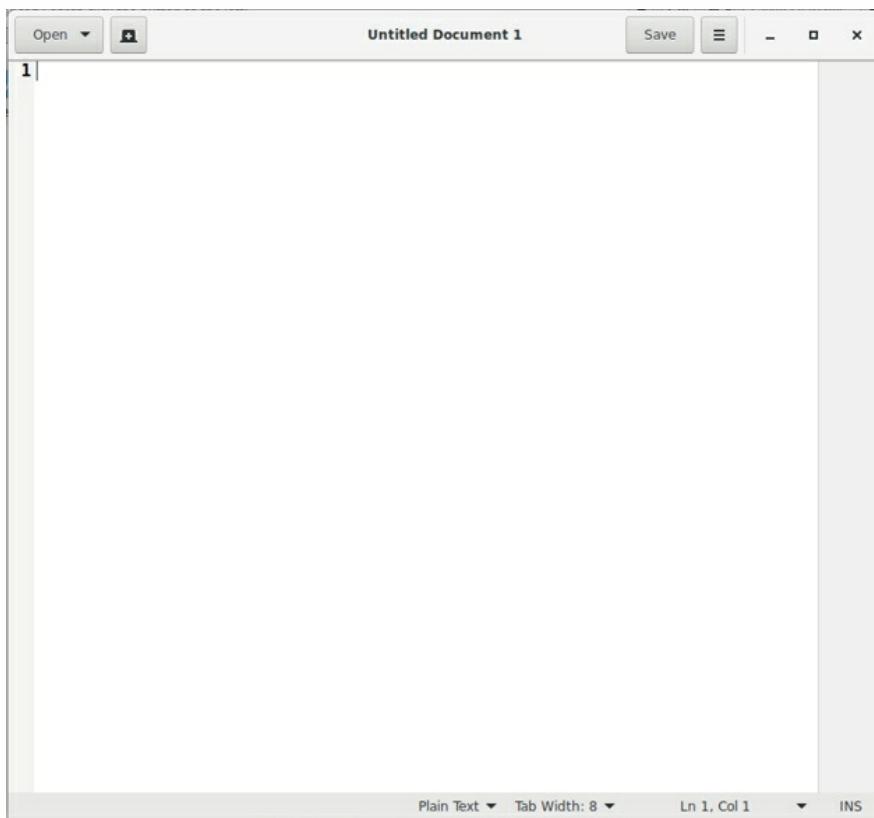
if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
```

```
$DOCKER_COMMAND run --rm \
$APPARMOR_FLAGS \
$DBUS_FLAGS \
-e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
-v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
gedit
```

It should display a window similar to the following.



Printing

So far our focus has been on getting the basic functionality working and we have skipped over printing from the application in order to avoid introducing too many new concepts. The ability to print is, however, a significant feature required by many applications, so now it is time to revisit gedit to add the finishing touches.

If we run geditVo.sh and attempt to print, because of the magic of D-bus integration we should actually see any printers attached to the host without needing to explicitly bind-mount the cups socket /var/run/cups/cups.sock, and printing should more or less work correctly albeit with some additional warning messages:

```
** (gedit:1): WARNING **: Couldn't connect to D-Bus system bus,  
Could not connect: No such file or directory  
  
** (gedit:1): WARNING **: failed to contact colord: Could not c  
onnect: No such file or directory
```

The first warning is informing us that the application cannot connect to the D-bus, system bus, this is because although we have already added D-bus support to the application that only enables us to connect to the D-bus *session bus*, in order to connect to the **system bus** we must bind-mount /var/run/dbus/system_bus_socket.

The second warning relates to colord, a system service that makes it easy to manage, install and generate colour profiles to accurately colour manage input and output devices

<https://www.freedesktop.org/software/colord/intro.html>.

The launch script [geditV1.sh](#) simply prepends the flags needed to bind-mount the system bus socket to the rest of the DBUS_FLAGS, which then makes colord on the host visible to the container.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

# Add flags for connecting to the D-bus system bus.
# This is necessary to connect to colord, a service
# that makes it easy to manage, install and generate colour
# profiles to accurately colour manage devices.
# https://www.freedesktop.org/software/colord/intro.html
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sys
tem_bus_socket:ro \"$DBUS_FLAGS

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi
```

```
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
gedit
```

4-2-3 evince

[Evince](#) is a document viewer for PDF, PostScript, DjVu, TIFF, XPS, and DVI formats designed for the GNOME desktop environment. It is the default document viewer for a large number of Linux distributions including Ubuntu, Fedora, and Linux Mint.

Following all of our work on [gnome-calculator](#) and [gedit](#) we should now be in a good position to create a fully working version of evince first time.

The [Dockerfile](#) is in the [evince](#) directory of [docker-gui](#) under [4-local-applications/gnome-dbus-and-apparmor](#).

```
FROM debian:stretch-slim

# Install evince
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        evince evince-common gir1.2-evince-3.0 libgxps2 && \
        rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["evince"]
```

To build the image simply run:

```
docker build -t evince .
```

The [evince.sh](#) launch script is as follows:

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

# Add flags for connecting to the D-bus system bus.
# This is necessary to connect to colord, a service
# that makes it easy to manage, install and generate colour
# profiles to accurately colour manage devices.
# https://www.freedesktop.org/software/colord/intro.html
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sy
stem_bus_socket:ro "$DBUS_FLAGS

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

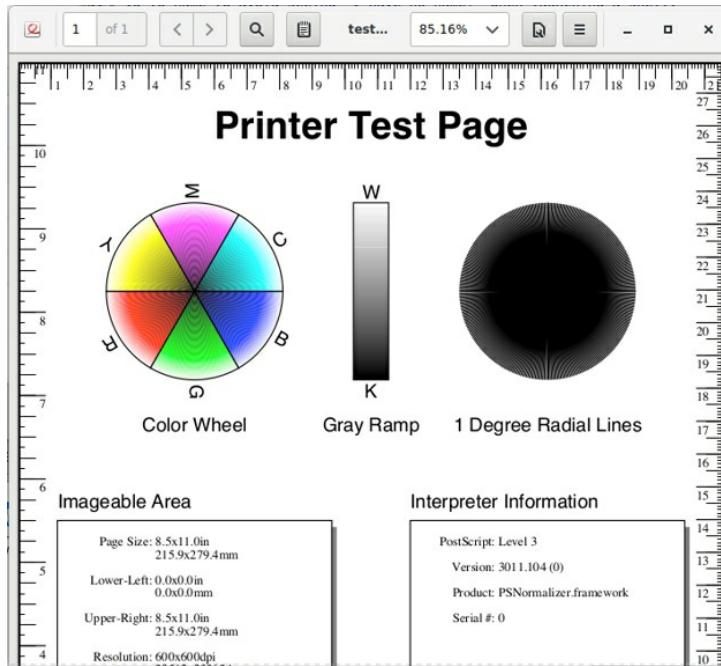
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY nmerge -
```

```

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un) /.config/dconf
$DOCKER_COMMAND run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
    -v $PWD/testfile.pdf:$HOME/testfile.pdf \
    evince $HOME/testfile.pdf

```

It should display a window similar to the following.



4-3 3D Accelerated Applications

4-3-1 Prerequisites

It should *hopefully* be obvious that for a section on 3D acceleration to make much sense it is necessary to have 3D acceleration enabled on the host that is running Docker.

Desktop or laptop

If you are using a desktop or laptop machine then in all likelihood it will come equipped with some sort of GPU and you will be good to go, to check the hardware run:

```
lspci | grep VGA
01:00.0 VGA compatible controller: NVIDIA Corporation GM200 [Ge
Force GTX TITAN X] (rev a1)
```

The displayed result will obviously vary depending on the hardware actually installed. In order to check the actual 3D capabilities the mesa-utils package needs to be installed. The first check is for direct rendering to be supported:

```
glxinfo | grep direct
direct rendering: Yes
    GL_AMD_multi_draw_indirect, GL_AMD_seamless_cubemap_per_texture,
    GL_ARB_direct_state_access, GL_ARB_draw_buffers,
    GL_ARB_draw_indirect, GL_ARB_draw_instanced, GL_ARB_enhanced_layouts,
    GL_ARB_indirect_parameters, GL_ARB_instanced_arrays,
    GL_ARB_map_buffer_range, GL_ARB_multi_bind, GL_ARB_multi_draw_indirect,
    ...
```

The next check is for the OpenGL version, vendor and renderer information, which should look something like:

```
glxinfo | egrep "OpenGL version*|OpenGL vendor|OpenGL renderer*"  
OpenGL vendor string: NVIDIA Corporation  
OpenGL renderer string: GeForce GTX TITAN X/PCIe/SSE2  
OpenGL version string: 4.5.0 NVIDIA 384.111
```

Clearly the displayed result will again depend on the actual hardware and driver versions installed.

VMware Virtual Machines

Support for 3D acceleration in VMware virtual machines seems to be fairly good. The author has tested the examples in this book using [VMware Workstation Player 14](#) with Linux Mint 18, Ubuntu 17, Debian 9, CentOS 7, and Fedora 26 virtual machines, though other VMware versions and distributions should work too.

When using VMware Workstation Player the 3D acceleration largely *just works* out of the box with no issues on fresh VM installations, provided that the *Accelerate 3D graphics* check box has been selected in the *Display* settings. Ubuntu, CentOS, and Fedora all seem to automatically install **open-vm-tools** and **open-vm-tools-desktop** (which seem to be recommended over *VMware Tools*), but Linux Mint and Debian VMs both need these to be manually installed via:

```
sudo apt-get install open-vm-tools open-vm-tools-desktop
```

In addition Ubuntu doesn't install mesa-utils by default, so it is necessary to install these via:

```
sudo apt-get install mesa-utils
```

in order to run:

```
glxinfo | egrep "OpenGL version*|OpenGL vendor|OpenGL renderer*"  
OpenGL vendor string: VMware, Inc.  
OpenGL renderer string: SVGA3D; build: RELEASE; LLVM;  
OpenGL version string: 3.0 Mesa 17.2.2
```

The different distributions return different Mesa versions and of particular note is that the default Mesa version for Linux Mint 18 is only 12.0.6, which is worth updating to a more recent version.

VirtualBox Virtual Machines

Compared to VMware the 3D support in [VirtualBox](#) seems to be a little bit *unloved*. Out of the box VirtualBox runs Mesa in software rendering mode so it *appears* to be working, but checking the OpenGL information tells a different story:

```
glxinfo | egrep "OpenGL version*|OpenGL vendor|OpenGL renderer*"  
OpenGL vendor string: VMware, Inc.  
OpenGL renderer string: Gallium 0.4 on llvmpipe (LLVM 3.8, 256  
bits)  
OpenGL version string: 3.0 Mesa 12.0.6
```

The **llvmpipe** part of the renderer string tells us that software rendering is being used.

Fortunately VirtualBox does actually have 3D acceleration support, though it requires *VirtualBox Guest Additions* to be installed, the procedure for this is distribution specific.

With Ubuntu first install mesa-utils:

```
sudo apt-get install mesa-utils
```

For Ubuntu 17 and Linux Mint 18 select *Devices*→*Insert Guest Additions CD image...* from the VirtualBox menu, which should autorun the install script, then follow the instructions in the VM to install the Guest Additions.

After installing Guest Additions, shutdown the VM then select *Settings*→*Display* in the Oracle VM VirtualBox Manager and tick Enable 3D Acceleration, finally restart the VM.

For Debian 9 the process is somewhat more involved as the default installation lacks the tools to build kernel modules. In the VM first run:

```
sudo apt-get update  
sudo apt-get upgrade  
sudo apt-get install build-essential module-assistant dkms  
sudo m-a prepare
```

Then select *Devices*→*Insert Guest Additions CD image...* from the VirtualBox menu. On Debian 9 the autorun fails, so in a console window run:

```
sudo sh /media/cdrom/VBoxLinuxAdditions.run
```

After installing Guest Additions, shutdown the VM then select *Settings*→*Display* in the Oracle VM VirtualBox Manager and tick Enable 3D Acceleration, finally restart the VM.

After this process has been completed, for Linux Mint 18, Ubuntu 17 and Debian 9 we should have 3D acceleration:

```
glxinfo | egrep "OpenGL version*|OpenGL vendor|OpenGL renderer*"  
OpenGL vendor string: Humper  
OpenGL renderer string: Chromium  
OpenGL version string: 2.1 Chromium 1.9
```

It is important to note however that the VirtualBox virtual graphics device is unfortunately limited to OpenGL 2.1.

Unfortunately the author has been unable to successfully achieve 3D acceleration for CentOS 7 and Fedora 26 VirtualBox VMs. For CentOS the Guest Additions compile and install, and the copy/paste integration works, but selecting *Enable 3D Acceleration* in the Oracle VM VirtualBox Manager results in a "*black screen of death*". For Fedora 26 it wasn't even possible to *compile* the Guest Additions.

There doesn't appear to be a good solution to these issues, so if you *have* to run CentOS or Fedora on VirtualBox you might be out of luck.

QEMU/KVM Virtual Machines

Apparently QEMU/KVM has support for 3D/OpenGL acceleration in the form of [VirGL](#) however the author hasn't yet tried this.

4-3-2 glxgears

In an *ideal world* we could use what we've learned building simple [x11-apps](#) to package a minimal OpenGL application like glxgears into this simple [Dockerfile](#) in the [glxgears](#) directory of [docker-gui](#) under [4-local-applications/3D-accelerated-applications](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    mesa-utils && \
    rm -rf /var/lib/apt/lists/*
ENV LIBGL_DEBUG verbose
ENTRYPOINT ["glxgears"]
```

We could then build it like this:

```
docker build -t glxgears -f DockerfileV0 .
```

Note that we've called it DockerfileV0 here as we don't expect this to be the final version.

The container could then be launched via the [glxgearsVo.sh](#) script, following the same pattern as the [xlogoV4.sh](#) script:

```
# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
```

```
docker run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
glxgears
```

Which results in a flood of errors similar to the following:

```
libGL: screen 0 does not appear to be DRI2 capable
libGL: OpenDriver: trying /usr/lib/x86_64-linux-gnu/dri/tls/swrast_dri.so
libGL: OpenDriver: trying /usr/lib/x86_64-linux-gnu/dri/swrast_dri.so
libGL: dlopen /usr/lib/x86_64-linux-gnu/dri/swrast_dri.so failed (/usr/lib/x86_64-linux-gnu/dri/swrast_dri.so: cannot open shared object file: No such file or directory)
libGL: OpenDriver: trying ${ORIGIN}/dri/tls/swrast_dri.so
libGL: OpenDriver: trying ${ORIGIN}/dri/swrast_dri.so
libGL: dlopen ${ORIGIN}/dri/swrast_dri.so failed (${ORIGIN}/dri/swrast_dri.so: cannot open shared object file: No such file or directory)
libGL: OpenDriver: trying /usr/lib/dri/tls/swrast_dri.so
libGL: OpenDriver: trying /usr/lib/dri/swrast_dri.so
libGL: dlopen /usr/lib/dri/swrast_dri.so failed (/usr/lib/dri/swrast_dri.so: cannot open shared object file: No such file or directory)
libGL error: unable to load driver: swrast_dri.so
libGL error: failed to load driver: swrast
X Error of failed request:  BadValue (integer parameter ...
Major opcode of failed request:  154 (GLX)
Minor opcode of failed request:  3 (X_GLXCreateContext)
Value in failed request:  0x0
Serial number of failed request:  35
Current serial number in output stream:  37
```

In order to resolve these errors it is necessary to give the container access to the host's GPU device and possibly also some device specific libraries. The procedure for doing this is unfortunately very much device specific, so we shall explore each device type in turn.

Nvidia GPU Acceleration

Nvidia GPUs are probably the fiddliest devices to get working with Docker, fortunately Nvidia have been working on a [Docker plugin](#) that makes it *relatively* straightforward to leverage the power of GPU acceleration.

Using nvidia-docker has been something of a moving target, the author had initial experience using version 1, however in October 2017 version 2 was released and version 1 became *deprecated*. For completeness this book shall cover both versions, but version 2 should be preferred.

Nvidia Docker Version 1

The [nvidia-docker V1](#) Quick Start instructions are pretty good and repeated below for convenience:

Ubuntu distributions

```
# Install nvidia-docker and nvidia-docker-plugin
wget -P /tmp https://github.com/NVIDIA/nvidia-docker/releases/download/v1.0.1/nvidia-docker_1.0.1-1_amd64.deb
sudo dpkg -i /tmp/nvidia-docker*.deb && rm /tmp/nvidia-docker*.deb

# Test nvidia-smi
nvidia-docker run --rm nvidia/cuda nvidia-smi
```

CentOS distributions

```
# Install nvidia-docker and nvidia-docker-plugin
wget -P /tmp https://github.com/NVIDIA/nvidia-docker/releases/download/v1.0.1/nvidia-docker-1.0.1-1.x86_64.rpm
sudo rpm -i /tmp/nvidia-docker*.rpm && rm /tmp/nvidia-docker*.rpm
sudo systemctl start nvidia-docker

# Test nvidia-smi
nvidia-docker run --rm nvidia/cuda nvidia-smi
```

Other distributions

```
# Install nvidia-docker and nvidia-docker-plugin
wget -P /tmp https://github.com/NVIDIA/nvidia-docker/releases/download/v1.0.1/nvidia-docker_1.0.1_amd64.tar.xz
sudo tar --strip-components=1 -C /usr/bin -xvf /tmp/nvidia-docker*.tar.xz && rm /tmp/nvidia-docker*.tar.xz

# Run nvidia-docker-plugin
sudo -b nohup nvidia-docker-plugin > /tmp/nvidia-docker.log

# Test nvidia-smi
nvidia-docker run --rm nvidia/cuda nvidia-smi
```

One issue encountered by the author was that the nvidia-docker-plugin requires nvidia-modprobe, which is available in some Linux repositories but in the author's case had to be built from [source](#) and installed manually.

Another issue was that on *upstart* based hosts the upstart configuration doesn't seem to be set correctly, so the nvidia-docker-plugin didn't start up automatically on reboot.

It is however possible to manually start nvidia-docker on upstart based systems via

```
sudo start nvidia-docker
```

With the nvidia-docker-plugin running *in theory* all that is then required with nvidia-docker V1 is to use `nvidia-docker run` instead of `docker run` when launching containers. One additional thing that the author discovered was that, for some reason, if a container needs to use `--net=host` then adding that flag will cause issues with OpenGL/WebGL. This can, however, be resolved by adding the flag `--device=/dev/nvidia-modeset` to the `nvidia-docker run` command (see <https://github.com/NVIDIA/nvidia-docker/issues/421>).

Unfortunately, in addition to the host-side tweaks described above it is necessary to make a few additions to application Dockerfiles in order to launch with nvidia-docker V1.

The first addition is to include:

```
LABEL com.nvidia.volumes.needed=nvidia_driver
```

This LABEL is used by `nvidia-docker run` to decide if the driver volume and the device files are required and things won't work correctly if it's not in place, unfortunately that important piece of information isn't at all obvious from the documentation as it's slightly [hidden away](#).

Nvidia accelerated containers also require the environment variable `LD_LIBRARY_PATH` to be correctly set:

```
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib  
64:${LD_LIBRARY_PATH}
```

This can either be set in the Dockerfile as above, or preferably (for more portable Dockerfiles) in the `nvidia-docker run` command via the `-e` or `--env` flags.

A [Dockerfile](#) with the additions required to support nvidia-docker version 1 is also available in the `glxgears` directory of [docker-gui](#) under [4-local-applications/3D-accelerated-applications](#).

```
FROM debian:stretch-slim  
  
# nvidia-docker hooks  
LABEL com.nvidia.volumes.needed=nvidia_driver  
  
# Install glxgears  
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \  
    # Add the packages used  
    apt-get install -y --no-install-recommends \  
    mesa-utils && \  
    rm -rf /var/lib/apt/lists/*  
  
ENV LIBGL_DEBUG verbose  
ENTRYPOINT ["glxgears"]
```

and may be built with:

```
docker build -t glxgears -f DockerfileV1 .
```

The container can then be launched via [glxgearsV1.sh](#), noting that this version of the glxgears launch script is nvidia-docker version 1 specific, so won't work with other GPU types or even nvidia-docker version 2.

```
if test -f "/usr/bin/nvidia-container-runtime"; then
    echo "This script is specifically intended to illustrate nv
idia-docker V1"
    echo "You appear to have nvidia-docker2 installed, try glxg
earsV2.sh."
    exit 1
fi

if test -c "/dev/nvidia-modeset"; then
    # Nvidia GPU
    DOCKER_COMMAND=nvidia-docker
    SRC=/usr/local/nvidia
    GPU_FLAGS="--device=/dev/nvidia-modeset "
    GPU_FLAGS+=" -e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
else
    echo "This version of the glxgears launch script is Nvidia
specific."
    exit 1
fi

# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY nmerge -
```

```
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
$GPU_FLAGS \
glxgears
```

Nvidia Docker Version 2

Around September 2017 Nvidia migrated towards a different approach for Docker integration. The new approach takes advantage of [custom container runtimes](#) that were added in Docker 1.12 and works by using [nvidia-container-runtime](#), which is a modified version of [runc](#) that adds a custom [pre-start hook](#) to all containers started by it.

The hook configures GPU access for the container by leveraging nvidia-container-cli from the other sub-project [libnvidia-container](#), based on several [OCI spec environment variables](#).

Nvidia recommend that users transition to nvidia-docker 2, with version 1 having been deprecated. Nvidia claim that 2 is already more stable than 1, although there is also a warning that libnvidia-container is currently an alpha release.

Another thing to be aware of is that *technically* neither nvidia-docker 2 nor nvidia-docker 1 have [support for OpenGL](#) and it isn't a priority for Nvidia, as their focus is [CUDA](#). However, *not supported* and *not possible* are entirely different things as we have shown for V1 and shall shortly show for V2.

The [nvidia-docker V2](#) Quick Start instructions are pretty good and repeated below for convenience:

Ubuntu 16.04 (Xenial) x86_64 based distributions

```
# Cleanly remove nvidia-docker 1.0 and existing containers.  
docker volume ls -q -f driver=nvidia-docker | xargs -r -I{} -n1  
docker ps -q -a -f volume={} | xargs -r docker rm -f  
sudo apt-get purge -y nvidia-docker  
# Add the package repositories  
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | \  
    sudo apt-key add -  
curl -s -L https://nvidia.github.io/nvidia-docker/ubuntu16.04/a  
md64/nvidia-docker.list | \  
    sudo tee /etc/apt/sources.list.d/nvidia-docker.list  
sudo apt-get update  
# Install nvidia-docker2 & reload Docker daemon config.  
sudo apt-get install -y nvidia-docker2  
sudo pkill -SIGHUP dockerd  
  
# Test nvidia-smi with the latest official CUDA image  
docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

Although these instructions specifically talk about Ubuntu 16.04 (Xenial) the author has successfully installed nvidia-docker2 on an Ubuntu 14.04 (Trusty) based system, though initially ran into an issue due to running a slightly older version of Docker, which was easily remedied by updating to the latest Docker version. According to a reply given by an Nvidia developer to issue #551 [can't install nvidia-docker 2.0 on ubuntu 14.04](#) on the nvidia-docker GitHub site “Using the 16.04 repository should be fine”, though it is an unsupported configuration.

Debian 9 (Stretch) x86_64 based distributions

```
# Cleanly remove nvidia-docker 1.0 and existing containers.  
docker volume ls -q -f driver=nvidia-docker | xargs -r -I{} -n1  
docker ps -q -a -f volume={} | xargs -r docker rm -f  
sudo apt-get purge -y nvidia-docker  
# Add the package repositories  
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | \  
    sudo apt-key add -  
curl -s -L https://nvidia.github.io/nvidia-docker/debian9/amd64  
/nvidia-docker.list | \  
    sudo tee /etc/apt/sources.list.d/nvidia-docker.list  
sudo apt-get update  
# Install nvidia-docker2 & reload Docker daemon config.  
sudo apt-get install -y nvidia-docker2  
sudo pkill -SIGHUP dockerd  
  
# Test nvidia-smi with the latest official CUDA image  
docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

CentOS/RHEL 7 x86_64 based distributions

```
# Cleanly remove nvidia-docker 1.0 and existing containers.  
docker volume ls -q -f driver=nvidia-docker | xargs -r -I{} -n1  
docker ps -q -a -f volume={} | xargs -r docker rm -f  
sudo yum remove nvidia-docker  
# Add the package repositories  
curl -s -L https://nvidia.github.io/nvidia-docker/centos7/x86_6  
4/nvidia-docker.repo | \  
    sudo tee /etc/yum.repos.d/nvidia-docker.repo  
# Install nvidia-docker2 & reload Docker daemon config.  
sudo yum install -y nvidia-docker2  
sudo pkill -SIGHUP dockerd  
  
# Test nvidia-smi with the latest official CUDA image  
docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

The nvidia-docker2 [installation](#) pages largely repeat the information from the Quick Start, however one important additional piece of information is included as follows, which relates to the use of nvidia-docker2 with older versions of Docker:

You must pin the versions of both `nvidia-docker2` and `nvidia-container-runtime` when installing, for instance:

```
sudo apt-get install -y nvidia-docker2=2.0.1+docker1.12.6-1 nvidia-container-runtime=1.1.0+docker1.12.6-1
```

To list the available versions, use:

```
apt-cache madison nvidia-docker2 nvidia-container-runtime
```

or

```
yum search --showduplicates nvidia-docker2 nvidia-container-runtime
```

The author of this book originally got an error of the form:

```
Depends: docker [...] but it is not installable or nothing provides docker [...]
```

which turned out to be due to this problem, however in the author's case it was resolved by simply upgrading to the latest Docker version as there was no compelling reason to remain at a specific older version.

The [Dockerfile](#) that we previously used for nvidia-docker version 1 can still be used with nvidia-docker2, though the LABEL is no longer necessary.

```
FROM debian:stretch-slim

# nvidia-docker hooks
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install glxgears
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    mesa-utils && \
    rm -rf /var/lib/apt/lists/*

ENV LIBGL_DEBUG verbose
ENTRYPOINT ["glxgears"]
```

as before it may be built with:

```
docker build -t glxgears -f DockerfileV1 .
```

According to the [usage](#) instructions, using nvidia-docker2 *should* be as simple as selecting `--runtime=nvidia` when using `docker run`, that is to say as per Nvidia's example:

```
docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

To be fair, that is indeed the case when using Nvidia's [official CUDA images](#), which contain the required [OCI spec environment variables](#) as ENV items in the Dockerfiles.

In order to use more general purpose images not obtained from Nvidia however, it is necessary to pass these environment variables to the `docker run` command via the `-e` or `--env` flags. The most important of these environment variables are:

NVIDIA_VISIBLE_DEVICES

This controls which GPUs will be made accessible inside the container. The possible values are:

- 0,1,2, GPU-fef8089b ...: a comma-separated list of GPU UUIDs or indexes,
- all: all GPUs will be accessible.
- none: capabilities enabled but no GPU will be accessible.
- void or empty or unset: nvidia-container-runtime will have the same behaviour as runc.

NVIDIA_DRIVER_CAPABILITIES

This controls which driver libraries and binaries will be mounted inside the container. The possible values are:

- compute,video, graphics,utility ...: a comma-separated list of driver features that the container needs. The supported driver capabilities are:
 - compute: required for CUDA and OpenCL.
 - compat32: required for 32-bit applications.
 - graphics: required for OpenGL and Vulkan.
 - utility: required for using nvidia-smi and NVML.
 - video: required for using the Video Codec SDK.
- all: enable all available driver capabilities.
- empty or unset: use default driver capability: utility.

Unfortunately, simply setting these environment variables and creating a launch script of the form:

```
docker run --rm --runtime=nvidia -e NVIDIA_VISIBLE_DEVICES=all  
-e NVIDIA_DRIVER_CAPABILITIES=graphics <X11-socket/DISPLAY/auth  
> <image> <command>
```

still isn't enough to get nvidia-docker2 displaying glxgears.

After a fair amount of trial and error the author discovered that what is required is for the container to have visibility of a number of shared libraries from the host that are Nvidia versions of the OpenGL libraries `libGL.so.1`, `libGLX.so.0` and `libGLdispatch.so.0`, which are found in the directory `/usr/lib/nvidia-<driver version>` on Ubuntu derived hosts. Note this path may be different on other distributions, in which case the SRC variable in subsequent scripts would need to be modified to reflect this.

The author also discovered that, similar to the case with nvidia-docker V1, if a container needs to make use of `--net=host` then adding that flag will cause issues with OpenGL/WebGL, this can be resolved by adding the flag `--device=/dev/nvidia-modeset` to the `docker run` command.

With all of these additions incorporated the launch script for nvidia-docker2 looks like [glxgearsV2.sh](#) below, noting that this version of the glxgears launch script is nvidia-docker version 2 specific, so won't work with other GPU types or even nvidia-docker version 1.

```

DOCKER_COMMAND=docker
DST=/usr/lib/x86_64-linux-gnu
if test -c "/dev/nvidia-modeset"; then
    # Nvidia GPU
    if test -f "/usr/bin/nvidia-container-runtime"; then
        # Nvidia Docker Version 2
        # Attempt to find the actual Nvidia library path.
        # It should be something like
        # /usr/lib/nvidia-<driver version>
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep
/lib/)
        GPU_FLAGS="--runtime=nvidia "
        GPU_FLAGS+="--device=/dev/nvidia-modeset "
        GPU_FLAGS+="--e NVIDIA_VISIBLE_DEVICES=all "
        GPU_FLAGS+="--e NVIDIA_DRIVER_CAPABILITIES=graphics "
        GPU_FLAGS+="--v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
        GPU_FLAGS+="--v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
        GPU_FLAGS+="--v $SRC/libGDispatch.so.0:$DST/libGDispatch.s
0.0:ro "
    else
        # Nvidia Docker Version 1
        echo "This script is specifically intended to illustrate nv
idia-docker V2"
        echo "You appear to have nvidia-docker V1 installed, try gl
xgearsV1.sh."
        exit 1
    fi
else
    echo "This version of the glxgears launch script is Nvidia sp
ecific."
    exit 1
fi

# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

```

```

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
$GPU_FLAGS \
glxgears

```

Nvidia Docker Version 2 - alternative approach

The nvidia-docker2 approach described previously, and in the [glxgearsV2.sh](#) launch script, work by ensuring that the relevant Nvidia vendor specific OpenGL shared libraries are bind-mounted into the container from the host, which the author believes is also the approach that is used, albeit a little more transparently, by nvidia-docker version 1.

A big advantage of that approach is that the Dockerfile can be minimal yet still support a range of GPU vendors, as the key vendor specific components are mounted from the host.

The main disadvantage is that by bind-mounting the vendor specific libraries we have reduced container sandboxing, though a number of drivers and libraries are transparently mounted just by enabling the various driver capabilities, so arguably it just *looks* worse because we have *explicitly* mounted the OpenGL libraries.

An alternative approach to bind-mounting the OpenGL libraries from the host is to use an appropriate set of libraries inside the container, in this case we shall create a container that builds Nvidia's [libglvnd](#) vendor-neutral dispatch layer for arbitrating OpenGL API calls between multiple vendors.

The [Dockerfile](#) first installs the libglvnd dependencies and a build environment, it then git clones the libglvnd source code, builds and installs it. The rest of the Dockerfile is as before.

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    git \
    ca-certificates \
    make \
    automake \
    autoconf \
    libtool \
    pkg-config \
    python \
    libxext-dev \
    libx11-dev \
    x11proto-gl-dev && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /opt/libglvnd
RUN git clone --branch=v1.0.0 https://github.com/NVIDIA/libglvnd.git . && \
    ./autogen.sh && \
    ./configure --prefix=/usr/local --libdir=/usr/local/lib/x86_64-linux-gnu && \
    make -j$(nproc) install-strip && \
    find /usr/local/lib/x86_64-linux-gnu -type f -name 'lib*.la' -delete
```

```

# Only the first line below seems to be needed with an
# Ubuntu base image, but all of it is needed when using
# debian:stretch-slim
RUN echo '/usr/local/lib/x86_64-linux-gnu' >> /etc/ld.so.conf.d
/glvnd.conf && \
    ldconfig && \
    echo '/usr/local/$LIB/libGL.so.1' >> /etc/ld.so.preload && \
    echo '/usr/local/$LIB/libEGL.so.1' >> /etc/ld.so.preload

# nvidia-container-runtime
ENV NVIDIA_VISIBLE_DEVICES all
ENV NVIDIA_DRIVER_CAPABILITIES graphics
#-----

# Install glxgears
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    mesa-utils && \
    rm -rf /var/lib/apt/lists/*

ENV LIBGL_DEBUG verbose
ENTRYPOINT ["glxgears"]

```

The image may be built with:

```
docker build -t glxgears-libglvnd -f Dockerfile-libglvnd .
```

Note that this image is sub-optimal as it could, but doesn't, clean up the build environment and remove the source code and dependencies after building libglvnd. A better approach might be to separate the build into a separate container whose job is to build libglvnd and export the built libraries into a libglvnd.tar.gz, which could then be ADDed into an image, which should result in much smaller images.

The launch script for the libglvnd approach looks like [glxgears-libglvnd.sh](#) below:

```
DOCKER_COMMAND=docker
# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

$DOCKER_COMMAND run --runtime=nvidia --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
glxgears-libglvnd
```

This launch script is somewhat simpler than the previous scripts for nvidia-docker, with the only GPU specific addition being

`--runtime=nvidia`. It is important to be aware however that we have really just traded complexity of the Dockerfile/image with complexity of the launch script, so it's not clear that there is a major gain to be had following this approach other than for those who wish to ensure the maximum possible levels of container isolation.

VirtualBox Guest Additions GPU Acceleration

The approach for enabling GPU acceleration in containers that are running on VirtualBox guest machines using VirtualBox Guest Additions is *somewhat* similar to that needed for Nvidia, though the detail is rather different.

As with nvidia-docker2 it is necessary to ensure that the Guest Additions OpenGL shared libraries are used *in preference* to the ones shipped with the container's OpenGL and that a number of VirtualBox shared libraries are available for the VirtualBox OpenGL implementation to use.

The files from the host that seem to be required are:

```
/var/lib/VBoxGuestAdditions/lib/libGL.so.1  
/usr/lib/x86_64-linux-gnu/VBoxEGL.so  
/usr/lib/x86_64-linux-gnu/VBoxOGL.so  
/usr/lib/x86_64-linux-gnu/VBoxOGLerrorspu.so  
/usr/lib/x86_64-linux-gnu/VBoxOGLpassthroughspu.so  
/usr/lib/x86_64-linux-gnu/VBoxOGLarrayspu.so  
/usr/lib/x86_64-linux-gnu/VBoxOGLfeedbackspu.so  
/usr/lib/x86_64-linux-gnu/VBoxOGLcrutil.so  
/usr/lib/x86_64-linux-gnu/VBoxOGLpackspu.so  
/usr/lib/x86_64-linux-gnu/libXcomposite.so.1
```

These shared libraries may be bind-mounted in the container, noting that `libGL.so.1` should be mounted in the container at `/usr/lib/x86_64-linux-gnu` rather than `/var/lib/VBoxGuestAdditions/lib`.

It is also necessary to ensure that `/dev/vboxuser` is visible to the container via `--device=/dev/vboxuser` .

Open Source Mesa Drivers GPU Acceleration

The Mesa Open Source drivers are the preferred 3D drivers for AMD, Intel, and VMware GPUs.

Compared to the effort required to use Nvidia or VirtualBox GPUs with Docker, using the Open Source Mesa Drivers from a container is very straightforward. The easiest approach is to include the `libgl1-mesa-glx` and `libgl1-mesa-dri` packages in the Dockerfile, giving our final `glxgears` [Dockerfile](#) as follows:

```
FROM debian:stretch-slim

# nvidia-docker hooks
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install glxgears
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        mesa-utils libgl1-mesa-glx libgl1-mesa-dri && \
        rm -rf /var/lib/apt/lists/*

ENV LIBGL_DEBUG verbose
ENTRYPOINT ["glxgears"]
```

which may be built with:

```
docker build -t glxgears .
```

As well as including the `libgl1-mesa-glx` and `libgl1-mesa-dri` packages in the Dockerfile it is also necessary to ensure that the `/dev/dri` device is visible to the container via `--device=/dev/dri`.

It is not *all* plain sailing with Mesa however.

If the container is run with root privileges, then adding the libgl1-mesa libraries to the Dockerfile and passing `--device=/dev/dri` to the `docker run` command as described previously is indeed all that is required. If we are running as an **unprivileged** user however, by using;

```
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
```

in the `docker run` command, then it is likely that it will still *appear* to work, but it would in fact be using the **software** renderer not the GPU.

The issue appears to be that users of the **drm** device need to be members of the *video* group, otherwise a “permission denied” error occurs similar to the following:

```
libGL error: failed to open drm device: Permission denied
libGL error: failed to load driver: vmwgfx
libGL: OpenDriver: trying /usr/lib/x86_64-linux-gnu/dri/tls/swr
ast_dri.so
libGL: OpenDriver: trying /usr/lib/x86_64-linux-gnu/dri/swrast_
dri.so
```

The *obvious* answer is to join the video group by adding `--group-add video` to the `docker run` command. Unfortunately this is unreliable and it seems necessary to pass the group ID of the video group rather than its name via:

```
--group-add $(cut -d: -f3 < <(getent group video))
```

The launch script [glxgearsV3.sh](#) below brings together all that we have learned in this chapter into a single script:

```
DOCKER_COMMAND=docker
DST=/usr/lib/x86_64-linux-gnu
if test -c "/dev/nvidia-modeset"; then
    GPU_FLAGS="--device=/dev/nvidia-modeset "
    if test -f "/usr/bin/nvidia-container-runtime"; then
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep
/lib/)
        GPU_FLAGS+="--runtime=nvidia "
        GPU_FLAGS+=" -e NVIDIA_VISIBLE_DEVICES=all "
        GPU_FLAGS+=" -e NVIDIA_DRIVER_CAPABILITIES=graphics "
        GPU_FLAGS+=" -v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
        GPU_FLAGS+=" -v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
        GPU_FLAGS+=" -v $SRC/libGLdispatch.so.0:$DST/libGLdispatch.s
0.0:ro "
    else
        DOCKER_COMMAND=nvidia-docker
        SRC=/usr/local/nvidia
        GPU_FLAGS+=" -e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
    fi
else
    if test -d "/var/lib/VBoxGuestAdditions"; then
        GPU_FLAGS="--device=/dev/vboxuser "
        GPU_FLAGS+=" -v /var/lib/VBoxGuestAdditions/lib/libGL.so.1:$
DST/libGL.so.1 "
        for f in $DST/VBox*.so $DST/libXcomposite.so.1
        do
            GPU_FLAGS+=" -v $f:$f "
        done
    else
        GPU_FLAGS="--device=/dev/dri "
        GPU_FLAGS+=" --group-add $(cut -d: -f3 < <(getent group vide
o)) "
    fi
fi
```

```
# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
$GPU_FLAGS \
glxgears
```



4-3-3 glmark

Now that we have explored [glxgears](#) in some detail we should be in a suitable position to apply what we have learned to other OpenGL applications to determine if the solutions that we have arrived at for each GPU family are general ones, or whether we have more work left to do.

The first application to try is glmark, which as its name would suggest, is an OpenGL benchmarking application that should hopefully provide a more representative view of our 3D graphics acceleration than glxgears.

The [Dockerfile](#) is in the [glmark](#) directory of the [docker-gui](#) repository under [4-local-applications/3D-accelerated-applications](#). Note that in this case we are using an ubuntu base image rather than debian:stretch-slim because glmark2 is not available in the Debian repository.

```
# glmark2 binaries not available in Debian repository
FROM ubuntu:16.04

# nvidia-docker hooks
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install glmark2
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        glmark2 glmark2-data libgl1-mesa-glx libgl1-mesa-dri && \
        rm -rf /var/lib/apt/lists/*

ENV LIBGL_DEBUG verbose
ENTRYPOINT ["glmark2"]
```

To build the image run:

```
docker build -t glmark .
```

The original intention in this chapter was to modify our final glxgears launch script [glxgearsV3.sh](#) and simply replace the image referenced in the `docker run` command from glxgears to glmark. This simple modification *should* have worked, and indeed it does work for mesa, but with Nvidia GPUs it unfortunately results in a *segmentation fault*.

Needless to say, given that the nvidia-docker plugin is primarily intended for compute acceleration and doesn't *officially* support OpenGL, the author initially assumed that something was missing from the libraries being bind-mounted from the host. Eventually however, an Ubuntu [bug report](#) was discovered describing a glmark2 segmentation fault with Nvidia drivers, pointing to a *packaging issue*, as there is no error if glmark is compiled from source.

The issue turns out to be due to a link error whereby `libpthread.so.0` is required, but is not linked. Aside from building glmark2 from source, a simple solution is to `LD_PRELOAD` the library:

```
-e LD_PRELOAD=/lib/x86_64-linux-gnu/libpthread.so.0
```

The complete [glmark.sh](#) launch script therefore mostly comprises what we have learned from glxgears, but with the addition of the `LD_PRELOAD` and the added knowledge that sometimes simple things can turn out complicated.

```

DOCKER_COMMAND=docker
DST=/usr/lib/x86_64-linux-gnu
if test -c "/dev/nvidia-modeset"; then
    # Nvidia GPU
    GPU_FLAGS="--device=/dev/nvidia-modeset "
    # There is a known problem with the glmark2 Ubuntu
    # packaging and Nvidia drivers
    # https://bugs.launchpad.net/ubuntu/+source/glmark2/+bug/1475
902.
    # In precis libpthread is needed, but is not linked so a
    # segfault occurs.
    # LD_PRELOAD of libpthread.so.0 fixes this.
    GPU_FLAGS+=" -e LD_PRELOAD=/lib/x86_64-linux-gnu/libpthread.so
.0 "
    if test -f "/usr/bin/nvidia-container-runtime"; then
        # Nvidia Docker Version 2
        # https://github.com/NVIDIA/nvidia-container-runtime.

        # Attempt to find the actual Nvidia library path.
        # It should be something like
        # /usr/lib/nvidia-<driver version>
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep
/lib/)

        GPU_FLAGS+=" --runtime=nvidia "
        GPU_FLAGS+=" -e NVIDIA_VISIBLE_DEVICES=all "
        GPU_FLAGS+=" -e NVIDIA_DRIVER_CAPABILITIES=graphics "
        GPU_FLAGS+=" -v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
        GPU_FLAGS+=" -v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
        GPU_FLAGS+=" -v $SRC/libGLdispatch.so.0:$DST/libGLdispatch.s
0.0:ro "
    else
        # Nvidia Docker Version 1
        DOCKER_COMMAND=nvidia-docker
        SRC=/usr/local/nvidia
        GPU_FLAGS="-e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
    fi

```

```

else # Non-Nvidia GPU path
  if test -d "/var/lib/VBoxGuestAdditions"; then
    # VirtualBox GPU
    GPU_FLAGS="--device=/dev/vboxuser "
    GPU_FLAGS+=" -v /var/lib/VBoxGuestAdditions/lib/libGL.so.1:$
DST/libGL.so.1 "
    for f in $DST/VBox*.so $DST/libXcomposite.so.1
    do
      GPU_FLAGS+="-v $f:$f "
    done
  else # Open Source Mesa GPU.
    GPU_FLAGS="--device=/dev/dri "
    GPU_FLAGS+=" --group-add $(cut -d: -f3 < <(getent group vide
o)) "
  fi
fi

# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
  DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY nmerge -

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
$GPU_FLAGS \
glmark

```



Although glmark runs well on Nvidia and mesa GPUs, including VMware, and despite glmark being described as an OpenGL 2.0 and ES 2.0 benchmark, unfortunately the author was unable to get glmark2 to run directly in a VirtualBox VM, let alone in a container. The errors were as follows:

```
OpenGL Warning: glXChooseVisual: ignoring attribute 0x22
OpenGL Warning: glXChooseFBConfig returning NULL, due to attrib
=0x2, next=0x1
Error: glXChooseFBConfig() failed
Error: Error: Couldn't get GL visual config!
Error: main: Could not initialize canvas
```

Sadly, this seems to be a common VirtualBox problem!

4-3-4 kcalc

The original intention was to have kcalc in the [Simple X11 Applications](#) section as a KDE counterpoint to [gnome-calculator](#). It is, in fact, a relatively simple application without the D-Bus related complexities of gnome-calculator, but it comes complete with a slight surprise.

Following a familiar pattern, the [Dockerfile](#) is in the [kcalc](#) directory of [docker-gui](#) under [4-local-applications/3D-accelerated-applications](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install kcalc
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    kcalc libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["kcalc"]
```

and it may be built in the familiar way:

```
docker build -t kcalc .
```

Our first attempt at a launch script is [kcalcVo.sh](#), which is very similar to [gnome-calculatorV1.sh](#), the gnome-calculator attempt before the D-bus and AppArmor issues were resolved.

```

DOCKER_COMMAND=docker
# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
    kcalc

```

When this is run, sure enough kcalc appears and seems to work, but the error below appears in the console:

```

libGL error: No matching fbConfigs or visuals found
libGL error: failed to load driver: swrast

```

This error clearly points to the fact that kcalc is trying to use OpenGL, which is surprising for a simple calculator application. The most likely explanation is that OpenGL forms part of the underlying KDE/Qt GUI toolkit rather than the application itself.

The [kcalcV1.sh](#) launch script adds the GPU flags identified previously, which resolves the error messages:

```
DOCKER_COMMAND=docker
DST=/usr/lib/x86_64-linux-gnu
if test -c "/dev/nvidia-modeset"; then # Nvidia GPU
    GPU_FLAGS="--device=/dev/nvidia-modeset "
    if test -f "/usr/bin/nvidia-container-runtime"; then
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep
/lib/)
        GPU_FLAGS+="--runtime=nvidia "
        GPU_FLAGS+=" -e NVIDIA_VISIBLE_DEVICES=all "
        GPU_FLAGS+=" -e NVIDIA_DRIVER_CAPABILITIES=graphics "
        GPU_FLAGS+=" -v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
        GPU_FLAGS+=" -v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
        GPU_FLAGS+=" -v $SRC/libGLdispatch.so.0:$DST/libGLdispatch.s
0.0:ro "
    else # Nvidia Docker Version 1
        DOCKER_COMMAND=nvidia-docker
        SRC=/usr/local/nvidia
        GPU_FLAGS+=" -e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
    fi
else # Non-Nvidia GPU path
    if test -d "/var/lib/VBoxGuestAdditions"; then
        GPU_FLAGS="--device=/dev/vboxuser "
        GPU_FLAGS+=" -v /var/lib/VBoxGuestAdditions/lib/libGL.so.1:$
DST/libGL.so.1 "
        for f in $DST/VBox*.so $DST/libXcomposite.so.1
        do
            GPU_FLAGS+=" -v $f:$f "
        done
    else # Open Source Mesa GPU.
        GPU_FLAGS="--device=/dev/dri "
        GPU_FLAGS+=" --group-add $(cut -d: -f3 < <(getent group vide
o)) "
    fi
fi
```

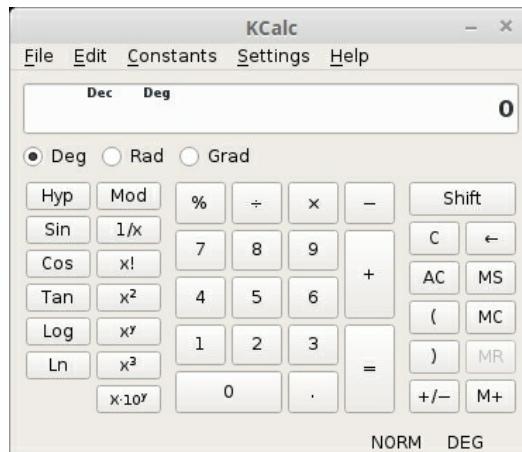
```

# If user isn't in docker group prefix docker with sudo
if ! $(id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
$GPU_FLAGS \
kcalc

```



4-3-5 gtk3-examples

Another example that is perhaps not the most obvious OpenGL application is the `gtk-3-examples` package. It does, however, actually incorporate an *OpenGL Area* widget and requires the D-bus and AppArmor support identified from the previous GNOME/GTK 3 examples, so represents a relatively simple application but with some complex dependencies.

The [Dockerfile](#) is in the `gtk3-examples` directory of [docker-gui](#) under [4-local-applications/3D-accelerated-applications](#). Of particular note is `libgl1-mesa-glx`, as it was whilst working on this particular example that the author realised that it was necessary to *explicitly* include this library in addition to `libgl1-mesa-dri`.

```
FROM debian:stretch-slim

# nvidia-docker hooks
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install gtk-3-examples
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    gtk-3-examples libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/*

CMD ["gtk3-demo"]
```

The image may be built with:

```
docker build -t gtk-3-examples .
```

The [gtk3-examples.sh](#) launch script requires GPU, D-bus, and AppArmor flags as follows:

```
if [ $# != 0 ] && ([ $1 == "-h" ] || [ $1 == "--help" ])
then
    echo "Usage: gtk3-examples [COMMAND]"
    echo "Example files and demonstration programs for GTK+3, including:"
    echo " - gtk3-demo"
    echo " - gtk3-demo-application"
    echo " - gtk3-icon-browser"
    echo " - gtk3-widget-factory"
    exit 0
fi

# Attempt to detect the type of GPU present on the host.
DOCKER_COMMAND=docker
DST=/usr/lib/x86_64-linux-gnu
if test -c "/dev/nvidia-modeset"; then
    # Nvidia GPU
    GPU_FLAGS="--device=/dev/nvidia-modeset "
    if test -f "/usr/bin/nvidia-container-runtime"; then
        # Nvidia Docker Version 2
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep
/lib/)
        GPU_FLAGS+="--runtimes=nvidia "
        GPU_FLAGS+=" -e NVIDIA_VISIBLE_DEVICES=all "
        GPU_FLAGS+=" -e NVIDIA_DRIVER_CAPABILITIES=graphics "
        GPU_FLAGS+=" -v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
        GPU_FLAGS+=" -v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
        GPU_FLAGS+=" -v $SRC/libGLdispatch.so.0:$DST/libGLdispatch.s
0.0:ro "
    else # Nvidia Docker Version 1
        DOCKER_COMMAND=nvidia-docker
        SRC=/usr/local/nvidia
        GPU_FLAGS="-e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
    fi
fi
```

```

else
    # Non-Nvidia GPU path
    if test -d "/var/lib/VBoxGuestAdditions"; then
        # VirtualBox GPU
        GPU_FLAGS="--device=/dev/vboxuser "
        GPU_FLAGS+=" -v /var/lib/VBoxGuestAdditions/lib/libGL.so.1:$
DST/libGL.so.1 "
        for f in $DST/VBox*.so $DST/libXcomposite.so.1
        do
            GPU_FLAGS+=" -v $f:$f "
        done
    else
        # Open Source Mesa GPU.
        GPU_FLAGS="--device=/dev/dri "
        GPU_FLAGS+=" --group-add $(cut -d: -f3 < <(getent group vide
o)) "
    fi
fi

# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro -
e NO_AT_BRIDGE=1"
fi

# Set appropriate apparmor flags.
if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

```

```

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
    $GPU_FLAGS \
    gtk-3-examples $@

```

Although the `gtk3-examples` application was relatively simple to compose an image for, it combines everything that we have learned so far about running GUI applications in Docker.

D-bus integration, AppArmor, and 3D acceleration are topics rarely covered in most books on building Docker applications, so by this stage of the book the reader should be in a relatively unique position to containerise some more interesting and unusual applications of their own.

Run OpenGL Area

Application Class
 Assistant
 ▶ Benchmark
 Builder
 Button Boxes
 Change Display
 Clipboard
 Color Chooser
 Combo Boxes
 Cursors
 Dialogs and Message Boxes
 Drawing Area
 ▶ Entry
 Expander
 Flow Box
 Foreign drawing
 Gestures
 Header Bar
 ▶ Icon View
 Images
 Info Bars
 Links
 List Box
 Menus
 Model Button
 ▶ Offscreen Windows
OpenGL Area
 ▶ Overlay
 Paned Widgets
 ▶ Pango

Info Source

```
#include <math.h>
#include <gtk/gtk.h>
#include <epoxy/gl.h>

static GtkWidget *demo_
```

/* the GtkGLArea widget

```
static GtkWidget *gl_ar
```

enum {
 X_AXIS,
 Y_AXIS,
 Z_AXIS,
 N_AXIS
};

/* Rotation angles on each axis */
static float rotation_a

/* The object we are drawing */
static const GLfloat ve
{
 0.f, 0.5f, 0.f, 1
 0.5f, -0.366f, 0.f, 1
 -0.5f, -0.366f, 0.f, 1
};

/* Initialize the GL buffers */
static void
init_buffers (GLuint *vao,
 GLuint *vbo,
 GLuint vao, buffer;

/* We only use one VAO */
glGenVertexArrays (1,

```
glBindVertexArray (va
```

/* This is the buffer that holds the vertices */



Animated Backgrounds

```
/* You can edit the text in this window to change the
 * appearance of this Window.
 * Be careful, if you screw it up, nothing might be visible
 * anymore. :)
```

```
/* This CSS resets all properties to their defaults values
 * and overrides all user settings and the theme in use
@import url("resource://css_pixbufs/reset.css");
@import url("resource://css_pixbufs/cssview.css");

@keyframes move-the-image {
0% { background-position: 50.00% 75.00%, 67.68% 67.68%, 7
3.125% { background-position: 55.19% 76.11%, 72.14% 64.79%
6.25% { background-position: 60.79% 76.04%, 76.04% 60.79%
9.375% { background-position: 66.46% 74.64%, 79.06% 55.78%
12.5% { background-position: 71.84% 71.84%, 80.89% 50.00%
15.625% { background-position: 76.55% 67.74%, 81.32% 43.7
18.75% { background-position: 80.21% 62.51%, 80.21% 37.49%
21.875% { background-position: 82.54% 56.47%, 77.58% 31.5
25% { background-position: 83.33% 50.00%, 73.57% 26.43%,
```

4-4 Adding Audio

Most of the Docker examples on the web are resolutely silent and including a chapter on audio is perhaps a little odd in a book about GUI applications, but this book is about running GUI applications in Docker, with the emphasis on **applications**. Many of the applications that people might wish to deploy in containers, such as browsers and games, are very much multimedia applications and it is important to include audio support in order for them to behave as expected.

4-4-1 PulseAudio

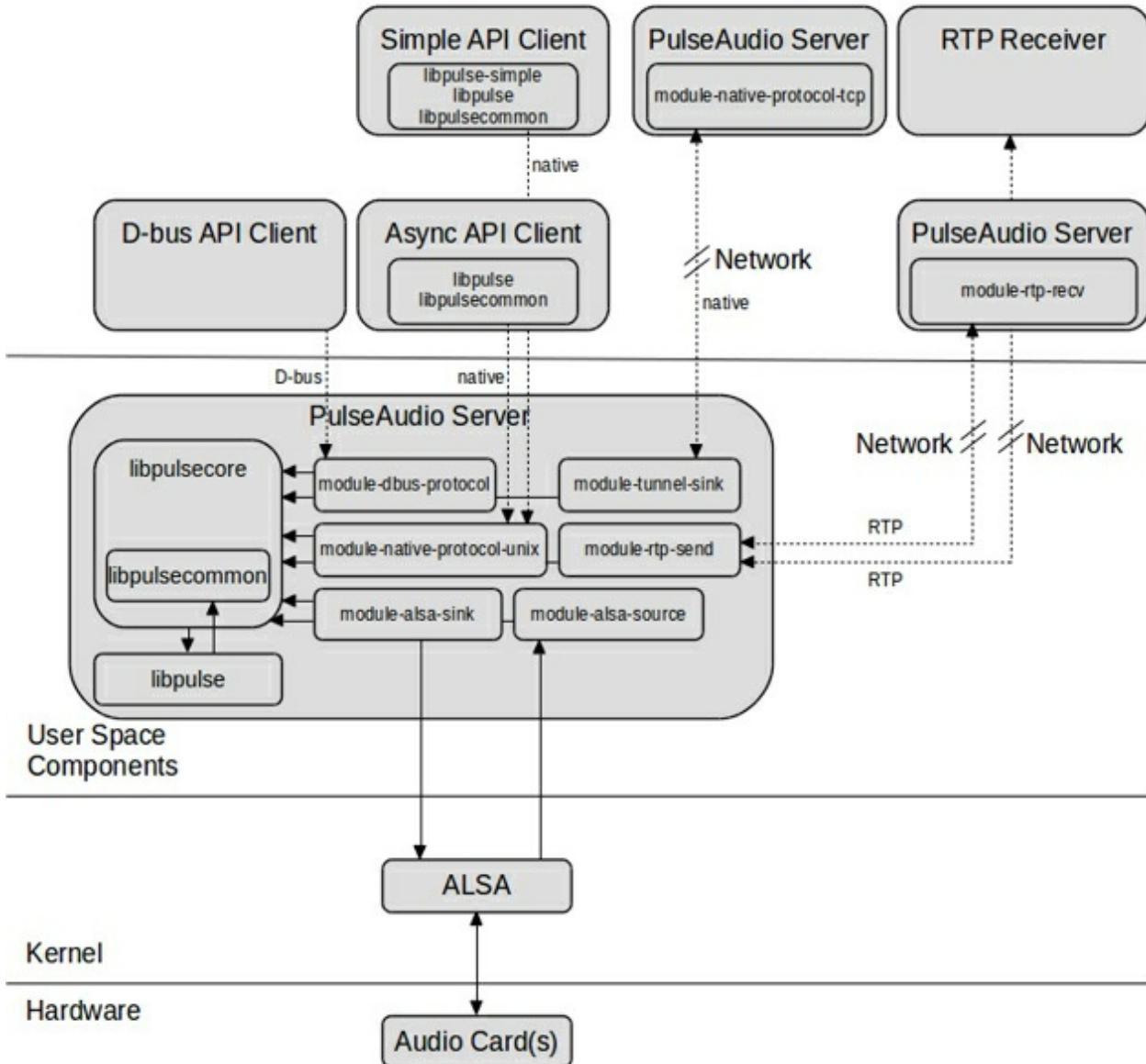
PulseAudio is the default audio subsystem of most modern Linux distributions; it has had a rather chequered history and somewhat of a “Marmite” reputation where it is either loved or hated, but regardless of that it is sufficiently ubiquitous for Firefox to have dropped direct ALSA support in favour of it.

PulseAudio acts as a network capable sound server, where a daemon is created that accepts input from multiple sound sources and redirects the audio to one or more sinks, and a comprehensive set of libraries and adapters are available to support a broad range of native and legacy clients.

Perhaps the best way to think of PulseAudio is as the audio equivalent of a compositing window manager where, rather than composing window buffers, audio buffers from multiple applications are composed by the server into a single buffer and written to the audio device.

The PulseAudio Architecture is illustrated below, note the clear similarities to the [X11 Architecture](#).

Applications



The diagram illustrates the most important components arranged as one local PulseAudio server, one “native” remote PulseAudio server and another connected via RTP, one remote RTP receiver, an ALSA back-end, and all required modules.

The main PulseAudio components are:

- **libpulse-simple** : A wrapper around libpulse providing a simple synchronous client library.
- **libpulse** : An asynchronous API for client or server applications that communicates with the server via a *native* protocol over either a Unix domain socket or TCP stream socket.
- **libpulsecommon** : Parts from libpulsecore needed by clients and server that are not part of the public API.
- **libpulsecore** : An internal server API for modules.
- **modules** : Server extensions such as network protocols, drivers, desktop integrations, etc.

PulseAudio also supports a rich variety of network protocols:

native

This is suitable for most client-server and server-server communications over a Unix domain or TCP stream socket. When the native protocol is used between client and server on the same host, the zero-copy mode may be employed, which requires a Unix domain socket to be used, and POSIX shared memory or memfd to be enabled.

PulseAudio will attempt to use a default location or the default-server in /etc/pulse/client.conf to find the daemon socket to connect to, it also uses window properties on the root window of the X11 server or the environment variable `$PULSE_SERVER` to help find the daemon. It is important to note that the precedence is environment variable then window properties and finally default location.

simple

The PulseAudio simple protocol may be used to send or receive raw PCM samples from the PulseAudio server without headers, however it is rarely useful or used.

mDNS (Zeroconf)

The Zeroconf multicast DNS protocol resolves names in the local network without using a name server. If an Avahi or Bonjour daemon is running and PulseAudio Zeroconf support is enabled, every sink and source on every PulseAudio server in the local network become automatically available to all other PulseAudio servers.

RTP/SDP/SAP

PulseAudio achieves RTP support for both senders and receivers through the Real-time Transport Protocol (RTP), Session Description Protocol (SDP) and Session Announcement Protocol (SAP). RTP supports multicasting of a single local source to any number of remote sinks.

RAOP

Remote Audio Output Protocol is used in Apple AirPlay devices and uses the Zeroconf multicast DNS protocol for discovery, so every AirPlay device in the local network automatically becomes available to PulseAudio.

ESound

PulseAudio allows interconnectivity with the playback, recording and control protocol used in the Enlightened Sound Daemon, which is now largely deprecated by most Linux distributions.

HTTP

PulseAudio allows audio streaming over HTTP. When a new HTTP client connects to the endpoint, PulseAudio uses the HTTP Content-Type header set with a MIME type corresponding to the sample format in use. After sending headers, PulseAudio then creates a new source output connected to the source or sink monitor which writes all new samples to the HTTP connection.

DLNA and Chromecast

The Digital Living Network Alliance (DLNA) is a set of interoperability guidelines for sharing digital media among multimedia devices using a variety of protocols such as Universal Plug and Play (UPnP), Real-time Transport Protocol (RTP), and custom HTTP APIs. Chromecast is a similar technology built on Google Cast, a proprietary stack that uses Google Protocol Buffers and mDNS.

D-bus

The PulseAudio D-bus API allows server-side objects to be inspected and controlled over D-bus, however the D-bus API is limited and can't be used for playback and recording and it is not actively maintained.

CLI protocol

Finally, PulseAudio implements its own Command Line Interface protocol, which is a simple text protocol providing a range of commands to inspect and control the server. The syntax of the PulseAudio CLI protocol may be seen in the `/etc/pulse/default.pa` script and is also used in the **pacmd** tool.

The [Dockerfile](#) for our first containerised PulseAudio application is in the [pulseaudio-utils](#) directory of [docker-gui](#) under [4-local-applications/adding-audio](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    pulseaudio-utils && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

CMD ["pacat", "-v", "/dev/urandom"]
```

and may be built with:

```
docker build -t pulseaudio-utils .
```

This is a trivial application whose default behaviour is to play noise from /dev/urandom (very loudly!), however the following lines require some explanation.

```
cp /etc/pulse/client.conf /etc/pulse/client-noshm.conf
sed -i "s/enable-shm = yes/enable-shm = no/g" \
    /etc/pulse/client-noshm.conf
```

The reason for creating an alternative `client.conf` that disables PulseAudio shared memory is due to a [PulseAudio bug](#) that affects versions 7 to 9, whereby PulseAudio's mechanism for cleaning up shm files is overly aggressive in a sandboxed environment such as Docker.

The PulseAudio developer [tanuk](#) explained the issue in a [Firejail bug report](#) thus:

Hmm... I have a likely explanation for the bug. Whenever PulseAudio creates a new shm file, it also cleans up any files that are left by crashed processes that didn't clean up the files they created. Each shm file contains the pid of the process that created it. The cleanup is triggered, if `kill(pid, 0)` fails with `ESRCH`, that is, if the pid does not exist. If firejail creates a new pid namespace, then processes outside the sandbox aren't visible to processes in the sandbox, and a sandboxed process will "clean up" all shm files.

So yes, this is a PulseAudio bug after all. With pid namespaces, we can't assume that a process is dead if `kill()` fails with `ESRCH`.

This will be resolved when we get memfd support in PulseAudio (currently being worked on). I'm not sure that it will be ready before the next release, so some other fix may be needed. I don't yet know what that fix might be, though. Not cleaning up the shm files isn't really a good solution.

The most common solution to this issue is to disable shared memory support entirely in the `client.conf` by setting `enable-shm = no`, which works but is perhaps slightly heavy handed because PulseAudio zero-copy mode is the most efficient and lowest-latency way to use PulseAudio when client and server are on the same host.

The reason that this issue first arose in PulseAudio version 7 was due to the Shared Ringbuffer Channel optimisation [srbchannel](#) being made the default, therefore instead of completely disabling shared memory an alternative option is to *disable srbchannel*.

Unfortunately it is not possible to disable srbchannel in the `client.conf`, it is necessary (on the host) to either modify `/etc/pulse/default.pa` or create a copy `default.pa` in `~/.config/pulse` replacing the line:

```
load-module module-native-protocol- unix
```

with

```
load-module module-native-protocol- unix srbchannel=no
```

PulseAudio version 9 introduced the memfd shared memory mechanism, which is viewed by the PulseAudio developers as the real solution to this problem. In PulseAudio version 10 memfd support is enabled by default, but with version 9 memfd support has to be manually enabled by adding `enable-memfd = yes` to `daemon.conf`.

Because enabling memfd in version 9 and disabling srbchannel are both solutions that require modifications to the PulseAudio daemon configuration they will affect *all* PulseAudio connections for a user, therefore a compromise approach may be to detect the PulseAudio version and only disable shared memory for versions 7, 8 and 9.

The `noise.sh` launch script detects the version and sets the `PULSE_CLIENTCONFIG` environment variable to point to the `client-noshm.conf` file created by the Dockerfile.

The script also illustrates the `PULSE_SERVER` variable used to pass the location of the PulseAudio Unix domain socket, and bind-mounting the volumes containing the PulseAudio socket and authentication cookie.

```
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# For pulseaudio versions 7 to 9 there is a bug whereby shm
# files get "cleaned up" incorrectly in containers. Force
# those versions to disable shared memory. Pulseaudio 10
# enables memfd by default, which apparently fixes this.
# See https://bugs.freedesktop.org/show_bug.cgi?id=92141
PULSE_VERSION=$(pulseaudio --version | sed 's/[^\0-9.]*/([0-9]*\.\*)*/\1/'')
if ([[ $PULSE_VERSION -gt 6 ]] && [[ $PULSE_VERSION -lt 10 ]]);
then
    PULSE_FLAGS="-e PULSE_CLIENTCONFIG=/etc/pulse/client-noshm.co
nf"
fi

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
-v $XDG_RUNTIME_DIR/pulse:$XDG_RUNTIME_DIR/pulse:ro \
-v $HOME/.config/pulse/cookie:$HOME/.config/pulse/cookie:ro \
$PULSE_FLAGS \
pulseaudio-utils
```

Another example following the same pattern is the [ok.sh](#) launch script, also found in the [pulseaudio-utils](#) directory, that uses paplay to say OK!

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# For pulseaudio versions 7 to 9 there is a bug whereby shm
# files get "cleaned up" incorrectly in containers. Force
# those versions to disable shared memory. Pulseaudio 10
# enables memfd by default, which apparently fixes this.
# See https://bugs.freedesktop.org/show_bug.cgi?id=92141
PULSE_VERSION=$(pulseaudio --version | sed 's/[^\0-9.]*\([0-9]*\)\.*/\1/'')
if ([[ $PULSE_VERSION -gt 6 ]] && [[ $PULSE_VERSION -lt 10 ]]);
then
    PULSE_FLAGS="-e PULSE_CLIENTCONFIG=/etc/pulse/client-noshm.co
nf"
fi

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
-v $XDG_RUNTIME_DIR/pulse:$XDG_RUNTIME_DIR/pulse:ro \
-v $HOME/.config/pulse/cookie:$HOME/.config/pulse/cookie:ro \
$PULSE_FLAGS \
-v $PWD/ok.wav:$HOME/ok.wav:ro \
pulseaudio-utils paplay $HOME/ok.wav
```

4-4-2 ALSA

ALSA is the Advanced Linux Sound Architecture, a software framework that provides kernel level sound card drivers and a user space library and API for application developers. Unlike the kernel API, ALSA's user-space library provides a standardised abstraction across disparate underlying hardware elements by using software plug-ins.

ALSA replaced the original Linux Open Sound System (OSS) drivers, in part due to OSS becoming proprietary for a while and in part because some key features were not supported by OSS at the time:

- Hardware MIDI synthesis.
- Hardware mixing.
- Full-duplex operation.
- Thread-safe SMP support.

Although [PulseAudio](#) is the default audio subsystem for most modern Linux distributions, there are still some occasions where it is necessary or preferable to be able to use ALSA directly to access the underlying low-level audio hardware from Docker containers. Some use cases for this include:

- Running a [PulseAudio](#) server in a container.
- Running a [JACK](#) server in a container.
- Running ALSA's own command-line utilities alsactl, amixer, arecord/aplay and alsamixer in a container.
- Niche applications requiring extremely low-latency.

ALSA can typically support up to eight audio cards, numbered 0 to 7, alternatively each card may also be addressed by an *id* string such as “PCH”, “ICH9” etc.

Each card presents an interface describing the “protocol” for accessing the card, such as *hw* , *plughw* , *default* , *dsnoop* , *hdmi* , *dmix* , *plug:dmix* , etc. The *hw* interface provides direct access to the kernel device, but has no software mixing or stream adaptation support. The *plughw* and *default* interfaces provide features such as sample-rate conversion and software mixing and typically enable sound output where the *hw* interface would produce an error.

Each card has a number of devices such as *playback* , *capture* , *control* , *timer* , or *sequencer* and devices may have sub-devices representing a specific device endpoint like a speaker pair. Devices and subdevices are numbered from 0.

An application typically addresses ALSA by a device string of the form
interface:card,device,subdevice e.g.

```
default:CARD=PCH
surround51:CARD=PCH,DEV=0
dmix:CARD=Nvidia,DEV=7
hw:CARD=Nvidia,DEV=7
plughw:CARD=Nvidia,DEV=7
...
```

ALSA streams are most commonly PCM and must be configured to match the underlying hardware by specifying sampling rate, sample width (8, 16, 24 or 32 bits per sample), sample encoding (endianness etc.) and number of channels.

It is obviously extremely useful that ALSA is able to support multiple different audio cards and their subdevices simultaneously, the flip side of this however is that it can often be *extremely* confusing trying to work out which logical device your speakers might be connected to.

In order to help determine which device does what, our ALSA example is a simple application that enumerates the ALSA logical PCM devices and attempts to play a sound on each.

The [Dockerfile](#) for our containerised ALSA application is in the [alsa-utils](#) directory of [docker-gui](#) under [4-local-applications/adding-audio](#).

```
FROM debian:stretch-slim

# Install alsa-utils
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    alsa-utils && \
    rm -rf /var/lib/apt/lists/*

COPY ./startup.sh /src/startup.sh

ENTRYPOINT ["/src/startup.sh"]
```

and may be built with:

```
docker build -t alsa-utils .
```

The Dockerfile is fairly trivial and simply installs the alsa-utils package then copies the following [startup.sh](#) script to the image's /src directory.

```

for i in $(aplay -L | grep CARD); do
    echo trying device $i
    aplay -D $i $HOME/ok.wav
    echo
    echo
    sleep 2
done

```

This script uses `aplay -L | grep CARD` to enumerate every ALSA PCM device, then iterates through them, printing the device name and attempting to play a sound.

The [try-pcm.sh](#) launch script is as follows:

```

# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# Check if pasuspender (and thus pulseaudio) is present
# if so then prefix with pasuspender to suspend pulseaudio
# for the duration of the test.
if test -f /usr/bin/pasuspender; then
    DOCKER_COMMAND="pasuspender -- \"$DOCKER_COMMAND"
fi

$DOCKER_COMMAND run --rm \
    --device=/dev/snd \
    --group-add $(cut -d: -f3 < <(getent group audio)) \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/ok.wav:$HOME/ok.wav:ro \
    alsa-utils

```

This launch script is mostly very straightforward, however there are some notable points:

1. `docker run` is prefixed by **pasuspender**, a tool that can be used to tell a local PulseAudio sound server to temporarily suspend access to the audio devices to allow other applications to access them directly. pasuspender will suspend access to the audio devices, fork a child process, and when the child process terminates resume access again.
2. The `--device=/dev/snd` flag is used to grant the container access to the audio devices used by ALSA.
3. The **audio** group is added to the container via

```
--group-add $(cut -d: -f3 < <(getent group audio))
```

The main reason we need to add the audio group is because we have, as per our previous examples, reduced the container's privileges by setting `-u $(id -u):$(id -g)`, which sets the container's user and group to those of the user launching the container.

Most Linux distributions use udev (or HAL) and ConsoleKit to dynamically give access to the currently “active” user, but allow that to be overridden using the audio group. However, because our container is sandboxed even though the container user is the same as the user launching the container we must use the audio group override. A less secure alternative is to run as group **root**, by only setting `-u $(id -u)`, that is setting the user *but not the group* of the container to that of the user launching the container.

PulseAudio as an ALSA Device

The previous example illustrated ALSA being used directly with one or more real hardware audio devices, but another use case for running ALSA in a container is where the application *only* has direct audio support for ALSA, but we wish to integrate it with PulseAudio for a seamless desktop audio experience.

This goal is actually very simple to achieve, requiring only that we include PulseAudio in the image, as that configures PulseAudio to be the default device for clients.

The [Dockerfile](#) for our containerised ALSA+PulseAudio application is in the [alsa-pulseaudio](#) directory of [docker-gui](#) under [4-local-applications/adding-audio](#).

```
FROM debian:stretch-slim

# Install alsa-utils and pulseaudio
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    alsa-utils pulseaudio && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

CMD ["aplay", "-L"]
```

and may be built with:

```
docker build -t alsa-pulseaudio .
```

The [ok.sh](#) launch script for running aplay with a PulseAudio back-end is practically identical to the ok.sh launch script for paplay, illustrated in the previous [PulseAudio](#) section. Note that by using PulseAudio as the ALSA device we don't need to add `--device=/dev/snd` to the container, nor do we need to add the audio group.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# For pulseaudio versions 7 to 9 there is a bug whereby shm
# files get "cleaned up" incorrectly in containers. Force
# those versions to disable shared memory. Pulseaudio 10
# enables memfd by default, which apparently fixes this.
# See https://bugs.freedesktop.org/show_bug.cgi?id=92141
PULSE_VERSION=$(pulseaudio --version | sed 's/[^\0-9.]*\([0-9]*\)\.*/\1/')
if ([[ $PULSE_VERSION -gt 6 ]] && [[ $PULSE_VERSION -lt 10 ]]);
then
    PULSE_FLAGS="-e PULSE_CLIENTCONFIG=/etc/pulse/client-noshm.co
nf"
fi

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
-v $XDG_RUNTIME_DIR/pulse:$XDG_RUNTIME_DIR/pulse:ro \
-v $HOME/.config/pulse/cookie:$HOME/.config/pulse/cookie:ro \
$PULSE_FLAGS \
-v $PWD/ok.wav:$HOME/ok.wav:ro \
alsa-pulseaudio aplay $HOME/ok.wav
```

4-4-3 JACK

JACK is a real-time low-latency sound server used for routing both audio and MIDI data between applications.

As a sound server JACK appears to have similar goals to PulseAudio, however PulseAudio is focused on consumer desktop and mobile audio needs. It doesn't try to address low latency usage but does provide seamless device switching, network routing, global per-application volume control.

JACK, on the other hand, is completely focused on the needs of pro-audio and music creation users. It offers the lowest possible latency, complete routing flexibility between applications and audio hardware, and all audio is always sample synchronized so applications don't run ahead of or behind others. It doesn't however provide the smooth and seamless desktop experience that PulseAudio is aiming at.

JACK has long been the de facto standard for professional audio on Linux, where *real-time* and *low-latency* are key differentiators. JACK's scheduling requirements to achieve sufficiently low latencies have been one of the driving forces behind the real-time optimisation effort for the Linux kernel.

Routing flexibility is another major feature. JACK not only deals with connections between programs, but also the connections *within* programs. The beauty of this is that these connections are also available for any other JACK aware programs to connect to. They are not restricted to only being “internal” input and outputs.

JACK is powerful, but it is also complex and confusing for new users coming to it from a consumer audio background. Before getting started with JACK it is a good idea to have at least a passing familiarity with ALSA, as configuring the JACK daemon requires knowledge of the ALSA devices. The simple device enumeration application from the [ALSA](#) section of this book lets the most relevant device be identified.

Another thing that can add to the confusion around JACK is that there are in fact **two** different implementations! Moreover, JACK 1 and JACK 2 are actually *equivalent* implementations of the same protocol, not different versions.

JACK 2 was originally written to add multi-processor support and to rewrite the internals using C++. Programs compiled against JACK 1 will work with JACK 2 without recompile (and vice versa). JACK 2 was initially planned to replace JACK 1, but this is no longer the case and they are considered to be equivalent implementations. A comprehensive list of the differences between and motivations for the different versions may be found in the [JACK FAQ](#).

Although JACK 1 and JACK 2 are equivalent and library compatible, this does not imply that they are *interoperable*. In particular the shared memory handles and Unix domain socket paths are different between the two versions, so a JACK 1 client cannot connect to a JACK 2 daemon nor vice versa. They also differ with respect to their dependencies, with JACK 2 requiring D-bus integration by default, although it may be recompiled to remove the D-bus dependency.

As well as JACK 1 and JACK 2 not being interoperable with each other. **it is important to note** that different *release versions* of either JACK 1 or JACK 2 might not be interoperable with earlier or later versions. In a normal desktop environment this might not be a common use case, however with Docker it is not at all uncommon to use a range of different distributions as base images.

The author encountered this issue himself whilst developing this chapter and spent *literally days* assuming that the reason why a test application produced no sound from JACK running on his host was due to a problem with container privileges, when it turned out that simply changing the Dockerfile base image to match his host distribution resolved the problem.

Enabling Kernel Realtime Scheduling

Docker > 1.12 supports real-time scheduling in containers, but to enable this the host's kernel must be configured correctly. Distributions such as Ubuntu have this enabled by default, but for others including CentOS 7 it must be explicitly enabled:

```
sudo sysctl -w kernel.sched_rt_runtime_us=-1
```

To permanently enable the real-time scheduler, edit /etc/sysctl.conf to include:

```
kernel.sched_rt_runtime_us=-1
```

Apply the changes to the running configuration with:

```
sudo sysctl -p
```

JACK 1

The [Dockerfile](#) for our containerised JACK 1 jackd and mplayer application is in the [jackd1-mplayer](#) directory of [docker-gui](#) under [4-local-applications/adding-audio](#).

```
FROM debian:stretch-slim

# Install mplayer and jack
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    mplayer jackd1 libjack0 && \
    rm -rf /var/lib/apt/lists/*
```

and may be built with:

```
docker build -t jackd1-mplayer .
```

The [jackd.sh](#) launch script for JACK 1 is as follows, although running this script may be unnecessary if jackd for JACK 1 is already running on the host (though noting the earlier comments that different releases of JACK, even for the same version, may not be interoperable).

```
# Replace with real device name
ALSA_DEVICE=hw:CARD=NVidia,DEV=7

# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi
```

```

# Check if pasuspender (and thus pulseaudio) is present
# if so then prefix with pasuspender to suspend pulseaudio
# for the duration of the test.
if test -f /usr/bin/pasuspender; then
    DOCKER_COMMAND="pasuspender -- \"$DOCKER_COMMAND"
fi

$DOCKER_COMMAND run --rm \
    --ulimit rtprio=99 \
    --ulimit memlock=83886080 \
    --ipc=host \
    --device=/dev/snd \
    --group-add $(cut -d: -f3 < <(getent group audio)) \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    jackd1-mplayer jackd -d alsa -d $ALSA_DEVICE -r 44100 -n 2 -p
2048

```

As was the case with the earlier ALSA example, this script prefixes `docker run` with **pasuspender**, adds the `--device=/dev/snd` flag and adds the audio group to the container. In addition a few other flags are necessary for JACK to run correctly in a container:

1. Setting `--ipc=host` is necessary in order to share SysV IPC between the host and container.
2. Setting `--ulimit rtprio=99` sets the maximum realtime priority allowed for the container and gives the container realtime priority.
3. Setting `--ulimit memlock=83886080` lets the container page-lock memory of the specified size. Page-locked memory is locked into physical memory and not paged out, which improves performance and reduces latency at the expense of poorer resource utilisation.

If all goes well, and the ALSA_DEVICE has been correctly set, the jackd.sh script should launch an instance of jackd with realtime priority and page locked memory of the required size that will be listening for inbound client connections.

N. B. this script is **unlikely to work out of the box**, because the ALSA_DEVICE variable will need to be set to match the reader's own configuration, which may be determined by running the [ALSA](#) example from the previous section.

To test our jackd container we can use a simple launch script [ok.sh](#) that uses mplayer to play a wav file over JACK.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# Check if pasuspender (and thus pulseaudio) is present
# if so then prefix with pasuspender to suspend pulseaudio
# for the duration of the test.
if test -f /usr/bin/pasuspender; then
    DOCKER_COMMAND="pasuspender -- \"$DOCKER_COMMAND"
fi

$DOCKER_COMMAND run --rm \
    --ulimit rtprio=99 \
    --ulimit memlock=661000000 \
    --ipc=host \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/ok.wav:$HOME/ok.wav:ro \
    jackd1-mplayer mplayer -ao jack $HOME/ok.wav
```

In ok.sh the container is configured to use the flags `--ipc=host` , `--ulimit rtprio=99` and `--ulimit memlock=661000000` in a similar way to that described previously for jackd.

JACK 2

The [Dockerfile](#) for our containerised JACK 2 jackd and mplayer application is in the [jackd2-mplayer](#) directory of [docker-gui](#) under [4-local-applications/adding-audio](#).

```
FROM debian:stretch-slim

# Install mplayer and jack
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    mplayer jackd2 libjack-jackd2-0 && \
    rm -rf /var/lib/apt/lists/*
```

and may be built with:

```
docker build -t jackd2-mplayer .
```

The [jackd.sh](#) launch script for JACK 2 is as follows.

```
# Replace with real device name
ALSA_DEVICE=hw:CARD=NVidia,DEV=7

# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi
```

```

if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro -e NO_AT_BRIDGE=1"
fi

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Check if pasuspender (and thus pulseaudio) is present
# if so then prefix with pasuspender to suspend pulseaudio
# for the duration of the test.
if test -f /usr/bin/pasuspender; then
    DOCKER_COMMAND="pasuspender -- \"$DOCKER_COMMAND"
fi

$DOCKER_COMMAND run --rm \
    --ulimit rtprio=99 \
    --ulimit memlock=83886080 \
    --ipc=host \
    --device=/dev/snd \
    --group-add $(cut -d: -f3 < <(getent group audio)) \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    jackd2-mplayer jackd -d alsa -d $ALSA_DEVICE -r 44100 -n 2 -p 2048

```

If all goes well, and the `ALSA_DEVICE` has been correctly set, the `jackd.sh` script should launch an instance of `jackd` with realtime priority and page locked memory of the required size that will be listening for inbound client connections.

This example again follows the same pattern that we employed for JACK 1, where the script prefixes the `docker run` command with **`pasuspender`**, then adds the `--device=/dev/snd` flag and adds the audio group to the container, as well as the `--ipc=host`, `--ulimit rtprio=99` and `--ulimit memlock=83886080` flags previously described.

In addition, because JACK 2 requires D-bus integration this script includes the D-bus and AppArmor flags that were described in detail in the section on [gnome-calculator](#).

N.B. as was the case with our JACK 1 `jackd.sh` script this script is **unlikely to work out of the box**, because the `ALSA_DEVICE` variable will need to be set to match the reader's own configuration, which may be determined by running the [ALSA](#) example from the previous section.

In order to test our JACK 2 `jackd` container we can use a simple launch script [ok.sh](#) that uses `mplayer` to play a wav file over JACK.

```

# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# Check if pasuspender (and thus pulseaudio) is present
# if so then prefix with pasuspender to suspend pulseaudio
# for the duration of the test.
if test -f /usr/bin/pasuspender; then
    DOCKER_COMMAND="pasuspender -- \"$DOCKER_COMMAND"
fi

$DOCKER_COMMAND run --rm \
    --ulimit rtprio=99 \
    --ulimit memlock=83886080 \
    --ipc=host \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/ok.wav:$HOME/ok.wav:ro \
    jackd2-mplayer mplayer -ao jack $HOME/ok.wav

```

In ok.sh the container is again configured to use `--ipc=host` , `--ulimit rtprio=99` and `--ulimit memlock=83886080` in a similar way to that described previously for jackd, but note that it does not require D-bus integration.

4-4-4 OSS

Prior to the introduction of [ALSA](#) the Linux Open Sound System (OSS) was the primary Linux audio interface.

The OSS API is designed to use the traditional Unix I/O framework of `open()`, `read()`, `write()`, and `ioctl()` via special device files. For instance, the default device for sound input and output is `/dev/dsp`.

In part the limitation of that simple API was a reason for the Linux audio community largely abandoning OSS in favour of ALSA, though another key motivation was that OSS support for newer audio devices became proprietary for a while.

OSS is rarely used in modern Linux applications, but some older applications may have only OSS support, so it is included here for completeness. Direct OSS support is rarely included by default in modern Linux distributions, so the approach taken here is to bridge between OSS and PulseAudio using the `padsp` utility. `padsp` replaces device files, such as `/dev/dsp`, via library preloading, tricking the applications into believing that they have exclusive control over the sound card. In fact, their output is rerouted through PulseAudio.

Although bridging OSS via `padsp` is likely to be the most useful way to support OSS applications in a modern Linux distribution, if OSS support **is** directly available then direct support should only require adding `--device=/dev/dsp` to the container.

The [Dockerfile](#) for our containerised OSS application is in the [pulseaudio-utils](#) directory of [docker-gui](#) under [4-local-applications/adding-audio](#). This is the same Dockerfile that we used for our first PulseAudio application.

```
FROM debian:stretch-slim

# Install pulseaudio-utils
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    pulseaudio-utils && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

CMD ["pacat", "-v", "/dev/urandom"]
```

which may be built with:

```
docker build -t pulseaudio-utils .
```

The [oss-noise.sh](#) launch script basically behaves like a PulseAudio application, so we have to follow the same procedure of mounting the PulseAudio socket and cookie and passing the PULSE_SERVER environment variable as we do for any other PulseAudio application.

Another point of note is that because we don't actually have a /dev/dsp special device file, we have to use the tee command to give padsp something to preload against.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# For pulseaudio versions 7 to 9 there is a bug whereby shm
# files get "cleaned up" incorrectly in containers. Force
# those versions to disable shared memory. Pulseaudio 10
# enables memfd by default, which apparently fixes this.
# See https://bugs.freedesktop.org/show_bug.cgi?id=92141
PULSE_VERSION=$(pulseaudio --version | sed 's/[^\0-9.]*/([0-9]*\.\*)/\1/')
if ([[ $PULSE_VERSION -gt 6 ]] && [[ $PULSE_VERSION -lt 10 ]]);
then
    PULSE_FLAGS="-e PULSE_CLIENTCONFIG=/etc/pulse/client-noshm.conf"
fi

$DOCKER_COMMAND run --rm -it \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -v $XDG_RUNTIME_DIR/pulse:$XDG_RUNTIME_DIR/pulse:ro \
    -v $HOME/.config/pulse/cookie:$HOME/.config/pulse/cookie:ro \
    \
    $PULSE_FLAGS \
    pulseaudio-utils bash -c "cat /dev/urandom | padsp tee /dev/dsp > /dev/null"
```

4-5 Scanner Support

Whilst exploring our containerisation of [gedit](#) we discovered that enabling cups based printer support for containers largely *just works* if D-bus support has been enabled, as described in detail in the [gnome-calculator](#) section. If D-bus support is not required by the application an alternative way to enable printer support is to bind-mount the cups socket /var/run/cups/cups.sock.

By way of complementing our ability to print from containers, in this section we shall consider the steps necessary to enable support for scanners, both locally and over a network.

4-5-1 simple-scan

In modern Linux distributions the most common scanner API is [SANE](#), the architecture of which is separated into front-ends (application programs) and back-ends (scanner drivers).

A consequence of this architecture is that network scanning is easily implemented with no special handling required in either the front-ends or back-ends. The **saned** daemon acts as a front-end locally, but simply passes requests and data between network connections and the local scanner.

To illustrate SANE scanning we shall use Simple Scan, which was originally written for Ubuntu as a simplified GUI for SANE that was intended to be easier to use than XSane and provide better integration with the GNOME desktop.

The [Dockerfile](#) for our containerised version of Simple Scan is in the [simple-scan](#) directory of [docker-gui](#) under [4-local-applications/scanner-support](#).

```
FROM debian:stretch-slim

# Install simple-scan
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        libcanberra-gtk3-module simple-scan && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["simple-scan"]
```

and may be built with:

```
docker build -t simple-scan .
```

The Dockerfile is trivial, as the dependency tracking of the apt package management system causes the SANE libraries and drivers to be automatically installed in the image in addition to the Simple Scan application. With the drivers installed there is very little left to do to enable local scanner support.

The best way for containers to directly access USB scanners is by adding `--device=/dev/bus/usb` to the `docker run` command. Adding this flag is not however sufficient to enable the scanner as the user running the application must also be a member of the lp group, which may be accomplished by adding the following flag:

```
--group-add $(cut -d: -f3 < <(getent group lp))
```

The [local-simple-scan.sh](#) launch script is nearly identical to our [geditV1.sh](#) script except for the two `docker run` flags described above and, obviously, the image being used.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == **"abstract"** ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

# Add flags for connecting to the D-bus system bus.
# This is necessary to connect to colord, a service
# that makes it easy to manage, install and generate colour
# profiles to accurately colour manage devices.
# https://www.freedesktop.org/software/colord/intro.html
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sys
tem_bus_socket:ro "$DBUS_FLAGS

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY nmerge -
```

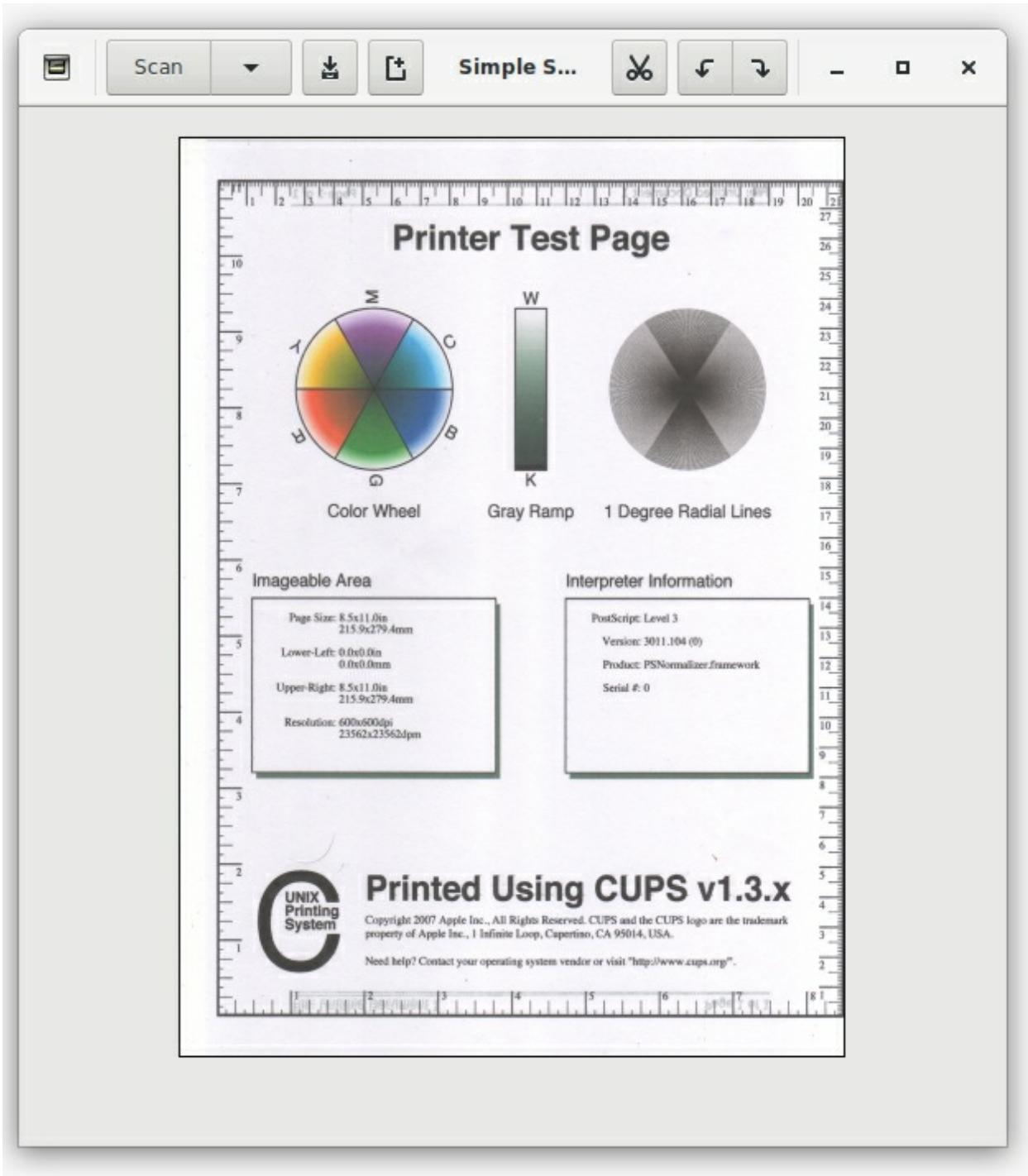
```

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un) /.config/dconf
$DOCKER_COMMAND run --rm \
--device=/dev/bus/usb \
--group-add $(cut -d: -f3 < <(getent group lp)) \
$APPARMOR_FLAGS \
$DBUS_FLAGS \
-e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
-v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix:ro \
-e XAUTHORITY=$DOCKER_XAUTHORITY \
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
simple-scan

```

As with our previous GNOME applications we have again followed the approaches described in detail in the [gnome-calculator](#) chapter to solve the dconf/D-bus and AppArmor issues, so Simple Scan should run without error.

All being well the script should display a window similar to the following, which illustrates Simple Scan scanning the test page printed from our earlier [evince](#) example. As with [evince](#) and the earlier [gedit](#) examples we should also be able to print from Simple Scan, showing us that given a little thought hosting an application in a Docker container need not constrain its features, even those with hardware dependencies.



4-5-2 Network Scanning and saned

Enabling network scanning is somewhat more involved than local scanning, however most of the extra effort required is expended setting up **saned**, which can be little bit fiddly.

The tutorials <https://feeding.cloud.geek.nz/posts/setting-up-a-network-scanner-using-sane/> and <https://help.ubuntu.com/community/SaneDaemonTutorial> were *extremely* useful to the author when setting up saned.

Before exploring saned in a container we shall first run it on the Ubuntu 18.04 VM used to test this section natively, as that installation has libsane and sane-utils installed by default. The following test should confirm that it detects the local scanner.

```
scanimage -L
```

This should give an output similar to the following:

```
device `epson2:libusb:001:004' is a Epson CX6000 flatbed scanner
```

Having verified that the scanner is detected locally, we need to edit /etc/sane.d/saned.conf and tell saned who to share with in the `## Access list` section.

```
## Access list
192.168.0.0/24
172.17.0.0/24
```

The configuration above tells saned to share with everyone on the local network and also everyone on the Docker bridge network. The latter is important if using a virtual machine with NAT networking as a Docker host, as in that case the service will not be exposed on the local network.

Having specified the addresses that we wish to expose we can tell SANE to run as a server listening on port 6566 by editing /etc/systemd/system/saned.socket to read:

```
# Defaults for the saned initscript, from sane-utils

# To enable under systemd please read README.Debian
# Set to yes to start saned under SysV
RUN=no

# Set to the user saned should run as
RUN_AS_USER=saned

[Unit]
Description=saned incoming socket

[Socket]
ListenStream=6566
Accept=yes
MaxConnections=64

[Install]
WantedBy=sockets.target
```

Note that setting `MaxConnections=64` is a workaround for a sane-utils [bug](#) that causes the error:

```
saned.socket: Too many incoming connections (1), dropping connection.
```

Then edit /etc/systemd/system/saned@.service:

```
[Unit]
Description=Scanner Service
Requires=saned.socket

[Service]
ExecStart=/usr/sbin/saned
User=saned
Group=saned
StandardInput=null
StandardOutput=syslog
StandardError=syslog
# Environment=SANE_CONFIG_DIR=/etc/sane.d SANE_DEBUG_DLL=255
```

To start the service execute the following commands:

```
sudo systemctl enable saned.socket
sudo systemctl start saned.socket
systemctl status saned.socket
```

Which should result in a message similar to:

```
● saned.socket
   Loaded: loaded (/etc/systemd/system/saned.socket; enabled; v
endor preset: ena
   Active: active (listening) since Sun 2018-07-15 02:18:05 PDT
; 4h 41min ago
     Listen: [::]:6566 (Stream)
   Accepted: 12; Connected: 0
     Tasks: 0 (limit: 4636)
    CGroup: /system.slice/saned.socket

Jul 15 02:18:05 ubuntu systemd[1]: Listening on saned.socket.
```

Note that the instructions above did not actually work first time for the author and it was necessary to add the saned user to the lp group with:

```
sudo usermod -a -G lp saned
```

and **reboot** in order to *finally* make saned visible to systemd.

The [network-simple-scan.sh](#) launch script is nearly identical to our local version, but omits the `--device` and `--group-add` flags described previously.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == **"abstract"** ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

# Add flags for connecting to the D-bus system bus.
# This is necessary to connect to colord, a service
# that makes it easy to manage, install and generate colour
# profiles to accurately colour manage devices.
# https://www.freedesktop.org/software/colord/intro.html
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sy
stem_bus_socket:ro "$DBUS_FLAGS
```

```

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    $APPPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
    simple-scan $@

```

To run the application we must pass the scanner's network URI:

```
./network-simple-scan.sh net:172.17.0.1:epson2:libusb:001:004
```

The actual URI will depend on the network address of saned and the USB bus and device numbers allocated to the scanner.

Enabling network scanning in a Docker container follows a very similar process to enabling it natively, though for simplicity we shall omit systemd integration.

The [Dockerfile](#) for our containerised version of saned is in the [saned](#) directory of [docker-gui](#) under [4-local-applications/scanner-support](#).

```
FROM debian:stretch-slim

# Install sane-utils
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    sane-utils && \
    rm -rf /var/lib/apt/lists/* && \
    # Set up /etc/sane.d/saned.conf in container
    # Tell saned to use 10000 and 10001 as data ports.
    sed -i "s/# data_portrange = 10000 - 10100/data_portrange = 1 \
0000 - 10001/g" /etc/sane.d/saned.conf && \
    # Tell saned to share with everyone on the local network
    # 192.168.0.0 and everyone on the Docker bridge network
    # 172.17.0.0.
    sed -i "s/scan-client.somedomain.firm/scan-client.somedomain.
firm\n192.168.0.0/24\n172.17.0.0/24/g" /etc/sane.d/saned.conf
&& \
    # Add saned user to lp group
    usermod -a -G lp saned && \
    # Add script to run saned with debug output
    echo "while true; do scanimage -f %d; echo; saned -d5; done"
> /usr/src/saned.sh && \
    chmod +x /usr/src/saned.sh

# Run saned as the non privileged user saned.
USER saned

CMD /usr/src/saned.sh
```

and may be built with:

```
docker build -t saned .
```

The Dockerfile starts simply by following our familiar pattern of installing the required packages via apt. We then update /etc/sane.d/saned.conf, as we did previously when running on the native host, in this case using sed to perform some simple string substitutions to set the required data port and access list values.

```
sed -i "s/# data_portrange = 10000 - 10100/data_portrange = 10000 - 10001/g" /etc/sane.d/saned.conf  
sed -i "s/scan-client.somedomain.firm/scan-client.somedomain.fi  
rm\n192.168.0.0\24\n172.17.0.0\24/g" /etc/sane.d/saned.conf
```

We next add the saned user (which was created when we installed sane-utils) to the lp group, which is necessary in order to correctly access the back-end scanner device.

```
usermod -a -G lp saned
```

We then create a simple script that runs saned in a loop, which we will use as our container entry point.

```
echo "while true; do scanimage -f %d; echo; saned -d5; done" >  
/usr/src/saned.sh  
chmod +x /usr/src/saned.sh
```

The loop is used because when saned is run with the -d option it terminates when the client connection closes.

Note that for a production system using an orchestration platform like Kubernetes or Docker Swarm might be a more reliable method of restarting saned upon termination than our simple loop, but that is beyond the scope of this book.

The [saned.sh](#) launch script is fortunately very simple. Key points of note are the use of `--device=/dev/bus/usb` to access the USB scanner device and the lines exposing the saned listen and data ports via the `-p` flags.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

$DOCKER_COMMAND run --rm -it \
    --device=/dev/bus/usb \
    -p 6566:6566 \
    -p 10000-10001:10000-10001 \
    saned
```

We may again use the [network-simple-scan.sh](#) launch script described previously and once again in order to run the application we must pass the scanner's network URI:

```
./network-simple-scan.sh net:172.17.0.1:epson2:libusb:001:004
```

The actual URI will depend on the network address of saned and the USB bus and device numbers allocated to the scanner.

4-6 Webcam Support

Webcams and other video capture devices have become popular in recent years for applications such as video calling and Vlogging, security and monitoring, and even input control via face and shape tracking.

Video capture on Linux is supported through the use of Video4Linux (V4L), which is a set of device drivers and an API that supports realtime capture from a wide range of devices including USB webcams and TV tuners.

4-6-1 camorama

To illustrate Docker webcam support we shall use camorama, which is a small GNOME utility with very few dependencies that may be used to view and save images from a webcam or any other Video4Linux device.

The [Dockerfile](#) is in the [camorama](#) directory of [docker-gui](#) under [4-local-applications/webcam-support](#).

```
FROM debian:stretch-slim

# Install camorama
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        libcanberra-gtk-module camorama && \
        rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["camorama"]
```

The camorama image may be built with:

```
docker build -t camorama .
```

Enabling video capture in Docker containers is simply a case of following the pattern that we have used previously to enable other hardware. That is by passing the required device, in this case /dev/video0, to the `docker run` command via `--device /dev/video0`.

In addition, in order for Video4Linux to run correctly the user running the application must be a member of the video group, which may be enabled by adding the flag:

```
--group-add $(cut -d: -f3 < <(getent group video))
```

The complete [camorama.sh](#) launch script is as follows:

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == **"abstract"** ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi
```

```

# Add flags for connecting to the D-bus system bus.
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/system_bus_socket:ro \"$DBUS_FLAGS"

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    --device /dev/video0 \
    --group-add $(cut -d: -f3 < <(getent group video)) \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
camorama

```

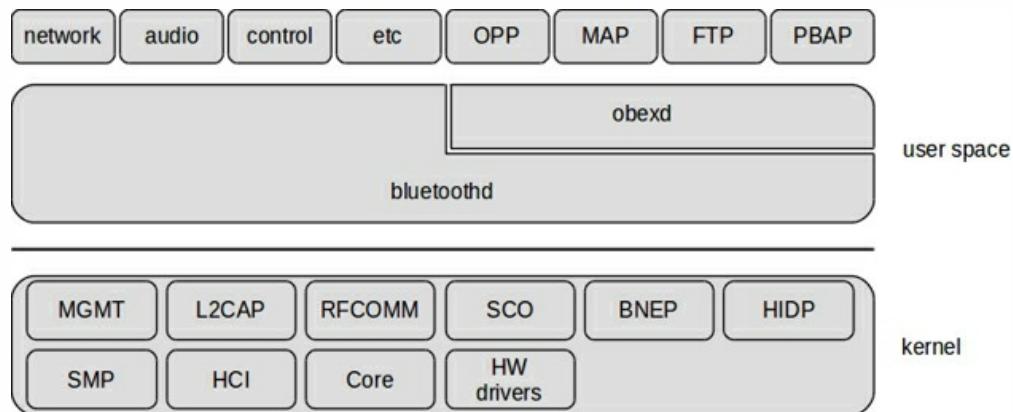


4-7 Bluetooth Applications

Bluetooth has become a fairly ubiquitous way of exchanging data wirelessly over short distances in a valiant attempt to do away with the jungle of wiring usually found connecting computers and devices.

Although accessing Bluetooth services from a container is a relatively uncommon use case and graphical Bluetooth applications are rare, it is worth covering in some detail in this book as integrating with Bluetooth can be quite fiddly and unintuitive and the topic is rarely covered in more mainstream Docker books.

On Linux the Bluetooth stack is split between kernel and user space. The kernel hosts the hardware drivers, low-level protocols and security and provides socket based interfaces to user space, which includes bluetoothd, obexd and tools like bluetoothctl. The primary means of communication between user applications and the Bluetooth stack is via D-bus.



4-7-1 blueman

Because Bluetooth applications primarily use D-bus to communicate with the Bluetooth stack it should, in theory, be relatively straightforward to develop a containerised Bluetooth application by following the patterns previously identified in the [GNOME, D-bus and AppArmor chapter](#).

To illustrate Bluetooth connectivity we shall use the GTK+ based Bluetooth Manager application Blueman, the [Dockerfile](#) for which is in the [blueman](#) directory of [docker-gui](#) under [4-local-applications/bluetooth-applications](#).

```
FROM debian:stretch-slim

# Install blueman
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        libcanberra-gtk3-module blueman && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["blueman-applet"]
```

The blueman image may be built with:

```
docker build -t blueman .
```

Note that the entrypoint for this is the blueman-applet program rather than blueman as the latter is not intended to be run independently of the applet.

The [blueman.sh](#) launch script follows a very similar pattern to that used in our earlier [gedit](#) example, using the D-bus session bus for dconf configuration management and now using the D-bus system bus to communicate with bluetoothd.

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

if [[ $DBUS_SESSION_BUS_ADDRESS == **"abstract"** ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

# Add flags for connecting to the D-bus system bus.
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sy
stem_bus_socket:ro "$DBUS_FLAGS

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY nmerge -
```

```
# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un) /.config/dconf
$DOCKER_COMMAND run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=unix$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
    -e XAUTHORITY=$DOCKER_XAUTHORITY \
    -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \
blueman
```

The Blueman applet may be launched by running `./blueman.sh`, which should result in an icon appearing in the top or bottom bar depending on the distribution being run on the host (on Fedora it is rather hidden bottom left).

There is however a lot that can go wrong. If a Bluetooth icon does not appear, the first thing to check is whether the Bluetooth service is actually running. If it is not, an error such as the following is likely to be present in the console from which blueman.sh was run:

```
dbus.exceptions.DBusException: org.freedesktop.systemd1.NoSuchUnit: Unit dbus-org.bluez.service not found.
```

If that is the case the most likely thing to resolve it is:

```
sudo service bluetooth restart
```

It is also possible that the Bluetooth service may be running with the radio blocked, in which case a solution might be:

```
rfkill unblock bluetooth
```

For the rest of this chapter we shall explore pairing and connecting Bluetooth speakers. This is one of the most common Bluetooth use cases, but also an awkward integration scenario due to the differences between Bluez stack versions and also PulseAudio versions. Of particular note is that the Linux Bluez Bluetooth stack had a major version uplift from 4 to 5 and many interfaces are incompatible.

The blueman image created earlier uses **debian:stretch-slim** as a base image, which uses Bluez version 5. This should “just work” with Linux host distributions that also use Bluez version 5, such as Ubuntu 16.04 and 18.04, CentOS 7 etc. It will not work correctly with Ubuntu 14.04 and its derivatives, which use Bluez version 4, where scanning and pairing work correctly but connecting to the audio device does not.

If a host distribution packaged with Bluez version 4 has to be used one option is to use a different base image for blueman, such as **ubuntu:14.04** or **debian:wheezy-slim** .

An alternative option when working with older host distributions is to use containerised instances of bluetoothd and the PulseAudio daemon that are also based on **debian:stretch-slim** . This second approach shall be described later in this chapter.

In order to use Bluetooth speakers, or indeed any other Bluetooth audio device, the package **pulseaudio-module-bluetooth** must be installed on the host. This may be found in the package repositories of all common Linux distributions.

When the package is installed we can check whether the PulseAudio module **module-bluetooth-discover** has been loaded by running:

```
pactl list | grep -i module-bluetooth
```

which should return:

```
Name: module-bluetooth-policy  
Name: module-bluetooth-discover
```

If the module is not present it may be loaded with:

```
pactl load-module module-bluetooth-discover
```

Now we have checked that the Bluez versions running on our Docker host system and our blueman container are compatible and have also verified that PulseAudio has the required Bluetooth modules loaded we can try pairing and connecting with the device.

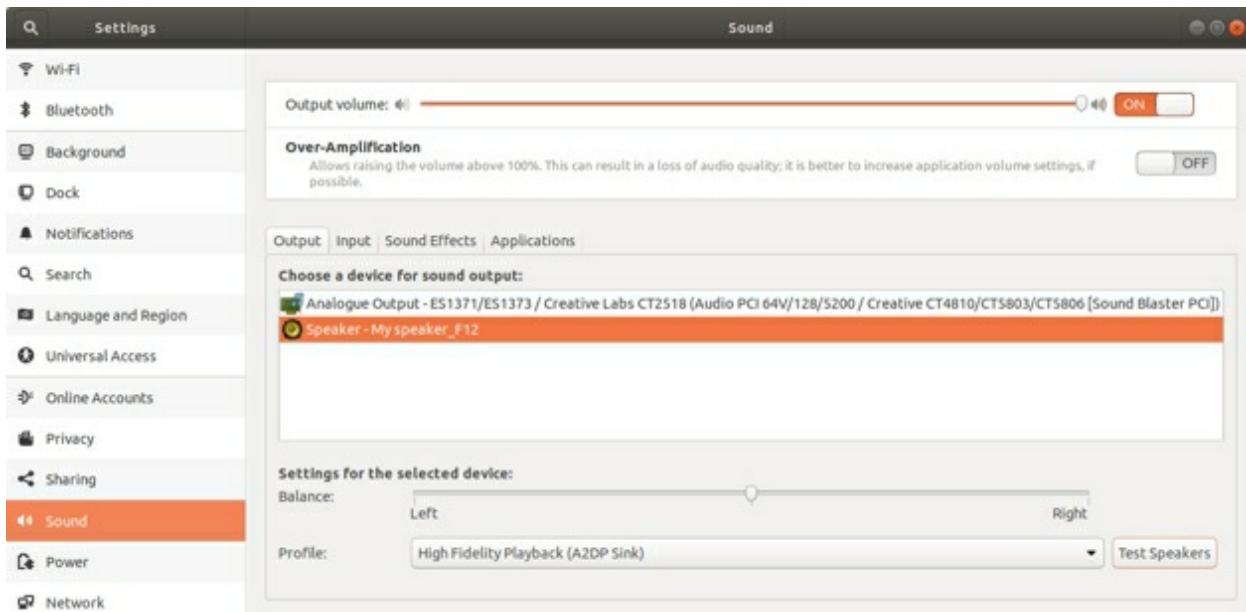
If we click the Bluetooth icon in the top or bottom bar (depending on the Linux distribution used by the host) we should see the Blueman applet menu appear.

Select “Setup New Device...“.

All being well Blueman should launch a wizard to guide the user through the process of pairing and connecting.



If pairing and connecting succeeds, the audio device should be available for selection like any other audio device in the sound settings of the host, such as the following from Ubuntu 18.04. Note the High Fidelity Playback (A2DP Sink) profile.



4-7-2 bluetoothd

Thus far we have shown that an application running in a Docker container can communicate with Bluetooth services running on the host via the D-bus interfaces of the Bluez Bluetooth stack *provided* that the Bluez versions running in the host and container are the same.

Although interesting, running Blueman in this way probably isn't particularly useful because the host most likely has a Bluetooth manager already well integrated. A more interesting challenge arises however if we wish to try a *different* version of Bluez to that packaged with the host, or indeed to run Bluetooth services on a host that doesn't already have them bundled. For this we also need to run bluetoothd in a container.

The [Dockerfile](#) is in the `bluetoothd` directory of [docker-gui](#) under [4-local-applications/bluetooth-applications](#).

```
FROM debian:stretch-slim

# Install bluetoothd
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        bluez && \
    rm -rf /var/lib/apt/lists/*

CMD ["nsenter", "--net=/rootns/net", \
    "bluetoothd", "-d", "-n"]
```

The bluetoothd image may be built with:

```
docker build -t bluetoothd .
```

One interesting point of note about this Dockerfile is the use of [nsenter](#) to run bluetoothd in a different namespace.

As mentioned previously the Bluez Bluetooth stack is split between kernel and user space and provides socket based interfaces from user to kernel space. Unfortunately however, the Linux kernel does not currently allow containers to create Bluetooth sockets, as detailed here <https://github.com/moby/moby/issues/16208>.

One solution is to run the container in the host's network namespace using `--net=host`, however this means that everything running in the container will be in the host's network namespace. An alternative approach is to use nsenter to run only the specified processes in the namespace.

The [bluetoothd.sh](#) launch script is as follows:

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# Add flags for connecting to the D-bus system bus.
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sys
tem_bus_socket:ro "$DBUS_FLAGS
```

```
if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

$DOCKER_COMMAND run --rm \
--device=/dev/rfkill --cap-add=NET_ADMIN \
--cap-add=SYS_ADMIN --cap-add=SYS_PTRACE \
-v /proc/1/ns:/rootns \
$APPARMOR_FLAGS \
$DBUS_FLAGS \
bluetoothd
```

This script is fairly straightforward, but has a few interesting points when compared to running most containers. Firstly the NET_ADMIN capability is required for Bluetooth sockets, secondly nsenter requires the SYS_ADMIN and SYS_PTRACE capabilities and we also need to bind-mount /proc/1/ns from the host to allow its net namespace to be entered.

Running our bluetoothd container should simply be a case of calling `./bluetoothd.sh`, however there are a few additional complications that we may have to consider.

The first complication is that in many cases the host will already have a native version of bluetoothd running, in which case running the container will fail and it is likely to produce the following error:

```
D-Bus setup failed: Name already in use
bluetoothd[1]: Unable to get on D-Bus
```

This error clearly shows the presence of another instance of bluetoothd and may be resolved by simply disabling the native instance by running the following command on the host:

```
sudo service bluetooth stop
```

Another complication on hosts running SELinux, such as CentOS or Fedora, is that the default security policies might block important actions needed by the container.

One such block observed by the author is that docker-runc is denied automount “mounton” access on the rootns directory by default by SELinux, which generates an error similar to the following when running bluetoothd:

```
docker: Error response from daemon: OCI runtime create failed:  
container_linux.go:296: starting container process caused "proc  
ess_linux.go:398: container init caused \"rootfs_linux.go:58: m  
ounting \\"/proc/1/ns\\" to rootfs \\"/var/lib/docker/overla  
y2/dc1da0fee674d89ed9782e2dd4ec36ae3406fc76974abf9b3eb5ccfec6cc  
83a5/merged\\" at \\"/var/lib/docker/overlay2/dc1da0fee674d89  
ed9782e2dd4ec36ae3406fc76974abf9b3eb5ccfec6cc83a5/merged/rootns  
\\" caused \\"permission denied\\\"": unknown.
```

Using information obtained from the SELinux audit logs this may be unblocked by running the following commands:

```
sudo ausearch -c 'runc:[2:INIT]' --raw | audit2allow -M my-runc  
2INIT  
sudo semodule -i my-runc2INIT.pp
```

On hosts running Bluez version 4 natively, yet another complication may be bluetoothd failing to run correctly because of D-bus errors due to the host's D-bus security policy being unaware of the Bluez 5 D-bus interfaces that are used by the blueman and bluetoothd containers.

To resolve this and expose the required interfaces it is necessary to edit the **host** XML file:

```
/etc/dbus-1/system.d/bluetooth.conf
```

and add the following elements as children of

```
<policy user="root">
```

```
<allow send_interface="org.bluez.Agent1"/>
<allow send_interface="org.bluez.MediaEndpoint1"/>
<allow send_interface="org.bluez.MediaPlayer1"/>
<allow send_interface="org.bluez.ThermometerWatcher1"/>
<allow send_interface="org.bluez.AlertAgent1"/>
<allow send_interface="org.bluez.Profile1"/>
<allow send_interface="org.bluez.HeartRateWatcher1"/>
<allow send_interface="org.bluez.CyclingSpeedWatcher1"/>
<allow send_interface="org.bluez.GattCharacteristic1"/>
<allow send_interface="org.bluez.GattDescriptor1"/>
<allow send_interface="org.freedesktop.DBus.ObjectManager"/>
<allow send_interface="org.freedesktop.DBus.Properties"/>
```

Because our containers are connecting to the host's native D-bus instance this change must be carried out on the **host**.

4-7-3 Bluetooth and PulseAudio

As shown earlier, in many cases we can simply run our [blueman](#) container and that will be enough to connect to the host's native Bluetooth services via D-bus.

In other cases we may have an incompatibility between the version of the Bluez Bluetooth stack running on the host and that running on our blueman container, requiring us to also run a compatible version of bluetoothd in a container in order to provide the required D-bus interfaces.

The final scenario that we may face is where there is also an incompatibility with the PulseAudio version running on the host. This can occur because the Bluez and PulseAudio versions required to successfully run Bluetooth audio services are quite tightly coupled.

In this scenario pairing might *appear* to be successful, but blueman connecting to the device will fail with a message “Device added successfully, but failed to connect”. In bluetoothd this Bluez/PulseAudio incompatibility is likely to yield the following error:

```
bluetoothd[1]: a2dp-sink profile connect failed for <Audio Device MAC Address>: Protocol not available
```

If this occurs, to resolve it we must also run an instance of PulseAudio daemon in a container, as described in this section.

The PulseAudio daemon [Dockerfile](#) may be found in the [bluetoothd](#) directory, noting that it has the name Dockerfile-pulseaudio to distinguish it from the bluetoothd Dockerfile.

```
FROM debian:stretch-slim

# Install pulseaudio
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        pulseaudio pulseaudio-module-bluetooth \
        pulseaudio-utils && \
    rm -rf /var/lib/apt/lists/* && \
    sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
        /etc/pulse/daemon.conf && \
    sed -i "s/#load-module module-native-protocol-tcp/load-module \
        module-native-protocol-tcp auth-ip-acl=127.0.0.1\:172.17.0.0\:1 \
        6\:192.168.0.0\:16 port=4714/g" /etc/pulse/default.pa && \
    sed -i "s/load-module module-console-kit/#load-module module- \
        console-kit/g" /etc/pulse/default.pa && \
    # Add user "pa" so we can run pulseaudio as a
    # non privileged user below.
    groupadd -r -g 1000 pa && \
    useradd -u 1000 -r -g pa -G audio pa && \
    mkdir /home/pa/ && \
    chown -R pa:pa /home/pa

COPY ./set-a2dp-sink.sh /src/set-a2dp-sink.sh

# Run pulseaudio daemon as the non privileged user pa.
USER pa
CMD pulseaudio -v 2>&1 | /src/set-a2dp-sink.sh
```

The pulseaudio image may be built with:

```
docker build -t pulseaudio -f Dockerfile-pulseaudio .
```

The Dockerfile has a few lines that require some additional explanation, with help from the PulseAudio documentation:
<https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation>

.

The following line modifies /etc/pulse/daemon.conf to change the value of **exit-idle-time** to -1, which keeps the PulseAudio daemon alive until the container exits. Without this change the daemon annoyingly exits after only 20 seconds of inactivity.

```
sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
/etc/pulse/daemon.conf
```

The following line modifies /etc/pulse/default.pa to enable **module-native-protocol-tcp**. The module is configured with Access Control List authentication allowing connections from localhost, the Docker subnet and the private IP address range allowing access from the local network, listening on port 4714 (the default PulseAudio TCP port is 4713).

```
sed -i "s/#load-module module-native-protocol-tcp/load-module m
odule-native-protocol-tcp auth-ip-acl=127.0.0.1\:172.17.0.0\:16
\:192.168.0.0\:16 port=4714/g" /etc/pulse/default.pa
```

The following line modifies /etc/pulse/default.pa to disable loading **module-console-kit**, as leaving that module enabled can cause D-bus errors on some Linux distributions.

```
sed -i "s/load-module module-console-kit/#load-module module-co
nsole-kit/g" /etc/pulse/default.pa
```

The following lines create a non-privileged user and group called “pa” so that we can run the PulseAudio daemon as a non-root user. We also create a “home directory” for the pa user as this is where PulseAudio will attempt to store various configuration files when it starts. Note that unlike most of our GUI applications we don’t need (or want) this unprivileged user to map to the user running the container.

```
groupadd -r -g 1000 pa
useradd -u 1000 -r -g pa -G audio pa
mkdir /home/pa/
chown -R pa:pa /home/pa
```

We next copy the script `set-a2dp-sink.sh`, the purpose of which shall be explained later, into the image as follows:

```
COPY ./set-a2dp-sink.sh /src/set-a2dp-sink.sh
```

Next we make the PulseAudio daemon run as the non-privileged user pa.

```
USER pa
CMD pulseaudio -v 2>&1 | /src/set-a2dp-sink.sh
```

This command runs the PulseAudio daemon in verbose mode `-v`, redirecting its stderr to stdout with `2>&1` and piping that output to the stdin of our `set-a2dp-sink.sh` script, which will parse and process the logs from the PulseAudio daemon.

The `set-a2dp-sink.sh` script used in the Dockerfile is as follows:

```
while IFS= read -r line; do
    if [[ $line = *[pulseaudio] card.c: Created* ]]; then
        NAME=$(echo "${line}" | grep -o -E "bluez_card.([[:xdigit:]]{2}_){5}([[:xdigit:]]{2})")
        echo "pacmd set-card-profile ${NAME} a2dp_sink"
        pacmd set-card-profile ${NAME} a2dp_sink
    fi
done
```

This script receives the verbose output from the PulseAudio daemon, parsing each line from the log as it occurs. When a Bluetooth A2DP sink connects, PulseAudio generates a log line containing the text

`[pulseaudio] card.c: Created .`

If we see that string in the PulseAudio log we then regex extract the bluez_card name that is also present in that log line, which can be used in the subsequent `pacmd set-card-profile` call to set the profile to `a2dp_sink`.

The reason that we need this script in the first place is because Bluetooth audio supports multiple profiles, but PulseAudio defaults to using the low-quality Headset Profile (HSP). In our application we detect the connection of a high-quality `a2dp_sink` device which represents the Advanced Audio Distribution Profile (A2DP) and use `pacmd` to set the card profile to that.

The [pulseaudio.sh](#) launch script is as follows:

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# Add flags for connecting to the D-bus system bus.
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sys-
tem_bus_socket:ro "$DBUS_FLAGS

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Unload module-bluetooth-discover from host, if present.
BLUETOOTH_DISCOVER=$(pactl list | grep module-bluetooth-disco-
ve
r)
if [ "${BLUETOOTH_DISCOVER}" != "" ]; then
    echo "pactl unload-module module-bluetooth-discover"
    pactl unload-module module-bluetooth-discover
fi

$DOCKER_COMMAND run --rm -it \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -p 4714:4714 \
    pulseaudio | ./create-tunnel-sink.sh

if [ "${BLUETOOTH_DISCOVER}" != "" ]; then
    echo "pactl load-module module-bluetooth-discover"
    pactl load-module module-bluetooth-discover
fi
```

This script mostly follows a familiar pattern of connecting to the D-bus system bus. One point of note is the test for **module-bluetooth-discover**, which is the module used on the host for establishing Bluetooth audio connections. As the host and the container cannot both establish a connection to the Bluetooth device simultaneously this test allows the script to disable discovery on the host and re-enable it on exit.

The strategy that we employ here is to run our containerised PulseAudio daemon *in addition* to the native PulseAudio daemon running on the host. The daemon in the container will be listening on a TCP socket on port 4714 exported via `-p 4714:4714` and when an A2DP connection is made we will create a tunnel to that daemon from the host's daemon via PulseAudio **module-tunnel-sink**.

The mechanism for establishing the tunnel is driven by the stdout of the pulseaudio container, which echoes the command used to set the card profile.

```
echo "pacmd set-card-profile ${NAME} a2dp_sink"
```

The stdout of the pulseaudio container is piped to the stdin of the script `create-tunnel-sink.sh` which is responsible for establishing the tunnel when required.

```
pulseaudio | ./create-tunnel-sink.sh
```

The [create-tunnel-sink.sh](#) script used in pulseaudio.sh is as follows:

```
while IFS= read -r line; do
    echo ${line}
    if [[ $line = *"a2dp_sink"* ]]; then
        sleep 1
        echo "Connecting module-tunnel-sink to localhost:4714"
        echo -e "\r"
        pactl load-module module-tunnel-sink server=localhost:4714
        sink_name=bluetooth > /dev/null
    fi
done
```

This script is somewhat similar to `set-a2dp-sink.sh` and parses each line of its stdin waiting for a line containing the string “a2dp_sink”, which occurs when the pulseaudio container logs that it has set the card profile to a2dp_sink. When this occurs the tunnel is established on the host via a call to:

```
pactl load-module module-tunnel-sink server=localhost:4714 sink
_name=bluetooth > /dev/null
```

On hosts running SELinux, such as CentOS or Fedora, our pulseaudio container is likely to see a similar complication to that which we encountered with bluetoothd, where the default security policies might block important actions needed by the container.

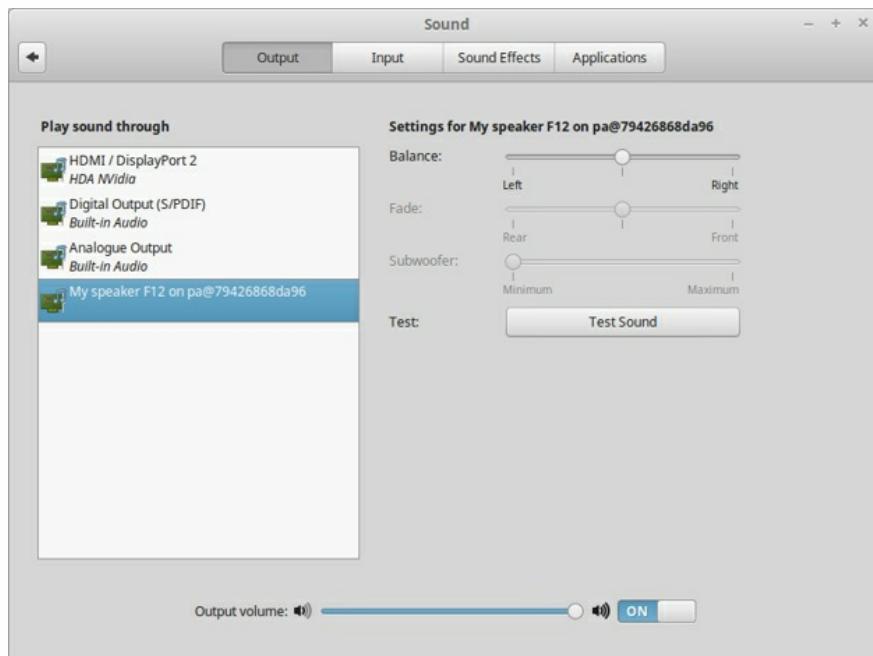
One such block observed by the author is that the dbus-daemon is denied read/write access on the socket “socket” by SELinux, which generates an error similar to the following when running pulseaudio:

```
dbus.exceptions.DBusException: org.freedesktop.DBus.Error.NoReply: Did not receive a reply. Possible causes include: the remote application did not send a reply, the message bus security policy blocked the reply, the reply timeout expired, or the network connection was broken.
```

Using information obtained from the SELinux audit logs this may be unblocked by running the following commands:

```
sudo ausearch -c 'dbus-daemon' --raw | audit2allow -M my-dbusdaemon  
sudo semodule -i my-dbusdaemon.pp
```

When the pulseaudio container is running correctly and an A2DP connection has been established, the tunnel sink audio device should be available for selection like any other audio device in the sound settings of the host.



4-8 Wayland and Weston

We have thus far only demonstrated X11 applications in this book, however we have previously briefly touched upon [Wayland and Weston](#) noting that Wayland is intended to be a simpler replacement for X.

Although Wayland might well be the future for running GUI applications on Linux, a number of potential obstacles remain before it can be considered mainstream.

The first obstacle is that Wayland uses uid to authenticate, which is fine for most applications however several “admin” applications require elevated privileges and won’t run under Wayland. From a security perspective Wayland’s approach is entirely correct and those applications should really be rewritten to separate the display from those parts requiring elevated privileges, but until this is done several important applications require workarounds limiting Wayland-native desktops to more experienced users.

The second, more serious, obstacle is that Wayland doesn’t yet work well with the Nvidia proprietary drivers. There is a work-in-progress implementation of an EGL External Platform library to add client-side Wayland support to EGL on top of the EGLDevice and EGLStream families of extensions <https://github.com/NVIDIA/egl-wayland>, but this is not yet supported by any Linux distribution, and Wayland compositors currently require extensive patching to use these extensions.

The third issue with Wayland is that only a small subset of applications currently support Wayland natively, with the majority of applications using the XWayland X11 fall-back.

4-8-1 gnome-calculator

All GNOME applications do support Wayland natively however, so we return to our [GNOME Calculator](#) application to illustrate native Wayland support in Docker.

The [Dockerfile](#) for our containerised GNOME Calculator is in the [gnome-calculator](#) directory of [docker-gui](#) under [4-local-applications/wayland-and-weston](#).

```
FROM debian:stretch-slim

# Install gnome-calculator
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    ca-certificates libcanberra-gtk3-module \
    packagekit-gtk3-module gnome-calculator && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["gnome-calculator"]
```

This is exactly the same as the Dockerfile that we developed in our earlier X11 [gnome-calculator](#) section, so if that image has already been built it may be reused, alternatively it may be built in the [gnome-calculator](#) directory with:

```
docker build -t gnome-calculator .
```

To run Wayland applications natively without X, it is necessary to have a running Wayland compositor such as GNOME Wayland or Weston. If a native Wayland compositor is not available never fear, as we shall create a Dockerised Weston in the next section supporting both a Wayland and X11 back-end.

The key to running native Wayland applications in Docker is sharing the Wayland Unix domain socket with the container, in a similar way to how we previously shared the X11 socket.

The Wayland socket is typically called something like `wayland-0` and is found in `XDG_RUNTIME_DIR` which is usually `/run/user/1000`, where the last number is the user's uid. The name of the socket is stored in the environment variable `WAYLAND_DISPLAY`. Note that both `XDG_RUNTIME_DIR` and `WAYLAND_DISPLAY` are required to be set by native Wayland applications.

As `XDG_RUNTIME_DIR` only allows access for its owner, the container needs to be running with the same uid as the compositor on the host.

The complete [`gnome-calculator-wayland.sh`](#) launch script is as follows:

```
WAYLAND_DISPLAY=${WAYLAND_DISPLAY:-wayland-0}

# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi
```

```

if [[ $DBUS_SESSION_BUS_ADDRESS == *"abstract"* ]]; then
    DBUS_FLAGS="--net=host"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm \
    $APPARMOR_FLAGS \
    $DBUS_FLAGS \
    -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS \
    -v $HOME/.config/dconf/user:$HOME/.config/dconf/user:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e XDG_RUNTIME_DIR=/tmp \
    -e WAYLAND_DISPLAY=$WAYLAND_DISPLAY \
    -v $XDG_RUNTIME_DIR/$WAYLAND_DISPLAY:/tmp/$WAYLAND_DISPLAY \
    gnome-calculator @@

```

The key points of note for connecting to Wayland are the following lines from the `docker run` call:

```

-e XDG_RUNTIME_DIR=/tmp \
-e WAYLAND_DISPLAY=$WAYLAND_DISPLAY \
-v $XDG_RUNTIME_DIR/$WAYLAND_DISPLAY:/tmp/$WAYLAND_DISPLAY \

```

4-8-2 weston

Weston is a reference implementation of a Wayland compositor also developed by the Wayland project. Weston has a number of back-ends that enable it to run directly on Linux KMS, another Wayland server, or on X11. Weston also supports X Clients via XWayland, which provides backwards compatibility to X applications in a Wayland stack.

XWayland is activated by instructing Weston to load the XWayland module, e.g. `weston --xwayland` or `weston --modules=xwayland.so`. Weston starts listening on a new X display socket, and exports it in the environment variable DISPLAY. When the first X client connects, Weston launches a special X server as a Wayland client to handle the X client and all future X clients.

The [Dockerfile](#) is in the [weston](#) directory of [docker-gui](#) under [4-local-applications/wayland-and-weston](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    weston xwayland dmz-cursor-theme \
    libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/*

CMD weston --modules=xwayland.so
```

Note that `debian:stretch` ships with Weston 1.12.0, which is a relatively old version of Weston.

The weston image may be built with

```
docker build -t weston .
```

and the launch script [weston.sh](#) will launch Weston with an X11 back-end.

```
DOCKER_COMMAND=docker
DST=/usr/lib/x86_64-linux-gnu
if test -c "/dev/nvidia-modeset"; then
    GPU_FLAGS="--device=/dev/nvidia-modeset "
    if test -f "/usr/bin/nvidia-container-runtime"; then
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep
/lib/)
        GPU_FLAGS+="--runtime=nvidia "
        GPU_FLAGS+=" -e NVIDIA_VISIBLE_DEVICES=all "
        GPU_FLAGS+=" -e NVIDIA_DRIVER_CAPABILITIES=graphics "
        GPU_FLAGS+=" -v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
        GPU_FLAGS+=" -v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
        GPU_FLAGS+=" -v $SRC/libGDispatch.so.0:$DST/libGDispatch.s
o.0:ro "
        GPU_FLAGS+=" -v $SRC/libEGL.so.1:$DST/libEGL.so.1:ro "
        GPU_FLAGS+=" -v $SRC/libGLESv1_CM.so.1:$DST/libGLESv1_CM.so.
1:ro "
        GPU_FLAGS+=" -v $SRC/libGLESv2.so.2:$DST/libGLESv2.so.2:ro "
    else # Nvidia Docker Version 1
        DOCKER_COMMAND=nvidia-docker
        SRC=/usr/local/nvidia
        GPU_FLAGS="-e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
    fi
else
    if test -d "/var/lib/VBoxGuestAdditions"; then
        GPU_FLAGS="--device=/dev/vboxuser "
        GPU_FLAGS+=" -v /var/lib/VBoxGuestAdditions/lib/libGL.so.1:$
DST/libGL.so.1 "
```

```

for f in $DST/VBox*.so $DST/libXcomposite.so.1
do
    GPU_FLAGS+=" -v $f:$f "
done
else
    # Open Source Mesa GPU.
    GPU_FLAGS="--device=/dev/dri "
    GPU_FLAGS+=" --group-add $(cut -d: -f3 < <(getent group video))"
fi
fi

# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

# Create .Xauthority.docker file with wildcarded hostname.
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -

```

\$DOCKER_COMMAND run --rm \
--ipc=host \
-u \$(id -u):\$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=unix\$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-v /tmp/.X0-lock:/tmp/.X0-lock \
-e XAUTHORITY=\$DOCKER_XAUTHORITY \
-v \$DOCKER_XAUTHORITY:\$DOCKER_XAUTHORITY:ro \
-e XDG_RUNTIME_DIR=/tmp \
-v \$XDG_RUNTIME_DIR:/tmp \
\$GPU_FLAGS \
weston

The first part of the script sets the GPU_FLAGS in a similar way to our previous [3D Accelerated Applications](#), with the notable addition of the following lines adding EGL/GLES:

```
GPU_FLAGS+=" -v $SRC/libEGL.so.1:$DST/libEGL.so.1:ro "
GPU_FLAGS+=" -v $SRC/libGLESv1_CM.so.1:$DST/libGLESv1_CM.so.1:ro "
GPU_FLAGS+=" -v $SRC/libGLESv2.so.2:$DST/libGLESv2.so.2:ro "
```

as EGL support is required to run Weston. Note that this means that the host requires EGL in order to run Weston but it *does not*, unfortunately, mean that Weston necessarily supports running OpenGL/EGL applications on an Nvidia based system, although it does on a mesa based system.

The remainder of the script follows a similar pattern to many of our other X11 applications, though note the line:

```
-v /tmp/.X0-lock:/tmp/.X0-lock \
```

This is needed by XWayland and without it XWayland will attempt to start its X Server on display unix:0 and will likely fail as that is most likely the host's display.

The following lines set up XDG_RUNTIME_DIR in the container (which is needed by Weston) and bind-mounts the host's XDG_RUNTIME_DIR read/write. This will allow the container to create the Wayland socket `wayland-0` and place it in the host's XDG_RUNTIME_DIR.

```
-e XDG_RUNTIME_DIR=/tmp \
-v $XDG_RUNTIME_DIR:/tmp \
```

The [weston.sh](#) script will run on both X11 based hosts and also on hosts running Wayland natively, as those will also be running XWayland and so our script will connect to the XWayland X Server in that case.

Alternatively, we can use the [weston-wayland.sh](#) launch script as follows to launch Weston with a native Wayland back-end.

```
WAYLAND_DISPLAY=${WAYLAND_DISPLAY:-wayland-0}

DOCKER_COMMAND=docker
DST=/usr/lib/x86_64-linux-gnu
if test -c "/dev/nvidia-modeset"; then
    GPU_FLAGS="--device=/dev/nvidia-modeset "
    if test -f "/usr/bin/nvidia-container-runtime"; then
        # Nvidia Docker Version 2
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep
/lib/)
        GPU_FLAGS+="--runtime=nvidia "
        GPU_FLAGS+="--e NVIDIA_VISIBLE_DEVICES=all "
        GPU_FLAGS+="--e NVIDIA_DRIVER_CAPABILITIES=graphics "
        GPU_FLAGS+="--v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
        GPU_FLAGS+="--v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
        GPU_FLAGS+="--v $SRC/libGLdispatch.so.0:$DST/libGLdispatch.s
0.0:ro "
        GPU_FLAGS+="--v $SRC/libEGL.so.1:$DST/libEGL.so.1:ro "
        GPU_FLAGS+="--v $SRC/libGLESv1_CM.so.1:$DST/libGLESv1_CM.so.
1:ro "
        GPU_FLAGS+="--v $SRC/libGLESv2.so.2:$DST/libGLESv2.so.2:ro "
    else
        # Nvidia Docker Version 1
        DOCKER_COMMAND=nvidia-docker
        SRC=/usr/local/nvidia
        GPU_FLAGS="--e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
    fi
fi
```

```

else # Non-Nvidia GPU path
    if test -d "/var/lib/VBoxGuestAdditions"; then
        # VirtualBox GPU
        GPU_FLAGS="--device=/dev/vboxuser "
        GPU_FLAGS+=" -v /var/lib/VBoxGuestAdditions/lib/libGL.so.1:$
DST/libGL.so.1 "
        for f in $DST/VBox*.so $DST/libXcomposite.so.1
        do
            GPU_FLAGS+="-v $f:$f "
        done
    else
        # Open Source Mesa GPU.
        GPU_FLAGS="--device=/dev/dri "
        GPU_FLAGS+=" --group-add $(cut -d: -f3 < <(getent group vide
o)) "
    fi
fi

# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

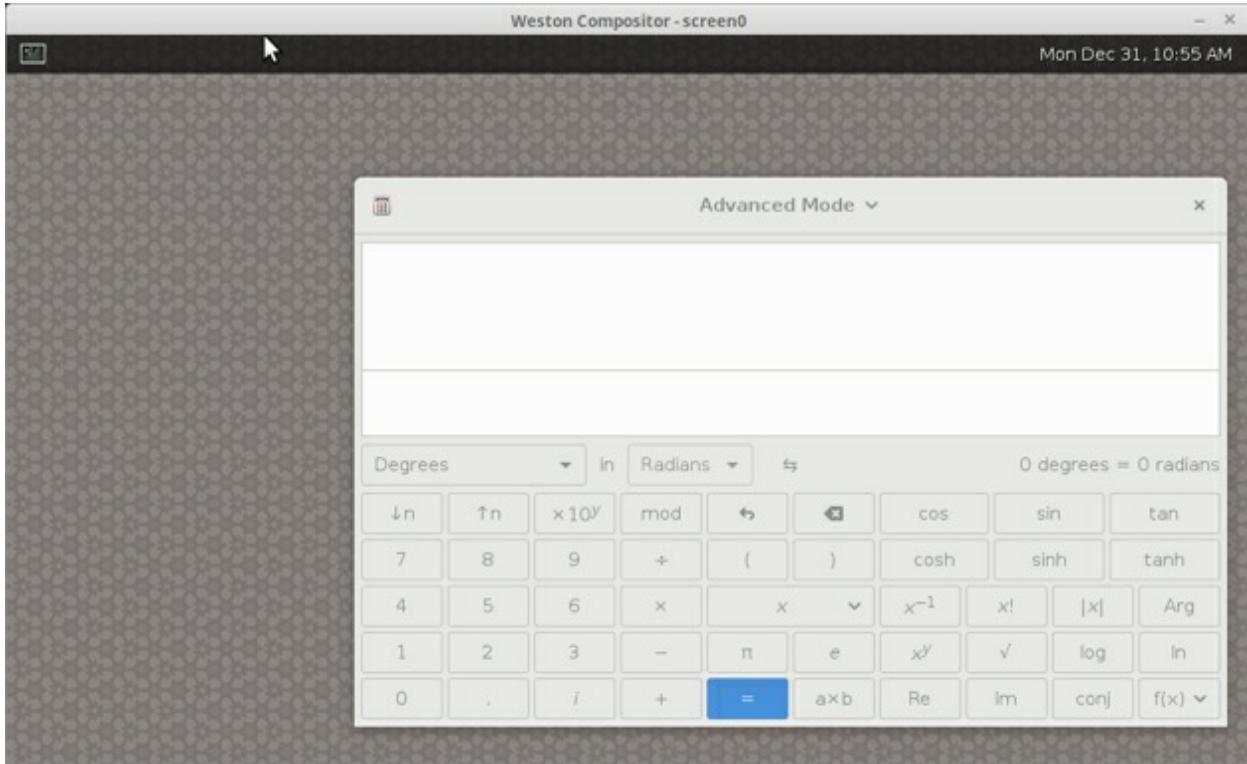
$DOCKER_COMMAND run --rm \
    --ipc=host \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    -v /tmp/.X0-lock:/tmp/.X0-lock \
    -e XDG_RUNTIME_DIR=/tmp \
    -e WAYLAND_DISPLAY=$WAYLAND_DISPLAY \
    -v $XDG_RUNTIME_DIR:/tmp\
$GPU_FLAGS \
weston

```

In this case we pass the host's WAYLAND_DISPLAY and remove the xauth code, but still need to bind-mount the X11 socket directory and lock file as they are used by XWayland.

A subtle point to note when running `weston-wayland.sh` is that it will result in a *nested* Wayland compositor being run on a Wayland native host, this means that when running Wayland Clients we need to target the *correct display* e.g.

```
WAYLAND_DISPLAY=wayland-1 ./gnome-calculator-wayland.sh
```

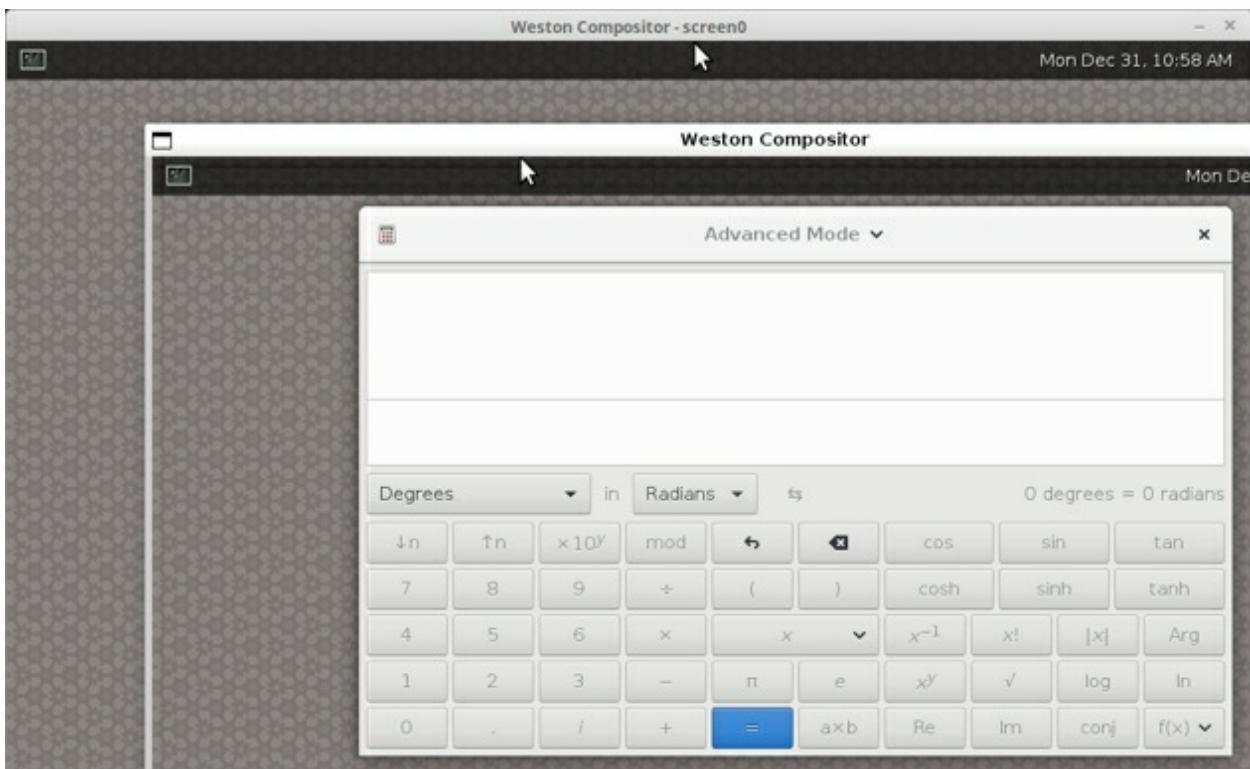


Running Weston with XWayland can be useful for a number of use cases, as it provides both a Wayland compositor and an X Server that are somewhat isolated from the host's display.

This can be advantageous when running less trusted GUI applications as they are unable to interact with the host's X Server to do nefarious things such as key logging, indeed preventing such interaction between windows is one benefit of running Wayland even without nesting.

Because of the diverse range of back-ends It is actually possible to nest Weston quite arbitrarily and we could run another Weston window directly on our first Weston using its Wayland back-end, or we could even run Weston on XWayland using the second Weston's X11 back-end.

The example below illustrates the first Weston window running on X11, with a second Weston window running on Wayland and a Wayland gnome-calculator targeting the second Weston window.



5 More Complex Applications

After completing the [Local Applications](#) chapter readers of this book should have developed a sound understanding of all the configuration settings needed to run sophisticated multimedia applications in Docker containers.

Whilst the [Local Applications](#) chapter attempted to cover each key topic individually to avoid overwhelming the reader with too many new concepts, the purpose of this chapter is to consolidate that knowledge by illustrating more complex and realistic applications that might combine many of the issues previously encountered within a single application.

The examples in this chapter have been chosen to illustrate the broadest range of multimedia applications, from browsers through to games and illustrate how Docker can help enable things that are often rather difficult to achieve in a Linux environment such as playing Blu-ray discs and Flash videos, which many sites still insist on using but often in a way that is broken on the old Flash player normally available for Linux.

This chapter also covers some sophisticated graphics applications that combine 3D accelerated OpenGL rendering with GPU accelerated compute using OpenCL or CUDA. We illustrate how Docker can be invaluable for compute accelerated applications as it provides a readily reproducible way of provisioning runtime environments that can often be quite fiddly to set up consistently and are even harder to upgrade to a later toolkit release.

5-1 Modular Scripting

The examples in the [Local Applications](#) chapter were intended to be self-contained as an aid to understanding, however that approach is not sustainable and would lead to a lot of repetition and copy and paste reuse if we were to follow it more generally.

To simplify the remainder of this book and allow us to focus more on details that have not been covered previously, in this section we shall present a modular set of scripts that may be selectively composed by applications as particular features are required.

These scripts may be found in the [docker-gui/bin](#) directory and are intended to be used by “sourcing” them into the main application script. For example, the following code snippet would enable the docker-xauth and docker-gpu script modules in the application script that was loading them.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-gpu.sh
```

The first line above looks complicated, but it is simply using bash command substitution and shortest ending match string processing to find the parent docker-gui directory path of the running script and then sets the BIN variable to the <path>/docker-gui/bin directory.

[docker-command.sh](#) populates DOCKER_COMMAND with docker or sudo docker depending whether the user running the script is in the “docker” group or not. This was first developed in [the docker Group and sudo](#) and is used as we wish to avoid running the whole application script as sudo.

```
DOCKER_COMMAND=docker
# If user isn't in docker group prefix docker with sudo
if ! (id -nG $(id -un) | grep -qw docker); then
    DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi
```

[docker-xauth.sh](#) is a ubiquitous script developed in [x11-apps](#), which first creates an additional .Xauthority file with a wildcard hostname then populates the X11_FLAGS variable as a short-cut instead of having to set the environment and volume flags individually in the docker run command.

```
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY.%%
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY.%% nmerge -
mv $DOCKER_XAUTHORITY.%% $DOCKER_XAUTHORITY

X11_XAUTH="-e XAUTHORITY=$DOCKER_XAUTHORITY "
X11_XAUTH+=" -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro "

X11_DISPLAY="-e DISPLAY=unix$DISPLAY "
X11_DISPLAY+=" -v /tmp/.X11-unix:/tmp/.X11-unix:ro "

X11_DISPLAY_RW="-e DISPLAY=unix$DISPLAY "
X11_DISPLAY_RW+=" -v /tmp/.X11-unix:/tmp/.X11-unix:rw "

X11_FLAGS=$X11_XAUTH$X11_DISPLAY
X11_FLAGS_RW=$X11_XAUTH$X11_DISPLAY_RW
```

[docker-wayland.sh](#) was developed in [Wayland and Weston](#) and sets environment and volume flags serving the same purpose for native Wayland Clients as docker-xauth.sh does for X11.

```
WAYLAND_DISPLAY=${WAYLAND_DISPLAY:-wayland-0}
WAYLAND_FLAGS="-e XDG_RUNTIME_DIR=/tmp "
WAYLAND_FLAGS+=" -e WAYLAND_DISPLAY=$WAYLAND_DISPLAY "
WAYLAND_FLAGS+=" -v $XDG_RUNTIME_DIR/$WAYLAND_DISPLAY:/tmp/$WAYLAND_DISPLAY "
```

[docker-dbus-all.sh](#) was developed in [GNOME, D-bus and AppArmor](#) and sets the [dconf](#) and [AppArmor](#) environment and volume flags that enable applications to connect to D-bus.

```
if [[ $DBUS_SESSION_BUS_ADDRESS == **"abstract"** ]]; then
    DBUS_FLAGS="--network=host -e NO_AT_BRIDGE=1"
else
    DBUS_FLAGS="-v $XDG_RUNTIME_DIR/bus:$XDG_RUNTIME_DIR/bus:ro
-e NO_AT_BRIDGE=1"
fi

DBUS_FLAGS+=" -v /var/run/dbus/system_bus_socket:/var/run/dbus/
system_bus_socket:ro "
DBUS_FLAGS+=" -e DBUS_SESSION_BUS_ADDRESS=$DBUS_SESSION_BUS_ADDRESS "

if test -f "/etc/apparmor.d/docker-dbus"; then
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"
else
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"
fi

DCONF_FLAGS=$DBUS_FLAGS
DCONF_FLAGS+=" -v $HOME/.config/dconf/user:$HOME/.config/dconf/u
ser:ro "
```

[docker-dbus-system-bus.sh](#) may be used in place of docker-dbus-all.sh for cases where only the D-bus system bus is required and not the session bus.

```
# Add flags for connecting to the D-bus system bus.  
DBUS_FLAGS="-v /var/run/dbus/system_bus_socket:/var/run/dbus/sys  
tem_bus_socket:ro \"$DBUS_FLAGS  
  
if test -f "/etc/apparmor.d/docker-dbus"; then  
    APPARMOR_FLAGS="--security-opt apparmor:docker-dbus"  
else  
    APPARMOR_FLAGS="--security-opt apparmor=unconfined"  
fi
```

[docker-gpu.sh](#) was developed in [3D Accelerated Applications](#) setting environment, volume and device flags enabling OpenGL and OpenCL acceleration on a variety of GPUs. On Nvidia GPUs CUDA compute and VDPAU video acceleration are also enabled. Note that the path to the Nvidia libraries set in the SRC variable in the script is correct for Ubuntu derived hosts, but it may be different in other distributions, in which case the SRC path would need to be modified.

```
DOCKER_COMMAND=docker  
DST=/usr/lib/x86_64-linux-gnu  
if test -c "/dev/nvidia-modeset"; then # Nvidia GPU  
    GPU_FLAGS="--device=/dev/nvidia-modeset "  
    if test -f "/usr/bin/nvidia-container-runtime"; then  
        # Nvidia Docker Version 2  
        # Attempt to find the actual Nvidia library path.  
        # It should be something like  
        # /usr/lib/nvidia-<driver version>  
        SRC=$(cat /etc/ld.so.conf.d/x86_64-linux-gnu_GL.conf | grep  
/lib/)  
        GPU_FLAGS+="--runtime=nvidia "
```

```

GPU_FLAGS+=" -e NVIDIA_VISIBLE_DEVICES=all "
GPU_FLAGS+=" -e NVIDIA_DRIVER_CAPABILITIES=all "
GPU_FLAGS+=" -v $SRC/libGL.so.1:$DST/libGL.so.1:ro "
GPU_FLAGS+=" -v $SRC/libGLX.so.0:$DST/libGLX.so.0:ro "
GPU_FLAGS+=" -v $SRC/libGLdispatch.so.0:$DST/libGLdispatch.s
0.0:ro "
GPU_FLAGS+=" -v $SRC/libEGL.so.1:$DST/libEGL.so.1:ro "
GPU_FLAGS+=" -v $SRC/libGLESv1_CM.so.1:$DST/libGLESv1_CM.so.
1:ro "
GPU_FLAGS+=" -v $SRC/libGLESv2.so.2:$DST/libGLESv2.so.2:ro "
GPU_FLAGS+=" -v $SRC/vdpau/libvdpau_nvidia.so:$DST/libvdpau_
nvidia.so:ro "
else # Nvidia Docker Version 1
DOCKER_COMMAND=nvidia-docker
SRC=/usr/local/nvidia
GPU_FLAGS+=" -e LD_LIBRARY_PATH=$SRC/lib:$SRC/lib64:${LD_LIB
RARY_PATH} "
fi
else # Non-Nvidia GPU path
if test -d "/var/lib/VBoxGuestAdditions"; then
GPU_FLAGS="--device=/dev/vboxuser "
GPU_FLAGS+=" -v /var/lib/VBoxGuestAdditions/lib/libGL.so.1:$
DST/libGL.so.1 "
for f in $DST/VBox*.so $DST/libXcomposite.so.1
do
GPU_FLAGS+=" -v $f:$f "
done
else # Open Source Mesa GPU.
GPU_FLAGS="--device=/dev/dri "
GPU_FLAGS+=" --group-add $(cut -d: -f3 < <(getent group vide
o)) "
fi
fi

if ! (id -nG $(id -un) | grep -qw docker); then
DOCKER_COMMAND="sudo $DOCKER_COMMAND"
fi

```

[docker-webcam.sh](#) is a simple script developed in [Webcam Support](#) that enumerates the available /dev/video* V4L devices and adds them to WEBCAM_FLAGS. This helps mitigate the issue where unplugging and replugging a USB webcam may cause it to be allocated to a new device.

```
WEBCAM_FLAGS="--group-add $(cut -d: -f3 < <(getent group video)
)
for f in /dev/video*
do
    WEBCAM_FLAGS+="--device=$f "
done
```

[docker-pulseaudio.sh](#) was developed in [PulseAudio](#) and sets environment and volume flags that enable applications to connect to the PulseAudio daemon. This script also disables shared memory for PulseAudio versions 7 to 9, where the mechanism for cleaning up shm files was overly aggressive.

```
PULSE_VERSION=$(pulseaudio --version | sed 's/[^\0-9.]*/([0-9]*\.
).*'\1/')
if ([[ $PULSE_VERSION -gt 6 ]] && [[ $PULSE_VERSION -lt 10 ]]);
then
    PULSE_FLAGS="-e PULSE_CLIENTCONFIG=/etc/pulse/client-noshm.
conf"
fi

# Populate the PULSEAUDIO_FLAGS variable as a short cut
# instead of having to set the flags individually.
PULSEAUDIO_FLAGS="-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/n
ative "
PULSEAUDIO_FLAGS+=" -v $XDG_RUNTIME_DIR/pulse:$XDG_RUNTIME_DIR/p
ulse:ro "
PULSEAUDIO_FLAGS+=" -v $HOME/.config/pulse/cookie:$HOME/.config/
pulse/cookie:ro "
PULSEAUDIO_FLAGS+="$PULSE_FLAGS "
```

[docker-jack.sh](#) was developed in the [JACK](#) chapter. This script suspends PulseAudio and enables realtime priority and memory [page-locking](#) in the container. The script also provides access to the host's IPC, which is used to enable communication between jackd and clients. For jackd the script enables container access to the host's ALSA sound device and adds the audio group to the set of groups that the container can run as.

```
# Check is pasuspender (and therefore pulseaudio) is
# present, if so then prefix with pasuspender to suspend
# pulseaudio for the duration of the test.
if test -f /usr/bin/pasuspender; then
    DOCKER_COMMAND="pasuspender -- \"$DOCKER_COMMAND"
fi

# Populate the JACK_FLAGS and JACKD_FLAGS variable as a
# short cut instead of having to set the environment and
# volume flags individually.
JACK_FLAGS="--ulimit rtprio=99 "
JACK_FLAGS+="--ulimit memlock=-1 "
JACK_FLAGS+="--ipc=host "

# For jackd we pass /dev/snd and add audio group to give
# access to ALSA
JACKD_FLAGS=$JACK_FLAGS
JACKD_FLAGS+="--device=/dev/snd "
JACKD_FLAGS+="--group-add $(cut -d: -f3 < <(getent group audio)
) "
```

5-2 Browsers

As use of the Internet has exploded over recent years web browsers have become probably *the* most indispensable and ubiquitous part of any modern desktop environment.

Although browsers began life in academia as simple text based tools to fetch hypertext information resources, they have evolved rapidly. Browsers have now become fully featured multimedia platforms that are capable of acting as full-blown execution environments, with the capabilities to run hardware accelerated video and even high-performance games using [WebAssembly](#) with [WebGL](#) enabling accelerated 3D graphics.

This section shall explore how to run a range of popular web browsers in Docker containers, providing access to the full range of features necessary to give a user experience comparable to running the browser natively on the desktop.

Important note on security and privacy . As mentioned above the principal aim of this section is to illustrate how to enable the key browser features that will provide the user with as rich a multimedia experience as running the browser natively, which inevitably comes at the cost of losing some of the powerful sandboxing capabilities available with Docker.

We shall touch upon some ways to improve application isolation when we look at running Flash Player, but for more complete application sandboxing we must employ many of the techniques used for running remote applications that will be described later in the book.

5-2-1 firefox

The [Dockerfile](#) for our containerised version of Firefox is in the [firefox](#) directory of [docker-gui](#) under [5-more-complex-applications/browsers](#).

```
FROM debian:stretch-slim
ENV FIREFOX_LANG=en-GB

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    wget bzip2 ca-certificates gnupg dirmngr procps \
    fonts-symbola fonts-lmodern fonts-freefont-ttf \
    fonts-liberation fonts-dejavu gsffonts \
    libgtk-3-0 libgtk2.0-0 libnss3 libxt6 libavcodec57 \
    libvpx4 libdbus-glib-1-2 libcanberra-gtk3-module \
    libpulse0 libv4l-0 libgl1-mesa-glx libgl1-mesa-dri && \
    # Debian ships with esr, so install latest Firefox.
    wget -qO - "https://download.mozilla.org/?product=firefox-latest&os=linux64&lang=${FIREFOX_LANG}" | tar -xjv -C /usr/lib &&
\
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf
ENTRYPOINT ["/usr/lib/firefox/firefox", "--no-remote"]
```

and may be built with:

```
docker build -t firefox .
```

This Dockerfile is somewhat more complicated than those developed in the [Local Applications](#) chapter, but it is nevertheless relatively straightforward when each part is considered separately.

The first part comprises an apt-get install block that installs the main package dependencies. For the majority of our previous simple examples many libraries were automatically installed by the apt package manager dependency tracking, but as we are manually installing Firefox in this Dockerfile we need to install the required packages more explicitly.

The next part of the Dockerfile installs Firefox. Debian ships with the Extended Support Release of Firefox (the firefox-esr package), so if we instead wish to install the latest release we must do so manually from the Firefox download web page, here using the FIREFOX_LANG environment variable to specify the preferred language (en-GB in this case):

```
wget -qO - "https://download.mozilla.org/?product=firefox-lates  
t&os=linux64&lang=${FIREFOX_LANG}" | tar -xjv -C /usr/lib
```

After downloading, the above command installs Firefox by unpacking the archive into the required directory via the tar command.

We next copy the fonts.conf file from the Docker build context directory to /etc/fonts/local.conf:

```
COPY fonts.conf /etc/fonts/local.conf
```

The font configuration that we use is shown below and is described more fully in this blog post on improving Debian font rendering
<http://blog.programster.org/debian-8-gnome-desktop-improve-font-rendering/>

```
<?xml version='1.0'?>
<!DOCTYPE fontconfig SYSTEM 'fonts.dtd'>
<fontconfig>
    <match target="font">
        <edit mode="assign" name="rgba">
            <const>rgb</const>
        </edit>
    </match>
    <match target="font">
        <edit mode="assign" name="hinting">
            <bool>true</bool>
        </edit>
    </match>
    <match target="font">
        <edit mode="assign" name="hintstyle">
            <const>hintslight</const>
        </edit>
    </match>
    <match target="font">
        <edit mode="assign" name="antialias">
            <bool>true</bool>
        </edit>
    </match>
    <match target="font">
        <edit mode="assign" name="lcdfilter">
            <const>lcddefault</const>
        </edit>
    </match>
</fontconfig>
```

Finally, the container entrypoint is set as follows:

```
ENTRYPOINT ["/usr/lib/firefox/firefox", "--no-remote"]
```

The `--no-remote` part is important as it explicitly forces Firefox to use the containerised instance rather than attempt to reuse any instance that might be active on the host, which would defeat the point of having a containerised instance.

The [firefox.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

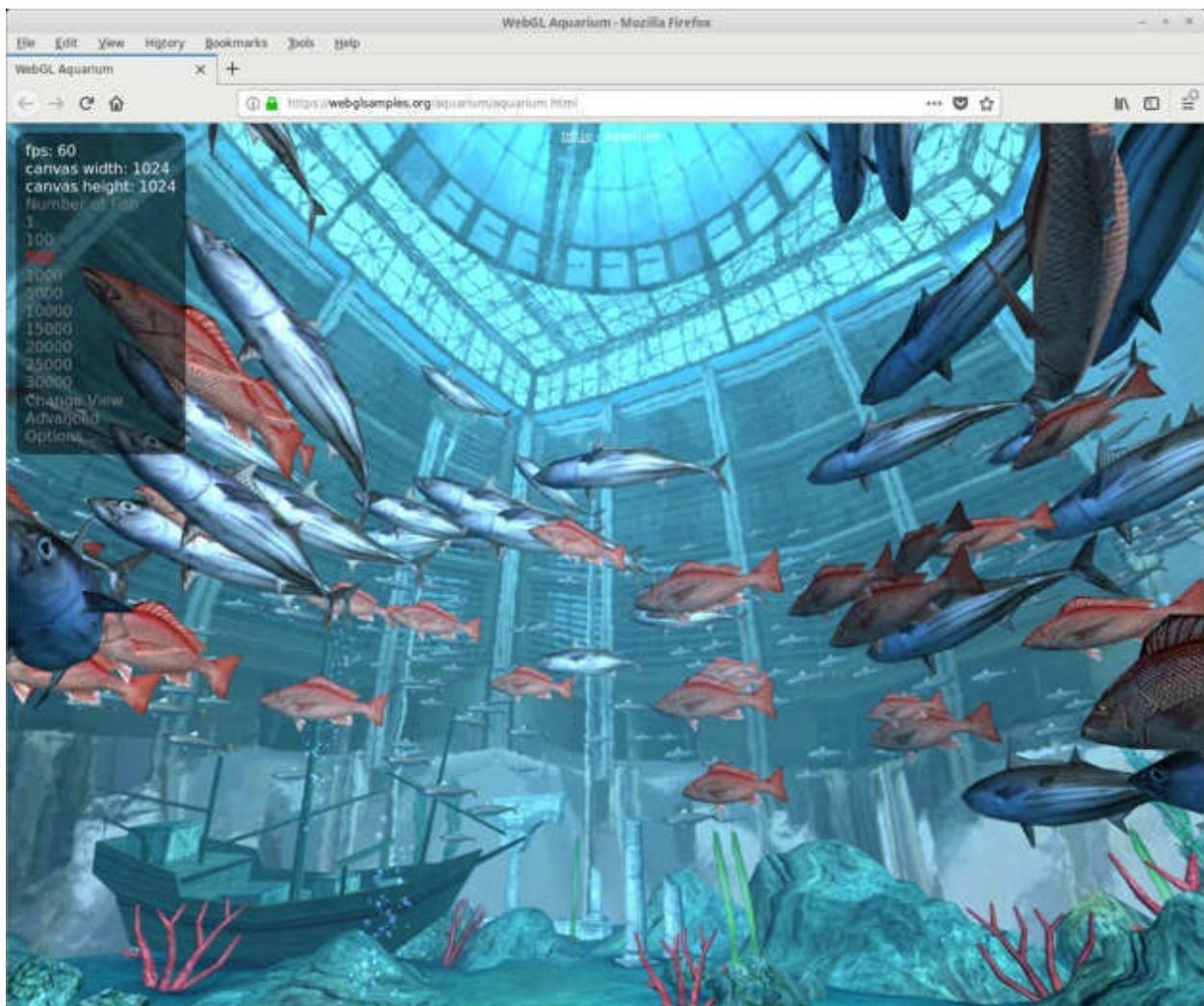
# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un) /.config/pulse
mkdir -p $(id -un) /.config/dconf
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    firefox
```

Note the use of several of the scripts described in [Modular Scripting](#) to enable audio, D-bus, and WebGL support.

One other rather interesting and subtle part of the launch script is the line:

```
--shm-size 2g
```

This increases the size of /dev/shm in the container from the default of 64MB to 2GB. This is important, as recent versions of Firefox have enabled multi-process windows that use shared memory to communicate and running Firefox with the default value causes tabs for many web sites to crash.



5-2-2 firefox-flash

[Adobe Flash Player](#) is the gift that keeps on giving. The once mighty engine that has powered multimedia content on the Internet since the 1990s has been in slow decline since 2010 when Apple announced that it would not be allowed on iOS, and Adobe itself has announced that it will officially end development of the Adobe Flash Player at the end of 2020.

Unfortunately however even in early 2020 some web sites stubbornly persist in delivering content via Flash, which is particularly annoying for Linux desktop users as Adobe's support for Linux has been consistently terrible for years.

Although the Pepper Plugin API ([PPAPI](#)) Flash plugin has continued to be supported for Google Chrome, the Netscape Plugin API ([NPAPI](#)) version that is used by Firefox was stuck at an old version for years. Whilst the NPAPI Flash plugin has now been updated, in practice both the PPAPI and NPAPI versions of Flash Player for Linux are essentially useless on the majority of the sites still serving Flash content.

The issue for Linux is that although both Firefox and Chrome on Linux actually do support DRM controlled content, for example in Firefox `about:preferences` there is a “Play DRM-controlled content” check box, unfortunately the official version of Flash Player available for Linux under both PPAPI and NPAPI is missing the necessary `PPB_Flash_DRM` code and it is unlikely ever to be supported by Adobe, especially given Flash Player’s impending end-of-life.

Until the sites serving Flash content *finally* upgrade to using HTML5 a workaround for this issue does in fact exist, however it is rather convoluted and a little obtuse. As luck would have it though that makes it an ideal candidate to illustrate the power of Docker for developing solutions that can automate tedious jobs and make them more reproducible.

The key to the workaround is the knowledge that whilst the PPAPI Flash Player for Linux does not contain the required PPB_Flash_DRM code, it turns out that the Flash Player available for [ChromeOS](#) actually *does* have the required code.

Given that Flash content is a well-known source of vulnerabilities the author wouldn't wish to encourage readers to blindly pull Flash libraries from the Internet so, in order for readers to make an informed assessment of the risk, the process that was used for finding and extracting the ChromeOS Flash Player library is described in detail below prior to presenting the Dockerfile that automates the process.

The starting point for this journey is Google's Chromebook support page <https://support.google.com/chromebook> where, if we enter the word "recovery" into the search box we should quickly find the "Recover your Chromebook" page
<https://support.google.com/chromebook/answer/1080595>.

Near the bottom of the "Recover your Chromebook" page there is a section "Use a Linux computer" which says: On the Linux computer, download the Recovery Tool here:
https://dl.google.com/dl/edgedl/chromeos/recovery/linux_recovery.s

For our purposes we won't actually be using Google's linux_recovery.sh script directly, however near the start of that script there is a link to another file that we do care about

<https://dl.google.com/dl/edgedl/chromeos/recovery/recovery.conf>.

This file contains a long list of Chromebook models along with the URL of their recovery image. The image that we need is the "zako" image that may be found by searching through recovery.conf, or in a more automated way via:

```
IMAGE_URL=$(wget -O - https://dl.google.com/dl/edgedl/chromeos/recovery/recovery.conf | grep zako | tail -n 1 | cut -d \= -f2)
```

if we run:

```
echo $IMAGE_URL
```

we should see the full URL of the image that we require, which will look something like

https://dl.google.com/dl/edgedl/chromeos/recovery/chromeos_1223c_channel_mp-v3.bin.zip

From the IMAGE_URL we see that the zip name is the part after the last / in the URL:

```
ZIP_NAME=$(echo "${IMAGE_URL}" | cut -d \/ -f8)
```

and if we remove the .zip extension we get the bin name:

```
BIN_NAME=$(echo $ZIP_NAME | rev | cut -d \. -f2- | rev)
```

The next step is to change to a temporary directory such as /mnt then download the recovery image:

```
wget -O ${ZIP_NAME} ${IMAGE_URL}
```

We then unzip it and remove the temporary zip file:

```
unzip ${ZIP_NAME} && rm ${ZIP_NAME}
```

By this point in the process we will have a filesystem image that we need to unpack. The easiest way to achieve this is to use 7z as this avoids the need to perform loopback mounting:

```
7z -y x ${BIN_NAME} -ochromeos
```

If we list the chromeos directory created by the previous command we will see:

```
EFI-SYSTEM.img  KERN-C.img  ROOT-B.img  STATE.img  
KERN-A.img      OEM.img     ROOT-C.img  reserved.img  
KERN-B.img      ROOT-A.img  RFWF.W.img
```

We next need to unpack the root filesystem in a similar way:

```
7z -y x ./chromeos/ROOT-A.img -orootfs
```

If we now inspect rootfs/opt/google/chrome/pepper we should see libpepflashplayer.so, which is the library we require and pepper-flash.info from which we will extract the version information and construct a manifest.

[Dockerfile-extract-libpepflashplayer](#) performs the procedure just described and may be found in the directory [firefox-flash/plugins](#) of [docker-gui](#) under [5-more-complex-applications/browsers](#).

```
FROM debian:stretch-slim

# Download the zako ChromeOS recovery image and extract
# libpepflashplayer.so from it.
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    wget p7zip-full unzip ca-certificates && \
    # First grok the URL of the zako ChromeOS recovery image
    IMAGE_URL=$(wget -O - https://dl.google.com/dl/edgedl/chromeos/recovery/recovery.conf | grep zako | tail -n 1 | cut -d \= -f 2) && \
    # The zip name is the part after the last / in the URL
    ZIP_NAME=$(echo "${IMAGE_URL}" | cut -d \/ -f8) && \
    # If we remove the .zip extension we get the bin name
    BIN_NAME=$(echo $ZIP_NAME | rev | cut -d \. -f2- | rev) && \
    # Download the ChromeOS recovery image and unzip
    cd /mnt && \
    wget -O ${ZIP_NAME} ${IMAGE_URL} && \
    unzip ${ZIP_NAME} && \
    rm ${ZIP_NAME} && \
    # Unpack the binary using p7zip
    7z -y x ${BIN_NAME} -ochromeos && \
    7z -y x ./chromeos/ROOT-A.img -orootfs && \
    mv ./rootfs/opt/google/chrome/pepper . && \
    VERSION=$(cat pepper/pepper-flash.info | grep VERSION= | head -n 1 | cut -d \" -f2) && \
    echo "{$\n      \"version\": \"${VERSION}\\"\n}" > pepper/manifest.json && \
    # Remove the packages used for extraction from the image
    rm ${BIN_NAME} && \
    rm -rf chromeos && \
    rm -rf rootfs && \
```

```
apt-get clean && \
apt-get purge -y wget p7zip-full \
    unzip ca-certificates && \
apt-get autoremove -y && \
# Display the URL of the Chrome OS image we've just used
echo "\nUsing Chrome OS image URL:\n${IMAGE_URL}\n" && \
echo "zip name:\n${ZIP_NAME}\n" && \
echo "bin name:\n${BIN_NAME}\n" && \
echo "Version: ${VERSION}\n" && \
rm -rf /var/lib/apt/lists/*
```

The image may be built with:

```
docker build -t extract-libpepflashplayer -f Dockerfile-extract \
-libpepflashplayer .
```

We now have a Docker image that will allow us to extract a Flash Player library with PPB_Flash_DRM support that library uses PPAPI. We shall however defer running it for the time being as although one might assume that library would “just work” with Google Chrome there are other Flash DRM issues with Chrome, so we shall instead look to use our library with Firefox.

Using a PPAPI Flash Player with Firefox requires some further work to “wrap” it in an NPAPI layer, for this we use <https://github.com/irinat/freshplayerplugin>. Note that although freshplayerplugin is available in package repositories for some Linux distributions those versions are out of date and won’t work with the ChromeOS Flash Player library extracted by the process described earlier, so we must instead build freshplayerplugin from source.

Dockerfile-build-libfreshwrapper will allow us to build freshplayerplugin, it may be found in the firefox-flash/plugins directory of docker-gui under 5-more-complex-applications/browsers

.

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ca-certificates git make cmake gcc g++ pkg-config ragel \
    libasound2-dev libpulse-dev libssl-dev libglib2.0-dev \
    libpango1.0-dev libgl1-mesa-dev libevent-dev \
    libgtk2.0-dev libxrandr-dev libxrender-dev libv4l-dev \
    libxcursor-dev libgles2-mesa-dev libavcodec-dev \
    libva-dev libvdpau-dev libdrm-dev libicu-dev && \
    mkdir /src && cd /src && \
    git clone https://github.com/i-rinat/freshplayerplugin.git &&
 \
    cd freshplayerplugin && mkdir build && cd build && \
    cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo .. && \
    make -j"$nproc" && \
    cd ../../ && \
    mv freshplayerplugin/build/libfreshwrapper-flashplayer.so . &
& \
    mv freshplayerplugin/data/freshwrapper.conf.example . && \
    rm -rf freshplayerplugin && \
    apt-get clean && \
    apt-get purge -y ca-certificates git make cmake gcc g++ \
    pkg-config ragel libasound2-dev libpulse-dev libssl-dev \
    libglib2.0-dev libpango1.0-dev libgl1-mesa-dev \
    libevent-dev libgtk2.0-dev libxrandr-dev libxrender-dev \
    libxcursor-dev libv4l-dev libgles2-mesa-dev libicu-dev \
    libavcodec-dev libva-dev libvdpau-dev libdrm-dev && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*
```

Although this Dockerfile again looks complex it is relatively straightforward. We start by installing the build dependencies, which are mostly described in the freshplayerplugin README.md page. Next we git clone the freshplayerplugin repository and perform the build. Finally, we copy the compiled library to a place we can later retrieve it then remove all of the build dependencies from the image.

The image may be build with:

```
docker build -t build-libfreshwrapper -f Dockerfile-build-libfreshwrapper .
```

We can now use the [create-plugins.sh](#) script to extract the libraries we require from our previously built images:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
# Extract the libfreshwrapper-flashplayer.so wrapper plugin  
# built by the build-libfreshwrapper container.  
$DOCKER_COMMAND run --rm build-libfreshwrapper cat /src/libfreshwrapper-flashplayer.so > libfreshwrapper-flashplayer.so  
  
# Extract libpepflashplayer.so and manifest.json created  
# by pulling the ChromeOS recovery image and unpacking it.  
$DOCKER_COMMAND run --rm extract-libpepflashplayer cat /mnt/pepper/libpepflashplayer.so > libpepflashplayer.so  
  
$DOCKER_COMMAND run --rm extract-libpepflashplayer cat /mnt/pepper/manifest.json > manifest.json
```

Now that we have extracted our plugin libraries we can create a Firefox image that supports Flash Player. The [Dockerfile](#) is in the directory [firefox-flash](#) of [docker-gui](#) under [5-more-complex-applications/browsers](#).

```
FROM firefox

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libevent-2.0-5 libevent-pthreads-2.0-5 && \
    rm -rf /var/lib/apt/lists/*

# Copy the libfreshwrapper-flashplayer.so wrapper plugin.
COPY plugins/libfreshwrapper-flashplayer.so /usr/lib/mozilla/plugins/libfreshwrapper-flashplayer.so

# Copy the libpepflashplayer.so plugin into the image.
COPY plugins/libpepflashplayer.so /usr/lib/adobe-flashplugin/libpepflashplayer.so

# Copy manifest.json into the image.
COPY plugins/manifest.json /usr/lib/adobe-flashplugin/manifest.json

ENTRYPOINT ["/usr/lib/firefox/firefox", "--no-remote"]
```

and may be built with:

```
docker build -t firefox-flash .
```

This Dockerfile uses our previous firefox as a base image then installs libevent, which is needed by freshplayerplugin. It then copies the libraries and manifest from our plugins directory.

The new [firefox.sh](#) launch script is as follows and is identical to our earlier firefox.sh with the exception of using our new firefox-flash image:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    firefox-flash
```

As mentioned previously, Flash content is a well-known source of vulnerabilities. Although the procedure for obtaining Flash Player described in this section will allow us to obtain an up-to-date legitimate ChromeOS Flash Player instance, and running in a container will provide some mitigations, we could still do better. In particular, to provide a relatively seamless user experience, we have bind-mounted a user directory which perhaps makes it a little *too* easy to export potential vulnerabilities to the rest of the system.

One way to lock things down a little bit more is to make the container instances fully ephemeral, preserving no user data between one run and the next. [Dockerfile-flash-ephemeral](#), the Dockerfile for a somewhat more ephemeral Firefox Flash image is as follows:

```
FROM firefox

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libevent-2.0-5 libevent-pthreads-2.0-5 && \
    rm -rf /var/lib/apt/lists/* && \
    # Add user "firefox" to run as non privileged user.
    groupadd -r -g 1000 firefox && \
    useradd -u 1000 -r -g firefox -G audio,video firefox && \
    mkdir /home/firefox/ && \
    chown -R firefox:firefox /home/firefox

# Copy the libfreshwrapper-flashplayer.so wrapper plugin.
COPY plugins/libfreshwrapper-flashplayer.so /usr/lib/mozilla/pl
ugins/libfreshwrapper-flashplayer.so

# Copy the libpepflashplayer.so plugin into the image.
COPY plugins/libpepflashplayer.so /usr/lib/adobe-flashplugin/li
bpepflashplayer.so

# Copy manifest.json into the image.
COPY plugins/manifest.json /usr/lib/adobe-flashplugin/manifest.
json

# Run application as non privileged user.
USER firefox
ENTRYPOINT ["/usr/lib/firefox/firefox", "--no-remote"]
```

and may be built with:

```
docker build -t firefox-flash-ephemeral -f Dockerfile-flash-ephemeral .
```

This Dockerfile is very similar to our earlier Firefox Flash Dockerfile, but contains additional instructions to add an unprivileged user directly into the image.

The [firefox-flash-ephemeral.sh](#) launch script is much simpler than some of our other launch scripts, as we have removed D-bus integration and all the bind-mounts to user directories:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

$DOCKER_COMMAND run --rm \
--shm-size 2g \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
$GPU_FLAGS \
firefox-flash-ephemeral
```

This approach is still not perfect however as Flash Player requires OpenGL, so we still have to maintain GPU support and we are still connecting to the host's X11 and PulseAudio Unix domain sockets.

We shall look at ways to provide greater application isolation later in the book when we explore remote applications.

5-2-3 chrome

A key use case for running GUI applications in Docker is to allow us to run multiple different versions of an application, enabling them to be tested more safely and cleanly removed. This can be especially important for web applications, where we might wish to test them using several different browsers and different versions of the same browser.

With that goal in mind, we shall now present the [Dockerfile](#) for a containerised version of [Google Chrome](#), which is currently the most popular browser. It may be found in the [chrome](#) directory of [docker-gui](#) under [5-more-complex-applications/browsers](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install Chrome
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        apt-transport-https wget gnupg ca-certificates \
        fonts-symbola fonts-lmodern fonts-freefont-ttf \
        fonts-liberation fonts-dejavu gsffonts \
        libpulse0 libv4l-0 libgl1-mesa-glx libgl1-mesa-dri && \
        wget -O - https://dl.google.com/linux/linux_signing_key.pub | \
        apt-key add - && \
        echo "deb [arch=amd64] https://dl.google.com/linux/chrome/deb \
        / stable main" > /etc/apt/sources.list.d/google.list && \
```

```
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends \
google-chrome-stable && \
rm -rf /var/lib/apt/lists/* && \
cp /etc/pulse/client.conf \
/etc/pulse/client-noshm.conf && \
sed -i "s/enable-shm = yes/enable-shm = no/g" \
/etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf

ENTRYPOINT ["google-chrome"]
```

and may be built with:

```
docker build -t chrome .
```

This Dockerfile is somewhat similar to the one we developed for [Firefox](#). We again install a range of font packages and must install apt-transport-https, wget, gnupg, and ca-certificates as Google Chrome is not packaged in the main distribution repositories so must be downloaded directly from Google.

The next section of the Dockerfile downloads the Chrome .deb package and its signing key from Google and adds it to the package manager sources list. Finally, we update the package manager and install google-chrome-stable from the repository we've just added.

We next copy the fonts.conf file from the Docker build context directory to /etc/fonts/local.conf as we did for Firefox and finally we set the container entrypoint to google-chrome.

The [chrome.sh](#) launch script is as follows, which we note is identical to our earlier [firefox.sh](#) launch script with the exception of the image being used:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    chrome
```

Once again note the use of:

```
--shm-size 2g
```

in order to increase the size of /dev/shm in the container from the default of 64MB to 2GB. As an aside, the choice of 2GB as the shared memory limit for several of the containers in this book was rather arbitrary and in many cases this could be safely reduced.

5-2-4 chromium

Although Google Chrome is currently the most popular web browser it has a proprietary licence and so a number of people prefer to use [Chromium](#) as a fully open source alternative.

The [Dockerfile](#) is in the [chromium](#) directory of [docker-gui](#) under [5-more-complex-applications/browsers](#).

```
FROM debian:stretch-slim
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN sed -i 's/main/main contrib/' \
    /etc/apt/sources.list && \
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends \
apt-transport-https gnupg ca-certificates \
fonts-symbola fonts-lmodern fonts-freefont-ttf \
fonts-liberation fonts-dejavu gsffonts \
libpulse0 libv4l-0 libgl1-mesa-glx libgl1-mesa-dri \
gsettings-desktop-schemas \
chromium chromium-l10n chromium-widevine && \
rm -rf /var/lib/apt/lists/* && \
cp /etc/pulse/client.conf \
    /etc/pulse/client-noshm.conf && \
sed -i "s/enable-shm = yes/enable-shm = no/g" \
    /etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf
ENTRYPOINT ["chromium"]
```

and may be built with:

```
docker build -t chromium .
```

As an open source product Chromium is available in the repositories of all major Linux distributions, though in Debian we must first enable the contrib repository via:

```
sed -i 's/main/main contrib/' /etc/apt/sources.list
```

The [chromium.sh](#) launch script is again the same as our chrome.sh and firefox.sh scripts except for the image being used:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    chromium
```

Again we increase the size of /dev/shm with:

```
--shm-size 2g
```

5-2-5 opera

The next browser we shall demonstrate is [Opera](#) which, although it has modest market share, is often innovative with interesting features such as a built-in VPN. The [Dockerfile](#) is in the [opera](#) directory of [docker-gui](#) under [5-more-complex-applications/browsers](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        apt-transport-https wget gnupg ca-certificates \
        fonts-symbola fonts-lmodern fonts-freefont-ttf \
        fonts-liberation fonts-dejavu gsffonts \
        libpulse0 libv4l-0 libgl1-mesa-glx libgl1-mesa-dri && \
        wget -O - https://deb.opera.com/archive.key | apt-key add - &
& \
    echo "deb [arch=amd64] http://deb.opera.com/opera-stable/ stable non-free" > /etc/apt/sources.list.d/opera.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
        opera-stable && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf

ENTRYPOINT ["opera"]
```

The opera image may be built with:

```
docker build -t opera .
```

The Opera Dockerfile is extremely similar to the one developed for Google Chrome, with the exception of the changes to the download repository that we have added and, obviously, that we are installing the opera-stable package rather than google-chrome-stable.

The [opera.sh](#) launch script again follows the pattern of our previous browser launch scripts.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    opera
```

5-2-6 epiphany

The final browser that we shall demonstrate is [Epiphany](#), also known as GNOME Web. Epiphany is billed as the web browser for the GNOME desktop and although it has a very small market share it has the advantage of very tight integration with the GNOME desktop, moreover, it is a pure GTK 3 application so renders natively on Wayland and Broadway.

The [Dockerfile](#) is in the [epiphany](#) directory of [docker-gui](#) under [5-more-complex-applications/browsers](#).

```
FROM debian:stretch-slim

LABEL com.nvidia.volumes.needed=nvidia_driver

# Install from testing as version in stable doesn't
# play YouTube videos and suffers with other glitches.
RUN echo "deb http://ftp.debian.org/debian testing main" \
    > /etc/apt/sources.list.d/stretch-testing.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends -t testing \
    ca-certificates epiphany-browser libpulse0 \
    libgl1-mesa-glx libgl1-mesa-dri gstreamer1.0-libav \
    gstreamer1.0-pulseaudio gstreamer1.0-plugins-good \
    gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf

ENTRYPOINT ["epiphany"]
```

The epiphany image may be built with:

```
docker build -t epiphany .
```

One significant point of note in this Dockerfile is the first line of the RUN statement and the -t testing flags of apt-get install. Those are necessary so that epiphany-browser is pulled from the Debian testing repository, as the version in stretch stable has issues playing YouTube videos and some other glitches.

The [epiphany.sh](#) launch script again follows the familiar pattern of our previous browser launch scripts.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un) /.config/pulse
mkdir -p $(id -un) /.config/dconf
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    epiphany
```

5-3 Compute Acceleration

In the [Local Applications](#) chapter earlier in this book we covered [3D Accelerated Applications](#) and showed how we could pass through a range of GPU devices into containers in order to accelerate OpenGL 3D graphics applications.

In this section however, we shall show it is possible to travel far beyond the obvious world of simple graphics acceleration into a more exciting world of general purpose compute acceleration, where we can harness the massive computational power of GPUs, many core CPUs or other devices such as FPGAs from within our Docker applications.

Compute acceleration is a rapidly growing field as it is used to power a range of technologies from machine learning through to the high performance graphics rendering required for movie CGI effects and game engines. We shall unfortunately only be able to scratch the surface as this is a topic that could comfortably fill several books of its own but, by the end of this section, readers should be able to develop Dockerised applications using the two most popular acceleration APIs, the Nvidia specific [CUDA](#) and the more heterogeneous [OpenCL](#).

The CUDA examples should work on any relatively modern Nvidia GPU, though for older GPUs it may be necessary to modify the Dockerfiles to use an earlier CUDA version. The OpenCL examples have been tested on Nvidia GPUs and on CPUs, however they *should* also work on GPUs that use the Mesa Open Source drivers, but this has not been tested.

5-3-1 nvidia-smi

Most users have a range of tools at their disposal to monitor the status of their CPUs or see how much system memory and disk space is free, but by contrast monitoring the health and status of GPUs has historically been much more difficult.

The Nvidia System Management Interface ([nvidia-smi](#)) is a command line utility that provides detailed management and monitoring capabilities for Fermi and later Nvidia GPU architecture families and can report query information as XML or plain text to either standard output or a file.

We have actually already encountered nvidia-smi very briefly before in the [3D Accelerated Applications](#) section when we covered Nvidia GPU Acceleration, as the installation instructions for [nvidia-docker](#) end with the following line, which will download Nvidia's **nvidia/cuda:9.0-base** image and run nvidia-smi from that:

```
# Test nvidia-smi with the latest official CUDA image
docker run --runtime=nvidia --rm nvidia/cuda:9.0-base nvidia-smi
```

Using Nvidia's image is perfectly reasonable, but in order to gain more control we shall create our own image. Note too that Nvidia's CUDA images contain the required [OCI spec environment variables](#) as ENV items in the Dockerfiles, whereas we have added them to the GPU_FLAGS we set as part of our [docker-gpu.sh](#) run time helper script.

The [Dockerfile](#) is in the [nvidia-smi](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

ENV CUDA_VERSION 9.0
ENV NVIDIA_REQUIRE_CUDA "cuda>=${CUDA_VERSION}"
ENV CUDA_FULL_VERSION ${CUDA_VERSION}.176

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver
LABEL com.nvidia.cuda.version="${CUDA_FULL_VERSION}"

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    apt-transport-https wget gnupg ca-certificates && \
    wget -O - https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub | apt-key add - && \
    echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64 /" > /etc/apt/sources.list.d/cuda.list \
&& \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    cuda-cudart-${CUDA_VERSION} && \
    ln -s cuda-${CUDA_VERSION} /usr/local/cuda && \
    apt-get clean && \
    apt-get purge -y apt-transport-https wget gnupg \
        ca-certificates && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["nvidia-smi"]
```

and may be built with:

```
docker build -t nvidia-smi .
```

The first part of the Dockerfile RUN command adds the packages needed to add and sign the required Nvidia CUDA repository.

The second part actually adds the repository from <https://developer.download.nvidia.com/compute/cuda/repos/>. Note that we are using the ubuntu1604 repository, which also works for Debian stretch.

After adding the repository we perform an apt update and add the CUDA runtime package cuda-cudart-9.0, we then symlink the version specific install to /usr/local/cuda.

Finally, to keep the image size as small as possible we remove all of the packages that were required only for installation of the Nvidia repository.

The [nvidia-smi.sh](#) launch script is very simple:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-gpu.sh  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$GPU_FLAGS \  
nvidia-smi $@
```

Detailed instructions on getting the best out of nvidia-smi may be found on Nvidia's web site

<http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf>

5-3-2 cuda-nbody

The [nvidia-smi](#) container was a very minimal example intended to provide a basic illustration of passing GPU compute capabilities through to a container. The [N-body simulation](#) example that we shall now cover is a lot more interesting, as it combines CUDA compute acceleration with OpenGL 3D graphics acceleration in a single application.

The [Dockerfile](#) for our CUDA accelerated N-body simulation is in the [cuda-nbody](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

ENV CUDA_VERSION 9.0
ENV NVIDIA_REQUIRE_CUDA "cuda>=${CUDA_VERSION}"
ENV CUDA_FULL_VERSION ${CUDA_VERSION}.176

# nvidia-docker hooks (Only needed for Nvidia Docker v1)
LABEL com.nvidia.volumes.needed=nvidia_driver
LABEL com.nvidia.cuda.version="${CUDA_FULL_VERSION}"

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used to install CUDA
    apt-get install -y --no-install-recommends \
    apt-transport-https wget gnupg ca-certificates && \
    # Add the CUDA packages and signing key
    wget -O - https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub | apt-key add - && \
    echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64 /" > /etc/apt/sources.list.d/cuda.list
    && \
```

```
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends \
libgl1-mesa-glx libgl1-mesa-dri libglu1-mesa freeglut3 \
cuda-cudart-${CUDA_VERSION} \
cuda-samples-${CUDA_VERSION} && \
ln -s cuda-${CUDA_VERSION} /usr/local/cuda && \
# Compile the sample. Note the tweak of findgllib.mk
# that is needed to find the OpenGL libraries on debian.
cd /usr/local/cuda/samples/5_Simulations/nbody && \
sed -i 's/grep -i ubuntu/grep -i "ubuntu\\|debian"/' \
findgllib.mk && \
make && cp nbody /usr/local/cuda/bin/. && make clean && \
# Remove the packages used for installation and build
apt-get clean && \
apt-get purge -y apt-transport-https wget gnupg \
ca-certificates cuda-samples-${CUDA_VERSION} && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*
```

```
ENTRYPOINT ["/usr/local/cuda/bin/nbody"]
```

and may be built with:

```
docker build -t cuda-nbody .
```

This Dockerfile starts off in exactly the same way as our nvidia-smi Dockerfile by adding the ubuntu1604 cuda repository. The main change from our previous example is adding the OpenGL runtime and cuda-samples-9.0 packages, which will also pull in all of the required development packages via the power of apt dependency management.

Because Nvidia only formally supports RHEL/CentOS and Ubuntu we have to perform some minor patching of the build system to support building on Debian as follows.

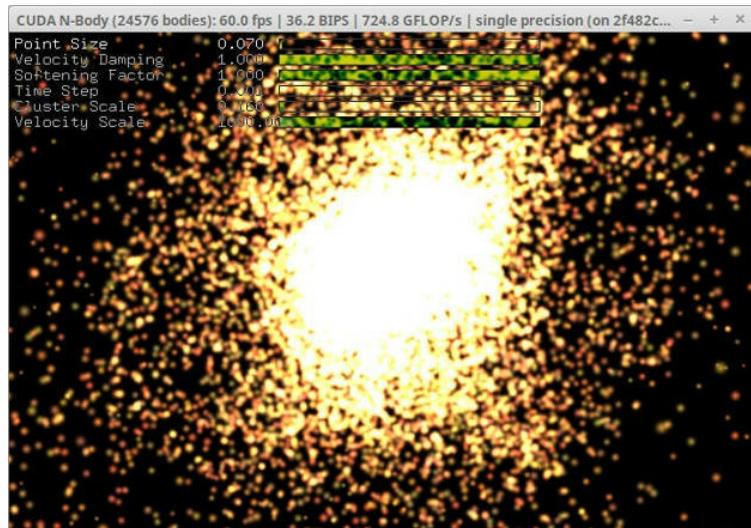
```
sed -i 's/grep -i ubuntu/grep -i "ubuntu\\|debian"/' findgllib.mk
```

We copy the built executable to /usr/local/cuda/bin/, then remove all of the packages that were only required for building, so we end up with a very modestly sized image.

The [cuda-nbody.sh](#) launch script is very simple, as most of the hard work is done by the [docker-gpu.sh](#) helper script:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
$GPU_FLAGS \
cuda-nbody $@
```



5-3-3 cuda-smoke

The [Dockerfile](#) for our CUDA accelerated [smoke simulation](#) is in the [cuda-smoke](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

ENV CUDA_VERSION 9.0
ENV NVIDIA_REQUIRE_CUDA "cuda>=${CUDA_VERSION}"
ENV CUDA_FULL_VERSION ${CUDA_VERSION}.176

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver
LABEL com.nvidia.cuda.version="${CUDA_FULL_VERSION}"

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used to install CUDA
    apt-get install -y --no-install-recommends \
    apt-transport-https wget gnupg ca-certificates && \
    # Add the CUDA packages and signing key
    wget -O - https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub | apt-key add - && \
    echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64 /" > /etc/apt/sources.list.d/cuda.list
    && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libgl1-mesa-glx libgl1-mesa-dri libglu1-mesa freeglut3 \
    cuda-cudart-${CUDA_VERSION} \
    cuda-samples-${CUDA_VERSION} && \
    ln -s cuda-${CUDA_VERSION} /usr/local/cuda && \
    # Compile the sample. Note the tweak of findgllib.mk
    # that is needed to find the OpenGL libraries on debian.
```

```
cd /usr/local/cuda/samples/5_Simulations && \
cd smokeParticles && \
sed -i 's/grep -i ubuntu/grep -i "ubuntu\\|debian"/' \
      findglLib.mk && \
make && cp smokeParticles /usr/local/cuda/bin/. && \
cp -R data /usr/local/cuda/bin/. && make clean && \
# Remove the packages used for installation and build
apt-get clean && \
apt-get purge -y apt-transport-https wget gnupg \
              ca-certificates cuda-samples-${CUDA_VERSION} && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

# Set WORKDIR and use relative path as this example uses
# Nvidia's sdkFindFilePath to find the data directory and
# using the absolute path doesn't work correctly.
WORKDIR /usr/local/cuda/bin
ENTRYPOINT ["./smokeParticles"]
```

and may be built with:

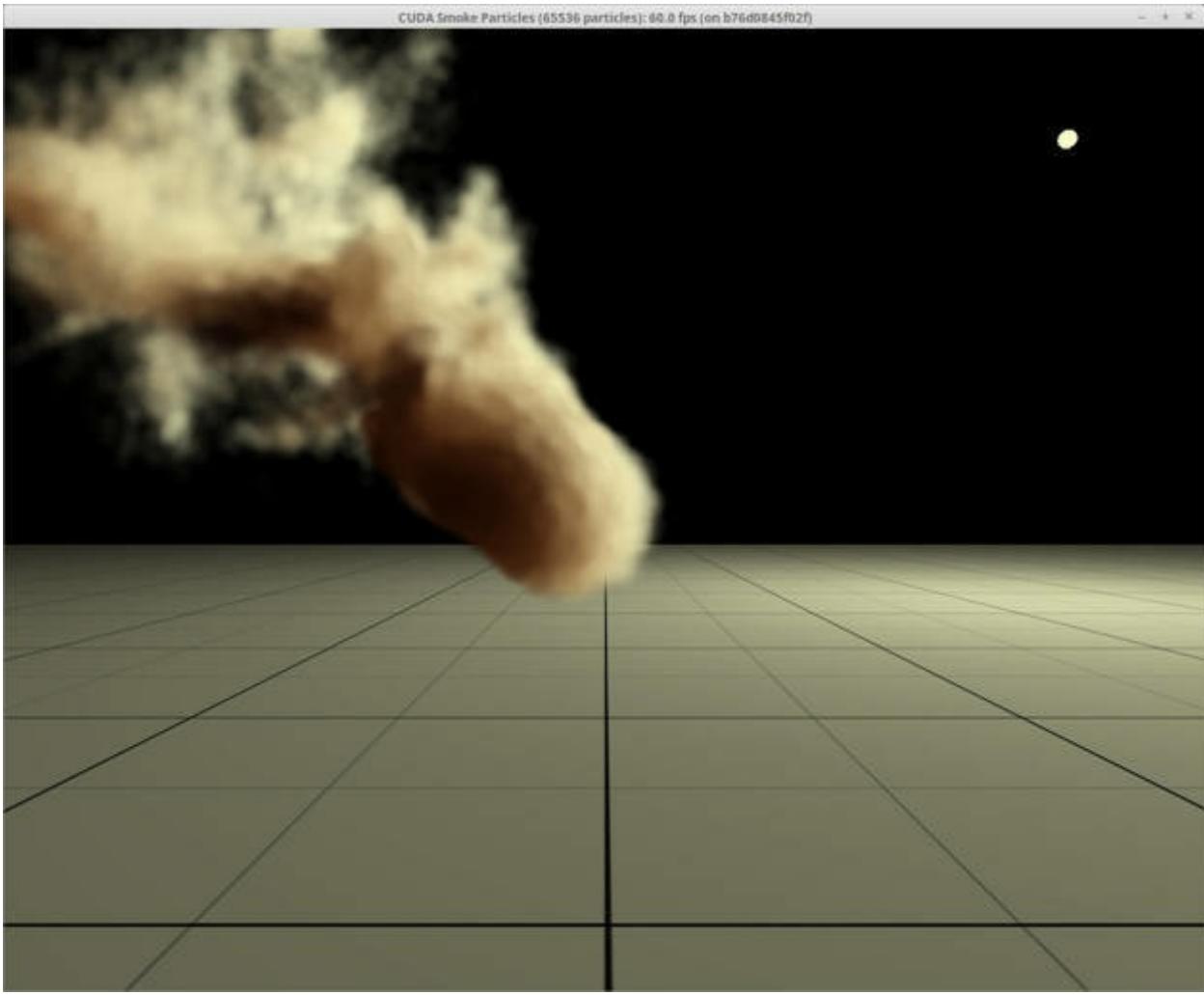
```
docker build -t cuda-smoke .
```

Again, note the simple patching of the build system that is required because Nvidia only formally supports RHEL/CentOS and Ubuntu. The main difference between this Dockerfile and our cuda-nbody Dockerfile is the need for the data directory relative to the bin directory that holds the assets required for rendering.

Hopefully these examples have illustrated how useful Docker can be for setting up a clean and consistent CUDA development environment.

The [cuda-smoke.sh](#) launch script is again very simple:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-gpu.sh  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$X11_FLAGS \  
$GPU_FLAGS \  
cuda-smoke $@
```



5-3-4 clinfo

By way of contrast with CUDA, which is very much Nvidia specific, the other major API for compute acceleration is [OpenCL](#) a framework for developing applications that execute across heterogeneous platforms including CPUs, GPUs, FPGAs or other hardware accelerators.

clinfo is a simple command-line application that enumerates all available information about all of the OpenCL platforms available on the system and the devices they expose. Like the nvidia-smi application for CUDA devices clinfo serves as a useful first step for working with OpenCL on Docker to check that we actually have working OpenCL platforms and devices.

Nvidia

OpenCL has implementations for a range of different compute platforms. The first of these is the [Dockerfile](#) for Nvidia OpenCL in the [clinfo](#) directory of [docker-gui](#) tree under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ocl-icd-libopencl1 clinfo && \
    mkdir -p /etc/OpenCL/vendors && \
    echo "libnvidia-opencl.so.1" > \
        /etc/OpenCL/vendors/nvidia.icd && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["clinfo"]
```

The Nvidia clinfo image may be built with:

```
docker build -t clinfo .
```

The Nvidia clinfo Dockerfile hopefully illustrates just how simple it is to enable Nvidia's OpenCL in Docker, which only requires the OpenCL ICD loader ocl-icd-libopencl1 to be installed and an ICD file referencing the vendor's OpenCL implementation library to be created in the directory /etc/OpenCL/vendors/.

The [clinfo.sh](#) launch script for running clinfo on Nvidia platforms is equally trivial.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-gpu.sh  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$GPU_FLAGS \  
clinfo $@
```

Running clinfo should yield a wealth of information about the available platform and devices and should display the following important information towards the end. Successfully creating an OpenCL context as illustrated below informs us that we have a usable OpenCL device.

clGetPlatformInfo(NULL, CL_PLATFORM_NAME, ...)	NVIDIA CUDA
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_ALL, ...)	Success [NV]
clCreateContext(NULL, ...)	[default] Success [NV]

Mesa

Whilst OpenCL has been designed as a vendor agnostic heterogeneous framework, unfortunately each vendor has taken a different approach to providing support. The previous example was Nvidia specific and tested by the author, but unfortunately some of the following examples are untested as the author only has access to Nvidia GPUs.

[Dockerfile-mesa](#) is in the [clinfo](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ocl-icd-libopencl1 clinfo mesa-opencl-icd && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["clinfo"]
```

and may be built with:

```
docker build -t clinfo-mesa -f Dockerfile-mesa .
```

The [clinfo-mesa.sh](#) launch script is equally trivial.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
.$BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$GPU_FLAGS clinfo-mesa $@
```

ROCM

The Open Source Mesa implementation described previously should provide OpenCL 1.1 on most AMD GPUs though, as far as the author is aware, a better approach for modern AMD GPUs is the [ROCM Platform](#). The ROCm GitHub page <https://github.com/RadeonOpenCompute/ROCM> provides comprehensive instructions for installing the required drivers on the host.

An experimental [Dockerfile-rocm](#) is in the [clinfo](#) directory of the [docker-gui](#) repository under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    wget gnupg2 libnuma-dev clinfo && \
    wget -qO - http://repo.radeon.com/rocm/apt/debian/rocm.gpg.ke
y | apt-key add - && \
echo 'deb [arch=amd64] http://repo.radeon.com/rocm/apt/debian
/ xenial main' > /etc/apt/sources.list.d/rocm.list && \
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends rocm-dev && \
apt-get clean && \
apt-get purge -y wget gnupg2 && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["clinfo"]
```

and may be built with:

```
docker build -t clinfo-rocm -f Dockerfile-rocm .
```

This Dockerfile mostly just follows the instructions given in the ROCm GitHub page for adding the required repository, but it installs the rocm-dev meta-package rather than rocm-dkms mentioned in the documentation as that is the full package that also tries to install the drivers.

Note that the experimental ROCm OpenCL Dockerfile presented here results in a rather large (~2GB) image being created as rocm-dev is a meta-package that installs the entire ROCm development stack. It should be possible to reduce the size of the image by being more selective about the installed [packages](#), but as the author is unable to test with any of the supported AMD GPUs it seems safest to simply install the main development meta-package.

The [clinfo-rocm.sh](#) launch script is similar to the others, but note the addition of `--device=/dev/kfd`, which the author believes might be necessary to run ROCm. Note that for some host distributions It might also be necessary to add `--security-opt seccomp=unconfined`.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-gpu.sh  
  
$DOCKER_COMMAND run --rm -it \  
--device=/dev/kfd \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$GPU_FLAGS \  
clinfo-rocm $@
```

Intel Graphics Compute Runtime

The [Intel Graphics Compute Runtime](#) for OpenCL replaces Beignet for Gen8 (Broadwell) and beyond. As with the case of AMD GPUs the author does not have access to Intel GPUs to test this section, so it is presented as-is based on information from the Intel Compute Runtime [releases](#) documentation.

[Dockerfile-intel](#) is in the [clinfo](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

ENV OCL_VERSION 19.43.14583
ENV GMM_VERSION 19.3.2
ENV IGC_VERSION 1.0.2714.1

RUN echo "deb http://ftp.debian.org/debian testing main" > /etc
/apt/sources.list.d/stretch-testing.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends -t testing \
    ocl-icd-libopencl1 clinfo wget ca-certificates && \
    mkdir neo && cd neo && \
    wget "https://github.com/intel/compute-runtime/releases/downl
oad/${OCL_VERSION}/intel-opencl_${OCL_VERSION}_amd64.deb" && \
    wget "https://github.com/intel/compute-runtime/releases/downl
oad/${OCL_VERSION}/intel-ocloc_${OCL_VERSION}_amd64.deb" && \
    wget "https://github.com/intel/compute-runtime/releases/downl
oad/${OCL_VERSION}/intel-gmmlib_${GMM_VERSION}_amd64.deb" && \
    wget "https://github.com/intel/compute-runtime/releases/downl
oad/${OCL_VERSION}/intel-igc-core_${IGC_VERSION}_amd64.deb" && \
    wget "https://github.com/intel/compute-runtime/releases/downl
oad/${OCL_VERSION}/intel-igc-opencl_${IGC_VERSION}_amd64.deb" &
& \
```

```
dpkg -i *.deb && \
# Remove packages used for installation
cd .. && rm -rf neo && \
apt-get clean && \
apt-get purge -y wget ca-certificates && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*
```

```
ENTRYPOINT ["clinfo"]
```

and may be built with:

```
docker build -t clinfo-intel -f Dockerfile-intel .
```

The [clinfo-intel.sh](#) launch script is a repeat of the others.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$GPU_FLAGS \
clinfo-intel $@
```

Intel CPU

In addition to providing an OpenCL implementation for GPUs Intel also provides an [implementation for CPUs](#) which works well on modern CPU architectures and can provide good OpenCL performance if sufficient CPU cores are available.

[Dockerfile-intel-cpu-18.1.0](#) for the Intel CPU OpenCL implementation is in the [clinfo](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

# Driver Version 18.1.0.0920
ENV RUNTIME l_opencl_p_18.1.0.013

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ocl-icd-libopencl1 libnuma1 clinfo libxml2 \
    wget ca-certificates tar rpm2cpio cpio && \
    # Add the Intel OpenCL Installable Client Driver
    wget -O ${RUNTIME}.tgz "http://registrationcenter-download.in\
tel.com/akdlm/irc_nas/13793/${RUNTIME}.tgz" && \
    tar zxfp ${RUNTIME}.tgz && \
    cd ${RUNTIME}/rpm && \
    rpm2cpio intel-openclrt-18.1.0.013-18.1.0-013.x86_64.rpm | cp\
io -idmv && \
    mkdir -p /opt/intel && \
    mv ./opt/intel/opencl_compilers_and_libraries_18.1.0.013/linu\
x/compiler/lib /opt/intel/opencl && \
    # Add ICD to the ICD registry
    mkdir -p /etc/OpenCL/vendors && \
    echo "/opt/intel/opencl/intel64_lin/libintelocl.so" > \
        /etc/OpenCL/vendors/intel_cpu.icd && \
    rm /${RUNTIME}.tgz && \
    rm -rf /${RUNTIME} && \
    apt-get clean && \
    apt-get purge -y wget ca-certificates rpm2cpio cpio && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["clinfo"]
```

and may be built with:

```
docker build -t clinfo-intel-cpu -f Dockerfile-intel-cpu-18.1.0
.
```

The Dockerfile looks quite complex, but is relatively straightforward when dissected. The first part installs the set of dependencies, most of which are obvious, but libnuma1 and libxml2 less so. Those two dependencies only came to light as the author was developing the application and when struggling to persuade it to work had to resort to running `ldd` on the ICD library libintelocl.so to see if any library dependencies were missing.

The next part fetches and extracts the OpenCL runtime.

```
wget -O ${RUNTIME}.tgz "http://registrationcenter-download.intel.com/akdlm/irc_nas/13793/${RUNTIME}.tgz"
tar zxvf ${RUNTIME}.tgz
```

The next part is a little unusual and awkward, where we use rpm2cpio piped through cpio to extract the runtime, as Intel have chosen to provide only rpm packaging.

```
rpm2cpio intel-openclrt-18.1.0.013-18.1.0-013.x86_64.rpm | cpio -idmv
```

The mechanism used to unpack and extract the runtime causes it to be placed in a rather awkward and untidy directory structure, so we next move the runtime to a more convenient location then create the ICD file based on the new location.

```
mv ./opt/intel/opencl_compilers_and_libraries_18.1.0.013/linux/compiler/lib /opt/intel/opencl
mkdir -p /etc/OpenCL/vendors
echo "/opt/intel/opencl/intel64_lin/libintelocl.so" > /etc/OpenCL/vendors/intel_cpu.icd
```

Fortunately the [clinfo-intel-cpu.sh](#) launch script is simple.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
clinfo-intel-cpu $@
```

AMD APP

AMD's Accelerated Parallel Processing SDK was the predecessor of AMDGPU-Pro and the new Open Source ROCm. The SDK includes OpenCL implementations that target AMD GPUs (including some that may not be supported by ROCm) and also CPUs. Although this SDK has been deprecated by AMD and is now only available through <https://web.archive.org> it is still useful as it enables support for older AMD GPUs and also older CPUs that are not supported by Intel's CPU drivers.

The [Dockerfile-amd-app](#) may be found in the [clinfo](#) directory of the [docker-gui](#) repository under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim  
  
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \  
apt-get install -y --no-install-recommends \  
ocl-icd-libopencl1 clinfo \  
wget bzip2 ca-certificates tar && \  
# Add the AMD APP OpenCL Installable Client Driver  
mkdir -p /opt/amd/opencl/install && \  
cd /opt/amd/opencl/install && \
```

```

wget -O AMD-APP-SDKInstaller-v3.0.130.136-GA-linux64.tar.bz2
"https://archive.org/download/AMDAPPSDK/AMD-APP-SDKInstaller-v3
.0.130.136-GA-linux64.tar.bz2" && \
tar xjf AMD-APP-SDKInstaller-v3.0.130.136-GA-linux64.tar.bz2
&& \
./AMD-APP-SDK-v3.0.130.136-GA-linux64.sh --tar xvfp && \
mv lib/x86_64/libamdocl12cl64.so /opt/amd/opencl && \
mv lib/x86_64/sdk/libamdocl64.so /opt/amd/opencl && \
mkdir -p /etc/OpenCL/vendors && \
echo "/opt/amd/opencl/libamdocl64.so" > \
/etc/OpenCL/vendors/amdocl64.icd && \
rm -rf /opt/amd/opencl/install && \
apt-get clean && \
apt-get purge -y wget bzip2 ca-certificates && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

```

```
ENTRYPOINT ["clinfo"]
```

The most interesting parts of the Dockerfile include the download location (<https://archive.org>) and that the SDK is packaged as an executable shell archive which normally expects user interaction, but by passing `--tar xvfp` can be used in unattended installs. The remainder of the Dockerfile is similar to the Intel CPU example, copying the libraries to a more convenient place.

The [clinfo-amd-app.sh](#) launch script repeats the usual pattern.

```

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$GPU_FLAGS clinfo-amd-app @@

```

5-3-5 opencl-nbody

We illustrated a CUDA implementation of an [N-body simulation](#) in the earlier [cuda-nbody](#) example, we shall now show how to build the equivalent OpenCL version.

This example uses the N-body example from Nvidia's OpenCL SDK, but we shall show that it also runs on Intel's CPU implementation and AMD's APP. It might well run on the other implementations illustrated in the previous chapter, but as the author is unable to test this it is left as an exercise for the reader. Dockerfiles for the other implementations should be straightforward to develop based on what the reader should hopefully have learned from previous chapters.

Nvidia

The [Dockerfile](#) for our Nvidia OpenCL accelerated N-body simulation is in the [opencl-nbody](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ocl-icd-libopencl1 ocl-icd-opencl-dev clinfo tar \
    libxmu-dev freeglut3-dev libglew-dev libgl1-mesa-glx \
    libgl1-mesa-dri wget ca-certificates gcc g++ make && \
    # Add the Nvidia OpenCL Installable Client Driver
    mkdir -p /etc/OpenCL/vendors && \
    echo "libnvidia-opencl.so.1" > \
        /etc/OpenCL/vendors/nvidia.icd && \
```

```

# Download Nvidia SDK which includes the OpenCL samples.
# The samples at https://developer.nvidia.com/opencl
# are incomplete, missing the "shared" and "common" code
# needed to compile! We therefore pull the 4.2.9 SDK
# here, because that is the last version to include the
# full compilable OpenCL source tree.
cd /usr/local/src && \
wget -O gpucomputingsdk_4.2.9_linux.run "https://developer.do
wnload.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.
9_linux.run" && \
chmod +x gpucomputingsdk_4.2.9_linux.run && \
./gpucomputingsdk_4.2.9_linux.run --tar xfp && \
rm gpucomputingsdk_4.2.9_linux.run && \
rm install-sdk-linux.pl && \
mv sdk/OpenCL sdk/shared . && \
rm -rf sdk && rm -rf shared/lib/linux && \
sed -i 's/GLEW_x86_64/GLEW/' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
sed -i 's/-Wimplicit//' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
# Fix issue with oclNbodyKernel.cl
sed -i 's/mul24(get_local_size(0), get_local_id(1))/mul24((ui
nt)get_local_size(0), get_local_id(1))/' /usr/local/src/OpenCL/
src/oclNbody/oclNbodyKernel.cl && \
# Fix issue with SobelFilter.cl
sed -i 's/mul24(get_local_size(1), get_global_size(0))/mul24(
(uint)get_local_size(1), get_global_size(0))/' /usr/local/src/O
penCL/src/oclSobelFilter/SobelFilter.cl && \
sed -i 's/mul24((get_group_id(0) + 1)/mul24((uint)(get_group_
id(0) + 1))/' /usr/local/src/OpenCL/src/oclSobelFilter/SobelFilt
er.cl && \
sed -i 's/mul24(get_group_id(0)/mul24((uint)get_group_id(0))/
/usr/local/src/OpenCL/src/oclSobelFilter/SobelFilter.cl && \
cd OpenCL && \

```

```

mv src/oclNbody . && rm -rf src/* && \
mv oclNbody src/. && \
make && \
chmod 777 /usr/local/bin && \
mkdir -p /usr/local/bin/src && \
cp /usr/local/src/OpenCL/bin/linux/release/oclNbody \
    /usr/local/bin/. && \
cp /usr/local/src/OpenCL/src/oclNbody/*.cl \
    /usr/local/bin/src/. && \
rm -rf /usr/local/src/* && \
apt-get clean && \
apt-get purge -y wget ca-certificates gcc g++ make && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*
# Uses Nvidia's shrFindFilePath to find kernel directory
# and using the absolute path doesn't work correctly.
WORKDIR /usr/local/bin/
ENTRYPOINT ["oclNbody"]

```

The opencl-nbody image may be built with:

```
docker build -t opencl-nbody .
```

Although complex the Dockerfile comprises two key parts. The first is a repeat of Nvidia clinfo, which installs the OpenCL runtime, and the second downloads the OpenCL SDK and builds the example. This Dockerfile would certainly be worth splitting into multiple more modular Dockerfiles if our intention wasn't to make these examples self-contained.

The Nvidia OpenCL SDK used in this example has a few quirks that make it an ideal candidate for building in a container as we note the need to use version 4.2.9 of the SDK because later versions are, frustratingly, “incomplete”.

Like the AMD APP SDK from the previous section, the Nvidia OpenCL SDK is packaged as an [executable shell archive](#) which normally expects user interaction, but by passing `--tar xfp` it can be used in unattended installs.

After unpacking the archive we rearrange the directory structure to make it more convenient to remove the parts that we don't require and for subsequent tidy-up stages.

```
mv sdk/OpenCL sdk/shared .
rm -rf sdk && rm -rf shared/lib/linux
```

We next use sed to “patch” the SDK build system and two of the example OpenCL kernels to fix compilation warnings. We then build the example and move the executable to /usr/local/bin before finally tidying up the image.

In contrast to the Dockerfile the [opencl-nbody.sh](#) launch script is fortunately very simple:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
$GPU_FLAGS \
opencl-nbody $@
```

Intel CPU

The [Dockerfile](#) in the `opencl-nbody` directory of [docker-gui](#) under [5-more-complex-applications/compute](#).

At face value it appears even more complex than the Nvidia example. In practice though, the first part is a repeat of the Intel CPU clinfo example which installs the OpenCL runtime and the second part downloads the Nvidia OpenCL SDK and builds the example and is the same as the previous Nvidia version except for the patch that replaces the hard-coded `CL_DEVICE_TYPE_GPU` with `CL_DEVICE_TYPE_CPU`.

```
FROM debian:stretch-slim

ENV RUNTIME l_opencl_p_18.1.0.013

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ocl-icd-libopencl1 ocl-icd-opencl-dev libnuma1 clinfo \
    libxml2 libxmu-dev freeglut3-dev libglew-dev \
    libgl1-mesa-glx libgl1-mesa-dri wget ca-certificates \
    tar rpm2cpio cpio gcc g++ make && \
    # Add the Intel OpenCL Installable Client Driver
    wget -O ${RUNTIME}.tgz "http://registrationcenter-download.intel.com/akdlm/irc_nas/13793/${RUNTIME}.tgz" && \
    tar zxfp ${RUNTIME}.tgz && \
    cd ${RUNTIME}/rpm && \
    rpm2cpio intel-openclrt-18.1.0.013-18.1.0-013.x86_64.rpm | cp
    io -idmv && \
    mkdir -p /opt/intel && \
    mv ./opt/intel/opencl_compilers_and_libraries_18.1.0.013/linux/compiler/lib /opt/intel/opencl && \
    # Add ICD to the ICD registry
    mkdir -p /etc/OpenCL/vendors && \
    echo "/opt/intel/opencl/intel64_lin/libintelocl.so" > \
        /etc/OpenCL/vendors/intel_cpu.icd && \
```

```
# Download Nvidia SDK which includes the OpenCL samples.
# The samples at https://developer.nvidia.com/opencl
# are incomplete, missing the "shared" and "common" code
# needed to compile! We therefore pull the 4.2.9 SDK
# here, because that is the last version to include the
# full compilable OpenCL source tree.
cd /usr/local/src && \
wget -O gpucomputingsdk_4.2.9_linux.run "https://developer.do
wnload.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.
9_linux.run" && \
chmod +x gpucomputingsdk_4.2.9_linux.run && \
./gpucomputingsdk_4.2.9_linux.run --tar xfp && \
rm gpucomputingsdk_4.2.9_linux.run && \
rm install-sdk-linux.pl && \
mv sdk/OpenCL sdk/shared . && \
rm -rf sdk && rm -rf shared/lib/linux && \
sed -i 's/GLEW_x86_64/GLEW/' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
sed -i 's/-Wimplicit//' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
# Fix issue with oclNbodyKernel.cl
sed -i 's/mul24(get_local_size(0), get_local_id(1))/mul24((ui
nt)get_local_size(0), get_local_id(1))/' /usr/local/src/OpenCL/
src/oclNbody/oclNbodyKernel.cl && \
# Fix issue with SobelFilter.cl
sed -i 's/mul24(get_local_size(1), get_global_size(0))/mul24(
(uint)get_local_size(1), get_global_size(0))/' /usr/local/src/O
penCL/src/oclSobelFilter/SobelFilter.cl && \
sed -i 's/mul24((get_group_id(0) + 1)/mul24((uint)(get_group_
id(0) + 1))/' /usr/local/src/OpenCL/src/oclSobelFilter/SobelFilt
er.cl && \
sed -i 's/mul24(get_group_id(0)/mul24((uint)get_group_id(0))/' \
/usr/local/src/OpenCL/src/oclSobelFilter/SobelFilter.cl && \
```

```

# Replace CL_DEVICE_TYPE_GPU with CL_DEVICE_TYPE_CPU
# so the samples will work on Intel CPU device.
for i in $(grep -rl /usr/local/src/OpenCL/src -e CL_DEVICE_T
YPE_GPU); do sed -i 's/CL_DEVICE_TYPE_GPU/CL_DEVICE_TYPE_CPU/' $i; done && \
cd OpenCL && \
mv src/oclNbody . && rm -rf src/* && \
mv oclNbody src/. && \
make && \
chmod 777 /usr/local/bin && \
mkdir -p /usr/local/bin/src && \
cp /usr/local/src/OpenCL/bin/linux/release/oclNbody \
/usr/local/bin/. && \
cp /usr/local/src/OpenCL/src/oclNbody/*.cl \
/usr/local/bin/src/. && \
# Remove packages used for installation
rm ${RUNTIME}.tgz && \
rm -rf ${RUNTIME} && \
rm -rf /usr/local/src/* && \
apt-get clean && \
apt-get purge -y wget ca-certificates rpm2cpio cpio \
gcc g++ make && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

# Uses Nvidia's shrFindFilePath to find kernel directory
# and using the absolute path doesn't work correctly.
WORKDIR /usr/local/bin/
ENTRYPOINT ["oclNbody"]

```

The opencl-nbody-intel-cpu image may be built with:

```

docker build -t opencl-nbody-intel-cpu -f Dockerfile-intel-cpu-
18.1.0 .

```

The [opencl-nbody-intel-cpu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-gpu.sh  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$X11_FLAGS \  
$GPU_FLAGS \  
opencl-nbody-intel-cpu $@
```

AMD APP

The AMD APP N-body [Dockerfile](#) is in the [opencl-nbody](#) directory of [docker-gui](#) under [5-more-complex-applications/compute](#). Like the Nvidia and Intel examples the first part repeats the clinfo example to install the OpenCL runtime and the second part repeats the Nvidia SDK install in the Intel CPU example.

```
FROM debian:stretch-slim  
  
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \  
apt-get install -y --no-install-recommends \  
ocl-icd-libopencl1 ocl-icd-opencl-dev libnuma1 clinfo \  
libxml2 libxmu-dev freeglut3-dev libglew-dev \  
libgl1-mesa-glx libgl1-mesa-dri wget ca-certificates \  
bzip2 tar gcc g++ make && \  
mkdir -p /opt/amd/opencl/install && \  
cd /opt/amd/opencl/install && \  
wget -O AMD-APP-SDKInstaller-v3.0.130.136-GA-linux64.tar.bz2  
"https://archive.org/download/AMDAPPSDK/AMD-APP-SDKInstaller-v3.  
.0.130.136-GA-linux64.tar.bz2" && \  
tar xjf AMD-APP-SDKInstaller-v3.0.130.136-GA-linux64.tar.bz2  
&& \  
 &&
```

```
./AMD-APP-SDK-v3.0.130.136-GA-linux64.sh --tar xvfp && \
mv lib/x86_64/libamdocl12cl64.so /opt/amd/opencl && \
mv lib/x86_64/sdk/libamdocl64.so /opt/amd/opencl && \
mkdir -p /etc/OpenCL/vendors && \
echo "/opt/amd/opencl/libamdocl64.so" > \
/etc/OpenCL/vendors/amdocl64.icd && \
# Download Nvidia SDK which includes the OpenCL samples.
# The samples at https://developer.nvidia.com/opencl
# are incomplete, missing the "shared" and "common" code
# needed to compile! We therefore pull the 4.2.9 SDK.
cd /usr/local/src && \
wget -O gpucomputingsdk_4.2.9_linux.run "https://developer.do
wnload.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.
9_linux.run" && \
chmod +x gpucomputingsdk_4.2.9_linux.run && \
./gpucomputingsdk_4.2.9_linux.run --tar xfp && \
rm gpucomputingsdk_4.2.9_linux.run && \
rm install-sdk-linux.pl && \
mv sdk/OpenCL sdk/shared . && \
rm -rf sdk && rm -rf shared/lib/linux && \
sed -i 's/GLEW_x86_64/GLEW/' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
sed -i 's/-Wimplicit//' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
# Fix issue with oclNbodyKernel.cl
sed -i 's/mul24(get_local_size(0), get_local_id(1))/mul24((ui
nt)get_local_size(0), get_local_id(1))/' /usr/local/src/OpenCL/
src/oclNbody/oclNbodyKernel.cl && \
# Fix issue with SobelFilter.cl
sed -i 's/mul24(get_local_size(1), get_global_size(0))/mul24(
(uint)get_local_size(1), get_global_size(0))/' /usr/local/src/O
penCL/src/oclSobelFilter/SobelFilter.cl && \
sed -i 's/mul24((get_group_id(0) + 1)/mul24((uint)(get_group_
id(0) + 1)/* /usr/local/src/OpenCL/src/oclSobelFilter/SobelFilt
er.cl && \
```

```

sed -i 's/mul24(get_group_id(0)/mul24((uint)get_group_id(0) '/'
/usr/local/src/OpenCL/src/oclSobelFilter/SobelFilter.cl && \
# Replace CL_DEVICE_TYPE_GPU with CL_DEVICE_TYPE_CPU
# so the samples will work on Intel CPU device.
for i in $(grep -rl /usr/local/src/OpenCL/src -e CL_DEVICE_T
YPE_GPU); do sed -i 's/CL_DEVICE_TYPE_GPU/CL_DEVICE_TYPE_CPU/' \
$i; done && \
cd OpenCL && \
mv src/oclNbody . && rm -rf src/* && \
mv oclNbody src/. && \
make && \
chmod 777 /usr/local/bin && \
mkdir -p /usr/local/bin/src && \
cp /usr/local/src/OpenCL/bin/linux/release/oclNbody \
/usr/local/bin/. && \
cp /usr/local/src/OpenCL/src/oclNbody/*.cl \
/usr/local/bin/src/. && \
rm -rf /opt/amd/opencl/install && \
rm -rf /usr/local/src/* && \
apt-get clean && \
apt-get purge -y wget bzip2 ca-certificates \
gcc g++ make && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

# Uses Nvidia's shrFindFilePath to find kernel directory
# and using the absolute path doesn't work correctly.
WORKDIR /usr/local/bin/
ENTRYPOINT ["oclNbody"]

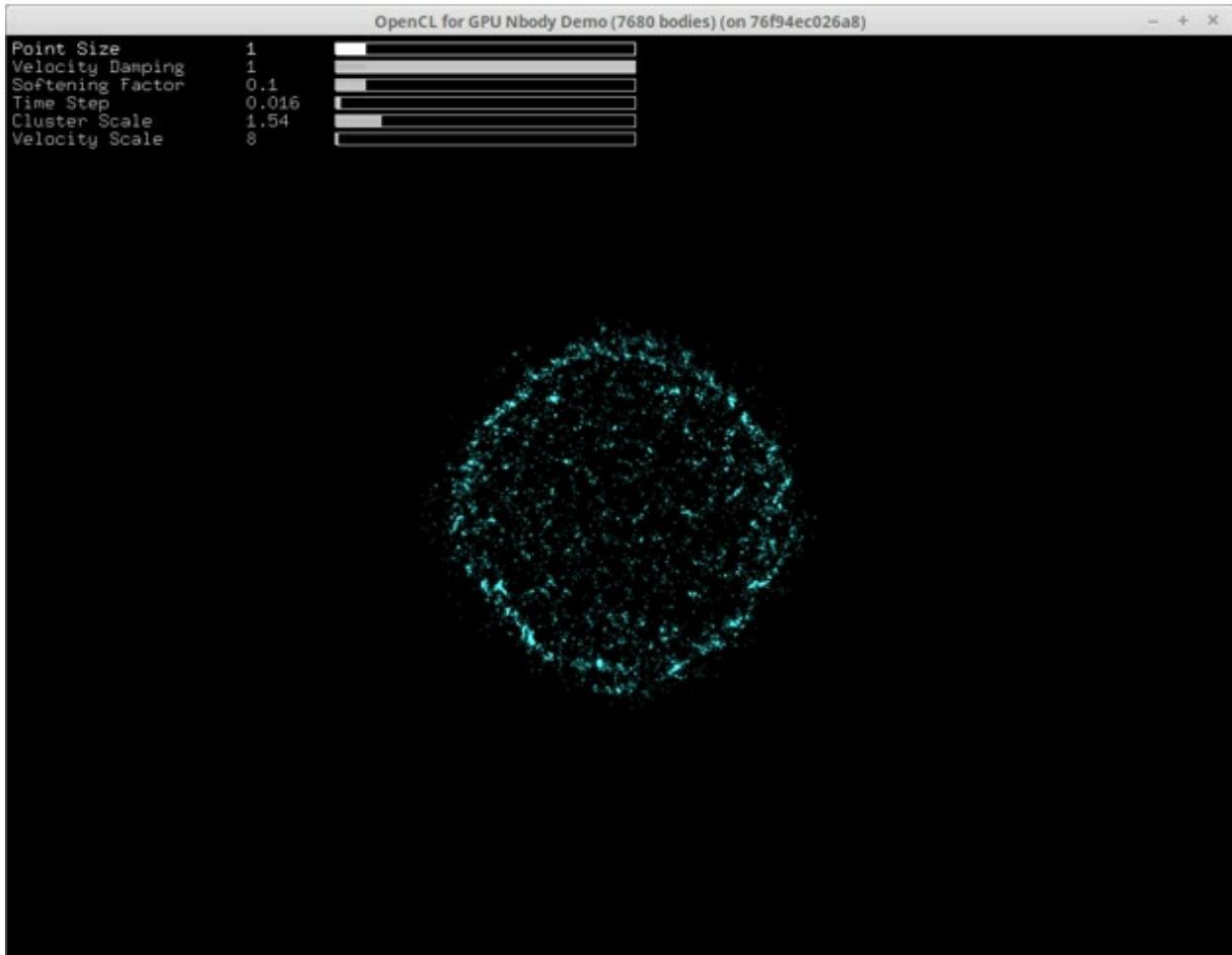
```

The opencl-nbody-amd-app image may be built with:

```
docker build -t opencl-nbody-amd-app -f Dockerfile-amd-app .
```

The [opencl-nbody-amd-app.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-gpu.sh  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$X11_FLAGS \  
$GPU_FLAGS \  
opencl-nbody-amd-app $@
```



5-3-6 opencl-particles

The examples in this section follow the same patterns employed in the [opencl-nbody](#) section for installing the OpenCL runtime and Nvidia OpenCL SDK, so will simply be presented here without further explanation.

Nvidia

The [Dockerfile](#) may be found in the [opencl-particles](#) directory of the [docker-gui](#) repository under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ocl-icd-libopencl1 ocl-icd-opencl-dev clinfo tar \
    libxmu-dev freeglut3-dev libglew-dev libgl1-mesa-glx \
    libgl1-mesa-dri wget ca-certificates gcc g++ make && \
# Add the Nvidia OpenCL Installable Client Driver
mkdir -p /etc/OpenCL/vendors && \
echo "libnvidia-opencl.so.1" > \
    /etc/OpenCL/vendors/nvidia.icd && \
# Download Nvidia SDK which includes the OpenCL samples.
# The samples at https://developer.nvidia.com/opencl
# are incomplete, missing the "shared" and "common" code
# needed to compile! We therefore pull the 4.2.9 SDK.
cd /usr/local/src && \
wget -O gpucomputingsdk_4.2.9_linux.run "https://developer.do
wnload.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.
9_linux.run" && \
chmod +x gpucomputingsdk_4.2.9_linux.run && \
./gpucomputingsdk_4.2.9_linux.run --tar xfp && \
```

```

rm gpucomputingsdk_4.2.9_linux.run && \
rm install-sdk-linux.pl && \
mv sdk/OpenCL sdk/shared . && \
rm -rf sdk && rm -rf shared/lib/linux && \
sed -i 's/GLEW_x86_64/GLEW/' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
sed -i 's/-Wimplicit//' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
cd OpenCL && \
mv src/oclParticles . && rm -rf src/* && \
mv oclParticles src/. && \
make && \
chmod 777 /usr/local/bin && \
mkdir -p /usr/local/bin/src && \
cp /usr/local/src/OpenCL/bin/linux/release/oclParticles \
/usr/local/bin/. && \
cp /usr/local/src/OpenCL/src/oclParticles/*.cl \
/usr/local/bin/src/. && \
# Remove packages used for installation
rm -rf /usr/local/src/* && \
apt-get clean && \
apt-get purge -y wget ca-certificates gcc g++ make && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

# Uses Nvidia's shrFindFilePath to find kernel directory
# and using the absolute path doesn't work correctly.
WORKDIR /usr/local/bin/
ENTRYPOINT ["oclParticles"]

```

and may be built with:

```
docker build -t opencl-particles .
```

The [opencl-particles.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
$GPU_FLAGS \
opencl-particles $@
```

Intel CPU

The [Dockerfile](#) may be found in the [opencl-particles](#) directory of the [docker-gui](#) repository under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

# Driver Version 18.1.0.0920
ENV RUNTIME l_opencl_p_18.1.0.013

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends \
ocl-icd-libopencl1 ocl-icd-opencl-dev libnuma1 clinfo \
tar libxml2 libxmu-dev freeglut3-dev libglew-dev \
libgl1-mesa-glx libgl1-mesa-dri wget ca-certificates \
rpm2cpio cpio gcc g++ make && \
# Add the Intel OpenCL Installable Client Driver
wget -O ${RUNTIME}.tgz "http://registrationcenter-download.intel.com/akdlm/irc_nas/13793/${RUNTIME}.tgz" && \
tar zxfp ${RUNTIME}.tgz && \
cd ${RUNTIME}/rpm && \
rpm2cpio intel-openclrt-18.1.0.013-18.1.0-013.x86_64.rpm | cp \
io -idmv && \
mkdir -p /opt/intel && \
```

```
mv ./opt/intel/opencl_compilers_and_libraries_18.1.0.013/linux/compiler/lib /opt/intel/opencl && \
mkdir -p /etc/OpenCL/vendors && \
echo "/opt/intel/opencl/intel64_lin/libintelocl.so" > \
/etc/OpenCL/vendors/intel_cpu.icd && \
# Download Nvidia SDK which includes the OpenCL samples.
# The samples at https://developer.nvidia.com/opencl
# are incomplete, missing the "shared" and "common" code
# needed to compile! We therefore pull the 4.2.9 SDK.
cd /usr/local/src && \
wget -O gpucomputingsdk_4.2.9_linux.run "https://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.9_linux.run" && \
chmod +x gpucomputingsdk_4.2.9_linux.run && \
./gpucomputingsdk_4.2.9_linux.run --tar xfp && \
rm gpucomputingsdk_4.2.9_linux.run && \
rm install-sdk-linux.pl && \
mv sdk/OpenCL sdk/shared . && \
rm -rf sdk && rm -rf shared/lib/linux && \
sed -i 's/GLEW_x86_64/GLEW/' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
sed -i 's/-Wimplicit//' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
# Replace CL_DEVICE_TYPE_GPU with CL_DEVICE_TYPE_CPU
for i in $(grep -rl /usr/local/src/OpenCL/src -e CL_DEVICE_TYPE_GPU); do sed -i 's/CL_DEVICE_TYPE_GPU/CL_DEVICE_TYPE_CPU/' $i; done && \
cd OpenCL && \
mv src/oclParticles . && rm -rf src/* && \
mv oclParticles src/. && \
make && \
chmod 777 /usr/local/bin && \
mkdir -p /usr/local/bin/src && \
cp /usr/local/src/OpenCL/bin/linux/release/oclParticles \
/usr/local/bin/. && \
cp /usr/local/src/OpenCL/src/oclParticles/*.cl \
/usr/local/bin/src/. && \
```

```

# Remove packages used for installation
rm /${RUNTIME}.tgz && \
rm -rf /${RUNTIME} && \
rm -rf /usr/local/src/* && \
apt-get clean && \
apt-get purge -y wget ca-certificates rpm2cpio cpio \
    gcc g++ make && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

# Uses Nvidia's shrFindFilePath to find kernel directory
# and using the absolute path doesn't work correctly.
WORKDIR /usr/local/bin/
ENTRYPOINT ["oclParticles"]

```

and may be built with:

```
docker build -t opencl-particles-intel-cpu -f Dockerfile-intel-cpu-18.1.0 .
```

The [opencl-particles-intel-cpu.sh](#) launch script is:

```

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    $X11_FLAGS \
    $GPU_FLAGS \
    opencl-particles-intel-cpu $@

```

AMD APP

The [Dockerfile](#) may be found in the [opencl-particles](#) directory of the [docker-gui](#) repository under [5-more-complex-applications/compute](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ocl-icd-libopencl1 ocl-icd-opencl-dev libnuma1 clinfo \
    libxmu-dev freeglut3-dev libglew-dev libgl1-mesa-glx \
    libgl1-mesa-dri wget bzip2 ca-certificates tar gcc g++ \
    make && \
    mkdir -p /opt/amd/opencl/install && \
    cd /opt/amd/opencl/install && \
    wget -O AMD-APP-SDKInstaller-v3.0.130.136-GA-linux64.tar.bz2 \
    "https://archive.org/download/AMDAPPSDK/AMD-APP-SDKInstaller-v3.0.130.136-GA-linux64.tar.bz2" && \
    tar xjf AMD-APP-SDKInstaller-v3.0.130.136-GA-linux64.tar.bz2 \
&& \
    ./AMD-APP-SDK-v3.0.130.136-GA-linux64.sh --tar xvfp && \
    mv lib/x86_64/libamdocl12cl64.so /opt/amd/opencl && \
    mv lib/x86_64/sdk/libamdocl64.so /opt/amd/opencl && \
    # Add ICD to the ICD registry
    mkdir -p /etc/OpenCL/vendors && \
    echo "/opt/amd/opencl/libamdocl64.so" > \
        /etc/OpenCL/vendors/amdocl64.icd && \
    # Download Nvidia SDK which includes the OpenCL samples.
    # The samples at https://developer.nvidia.com/opencl
    # are incomplete, missing the "shared" and "common" code
    # needed to compile! We therefore pull the 4.2.9 SDK.
    cd /usr/local/src && \
    wget -O gpucomputingsdk_4.2.9_linux.run "https://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.9_linux.run" && \
    chmod +x gpucomputingsdk_4.2.9_linux.run && \
    ./gpucomputingsdk_4.2.9_linux.run --tar xfp && \
    rm gpucomputingsdk_4.2.9_linux.run && \
    rm install-sdk-linux.pl && \
```

```

mv sdk/OpenCL sdk/shared . && \
rm -rf sdk && rm -rf shared/lib/linux && \
sed -i 's/GLEW_x86_64/GLEW/' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
sed -i 's/-Wimplicit//' \
/usr/local/src/OpenCL/common/common_opencl.mk && \
# Replace CL_DEVICE_TYPE_GPU with CL_DEVICE_TYPE_CPU
for i in $(grep -rl /usr/local/src/OpenCL/src -e CL_DEVICE_T
YPE_GPU); do sed -i 's/CL_DEVICE_TYPE_GPU/CL_DEVICE_TYPE_CPU/' \
$i; done && \
cd OpenCL && \
mv src/oclParticles . && rm -rf src/* && \
mv oclParticles src/. && \
make && \
chmod 777 /usr/local/bin && \
mkdir -p /usr/local/bin/src && \
cp /usr/local/src/OpenCL/bin/linux/release/oclParticles \
/usr/local/bin/. && \
cp /usr/local/src/OpenCL/src/oclParticles/*.cl \
/usr/local/bin/src/. && \
rm -rf /opt/amd/opencl/install && \
rm -rf /usr/local/src/* && \
apt-get clean && \
apt-get purge -y wget bzip2 ca-certificates gcc g++ \
make && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

# Uses Nvidia's shrFindFilePath to find kernel directory
# and using the absolute path doesn't work correctly.
WORKDIR /usr/local/bin/
ENTRYPOINT ["oclParticles"]

```

and may be built with:

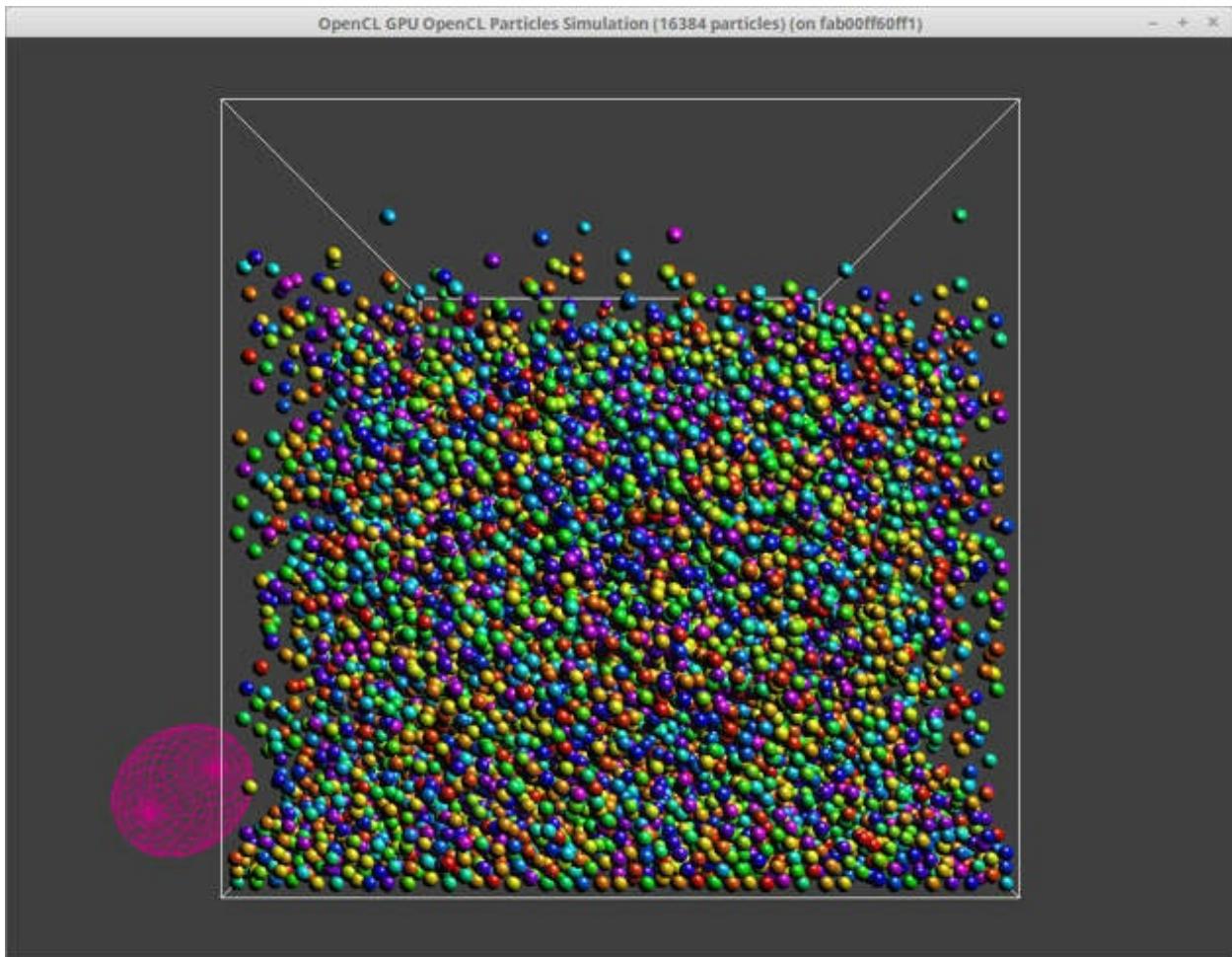
```

docker build -t opencl-particles-amd-app -f Dockerfile-amd-app
.
```

The [opencl-particles-amd-app.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
$GPU_FLAGS \
opencl-particles-amd-app $@
```



5-4 Video and Graphics

This section aims to reinforce what we have learned so far by covering a range of sophisticated graphics applications. We shall show that it is possible containerise almost anything relatively simply by using our [Modular Scripting](#) to reduce much of the Docker boilerplate.

5-4-1 gimp

No book on graphical applications would be complete without including the GNU Image Manipulation Program ([GIMP](#)). GIMP is one of the leading image editor applications and is available for a range of platforms. As we shall show here it is relatively simple to run GIMP in a Docker container too.

The [Dockerfile](#) is in the `gimp` directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    gimp && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["gimp"]
```

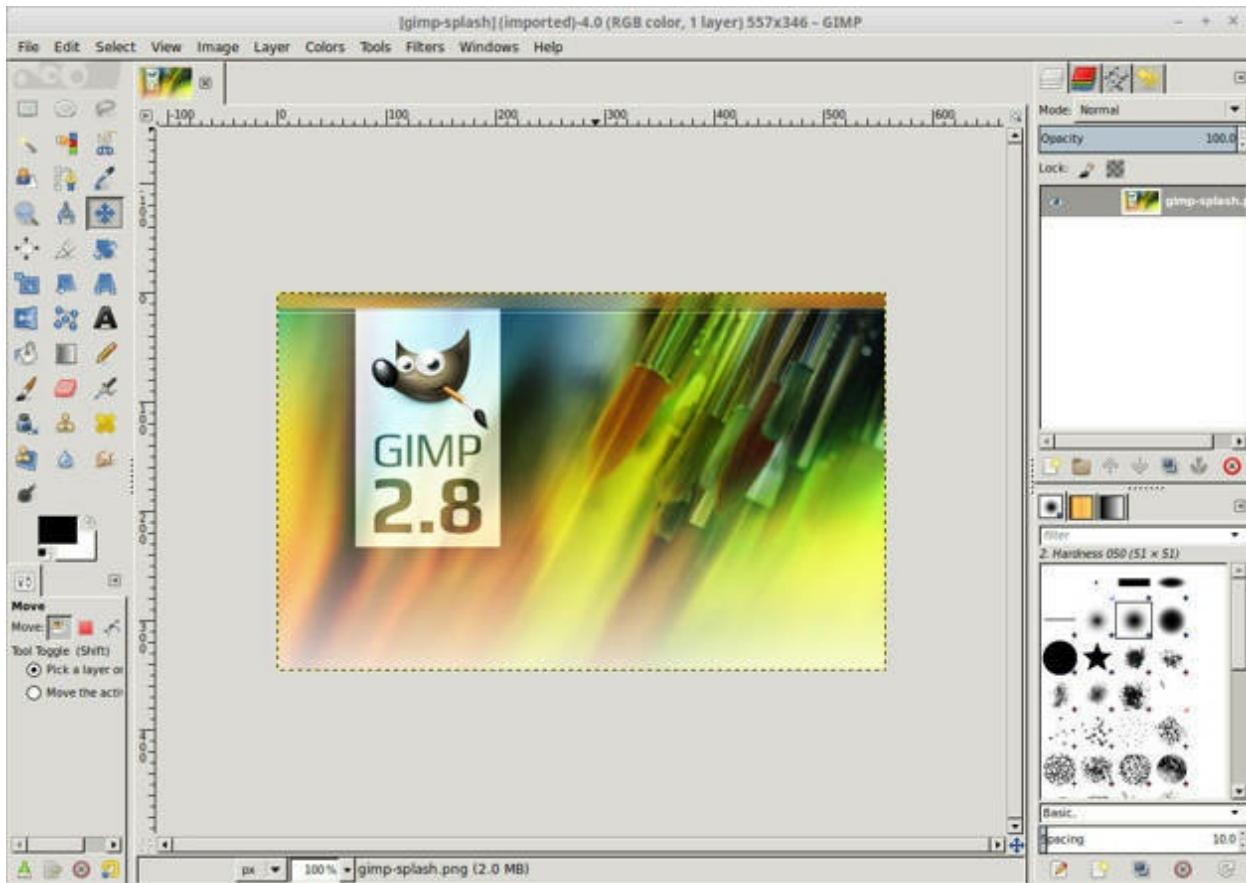
The gimp image may be built with:

```
docker build -t gimp .
```

The [gimp.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$X11_FLAGS \
gimp $@
```



5-4-2 inkscape

To complement the powerful [raster graphics](#) capabilities of GIMP, by contrast [Inkscape](#) is a free and open-source professional quality [vector graphics](#) editor with features similar to Adobe Illustrator and Corel Draw.

The [Dockerfile](#) may be found in the [inkscape](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM debian:stretch-slim

# Install inkscape
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    inkscape && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["inkscape"]
```

The inkscape image may be built with:

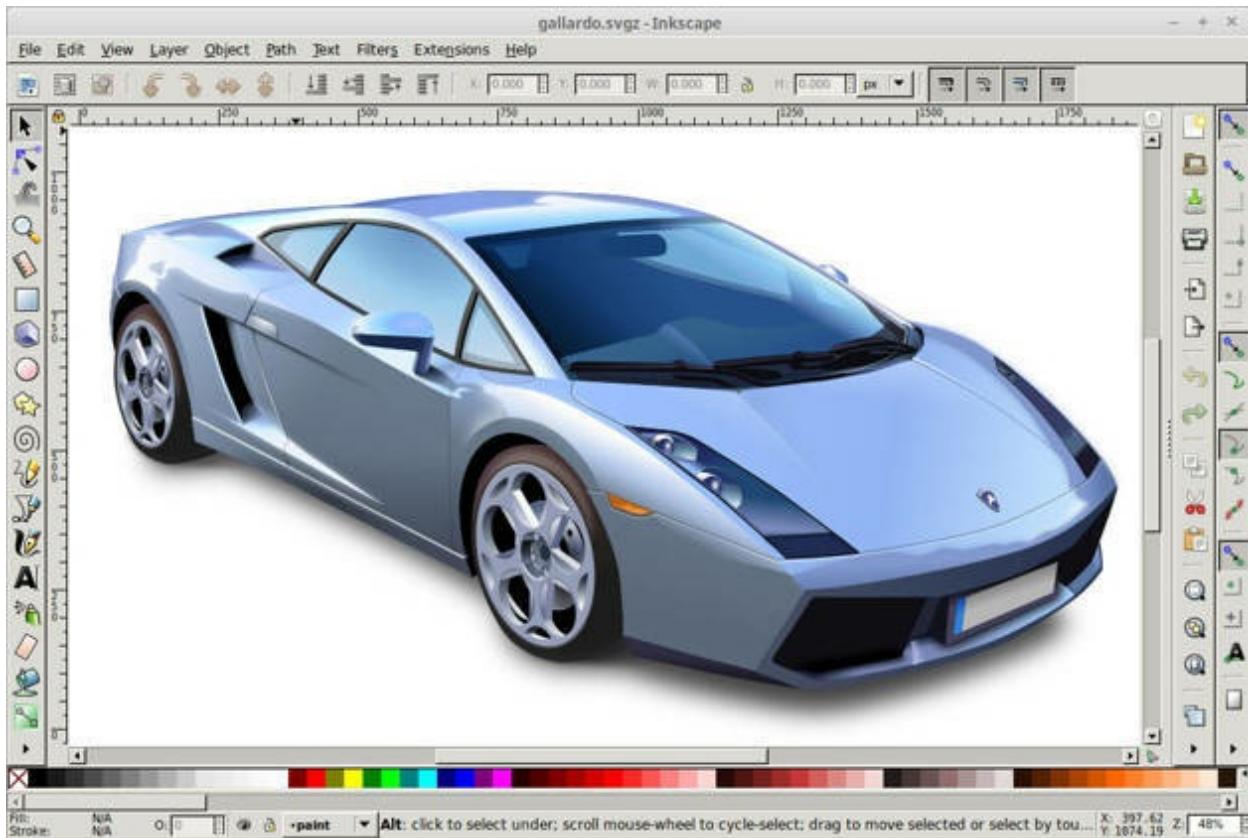
```
docker build -t inkscape .
```

Due to the powerful dependency tracking capabilities of apt we can see that the Dockerfile for Inkscape is more or less identical to that for GIMP and illustrates just how simple it can be to containerise some applications. Running the application is almost as simple too, though note that Inkscape requires an additional \$HOME/.local/share directory for it to store state such as references to recently used files.

The [inkscape.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/dconf
mkdir -p $(id -un)/.local/share
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$X11_FLAGS \
inkscape $@
```



5-4-3 guvcview

In the [Local Applications](#) section, we previously covered the simple GNOME webcam application [camorama](#). Here we cover [guvcview](#), a rather more comprehensive webcam application that also supports audio yet still uses fewer resources than many other webcam applications.

The [Dockerfile](#) may be found in the [guvcview](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#). Note the mesa libraries to support OpenGL and pulseaudio for audio support in addition to the basic guvcview package.

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install guvcview
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libcanberra-gtk-module libcanberra-gtk3-module \
    guvcview pulseaudio libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["guvcview"]
```

The guvcview image may be built with:

```
docker build -t guvcview .
```

The [guvcview.sh](#) launch script is:

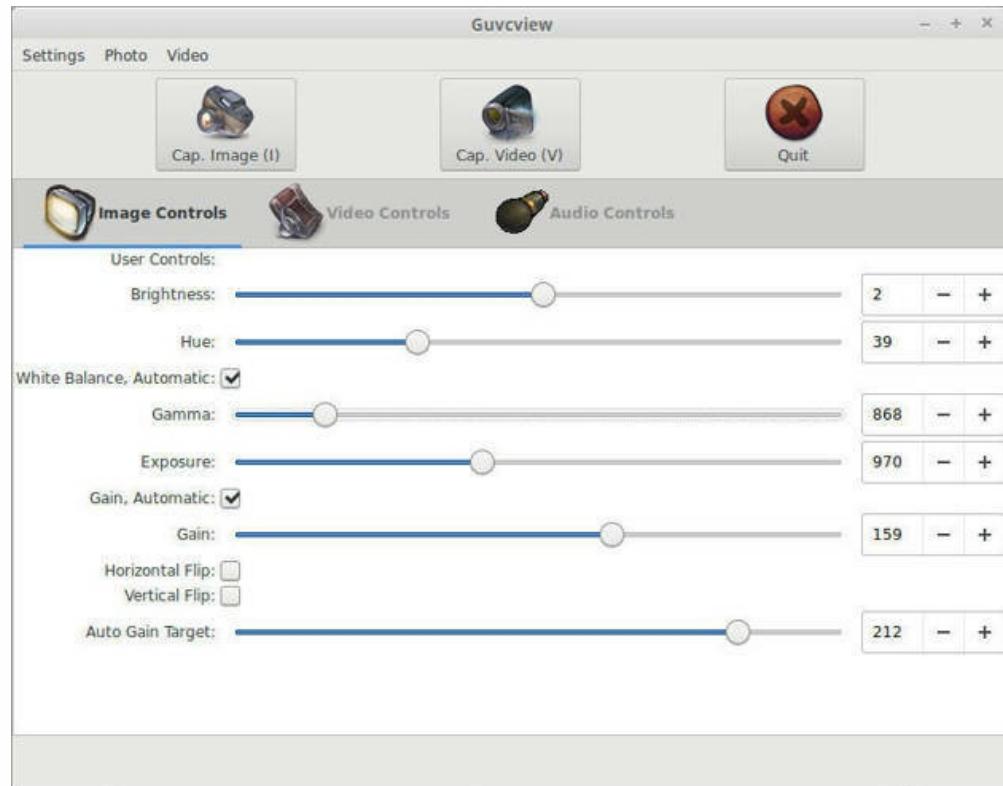
```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh
. $BIN/docker-webcam.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    $WEBCAM_FLAGS \
    guvcview $@
```

Although in many ways guvcview is a relatively simple application we can see from the launch script that *in Docker terms* it is actually quite sophisticated. We can see above everything from dconf/D-bus integration, PulseAudio, OpenGL/GPU support and webcam connectivity being used to integrate this application fully with the user's desktop.

To avoid ALSA errors run guvcview with:

```
./guvcview.sh -a pulse
```



5-4-4 vlc

In addition to the range of graphical applications illustrated thus far it is also possible to deploy fully featured, GPU accelerated, video players into Docker containers. VLC media player ([VLC](#)) is one of the most popular cross-platform media players and supports a wide range of audio and video compression methods and file formats including DVD video.

The [Dockerfile](#) is in the [vlc](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM debian:stretch-slim
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    vlc apt-transport-https wget gnupg ca-certificates \
    libv4l-0 libvdpau1 mesa-vdpau-drivers \
    libgl1-mesa-glx libgl1-mesa-dri && \
    wget -O - https://download.videolan.org/pub/debian/videolan-a
pt.asc | apt-key add - && \
    echo "deb [arch=amd64] https://download.videolan.org/pub/debi
an/stable/" > /etc/apt/sources.list.d/videolan.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libdvdcss2 && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["vlc"]
```

The vlc image may be built with:

```
docker build -t vlc .
```

In addition to the normal vlc package dependencies, in this Dockerfile we install the videolan repository so we are able to install the libdvdcss2 library needed to play DVD movies:

```
wget -O - https://download.videolan.org/pub/debian/videolan-apt.asc | apt-key add -
echo "deb [arch=amd64] https://download.videolan.org/pub/debian/stable/ /" > /etc/apt/sources.list.d/videolan.list
```

Note that adding the videolan repository is a necessary prerequisite for installing libdvdcss2 on Debian and is therefore included here to illustrate the process. Some other distributions however, such as Ubuntu and its derivatives, already have libdvdcss2 in their repositories by default.

In addition to enabling DVD support another notable feature of this Dockerfile is the addition of libv4l0 as VLC supports webcams and other video devices and we wish to make our containerised VLC as full-featured as possible.

Finally, we include libvdpau1 and mesa-vdpau-drivers to enable [VDPAU](#) support in the container. VDPAU is the Video Decode and Presentation API for Unix and gives applications standardised access to available video decompression acceleration hardware such as Nvidia's PureVideo or AMD's Unified Video Decoder.

The [vlc.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh
. $BIN/docker-webcam.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    --device=/dev/sr0 \
    --group-add $(cut -d: -f3 < <(getent group cdrom)) \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    $WEBCAM_FLAGS \
    vlc $@
```

Like our earlier [guvcview](#) this launch script contains a wide range of integration flags aimed at making VLC as full-featured and close to “native” as possible. If particular features are not required by the reader it is clearly a simple matter to remove those from the launch script.

Note the first two lines giving access to the DVD device:

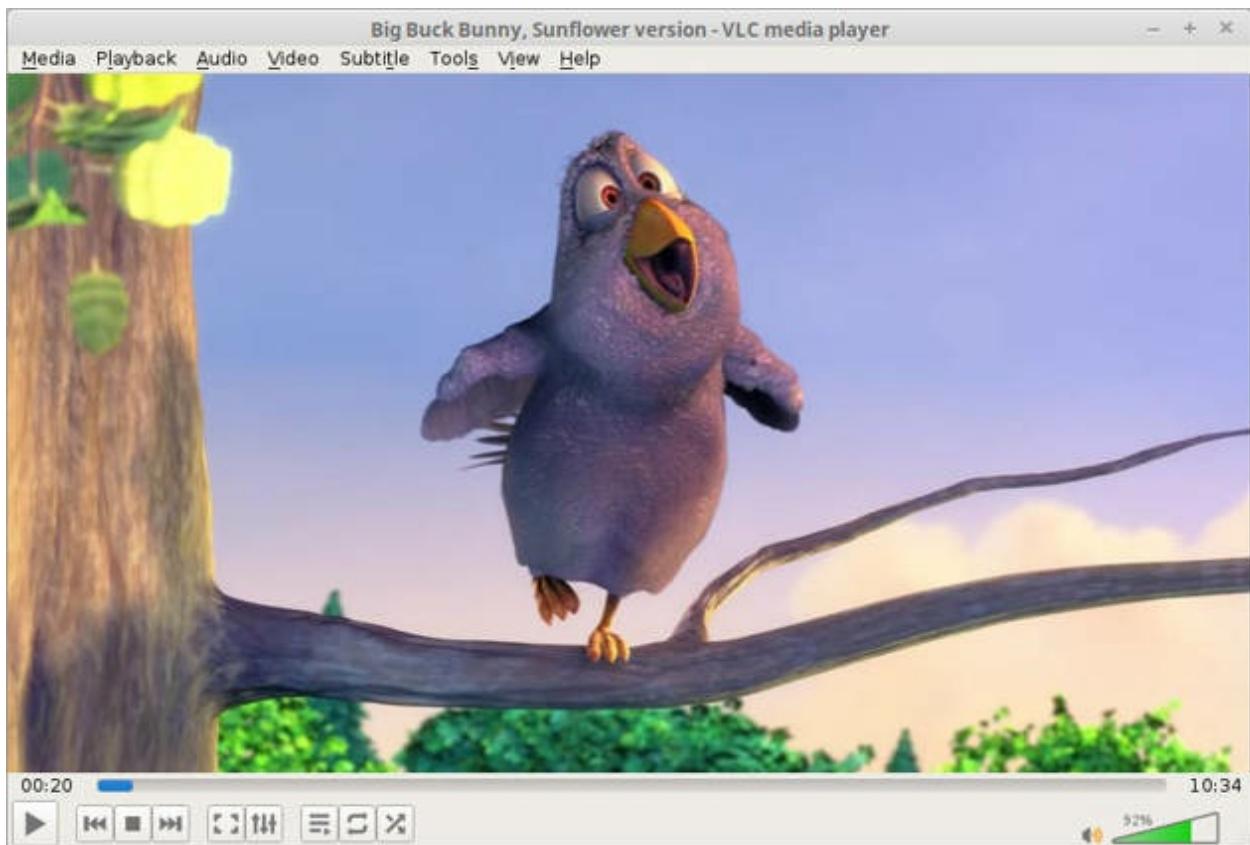
```
--device=/dev/sr0 \
--group-add $(cut -d: -f3 < <(getent group cdrom)) \
```

In the author's case VLC logs the following message when playing DVD videos indicating that Nvidia hardware decoding is being used:

```
[00007f06dcc0a500] avcodec decoder: Using NVIDIA VDPAU Driver Shared Library 384.130 Wed Mar 21 02:59:00 PDT 2018 for hardware decoding
```

and on a laptop equipped with an AMD GPU VLC logs the following message, the initial absence of which prompted the author to realise the need for the mesa-vdpau-drivers package to be included in the Dockerfile.

```
[00007f11f008f9a0] avcodec decoder: Using G3DVL VDPAU Driver Shared Library version 1.0 for hardware decoding
```



5-4-5 smplayer

[SMPlayer](#) is another multi-platform media player, this time built with Qt and based on MPlayer.

The [Dockerfile](#) may be found in the [smplayer](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install smplayer
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
        apt-transport-https wget gnupg ca-certificates \
        smplayer smtube mplayer libvdpau1 mesa-vdpau-drivers \
        libgl1-mesa-glx libgl1-mesa-dri && \
        wget -O - https://download.videolan.org/pub/debian/videolan-a
pt.asc | apt-key add - && \
    echo "deb [arch=amd64] https://download.videolan.org/pub/debi
an/stable/ /" > /etc/apt/sources.list.d/videolan.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
        libdvdcss2 && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["smplayer"]
```

As with our VLC example, our SMPlayer Dockerfile adds the videolan repository so we can install libdvdcss2 for DVD support and adds libvdpau1 and mesa-vdpau-drivers to enable VDPAU hardware accelerated video decoding.

The smplayer image may be built with:

```
docker build -t smplayer .
```

The [smplayer.sh](#) launch script is:

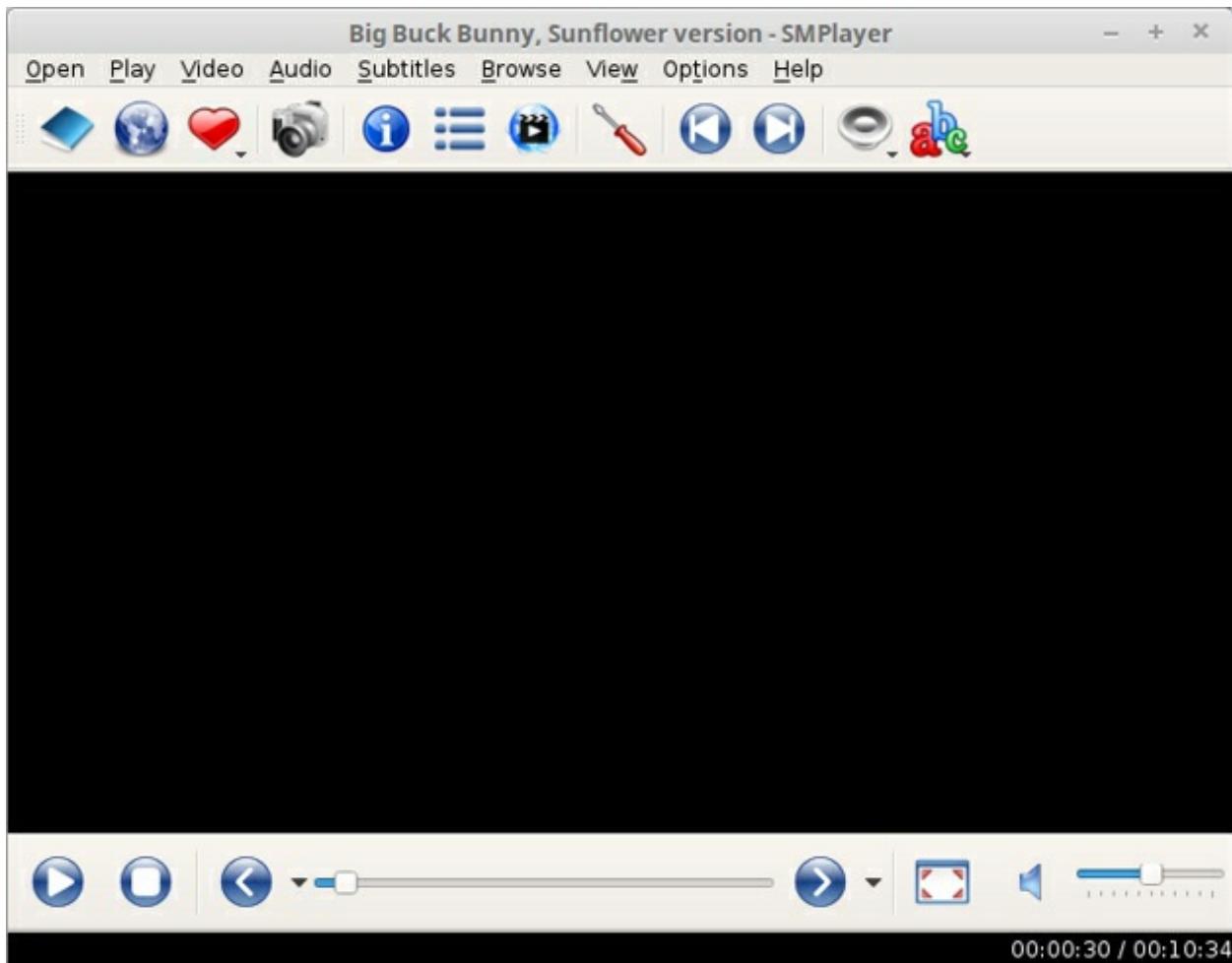
```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
$DOCKER_COMMAND run --rm \
    --device=/dev/sr0 \
    --group-add $(cut -d: -f3 < <(getent group cdrom)) \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    smplayer $@
```

This is very similar to the launch script for VLC, however as SMPlayer is a Qt based application it doesn't need D-bus/dconf integration nor do we include webcam support.

With SMPlayer it is somewhat harder than was the case with VLC to verify that VDPAU acceleration is indeed being used, as there is no terminal logging such as we saw previously.

We can however open both the Mplayer and SMPlayer logs via the View menu and moreover in the Preferences menu in the Video tab of General settings it is possible to explicitly set the output driver to VDPAU.



5-4-6 kodi

Whereas VLC and SMPlayer are media players [Kodi](#) bills itself more as a multi-platform home theatre PC (HTPC) application featuring a “10-foot user interface” for use with televisions and remote controls.

The [Dockerfile](#) is in the [kodi](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    apt-transport-https wget gnupg ca-certificates \
    kodi libvdpau1 mesa-vdpau-drivers \
    libgl1-mesa-glx libgl1-mesa-dri && \
    wget -O - https://download.videolan.org/pub/debian/videolan-a
pt.asc | apt-key add - && \
    echo "deb [arch=amd64] https://download.videolan.org/pub/debi
an/stable/ /" > /etc/apt/sources.list.d/videolan.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libdvdcss2 && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["kodi"]
```

As with VLC and SMPlayer, our Kodi Dockerfile adds the videolan repository so we can install libdvdcss2 for DVD support and adds libvdpau1 and mesa-vdpau-drivers to enable VDPAU hardware accelerated video decoding.

The kodi image may be built with:

```
docker build -t kodi .
```

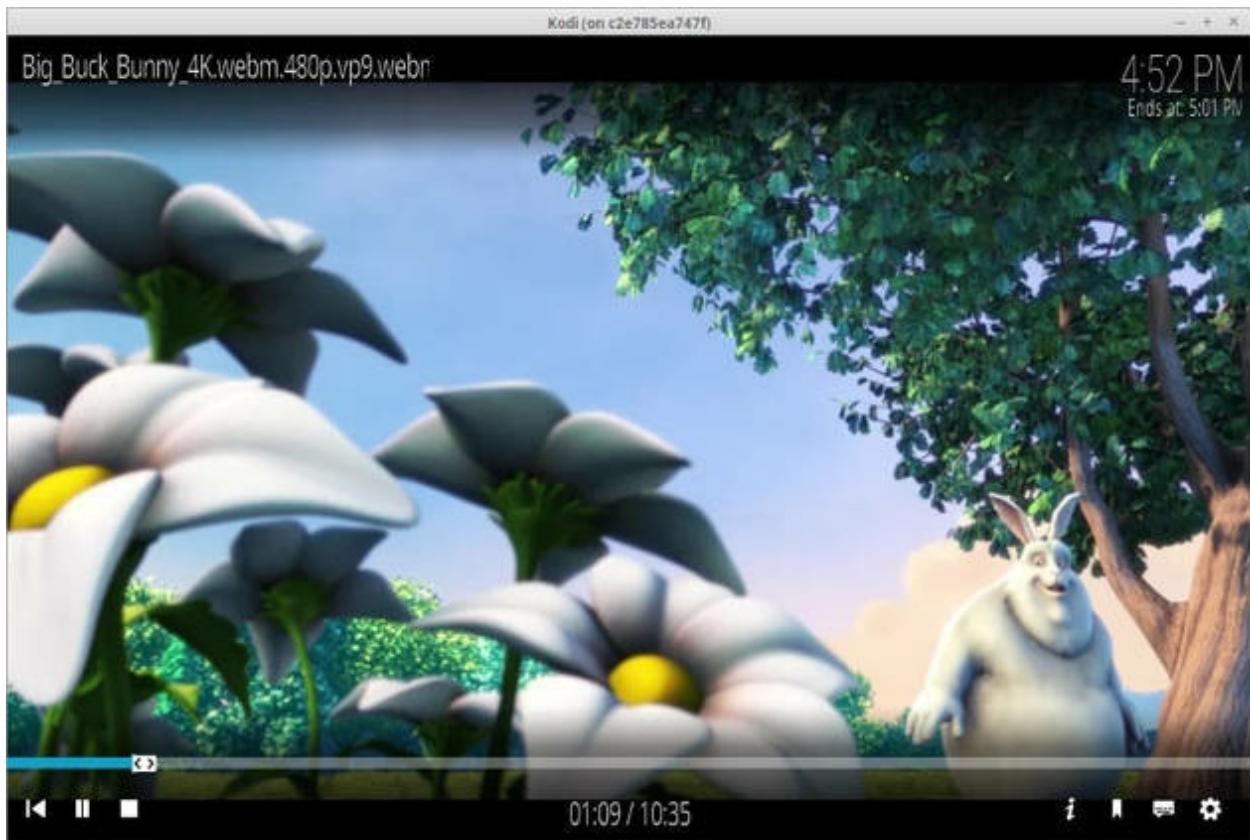
The [kodi.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
$DOCKER_COMMAND run --rm \
    --device=/dev/sr0 \
    --group-add $(cut -d: -f3 < <(getent group cdrom)) \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    kodi $@
```

This launch script is more or less identical to the one used for SMPlayer except for the image name, again we don't need D-bus or webcam integration for Kodi, but it makes a very nice DVD player so we do bind-mount /dev/sro.

As mentioned in the preamble unlike VLC and SMPlayer Kodi is really primarily intended as an HTPC application rather than a desktop application, so be aware that by default it launches in full-screen mode with its “10-foot user interface”. Kodi is extremely configurable however and if so desired it is also possible to change the settings to enable it to display in windowed mode:



5-4-7 Blu-ray

Although several DVD video applications are available for Linux, the ability to play legally purchased Blu-ray discs has proved rather more elusive. In this section we shall explore how Docker can help automate the somewhat tedious hoops that are necessary to jump through to play Blu-ray discs.

Before we begin there are a few important points to note about the process. The first point is that this process is primarily intended to demonstrate how Docker can be used to overcome awkward problems and is intended only to be used for the purposes of watching one's own legally purchased Blu-ray discs. The author absolutely does not condone the use of anything presented in this section for any other purpose.

The next point is that the process makes use of libraries from the [MakeMKV](#) application. It is important to note that MakeMKV is not a free and open-source application, however it is available from its authors as a free “beta” version.

Finally, in order to simplify the process of installing MakeMKV and also for a little variety, we use an Ubuntu 18.04 base image and install MakeMKV via a [PPA](#).

The use of non-free beta software from a PPA archive and some symbolic link library manipulation make this application an ideal candidate to run in a container. Moreover, as the MakeMKV beta licence expires fairly regularly it might be necessary to rebuild the image periodically if the application fails to play previously working Blu-ray discs.

The [Dockerfile](#) may be found in the [bluray](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM ubuntu:18.04

LABEL com.nvidia.volumes.needed=nvidia_driver

# Install vlc and makemkv
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    software-properties-common wget gnupg \
    vlc libvdpau1 mesa-vdpau-drivers \
    libgl1-mesa-glx libgl1-mesa-dri && \
    wget -O - https://download.videolan.org/pub/debian/videolan-a
pt.asc | apt-key add - && \
    echo "deb [arch=amd64] https://download.videolan.org/pub/debi
an/stable/" > /etc/apt/sources.list.d/videolan.list && \
    add-apt-repository ppa:heyarje/makemkv-beta && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y libdvdcss2 \
    makemkv-bin makemkv-oss && \
    rm -rf /var/lib/apt/lists/* && \
# https://www.makemkv.com/forum/viewtopic.php?t=7009
    cd /usr/lib/x86_64-linux-gnu && \
    ln -s libmmbd.so.0 libaacs.so.0 && \
    ln -s libmmbd.so.0 libbdplus.so.0 && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["vlc"]
```

and may be built with:

```
docker build -t vlc-bluray .
```

As with our previous Debian VLC Dockerfile this adds the videolan repository so we can install libdvdcss2 for DVD support and adds libvdpau1 and mesa-vdpau-drivers to enable VDPAU hardware accelerated video decoding. In addition to videolan we add the ppa:heyarje/makemkv-beta PPA to install MakeMKV and later, following instructions from

<https://www.makemkv.com/forum/viewtopic.php?t=7009> we make symlinks from libaacs and libbdplus to libmmbd.

```
ln -s libmmbd.so.0 libaacs.so.0
ln -s libmmbd.so.0 libbdplus.so.0
```

The [vlc-bluray.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    --device=/dev/sr0 \
    --device=/dev/sg1 \
    --group-add $(cut -d: -f3 < <(getent group cdrom)) \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    vlc-bluray
```

5-4-8 frameworks graphics samples

These examples illustrate the use of Docker to perform reproducible builds of the Nvidia GameWorks Graphics Samples <https://github.com/NVIDIAGameWorks/GraphicsSamples>. Note that these examples require a recent OpenGL version.

The [Dockerfile](#) is in the [frameworks-graphics-samples](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    git ca-certificates build-essential g++ \
    libxinerama-dev libxext-dev libxrandr-dev libxi-dev \
    libxcursor-dev libxxf86vm-dev libvulkan-dev \
    libgl1-mesa-dri libgl1-mesa-dev && \
    cd /usr/src && \
    git clone https://github.com/NVIDIAGameWorks/GraphicsSamples.
git && \
    cd GraphicsSamples && \
    # debian:stretch Vulkan version is 1.0.39.0 so later
    # versions of GameWorks won't compile with stretch.
    git checkout 61350521f6b183d9f694f5c72c11efcf1a0cc665 -b build && \
    cd samples/build/linux64 && \
    make -j$(getconf _NPROCESSORS_ONLN) release && \
    rm -rf build && \
    apt-get clean && \
    apt-get purge -y \
    git ca-certificates build-essential g++ && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*
WORKDIR /usr/src/GraphicsSamples/samples/bin/linux64
```

The gameworks-graphics-samples image may be built with:

```
docker build -t gameworks-graphics-samples .
```

Whilst most of the Dockerfiles in this book build their images by pulling **binary** packages from the main repositories of the base distributions this example is slightly different. Here we are actually initially pulling a *build environment* and development libraries from the repositories, as we have to build GameWorks ourselves directly from the source code.

The main part of this Dockerfile performs a `git clone` of the GameWorks Git repository and then checks out a specific version, because later versions of GameWorks require Vulkan 1.1 and won't compile directly on `debian:stretch`.

After checking out the required version we next perform a parallel build via:

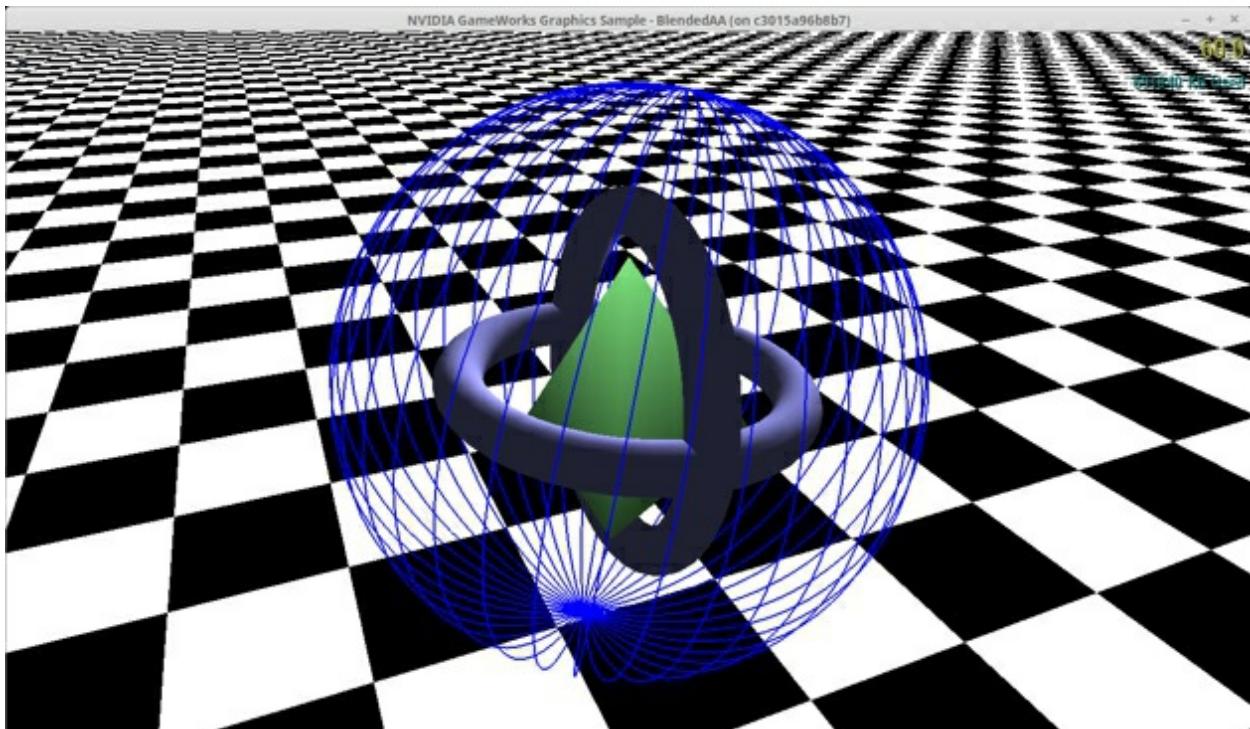
```
make -j$(getconf _NPROCESSORS_ONLN) release
```

The `getconf _NPROCESSORS_ONLN` command returns the number of available processor cores and `make -j` allows make to utilise all of the available cores in order to speed up a potentially lengthy build.

We then tidy up the image by removing all of the packages that were only required to build the GameWorks samples, though note that the final image is still relatively large weighing in at around 1GB.

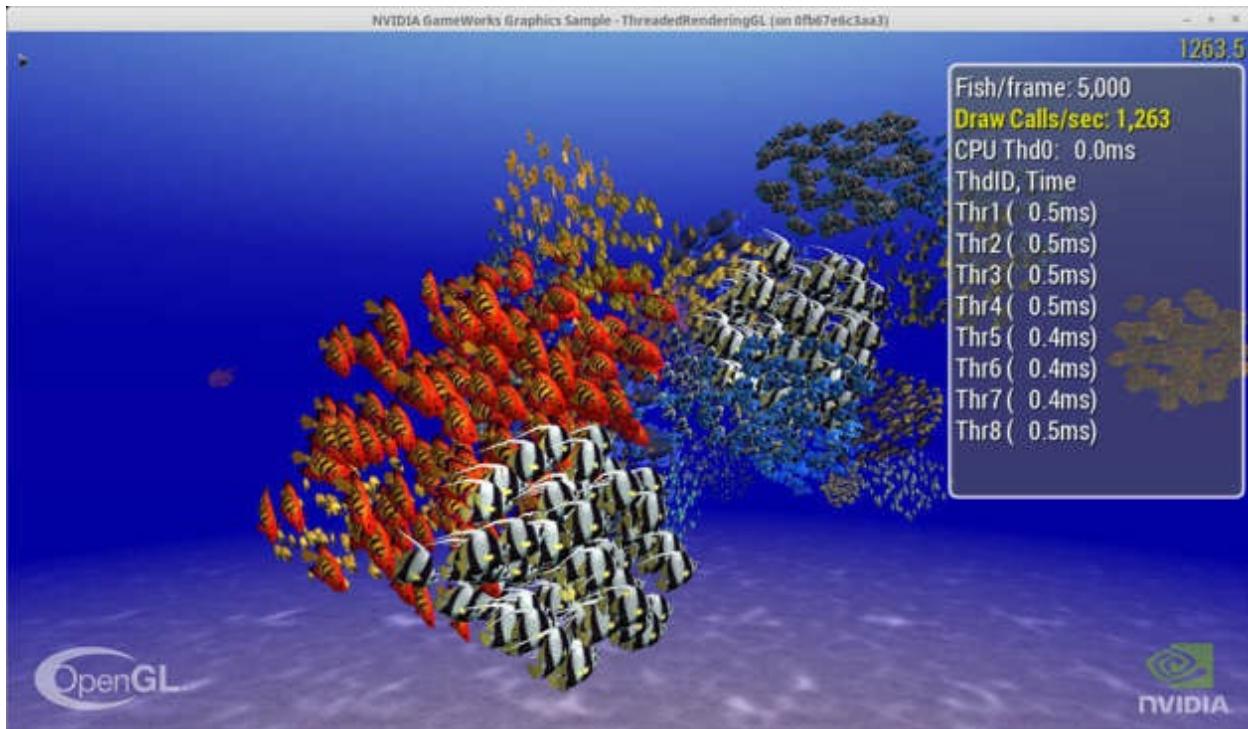
The [blended-anti-aliasing.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-gpu.sh  
  
docker run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$X11_FLAGS \  
$GPU_FLAGS \  
gameworks-graphics-samples ./BlendedAA
```



The [threaded-rendering.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-gpu.sh  
  
docker run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$X11_FLAGS \  
$GPU_FLAGS \  
gameworks-graphics-samples ./ThreadedRenderingGL
```



Note that the BlendedAA and ThreadedRenderingGL examples were chosen simply to be representative and clearly these scripts are easily modified to be able to run any of the other GameWorks examples.

5-4-9 blender

To finish off the Video and Graphics chapter we shall explore [Blender](#), the free and open source 3D creation suite.

The [Dockerfile](#) may be found in the [blender](#) directory of [docker-gui](#) under [5-more-complex-applications/video-and-graphics](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install blender
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    blender libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["blender"]
```

and may be built with:

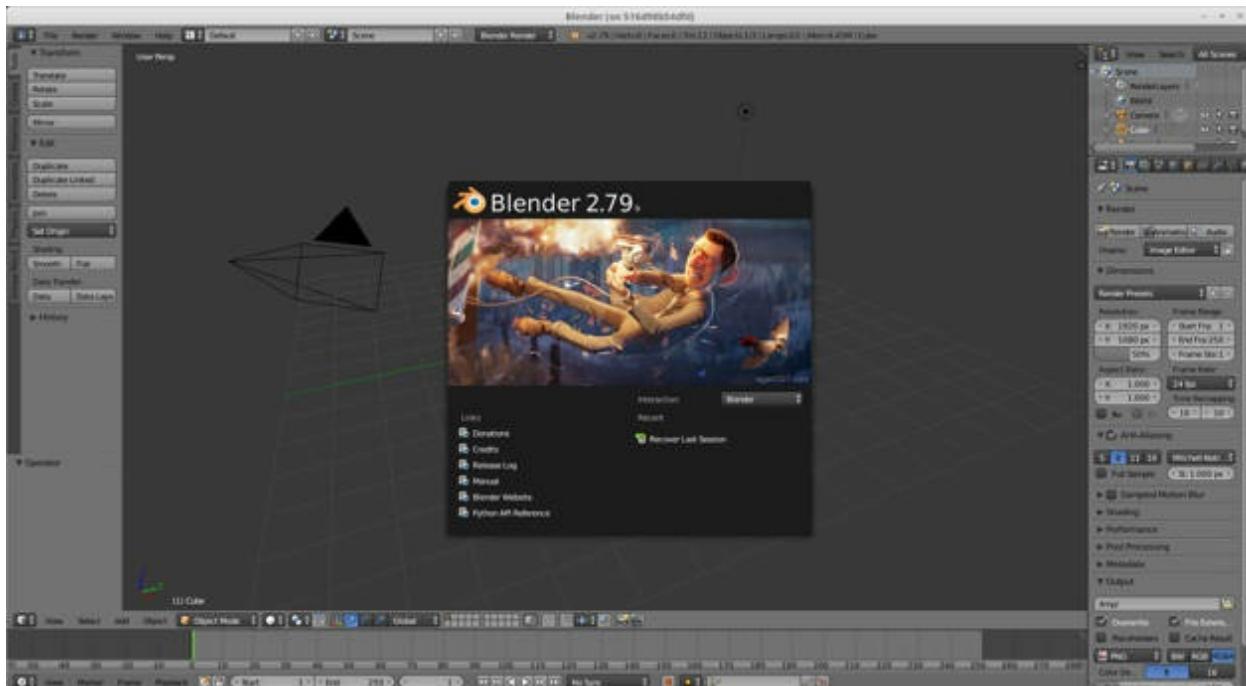
```
docker build -t blender .
```

This Dockerfile is comparatively straightforward and follows the by now hopefully quite familiar pattern employed by many of our previous OpenGL applications of including the libgl1-mesa-glx and libgl1-mesa-dri libraries along with the blender application package itself.

The [blender.sh](#) launch script is equally straightforward:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
$GPU_FLAGS blender $@
```



Our basic Blender application is, from a Docker perspective at least, quite simple given the knowledge that readers should hopefully have gained by now. We can, however, make things rather more exciting by adding the knowledge that we have gained from the [Compute Acceleration](#) chapter into the mix because, as it happens, Blender supports both CUDA and OpenCL accelerated rendering.

CUDA

The [Dockerfile](#) for our CUDA accelerated Blender is:

```
FROM debian:stretch-slim

ENV CUDA_VERSION 9.0
ENV NVIDIA_REQUIRE_CUDA "cuda>=${CUDA_VERSION}"
ENV CUDA_FULL_VERSION ${CUDA_VERSION}.176

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver
LABEL com.nvidia.cuda.version="${CUDA_FULL_VERSION}"

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    blender libgl1-mesa-glx libgl1-mesa-dri \
    apt-transport-https wget gnupg ca-certificates && \
    # Add the CUDA packages and signing key
    wget -O - https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub | apt-key add - && \
    echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64 /" > /etc/apt/sources.list.d/cuda.list
&& \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    cuda-cudart-${CUDA_VERSION} \
    cuda-command-line-tools-${CUDA_VERSION} && \
    ln -s cuda-${CUDA_VERSION} /usr/local/cuda && \
```

```
apt-get clean && \
apt-get purge -y apt-transport-https wget gnupg \
ca-certificates && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/* && \
cp /etc/pulse/client.conf \
/etc/pulse/client-noshm.conf && \
sed -i "s/enable-shm = yes/enable-shm = no/g" \
/etc/pulse/client-noshm.conf
```

```
ENTRYPOINT ["blender"]
```

The blender-cuda image may be built with:

```
docker build -t blender-cuda -f Dockerfile-cuda .
```

This is largely a mash-up of our previous Blender Dockerfile and our earlier [nvidia-smi](#) and could be rewritten in a more modular form by using one as the base image for the other.

The [blender-cuda.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
$GPU_FLAGS blender-cuda $@
```

OpenCL

The [Dockerfile](#) for our OpenCL accelerated Blender is:

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

# Install blender
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    blender libgl1-mesa-glx libgl1-mesa-dri \
    ocl-icd-libopencl1 ocl-icd-opencl-dev && \
# Add the Nvidia OpenCL Installable Client Driver
mkdir -p /etc/OpenCL/vendors && \
echo "libnvidia-opencl.so.1" > \
    /etc/OpenCL/vendors/nvidia.icd && \
rm -rf /var/lib/apt/lists/* && \
cp /etc/pulse/client.conf \
    /etc/pulse/client-noshm.conf && \
sed -i "s/enable-shm = yes/enable-shm = no/g" \
    /etc/pulse/client-noshm.conf

# Earlier blender versions used CYCLES_OPENCL_TEST=all
ENV CYCLES_OPENCL_SPLIT_KERNEL_TEST 1
ENTRYPOINT ["blender"]
```

and may be built with:

```
docker build -t blender-opencl -f Dockerfile-opencl .
```

Like our CUDA accelerated Blender this is largely a mash-up of our previous Blender Dockerfile and our earlier Nvidia [clinfo](#) and could be rewritten in a more modular form by using one as the base image for the other.

It's worth pointing out that OpenCL support in Blender is less advanced than its CUDA counterpart and we are illustrating Nvidia's OpenCL here purely because that is only hardware the author has access to. One would normally use Blender's OpenCL with AMD or Intel hardware, but to use with AMD readers would need to create their own Dockerfiles based on the ROCm, Mesa or AMD APP examples presented in [clinfo](#).

Another point of note is that enabling OpenCL on blender is not particularly obvious. This site: <https://blenderartists.org/t/cycles-amd-opencl-test/598060> eventually yielded the answer, hence the following lines near the end of the Dockerfile:

```
# Earlier blender versions used CYCLES_OPENCL_TEST=all  
ENV CYCLES_OPENCL_SPLIT_KERNEL_TEST 1
```

The [blender-opencl.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-gpu.sh  
. $BIN/docker-pulseaudio.sh  
  
mkdir -p $(id -un)/.config/pulse  
$DOCKER_COMMAND run --rm \  
    -u $(id -u):$(id -g) \  
    -v $PWD/$(id -un):/home/$(id -un) \  
    -v /etc/passwd:/etc/passwd:ro \  
    $PULSEAUDIO_FLAGS \  
    $X11_FLAGS \  
    $GPU_FLAGS \  
    blender-opencl $@
```

5-5 Office

Whilst office applications typically utilise fewer exotic features like the GPU acceleration we explored in the [Video and Graphics](#) section, they still expose a number of challenges to successful containerisation like enabling printer connectivity and [language](#) and timezone localisation.

Printing

Printer connectivity is relatively straightforward, and observant readers will recall that it was first explored when we covered [gedit](#). The “trick” is that in most cases printing “just works” for applications that have included D-bus support, so we simply have to include our [docker-dbus-all.sh](#) script and add \$DCONF_FLAGS to the docker run command.

For those applications that don’t utilise D-bus, or for cases where we’d prefer to avoid D-bus, it is generally possible to enable printing by explicitly bind-mounting the cups socket /var/run/cups/cups.sock or by adding the [mDNS](#) / [Avahi](#) discovery libraries to the image to support network printing.

Localisation

In this book we have generally specified the required language and timezone as environment variables in the Dockerfiles, so for the case of UK English we would use:

```
ENV LANG=en_GB.UTF-8  
ENV TZ=Europe/London
```

Setting the language locale requires that the locales package be installed and the process seems to be different depending on the Linux distribution being used as the base image.

For Debian the process involves modifying /etc/locale.gen then recompiling the database with locale-gen:

```
sed -i "s/^# *\$LANG/\$1/" /etc/locale.gen && locale-gen
```

For Ubuntu the process involves writing the language information to /etc/default/locale then calling update-locale:

```
echo LANG=\$LANG > /etc/default/locale  
update-locale LANG=\$LANG
```

Setting up the required timezone in Docker is largely distribution agnostic as follows:

```
echo $TZ > /etc/timezone  
ln -snf /usr/share/zoneinfo/\$TZ /etc/localtime
```

With Debian and Ubuntu we also however include the following step, which also requires the tzdata package to be installed. This step is the normal mechanism used in standard Debian and Ubuntu deployments, though might well be unnecessary in Docker containers as described here: <https://serverfault.com/questions/683605/docker-container-time-timezone-will-not-reflect-changes>.

```
DEBIAN_FRONTEND=noninteractive dpkg-reconfigure tzdata
```

5-5-1 thunderbird

[Thunderbird](#) is the email client included by default in many Linux distributions.

The [Dockerfile](#) is in the [thunderbird](#) directory of [docker-gui](#) under [5-more-complex-applications/office](#).

```
FROM debian:stretch-slim
ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    thunderbird thunderbird-l10n-en-gb \
    locales tzdata hunspell-en-gb && \
    rm -rf /var/lib/apt/lists/* && \
    sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
    locale-gen && \
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive dpkg-reconfigure tzdata

ENTRYPOINT ["/usr/bin/thunderbird", "--no-remote"]
```

and may be built with:

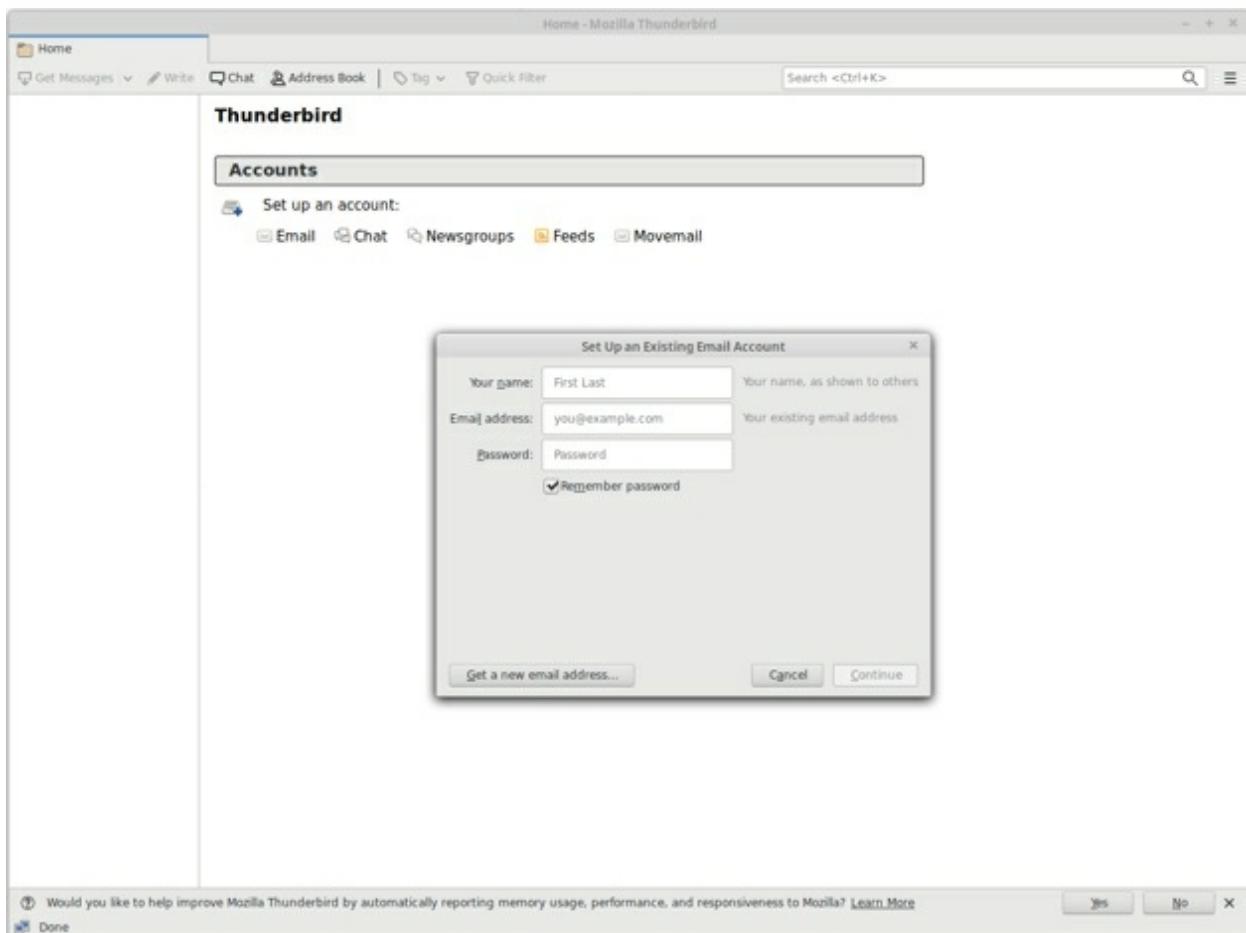
```
docker build -t thunderbird .
```

This Dockerfile is mostly simple and unsurprising. The most interesting parts are the inclusion of thunderbird-l10n-en-gb and hunspell-en-gb to provide spell checking and calendar information specific to our locale, and the addition of the language locale and timezone steps described previously.

The [thunderbird.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$X11_FLAGS thunderbird
```



5-5-2 libreoffice

LibreOffice is an office application suite for Linux.

The Dockerfile is in the libreoffice directory of docker-gui under 5-more-complex-applications/office .

```
FROM debian:stretch-slim
ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN sed -i 's/main/main contrib/' \
    /etc/apt/sources.list && \
    mkdir -p /usr/share/man/man1 && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libreoffice libreoffice-pdfimport \
    libreoffice-ogltrans libreoffice-gtk3 \
    ca-certificates locales tzdata default-jre fonts-noto \
    fonts-symbola fonts-lmodern fonts-freefont-ttf \
    fonts-liberation fonts-dejavu ttf-mscorefonts-installer \
    libpulse0 libgl1-mesa-glx libgl1-mesa-dri && \
    apt-get -y autoremove && apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    sed -i "s/^# *\$LANG\$/\1/" /etc/locale.gen && \
    locale-gen && \
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf
ENTRYPOINT ["libreoffice"]
```

The libreoffice image may be built with:

```
docker build -t libreoffice .
```

This Dockerfile looks complicated, but mostly just builds on some of the patterns we have already covered like setting the locale and including libgl1-mesa-glx and libgl1-mesa-dri for OpenGL support. Installing the ttf-mscorefonts-installer package requires the Debian contrib repository to be enabled and also, somewhat strangely but due to us using a “slim” base image, when using default-jre we must add:

```
mkdir -p /usr/share/man/man1
```

The [libreoffice.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
$GPU_FLAGS \
libreoffice
```

LibreOffice, perhaps surprisingly, also includes OpenCL support to accelerate computation of complex spreadsheets.

The [Dockerfile](#) is in the `libreoffice` directory of [docker-gui](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN sed -i 's/main/main contrib/' \
    /etc/apt/sources.list && \
    mkdir -p /usr/share/man/man1 && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libreoffice libreoffice-pdfimport \
    libreoffice-ogltrans libreoffice-gtk3 \
    ca-certificates locales tzdata default-jre fonts-noto \
    fonts-symbola fonts-lmodern fonts-freefont-ttf \
    fonts-liberation fonts-dejavu ttf-mscorefonts-installer \
    libpulse0 libgl1-mesa-glx libgl1-mesa-dri \
    ocl-icd-libopencl1 ocl-icd-opencl-dev && \
# Add the Nvidia OpenCL Installable Client Driver
    mkdir -p /etc/OpenCL/vendors && \
    echo "libnvidia-opencl.so.1" > \
        /etc/OpenCL/vendors/nvidia.icd && \
    apt-get -y autoremove && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
# Generate locales
    sed -i "s/^# *`($LANG)`/\1/" /etc/locale.gen && \
    locale-gen && \
# Set up the timezone
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
```

```
cp /etc/pulse/client.conf \
    /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
    /etc/pulse/client-noshm.conf
```

```
ENTRYPOINT ["libreoffice"]
```

and may be built with:

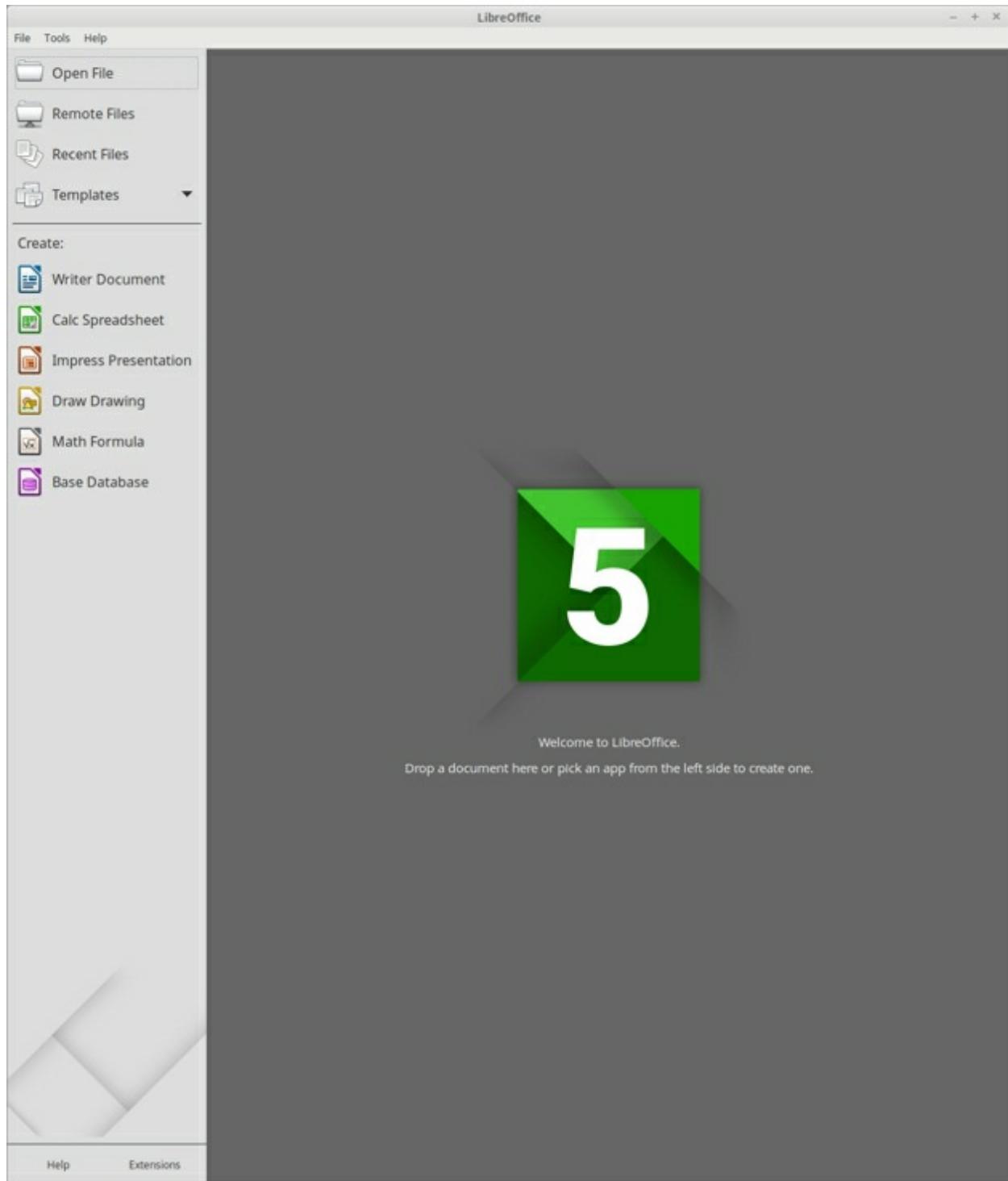
```
docker build -t libreoffice-opencl -f Dockerfile-opencl .
```

Like our earlier OpenCL accelerated [blender](#) this is largely a mash-up of our previous LibreOffice Dockerfile and our earlier Nvidia [clinfo](#) and could be rewritten in a more modular form by using one as the base image for the other.

The [libreoffice-opencl.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS libreoffice-opencl
```



5-5-3 calibre

[Calibre](#) is a cross-platform free and open-source suite of e-book software with a built-in e-book viewer that can display all the major e-book formats.

The [Dockerfile](#) is in the [calibre](#) directory of [docker-gui](#) under [5-more-complex-applications/office](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

LABEL com.nvidia.volumes.needed=nvidia_driver

RUN sed -i 's/main/main contrib/' \
    /etc/apt/sources.list && \
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends \
calibre ca-certificates locales tzdata fonts-noto \
fonts-symbola fonts-lmodern fonts-freefont-ttf \
fonts-liberation fonts-dejavu \
ttf-mscorefonts-installer && \
apt-get -y autoremove && \
apt-get clean && \
rm -rf /var/lib/apt/lists/* && \
# Generate locales
sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive dpkg-reconfigure tzdata

ENTRYPOINT ["calibre"]
```

The calibre image may be built with:

```
docker build -t calibre .
```

The Dockerfile for calibre again includes the language locale and timezone setup that we have previously covered and, as with LibreOffice, we have added a range of font packages though it is not clear which of these are *actually* necessary.

Whilst its relatively easy to identify *library* dependencies using ldd, unfortunately the author hasn't yet discovered a convenient way to work out such things as the font dependencies of containerised applications, so in many cases a range of font packages are included somewhat speculatively.

The [calibre.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    calibre $@
```

calibre - || Calibre Library ||

Add books Edit metadata Convert books View Get books Fetch news Help Remove books Calibre Library Save to disc

Virtual Library Search (For Advanced Search click the button to the left) Go! Saved Search... 

Find item in t... Find

	Title	Author(s)	Date	Size (MB)	Rating	Tags	Series	Publisher
1	Made with Rem...	Unknown	05 Jan 2020	3.8				
2	Quick Start Guide	John Schember	05 Jan 2020	0.1				

Authors Languages Series Formats Publisher Rating News Tags Identifiers

Running GUI Applications in Docker

Formats: EPUB Path: Click to open

Alter Tag Browser

calibre 2.75.1 created by Kovid Goyal [2 books, 1 selected]

Jobs: 0

5-6 Audio

Although many of the applications already covered in this book actually do have good audio support provided by [PulseAudio](#), the purpose of this chapter is to illustrate low-latency pro-audio graphical applications based on [JACK](#).

5-6-1 qjackctl

The first JACK based GUI application we shall cover is [QjackCtl](#). QjackCtl is a Qt application that is able to control jackd and includes a patchbay and connection control features. In Debian and many other distributions QjackCtl defaults to using JACK 2 as described in the [JACK](#) section.

The [Dockerfile](#) is in the [qjackctl](#) directory of [docker-gui](#) under [5-more-complex-applications/audio](#).

```
FROM debian:stretch-slim
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    qjackctl libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["qjackctl"]
```

and may be built with:

```
docker build -t qjackctl .
```

The Dockerfile is rather simple as installing the qjackctl package also installs its dependencies, one of which is jackd2.

The [qjackctl.sh](#) launch script is:

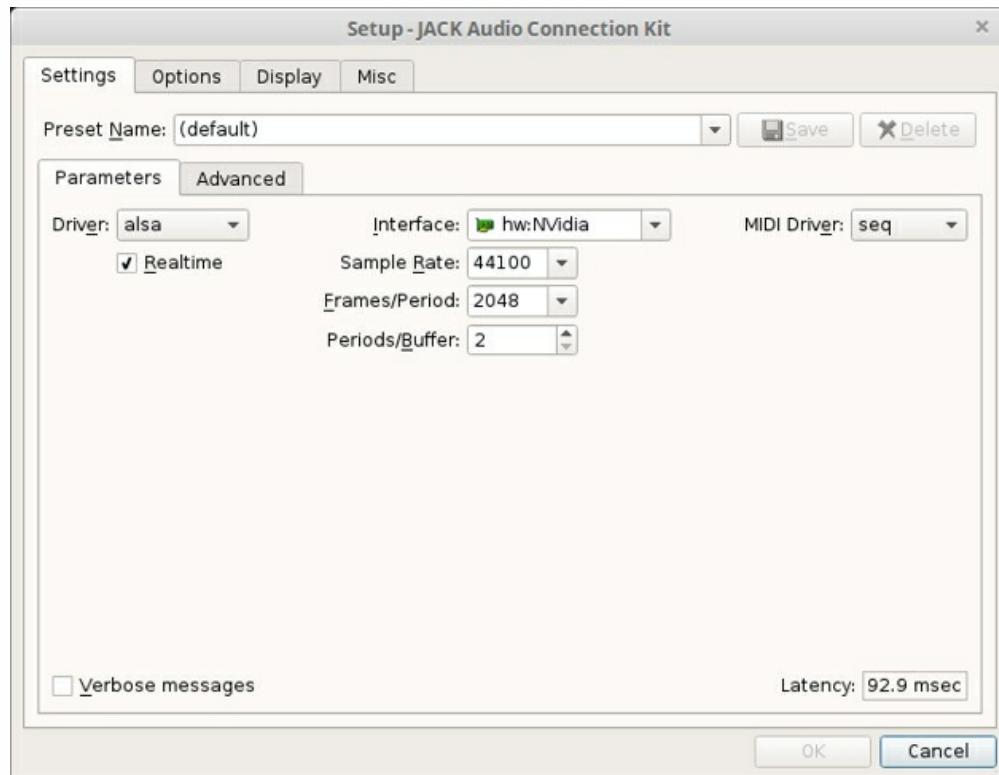
```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-jack.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$JACKD_FLAGS \
$X11_FLAGS \
$GPU_FLAGS \
qjackctl $@
```

This launch script includes D-bus/dconf support as that is used by jackd2 and it also uses our [docker-jack.sh](#) script, which uses pasuspender to suspend PulseAudio and adds realtime and /dev/snd support via the JACKD_FLAGS.



As mentioned in the earlier [JACK](#) section configuring JACK is often somewhat more involved than dealing with PulseAudio. In particular, if there are multiple audio devices available it is often not obvious which device to use. In the author's case, with two GPUs and a built-in sound card, trying to work out which device was actually connected to the monitor speakers via HDMI was quite tedious and the required configuration was discovered mostly by trial and error.



Once configured and started QjackCtl doesn't really do too much on its own other than control jackd, and it requires applications like the following ones to actually play anything interesting.

5-6-2 qsynth

[Qsynth](#) is a graphical front-end to [FluidSynth](#), a real-time software synthesiser based on the SoundFont 2 specification.

The [Dockerfile](#) is in the [qsynth](#) directory of [docker-gui](#) under [5-more-complex-applications/audio](#).

```
FROM debian:stretch-slim
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    qsynth fluid-soundfont-gm \
    libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/*
ENTRYPOINT ["qsynth"]
```

and may be built with:

```
docker build -t qsynth .
```

This is another simple Dockerfile where we benefit from apt dependency management doing much of the heavy lifting, so we don't have to explicitly include the fluidsynth package. Note, however, that we do include fluid-soundfont-gm as this is a large general MIDI SoundFont and without a suitable SoundFont Qsynth/FluidSynth won't play anything.

Note too libgl1-mesa-glx and libgl1-mesa-dri and the GPU flags used when running, this is because Qt applications use OpenGL rendering as first covered in our [kcalc](#) application.

The [qsynth.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-jack.sh

mkdir -p $(id -un)
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$JACK_FLAGS \
$X11_FLAGS \
$GPU_FLAGS qsynth $@
```



Qsynth also needs some additional configuration to set the MIDI driver to jack and add the SoundFont.



5-6-3 jack-keyboard

[jack-keyboard](#) is a virtual MIDI keyboard that allows JACK MIDI events to be sent by simply using a PC keyboard.

By using jack-keyboard in conjunction with Qsynth and a suitable SoundFont, and with QjackCtl controlling jackd we should be in a position to demonstrate both the audio capabilities (the output of Qsynth) and MIDI capabilities (between jack-keyboard and Qsynth) of JACK.

The [Dockerfile](#) is in the [jack-keyboard](#) directory of [docker-gui](#) under [5-more-complex-applications/audio](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    jack-keyboard && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["jack-keyboard"]
```

and may be built with:

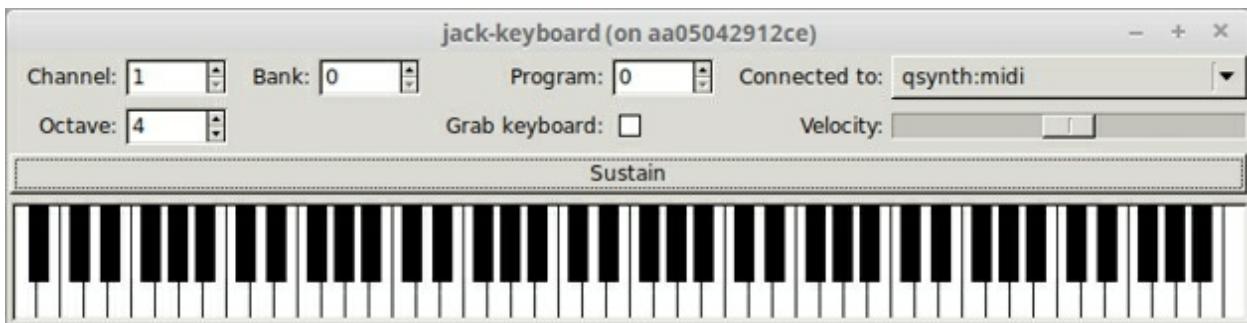
```
docker build -t jack-keyboard .
```

Once again this Dockerfile is rather trivial, as installing the jack-keyboard package will also automatically install the required JACK libraries and any other dependencies.

The [jack-keyboard.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh
. $BIN/docker-jack.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$JACK_FLAGS \
$X11_FLAGS \
jack-keyboard $@
```



If QjackCtl and Qsynth are both running correctly, when we start jack-keyboard and select “Connected to:” we should see an option for qsynth:midi, which should be selected. We should now be able to “play” a virtual piano (the default MIDI patch). The virtual instrument being played can be changed via the “Program” value on jack-keyboard.

Although jack-keyboard is quite fun and is a good way to try things out, if one wants to use real (physical) MIDI devices then the launch script for **QjackCtl** will need to be modified to add
`--device=/dev/sequencer` to pass the MIDI device through to the container.

5-6-4 ardour

[Ardour](#) is a powerful [Digital audio workstation](#) application that can be used to control, record, edit and run complex audio setups. Ardour uses [JACK](#), though it can also use [ALSA](#) directly if it has access to /dev/snd and the user running the application is a member of the audio group.

The [Dockerfile](#) is in the [ardour](#) directory of [docker-gui](#) under [5-more-complex-applications/audio](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    ardour binutils && \
    # Tidy up
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["ardour5"]
```

and may be built with:

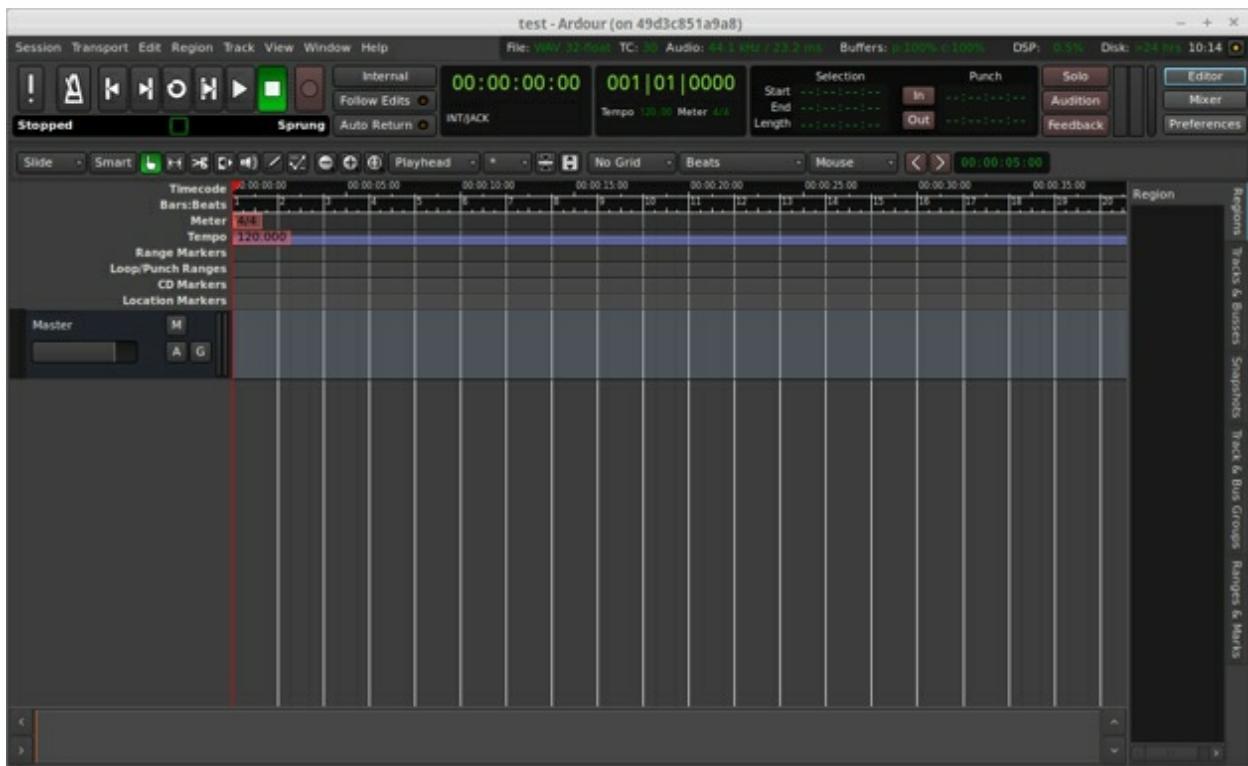
```
docker build -t ardour .
```

Note that the binutils package is included in the Dockerfile because at startup Ardour uses [nm](#), presumably to identify plugins. By default Ardour only comes with a small number of built-in [plugins](#), but many other plugins are available so readers will almost certainly wish to extend this basic Dockerfile to include their own favourite Ardour plugins.

The [ardour.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh
. $BIN/docker-jack.sh

# Deliberately using JACKD_FLAGS below as this also gives
# the option of connecting directly to ALSA as it enables
# /dev/snd and adds the audio group
mkdir -p $(id -un)/.local/share
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $JACKD_FLAGS \
    $X11_FLAGS \
    ardour $@
```



5-7 Games

Although the [Video and Graphics](#) section provided several good examples of GPU accelerated OpenGL applications, probably the best way to get a *real* feel for how we can run such applications in containers with near native performance is to try out a few games.

5-7-1 extremetuxracer

[Extreme Tux Racer](#) is a downhill racing game starring Tux, the Linux penguin.

The [Dockerfile](#) is in the `extremetuxracer` directory of [docker-gui](#) under [5-more-complex-applications/games](#).

```
FROM debian:stretch-slim

LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    extremetuxracer libpulse0 \
    libgl1-mesa-glx libgl1-mesa-dri && \
    apt-get -y autoremove && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["/usr/games/etr"]
```

The `extremetuxracer` image may be built with:

```
docker build -t extremetuxracer .
```

The Extreme Tux Racer Dockerfile is very simple, but one interesting part is the need to explicitly include `libpulseo` which wasn't included in the dependency tree for the `extremetuxracer` package.

This illustrates an advantage of the pattern we've been following where we create a `/etc/pulse/client-noshm.conf` file to disable shared memory for certain PulseAudio versions. With that in place the docker build actually fails if it can't find `client.conf`, so it was a simple matter to determine that the PulseAudio client library was initially missing from the image.

The [`extremetuxracer.sh`](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
    --device=/dev/input -v /dev/input:/dev/input:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    extremetuxracer $@
```

The launch script largely follows the common pattern of including X11, PulseAudio and GPU flags which we have encountered in many applications. In this case, however, we have also enabled `--device=/dev/input` and added `-v /dev/input:/dev/input:ro`. This allows game controllers or joysticks to be used to control the game.

The bind-mount is added because `/dev/input` is actually a directory. We could perhaps be more explicit and use `--device=/dev/input/js0`, but that would fail if we don't have a controller attached.



5-7-2 supertuxkart

SuperTuxKart is a free and open-source kart racing game featuring the mascots of various open-source projects.

The Dockerfile is in the supertuxkart directory of docker-gui under [5-more-complex-applications/games](#).

```
FROM debian:stretch-slim

LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    supertuxkart libpulse0 \
    libgl1-mesa-glx libgl1-mesa-dri && \
    apt-get -y autoremove && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["/usr/games/supertuxkart"]
```

and may be built with:

```
docker build -t supertuxkart .
```

The Dockerfile (and the launch script) of SuperTuxKart are nearly identical to those for Extreme Tux Racer with just the package, ENTRYPOINT and image names differing.

The [supertuxkart.sh](#) launch script is below, again with game controller or joystick support enabled:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
    --device=/dev/input -v /dev/input:/dev/input:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    supertuxkart $@
```



5-7-3 sauerbraten

Sauerbraten is a Quake-like first-person shooter.

The Dockerfile is in the sauerbraten directory of docker-gui under 5-more-complex-applications/games.

```
FROM debian:stretch-slim
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN sed -i 's/main/main non-free/' \
    /etc/apt/sources.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    sauerbraten libpulse0 \
    libgl1-mesa-glx libgl1-mesa-dri && \
    apt-get -y autoremove && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf
ENTRYPOINT ["/usr/games/sauerbraten"]
```

and may be built with:

```
docker build -t sauerbraten .
```

The Dockerfile is extremely similar to the SuperTuxKart and Extreme Tux Racer Dockerfiles, however Sauerbraten requires the Debian non-free repository to be enabled in /etc/apt/sources.list as the game media and assets are released under various non-free licenses.

The [sauerbraten.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
$GPU_FLAGS \
sauerbraten $@
```



5-7-4 dolphin-emu

[Dolphin Emulator](#) is a console emulator for the GameCube and Wii that is noted for its high compatibility with the original consoles. For the legalities of emulation it is important to read the Dolphin Emulator [FAQ](#).

The [Dockerfile](#) is in the [dolphin-emu](#) directory of [docker-gui](#) under [5-more-complex-applications/games](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    locales tzdata dolphin-emu pulseaudio \
    libgl1-mesa-glx libgl1-mesa-dri && \
    apt-get -y autoremove && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    sed -i "s/^# *\$LANG\$/\$1/" /etc/locale.gen && \
    locale-gen && \
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/\$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["/usr/games/dolphin-emu"]
```

The dolphin-emu image may be built with:

```
docker build -t dolphin-emu .
```

Dolphin Emulator itself is free and open-source software and unlike sauerbraten the dolphin-emu package is available from the main Debian repository, so the Dockerfile is largely simple and unsurprising.

Dolphin Emulator has good localisation support, so this Dockerfile sets the language locale and timezone as described in the [Office](#) section.

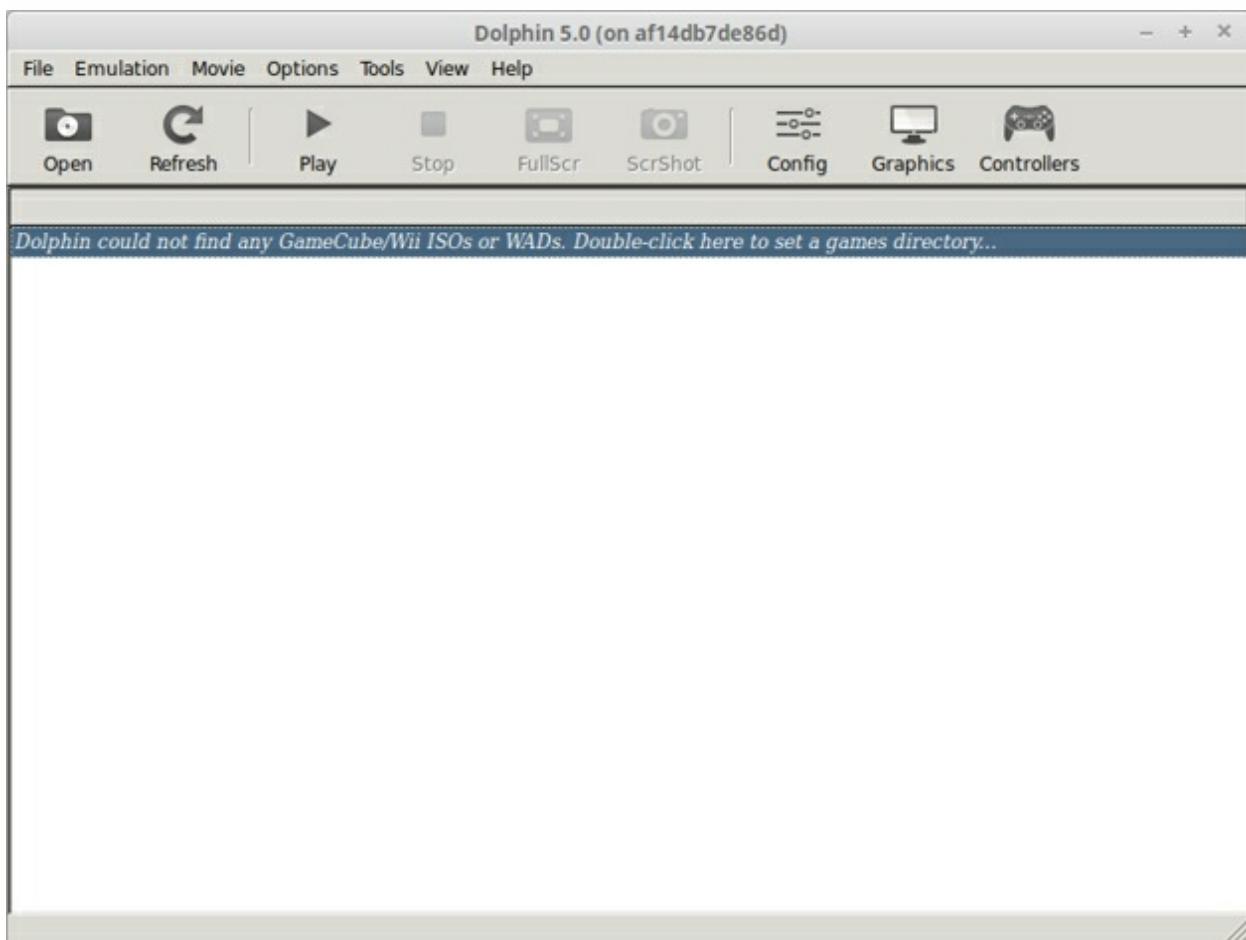
The [dolphin-emu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
    --device=/dev/input -v /dev/input:/dev/input:ro \
    --device=/dev/cdrom \
    --group-add $(cut -d: -f3 < <(getent group cdrom)) \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    dolphin-emu $@
```

As with Extreme Tux Racer and SuperTuxKart we have also enabled `--device=/dev/input` and added `-v /dev/input:/dev/input:ro` to allow game controllers or joysticks to be used to control the games. With Dolphin Emulator a range of different controller types may be used which need to be configured from the Dolphin's main window, either by navigating to Options→Controller Settings, or simply clicking the "Controllers" button. Full instructions on configuring controllers are in the Dolphin Emulator [Wiki](#).

The script also enables `--device=/dev/cdrom`, so Dolphin should behave largely as if it were installed natively.



6 Remote Applications

So far all of the examples in this book have followed the common pattern of running our containers on a local machine and communicating directly with a local display server.

By this stage readers should have gained sufficient knowledge to handle even the most complex local deployment scenarios, so the time has come to tackle the next stage of our journey and solve the problem of running our applications remotely, whilst still delivering the display to our local display server.

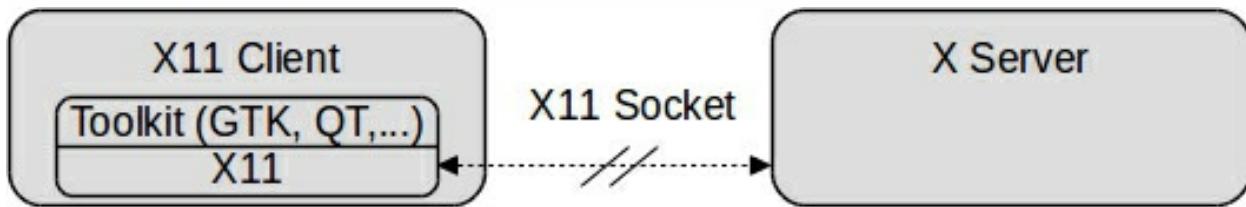
For the remainder of this book we shall assume that the reader has access to one or more remote servers, or has at least the ability to simulate the use of remote servers via virtual machines.

Some basic familiarity with [ssh](#) is also assumed as we shall use this as the primary mechanism for gaining access to our remote servers and launching containers on them. As we shall primarily be using ssh to access our remote servers it should hopefully be obvious that we will require those to be running an ssh server such as openssh-server, which is available in the package repositories of all major Linux distributions.

6-1 X11 Forwarding

In our earlier [X11 Overview](#) chapter we noted that X11 is based on a *client-server* architecture that is both device independent and network transparent.

To understand X11 network transparency consider again the simplified diagram comprising just the X Client and X Server.



In the examples covered so far the X11 socket that we have been using has been a [Unix domain socket](#), however X11 makes no assumptions about this and we could have used a TCP network socket almost as easily, as follows.

To fully understand all of the stages required to enable X11 forwarding from a container we shall cover each step separately, beginning with running a non-containerised X11 application *directly* on the remote server. For this we assume that the `x11-apps` package (or `xorg-x11-apps` on RHEL or CentOS) has already been installed on the remote server.

First log on to the remote server via ssh, e.g.

```
ssh <user>@<remote-host>
```

If this is successful we should be greeted by a welcome from the remote server and a command prompt that reflects the remote host's name rather than that of our local workstation.

If we now run

```
xeyes
```

That should (hopefully) fail with:

```
Error: Can't open display:
```

unless something has been previously (mis)configured.

If we now run:

```
echo $DISPLAY
```

we should see the first reason for the failure, as the DISPLAY environment variable on the remote server is empty.

The most obvious next step is to set the DISPLAY variable to refer to our workstation-host, that is to say the host, local to us, that is running our display server e.g.

```
DISPLAY=<workstation-host>:0 xeyes
```

Unfortunately however this doesn't work either, though the error (and the reason for the error) is slightly different.

```
Error: Can't open display: <workstation-host>:0
```

Although X11 was designed to be a network transparent protocol, back in 1984 most networks were relatively isolated and security was somewhat less of a concern. Now we have ubiquitous Internet connectivity and it becomes possible to accidentally expose the X Server to the Internet, so modern X Servers have *disabled direct network access by default*.

If both the workstation host and remote host are on a private network behind a router firewall that is blocking port 6000 from external networks then it is possible to *temporarily* illustrate how X11 worked *back in the day*.

Warning do not do the following if there is a risk that the workstation host has port 6000 publicly exposed (check first) and in any case only do it temporarily.

On the workstation host (not the remote host) run:

```
socat TCP-LISTEN:6000,reuseaddr,fork UNIX-CLIENT:/tmp/.X11-unix/X0
```

This command will run `socat` as a proxy, listening on port 6000 (the well-known port for X11) and forwarding connections to the X11 Unix domain socket `/tmp/.X11-unix/X0`, which obviously may need to be changed if the display server is listening on a different Unix domain socket.

If we again run (substituting the real workstation host name):

```
DISPLAY=<workstation-host>:0 xeyes
```

we will see a slightly different error:

```
No protocol specified
Error: Can't open display: <workstation-host>:0
```

The `No protocol specified` error was first encountered in the [x11-apps](#) section and relates to X11 authentication, and as described in that earlier x11-apps section the simplest (and least secure) way to resolve it is to run the following on the workstation host:

```
xhost +local:
```

and if we again run the following command (again substituting the real workstation host name):

```
DISPLAY=<workstation-host>:0 xeyes
```

we should *finally* see the display for the xeyes application visible locally, if so quit out and run:

```
xhost -
```

to re-enable authentication.

As discussed in the earlier [x11-apps](#) section a better way to enable authentication is via .Xauthority. However, we might well not have the required .Xauthority available on the remote host using this approach, so we will need to manually copy a .Xauthority.docker file created from our local applications to the remote host via scp or sftp. If we then run:

```
DISPLAY=<workstation-host>:0 XAUTHORITY=$HOME/.Xauthority.docker  
r xeyes
```

we should again see the xeyes application displayed locally.

To reiterate, the preceding section was intended to illustrate how X11 network transparency was *originally* intended to work, but is now really an illustration of **how not to do it**.

There is, however, a more modern, more secure, and simpler way to achieve X11 forwarding via ssh X11 forwarding, which needs to be enabled on both the client and the server side.

On the client side, the `-X` option to ssh enables X11 forwarding. This can be made the default (for all connections or for a specific connection) by setting `ForwardX11 yes` in `~/.ssh/config`.

On the server side, `x11Forwarding yes` must be specified in `/etc/ssh/sshd_config`, which seems to be the default in Ubuntu based distributions, but is off by default in others. Note that the user cannot override this setting and it must be set by an administrator. The xauth program must also be installed on the server, though this should be installed as a dependency when `x11-apps` is installed on the server.

Assuming that the remote server has been correctly configured, if we now run (with a capital `-X`):

```
ssh -X <user>@<remote-host>
```

then simply run (on the remote host):

```
xeyes
```

we should again see the xeyes application displayed locally.

6-1-1 x11-apps

Although from a user experience perspective using ssh -X is extremely simple compared to the legacy X11 forwarding previously illustrated, it is important to understand that there is a lot of work happening behind the scenes that has been transparently enabled by the ssh client and server.

Understanding the mechanics of ssh X11 forwarding is important if we are to identify a way to enable it from a container, so we shall now explore what is actually happening.

Recall that when basic ssh (without the -X flag) was used and we ran the command:

```
echo $DISPLAY
```

On the remote host the DISPLAY environment variable was empty. If this is repeated after using ssh -X the DISPLAY variable on the remote host should look something like:

```
localhost:10.0
```

If it is still blank it means that ssh X11 forwarding has not been correctly set up, the most likely reasons are because the ssh daemon has been incorrectly configured (missing `X11Forwarding yes` or `x11DisplayOffset 10` in /etc/ssh/sshd_config) or the remote server doesn't have xauth installed.

If we consider the DISPLAY for a moment, the “localhost” reference on the remote server is actually quite profound, as the display from the X Client running on the remote host is *actually* being rendered on our local **workstation** host’s display server, which is remote from the perspective of our remote host, so using “localhost” seems quite curious.

What is *actually* happening is that ssh has created a secure tunnel from the remote host back to the workstation host and the endpoint to that tunnel has been exposed over TCP to the remote host on its localhost interface. This is a very important detail when considering connections from remote containers.

To explore remote applications further we shall again use x11-apps for simplicity. The [Dockerfile](#) may be found in the [x11-apps](#) directory of [docker-gui](#) under [6-remote-applications/X11-forwarding](#).

```
FROM debian:stretch-slim

# Install x11-apps
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    x11-apps && \
    rm -rf /var/lib/apt/lists/*

CMD ["xlogo"]
```

and may be built **on the remote host** with:

```
docker build -t x11-apps .
```

This Dockerfile is the same as the one used earlier in [Simple X11 Applications](#) and it is definitely worth spelling out that this needs to be built on the remote host, as it's surprisingly easy to get confused about which end of the system does what when building applications spanning multiple hosts.

Our first attempt at a launch script for a remotely executed containerised GUI application looks like [remote-xlogoVo.sh](#):

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
$DOCKER_COMMAND run --rm \  
--network=host \  
-e DISPLAY=$DISPLAY \  
-v $HOME/.Xauthority:/root/.Xauthority \  
x11-apps
```

This needs to be launched on the remote host, either by running (on the local host):

```
ssh -X <user>@<remote-host>
```

Then on the remote host cd to <path-to>/docker-gui/6-remote-applications/X11-forwarding/x11-apps and call:

```
./remote-xlogoV0.sh
```

Or perhaps more directly on the local host:

```
ssh -X <user>@<remote-host> <path-to>/docker-gui/6-remote-applications/X11-forwarding/x11-apps/remote-xlogoV0.sh
```

The <path-to> is simply the path (on the remote host) to the directory where the docker-gui source tree was git cloned.

The most important thing to note about this launch script is the line:

```
--network=host
```

This means that the container is directly using the remote host's network namespace with no isolation and, as it currently stands, the container won't work without that line as it needs to be able to connect to the secure tunnel back to the X Server on the local workstation.

The next important point of note is the line:

```
-v $HOME/.Xauthority:/root/.Xauthority
```

That is to say we are bind-mounting the remote host's .Xauthority directly. This illustrates another interesting artefact of ssh X11 forwarding, which is that the ssh daemon automatically and transparently creates (or modifies) the user's .Xauthority on the remote host.

Whilst this launch script *hopefully* works perfectly well, launching xlogo in a container on the remote host and forwarding its display back via the ssh X11 tunnel to our local workstation's X Server, it is far from perfect. The two main problems are that the container is running with more privileges than it really needs and that it requires

```
--network=host .
```

Solving the first of these problems and reducing the container's privileges is accomplished relatively easily as shown in the [remote-xlogoV1.sh](#) launch script:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
$DOCKER_COMMAND run --rm \  
--network=host \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-e DISPLAY=$DISPLAY \  
-v $HOME/.Xauthority:$HOME/.Xauthority \  
x11-apps
```

In this launch script we have followed the same pattern as we developed for local applications and used the docker run -u flag to set the user and group id that we wish to run the container as.

```
-u $(id -u):$(id -g)
```

and again, as with the local applications, the line

```
-v /etc/passwd:/etc/passwd:ro
```

is not strictly necessary and simply provides a friendly user name should we exec into the container.

Finally, the .Xauthority is placed in the correct path:

```
-v $HOME/.Xauthority:$HOME/.Xauthority
```

In order to remove the need to run the container with:

```
--network=host
```

we must first consider why we needed it in the first place.

we previously noted that ssh X11 forwarding sets the remote host's DISPLAY environment as follows:

```
localhost:10.0
```

This is key to our understanding, as we should hopefully know that “localhost” means something different inside a container than it does for the Docker host. That is to say that X Clients inside a container will not be able to see the X11 tunnel, and thus the X Server, by referring to “localhost”.

In order to resolve this we need to replace “localhost” inside the container with an actual IP address of the Docker host that is accessible from the container. The most obvious IP address to use is the address of the “dockero” interface, which is usually “172.17.0.1” or similar.

Another issue that arises from having to connect to an interface other than “localhost” is that ssh servers are most commonly not configured to accept remote connections to the X11 tunnel. This must generally be explicitly enabled by editing /etc/ssh/sshd_config and setting:

```
X11UseLocalhost no
```

If the sshd_config is changed the ssh server will need to be restarted with (on systemd based distributions):

```
sudo systemctl restart ssh
```

or (on non-systemd based distributions)

```
sudo service ssh restart
```

or

```
sudo /etc/init.d/ssh restart
```

Then re-login to the remote host with:

```
ssh -X <user>@<remote-host>
```

Unfortunately, changing the DISPLAY that the X Client is connecting to from “localhost” to “172.17.0.1” introduces *yet another* problem. The ssh daemon has been setting the .Xauthority on the remote server such that X Clients can authenticate using the DISPLAY that has also been set by the ssh daemon, but as we now need to change that the authentication will fail as it is host based.

Fortunately we have already developed the solution to this problem and we must once again create a modified .Xauthority.docker that has set a wildcarded hostname in exactly the same way as we have previously done for local applications.

The script [remote-xlogoV2.sh](#) brings all of this together:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}  
DOCKER_XAUTHORITY=${XAUTH}.docker  
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY  
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKER_XAUTHORITY nmerge -  
  
DOCKER_NETWORK=172.17.0.1  
DISPLAY=$(echo $DISPLAY | sed "s/^[:]*\(.*\)\/$DOCKER_NETWORK\1/")  
  
if ! grep -Fxq "X11UseLocalhost no" /etc/ssh/sshd_config; then  
    echo "Warning /etc/ssh/sshd_config does not contain \"X11UseLocalhost no\""  
    echo "Containers will therefore need --network=host to do X11 forwarding."  
fi  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-e DISPLAY=$DISPLAY \  
-e XAUTHORITY=$DOCKER_XAUTHORITY \  
-v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro \  
x11-apps
```

The main line of interest is the one that replaces the DISPLAY host with the IP address of the Docker network:

```
DISPLAY=$(echo $DISPLAY | sed "s/^[:]*\(.*\)\/$DOCKER_NETWORK\1/")
```

In order to avoid tedious copy and paste reuse in later scripts much of this work may be moved to a helper script [`docker-remote-xauth.sh`](#) in the [`docker-gui/bin`](#) directory.

```
# Copies .Xauthority to a temporary location based on the
# pid of this shell then merges an nlist from DISPLAY,
# with the hostname wildcarded, into the temporary
# .Xauthority before finally renaming that to
# .Xauthority.docker
XAUTH=${XAUTHORITY:-$HOME/.Xauthority}
DOCKER_XAUTHORITY=${XAUTH}.docker
cp --preserve=all $XAUTH $DOCKER_XAUTHORITY.$$
xauth nlist $DISPLAY | sed -e 's/^....ffff/' | xauth -f $DOCKE
R_XAUTHORITY.$$ nmerge -
mv $DOCKER_XAUTHORITY.$$ $DOCKER_XAUTHORITY

# Modify the DISPLAY to replace the hostname part with the
# IP of the docker0 interface on the remote host.
DOCKER_NETWORK=172.17.0.1
DISPLAY=$(echo $DISPLAY | sed "s/^[:]*\(.*\)\/$DOCKER_NETWORK\1
/")

if ! grep -Fxq "X11UseLocalhost no" /etc/ssh/sshd_config; then
    echo "/etc/ssh/sshd_config does not contain \"X11Us
eLocalhost no\""
    echo "Containers will therefore need --network=host to do X
11 forwarding."
fi

# Populate the X11_FLAGS variable as a short cut instead of
# having to set the environment and volume flags
# individually in the docker run command.

X11_FLAGS="-e DISPLAY=$DISPLAY "
X11_FLAGS+=" -e XAUTHORITY=$DOCKER_XAUTHORITY "
X11_FLAGS+=" -v $DOCKER_XAUTHORITY:$DOCKER_XAUTHORITY:ro "
```

Our final xlogo launch script [remote-xlogoV3.sh](#) then becomes greatly simplified:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-remote-xauth.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
x11-apps
```

and we can generalise it into [remote-x11-apps.sh](#), a script capable of running all of the applications in the x11-apps package, simply by allowing the application name to be passed in the command line:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-remote-xauth.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
x11-apps $@
```

We can now, for example, launch xeyes remotely with:

```
ssh -X <user>@<remote-host> <path-to>/docker-gui/6-remote-applications/X11-forwarding/x11-apps/remote-x11-apps.sh xeyes
```

6-2 Remote Applications and D-bus

6-2-1 gnome-calculator

We have previously covered the complications that [D-bus and AppArmor](#) can cause for [Local Applications](#) as we explored [gnome-calculator](#). In this section we shall discover that D-bus can prove even more awkward for remote applications.

For this section the [Dockerfile](#) for our containerised version of GNOME Calculator is in the [gnome-calculator](#) directory of [docker-gui](#) under [6-remote-applications/remote-applications-and-dbus](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    ca-certificates libcanberra-gtk3-module \
    packagekit-gtk3-module gnome-calculator && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["gnome-calculator"]
```

and may be built **on the remote host** with:

```
docker build -t gnome-calculator .
```

This Dockerfile is exactly the same as the one used earlier for our locally executed GNOME Calculator.

When running local applications the pattern that has largely been followed has been to integrate with the user's desktop session D-bus instance. That approach sidesteps a lot of potential complexity and cleanly integrates containerised applications with the desktop session albeit at the expense of some reduced sandboxing, especially where D-bus has been configured to use [abstract sockets](#).

When running remote applications however, the applications clearly aren't run in the user's desktop session, they are run in an ssh session with a completely different environment. Unlike a desktop session, ssh *does not* automatically launch a D-bus session bus instance per ssh session.

If one were to run gnome-calculator *directly* on the remote host via ssh X11 forwarding, by default the process will attempt to invoke dbus-launch with the –autolaunch option to start up a new session bus or find the existing bus address on the X display. Whilst this might appear to be ideal, in practice the autolaunched instance doesn't have an easily discoverable address and tends not to exit cleanly when the ssh session ends. Moreover, when an application is launched in a container autolaunch attempts to launch D-bus in the container and the image may well not have D-bus installed.

Another problem with autolaunch is, as described in the [dbus-launch](#) manual, whenever autolaunch occurs, the application that had to start a new bus will be in its own little world; it can effectively end up starting a whole new session if it tries to use a lot of bus services. This can be suboptimal or even totally broken, depending on the application and what it tries to do.

In an attempt to overcome some of these issues we shall create a D-bus session launch script that will launch a D-bus session bus instance *for the duration of the ssh session that the application is running in*. If multiple applications are launched in the same ssh session they will therefore all be able to connect to the **same** D-bus instance using this script.

In order to provide some simple lifecycle management the script will use the `SSH_CLIENT` environment variable set by `sshd`. As `SSH_CLIENT` holds the client's IP address and ephemeral source port it can act as a useful ssh session key.

The main premise of the script is the idea of a “key file”, that is to say it creates a file with a name derived from `SSH_CLIENT` and places it in the directory `$XDG_RUNTIME_DIR/ssh-dbus/`, where `XDG_RUNTIME_DIR` is generally something like `/run/user/1000` and is in a temporary filesystem.

The script will check for the presence of that file and if it is present it will read the contents into the variable `DBUS_SESSION_BUS_ADDRESS` if, on the other hand, the file is not present we will then launch the D-bus instance, retrieve its `DBUS_SESSION_BUS_ADDRESS` and create a new key file holding that address.

Although the approach described so far will allow all clients in the same ssh session to share the same D-bus session bus instance, we are still left with the problem that those D-bus instances are daemonised and are not cleaned up when the ssh session closes.

The way that the D-bus launch script will overcome this is to first “reap” any D-bus instances that may belong to “dead” ssh sessions whenever the script runs.

The reaping approach is actually relatively simple, it first works out what the current established ssh connections are by running

`netstat -t | grep ssh`, the script then iterates through the files in the `$XDG_RUNTIME_DIR/ssh-dbus` directory whose names are, as described earlier, derived from the `SSH_CLIENT` key that created them.

If we check the key file name against current established ssh connections, any that don’t match are “dead” ssh sessions. We can therefore use the associated .pid file to retrieve the PID of the D-bus instance associated with that session and kill it.

The approach taken here for reaping D-bus instances associated with defunct ssh sessions is deliberately simple, though fairly effective. A somewhat more elegant approach might be to place some of the logic in a script that is run on ssh login and logout. One option might be to call it in `.bashrc` or `.bash_logout`, though if the remote host also has local logins that may result in unintended consequences.

A potentially cleaner approach might be to use [PAM](#), where one could modify `/etc/pam.d/sshd` to include:

```
session optional pam_exec.so quiet /etc/pam_session.sh
```

and create an executable `pam_session.sh` that will be run whenever someone logs in or out.

[docker-dbus-session-launch.sh](#) is in [docker-gui/bin](#).

```
UNIX_SOCKET_DIR=$( [ -d $XDG_RUNTIME_DIR/bus ] && [ $(ls -A $XDG_RUNTIME_DIR/bus) ] && echo "true")\n\nif [ [ $DBUS_SESSION_BUS_ADDRESS == "" || ($DBUS_SESSION_BUS_ADDRESS == "unix:path=$XDG_RUNTIME_DIR/bus" && $UNIX_SOCKET_DIR == "") ]]; then\n\n    mkdir -p $XDG_RUNTIME_DIR/ssh-dbus\n    NETWORK_STATUS=$(netstat -t | grep ssh)\n    pushd $XDG_RUNTIME_DIR/ssh-dbus/ > /dev/null\n        for f in *; do\n            if [ -f $f ] && [[ $f != *.pid ]]; then\n                if ! [[ $NETWORK_STATUS == *"$f"* ]]; then\n                    PID=$(cat ${f}.pid)\n                    echo "Session $f is closed, killing D-bus PID=$PID"\n                    kill -9 $PID\n                    rm $f\n                    rm ${f}.pid\n                fi\n            fi\n        done\n    popd > /dev/null\n\n    CLIENT_KEY=${SSH_CLIENT% *}\n    CLIENT_KEY=${CLIENT_KEY// /:}\n    KEY_FILE=$XDG_RUNTIME_DIR/ssh-dbus/$CLIENT_KEY\n    if [ -f $KEY_FILE ]; then\n        DBUS_SESSION_BUS_ADDRESS=$(cat $KEY_FILE)\n        DBUS_SESSION_BUS_PID=$(cat $KEY_FILE.pid)\n    else\n        echo "Launching D-bus"\n        eval $(dbus-launch)\n        echo $DBUS_SESSION_BUS_ADDRESS > $KEY_FILE\n        echo $DBUS_SESSION_BUS_PID > $KEY_FILE.pid\n    fi\nfi
```

Now that we have [docker-dbus-session-launch.sh](#) to create a D-bus instance per ssh session and [docker-remote-xauth.sh](#) to handle authentication to the remote X Server, the [remote-gnome-calculator.sh](#) launch script to allow us to use ssh X11 forwarding with a GNOME application becomes simple:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh
. $BIN/docker-command.sh
. $BIN/docker-remote-xauth.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$X11_FLAGS \
gnome-calculator
```

One refinement that was considered by the author was the idea of setting up a [reverse ssh tunnel](#) to allow the remote application to connect to the user's desktop D-bus session in a similar way to the [gnome-calculator-using-dbus-proxy.sh](#).

This approach was subsequently discounted however, because although it is technically possible to achieve the D-bus connection it doesn't help with **dconf**, because *reads are performed by direct mmap access to the database*.

6-2-2 firefox

Although X11 forwarding works well for simple applications, it is important to remember that X11 has a heritage dating back to the late 1980s. What this means in particular is that the X11 protocol was never really designed to handle graphically intensive operations requiring lots of bitmaps and textures.

At its core X11 sends a stream of display instructions, such as “draw line”, “draw rectangle” etc. from the X Client to the X Server so it can re-create the display on a local workstation, all of which needs to be done on each display refresh. This approach can be very efficient if the application comprises simple graphical primitives with low refresh rates, but modern applications tend to comprise many bitmaps, textures and fonts and often lots of animations and other eye-candy.

Another issue with X11 is that it is quite a “chatty” protocol, which is made worse by the [Xlib](#) implementation, which has a pseudo-synchronous API and implements considerable buffering. Mitigating some of this chattiness in a naturally asynchronous interface is one of the reasons [XCB](#) was designed as a potential replacement for Xlib.

Although X has had several enhancements over the years to deal with some of these issues, most of those have relied on shared memory or other IPC only available when the X Client and X Server are on the same host. When dealing with remote applications however, bandwidth and latency both start to severely impact X11 applications using X forwarding.

By way of illustration, we shall follow the pattern from the [gnome-calculator](#) that we have just developed to create a remote, containerised Firefox that is implemented using basic X11 forwarding.

The [Dockerfile](#) is in the [firefox](#) directory of [docker-gui](#) under [6-remote-applications/remote-applications-and-dbus](#) and is exactly the same Dockerfile as used in our earlier local [firefox](#) example.

```
FROM debian:stretch-slim
ENV FIREFOX_LANG=en-GB

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    wget bzip2 ca-certificates gnupg dirmngr procps \
    fonts-symbola fonts-lmodern fonts-freefont-ttf \
    fonts-liberation fonts-dejavu gsfonts \
    libgtk-3-0 libgtk2.0-0 libnss3 libxt6 libavcodec57 \
    libvpx4 libdbus-glib-1-2 libcanberra-gtk3-module \
    libpulse0 libv4l-0 libgl1-mesa-glx libgl1-mesa-dri && \
    # Debian ships with esr, so install latest Firefox.
    wget -O firefox.tar.bz2 "https://download.mozilla.org/?product=firefox-latest&os=linux64&lang=${FIREFOX_LANG}" && \
    tar xjf firefox.tar.bz2 -C /usr/lib/ && \
    rm firefox.tar.bz2 && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf
ENTRYPOINT ["/usr/lib/firefox/firefox", "--no-remote"]
```

The image may be built **on the remote host** with:

```
docker build -t firefox .
```

The [remote-firefox.sh](#) launch script is very similar to our gnome-calculator.sh script.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh
. $BIN/docker-command.sh
. $BIN/docker-remote-xauth.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$X11_FLAGS \
firefox
```

We can launch remotely with (note the syntax):

```
ssh -X <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/remote-applications-and-dbus/firefox; ./remote-firefox.sh"
```

Whilst it runs, it takes a **long** time to launch and is painfully slow to use, and although tweaks to the ssh parameters may help a fundamentally better approach to remoting is required.

6-3 Remote Audio

Before looking at alternatives to basic X11 forwarding we shall first take a short diversion and consider the problem of delivering audio from remote multimedia applications, as many of the remote display solutions that we shall cover don't include a native solution for audio forwarding.

6-3-1 pulseaudio-utils

The most obvious mechanism to enable remote audio over a network is to use PulseAudio, as it is fairly ubiquitous on most Linux distributions and desktops and is supported by *most* applications. The PulseAudio documentation does a good job of describing the various possible approaches

<https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation>

.

Note that whichever method is used to enable audio over the network, be it direct, a tunnel or Zeroconf, the PulseAudio daemon must have module-native-protocol-tcp loaded. This may be done by setting “Enable network access to local sound devices” in the the PulseAudio Preferences application (paprefs) or by updating either the /etc/pulse/default.pa or the ~/.pulse/default.pa startup script to include:

```
load-module module-native-protocol-tcp auth-ip-acl=127.0.0.1;17  
2.17.0.0/16;192.168.0.0/16 port=4713
```

With this module loaded, the server listens on port 4713 for incoming client connections. Observant readers might recall this configuration from Dockerfile-pulseaudio in the [Bluetooth and PulseAudio](#) section, though in that case our Dockerised PulseAudio daemon deliberately listened on port 4714 to avoid a collision with the default PulseAudio port.

For simplicity we shall return to the pulseaudio-utils that we explored in the [PulseAudio](#) section. The [Dockerfile](#) is in the [pulseaudio-utils](#) directory of [docker-gui](#) under [6-remote-applications/remote-audio](#) and is the same as used previously.

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    pulseaudio-utils && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

CMD ["pacat", "-v", "/dev/urandom"]
```

The image may be built on the **remote host** with:

```
docker build -t pulseaudio-utils .
```

With the PulseAudio daemon correctly set up with module-native-protocol-tcp it is, in some respects, easier to connect to from a container than when we did it locally, as we don't have to bind-mount the Unix domain socket and simply have to set PULSE_SERVER on the environment.

The main complication for setting up remote audio is actually finding a convenient mechanism for passing the address of the PulseAudio daemon to PULSE_SERVER.

It turns out however that we have already discovered that when we looked at [Remote Applications and D-bus](#), where we noted that the SSH_CLIENT environment variable is set automatically and contains the ssh client IP, which should be the IP of the user's workstation.

```
PA_HOST=$(echo ${SSH_CLIENT%% *} )
```

The [remote-noise.sh](#) launch script is very simple:

```
# If user isn't in docker group prefix docker with sudo
if id -nG $(id -un) | grep -qw docker; then
    DOCKER_COMMAND=docker
else
    DOCKER_COMMAND="sudo docker"
fi

# Get the ssh client IP so we know where to send the audio.
PA_HOST=$(echo ${SSH_CLIENT%% *} )

if [ -z $PA_HOST ]; then
    echo "This example needs to be launched via ssh to work."
else
    echo "PULSE_SERVER=$PA_HOST:4713"
    $DOCKER_COMMAND run --rm \
        -u $(id -u):$(id -g) \
        -v /etc/passwd:/etc/passwd:ro \
        -e PULSE_SERVER=$PA_HOST:4713 \
        pulseaudio-utils
fi
```

We can launch this remotely with the following, noting that the actual user name and host need to be supplied and that the path depends on where it was installed on the remote host.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/remote-audio/pulseaudio-utils; ./remote-noise.sh"
```

Note the syntax of the ssh command. The pattern of calling cd to change the working directory on the remote host means any files created by the application are relative to that directory. Note too the -t used to launch this application interactively, as it is **very loud** and killing it quickly is rather desirable.

One thing to be aware of is that whilst simply using the ssh client IP obtained from the SSH_CLIENT environment variable works very well on a private LAN network if, however, the local desktop workstation is behind a firewall, NAT, or is otherwise protected from incoming connections then the approach that has just been described won't work.

The solution to the problem is very similar to how ssh X11 forwarding works, that is to say it is necessary to set up an ssh tunnel from the remote server, forwarding port 4713 through the tunnel. The syntax for establishing the required tunnel is:

```
ssh -R 172.17.0.1:4714:localhost:4713 <user>@<remote-host>
```

This tells the remote host to bind to 172.17.0.1:4714 and forward to localhost:4713 on the client side.

Unfortunately this command normally won't "just work" as, by default, TCP listening sockets on the server will be bound to the loopback interface only and specifying a remote bind address will only succeed if the server has explicitly enabled:

```
GatewayPorts yes
```

in the /etc/ssh/sshd_config. Again, if changes are made to sshd_config then the ssh server will need to be restarted with (on systemd based distributions):

```
sudo systemctl restart ssh
```

or (on non-systemd based distributions)

```
sudo service ssh restart
```

When the server has been correctly configured then using

```
PULSE_SERVER=172.17.0.1:4714
```

will forward the audio over a secure tunnel back to the PulseAudio daemon on the local workstation.

In order to simplify our later applications and allow PulseAudio to work with both local and remote applications, we can now take what we have learned and extend our earlier docker-pulseaudio.sh into a new [docker-pulseaudio-all.sh](#) script that we can add to the [docker-gui/bin](#) directory with the other helper scripts.

```

# Get the ssh client IP so we know where to send the audio.
PA_HOST=$(echo ${SSH_CLIENT%% *})

if [ -z $PA_HOST ]; then
# SSH_CLIENT not set, so set flags for local application.

PULSE_VERSION=$(pulseaudio --version | sed 's/[^\0-9]*\([0-9]*\)\.*/\1/')
if ([[ $PULSE_VERSION -gt 6 ]] && [[ $PULSE_VERSION -lt 10 ]]);
then
    PULSE_FLAGS="-e PULSE_CLIENTCONFIG=/etc/pulse/client-noshm.
conf"
fi

# Populate PULSEAUDIO_FLAGS as a short cut instead of
# having to set the flags individually.
PULSEAUDIO_FLAGS="-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/n
ative "
PULSEAUDIO_FLAGS+=" -v $XDG_RUNTIME_DIR/pulse:$XDG_RUNTIME_DIR/p
ulse:ro "
PULSEAUDIO_FLAGS+=" -v $HOME/.config/pulse/cookie:$HOME/.config/
pulse/cookie:ro "
PULSEAUDIO_FLAGS+="$PULSE_FLAGS "

else
# SSH_CLIENT set, so set flags for remote application.
TUNNEL=$(netstat -nl | grep "0.0.0.0:4714")
TUNNEL=${TUNNEL// /}
if [ -z $TUNNEL ]; then
    # No ssh tunnel set so forward directly to client IP.
    PULSEAUDIO_FLAGS="-e PULSE_SERVER=$PA_HOST:4713 "
else
    # Tunnel set so send to tunnel endpoint.
    echo "Using SSH Tunnel PULSE_SERVER=172.17.0.1:4714"
    PULSEAUDIO_FLAGS="-e PULSE_SERVER=172.17.0.1:4714 "
fi
fi

```

The [remote-ok.sh](#) launch script is then extremely simple:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
. $BIN/docker-pulseaudio-all.sh  
  
$DOCKER_COMMAND run --rm \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
$PULSEAUDIO_FLAGS \  
-v $PWD/ok.wav:$HOME/ok.wav:ro \  
pulseaudio-utils paplay $HOME/ok.wav
```

Which may be launched remotely on a remote server that has direct network access back to the client workstation with:

```
ssh <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/remote-audio/pulseaudio-utils; ./remote-ok.sh"
```

Alternatively, where the client workstation is behind a firewall or is otherwise not directly accessible to the remote host the following command:

```
ssh -R 172.17.0.1:4714:localhost:4713 <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/remote-audio/pulseaudio-utils; ./remote-ok.sh"
```

will launch the application and forward the audio back to the client over an ssh tunnel.

6-4 Exporting the Framebuffer

Although [X11 forwarding](#) works very well for applications that render simple graphics primitives, our remote [Firefox](#) example using X forwarding neatly illustrates the limitations of that approach. The rest of this chapter shall consider, in some detail, a broad range of alternative approaches to running graphical applications remotely. All of these offer significantly better performance on average than basic X11 forwarding by having remote clients render to a *remote* framebuffer, then exporting that framebuffer in some way.

At first glance this idea seems counter-intuitive, as *clearly* sending simple primitives describing lines, rectangles, and triangles over a network is much more efficient than sending a whole framebuffer's worth of pixel data. The reality, however, is not as simple as that superficial analysis.

In modern real world scenarios very few applications actually work by sending simple X11 primitives, as illustrated by our Firefox example. Moreover, exporting an entire framebuffer as raw pixel data is a naive approach that is rarely employed by modern remote access protocols and their associated servers.

The approach actually taken by most remote access protocols is to use various techniques to efficiently compress the framebuffer data before exporting it. Many servers also attempt to send incremental updates that only export the deltas between each frame, and very often those updates are themselves also highly compressed.

On Linux based systems the typical approach employed by remote access servers is to expose an X Server on the remote host, to which remote applications can connect locally (from their perspective). In some cases this remote X Server is the remote host's own display server, but often it is a headless X Server, with some systems using a generic virtual frame buffer and others using custom X Servers (or Xorg drivers) to optimise the computation of frame deltas.

One subtle point of note is that, with many of the approaches we shall be covering, the principal aim of the remote access technology is to provide *remote desktop sessions*. Our initial goal, however, is primarily to export remote **applications** in an analogous way to X forwarding, though virtual and remote desktops will be covered in later chapters.

We begin by remoting discrete applications because Docker is typically viewed as an application abstraction, where each container has a single purpose. We shall, however, see that this can sometimes mean employing “tricks” to persuade the servers to behave in the way that we’d like.

Another point of note is that because many remote access technologies were primarily intended as remote desktop solutions most will display the application as a child of a desktop background rather than as a seamless window like X forwarding. Using Docker gives us flexibility to either live with this and have a server container serving multiple applications analogous to a remote desktop or, alternatively, have a server per application exposing each application on a different port and maximised to simulate seamless windows.

One question that might be tempting to ask is which of these technologies is “best”. The answer, however, is “it depends!”. Each technology and protocol trades a number of different factors such as bandwidth, latency, and server and client computational complexity and it will depend on the reader’s own particular set of circumstances which of these is most important to be optimal. This book simply aims to make it easy to experiment with the many different permutations.

To provide a fair comparison between the technologies covered, we shall illustrate each approach using Firefox. In part this is to allow us to focus on the *remoting* aspects, but it is also because Firefox is a fairly representative multimedia application. The author has found that use cases like playing YouTube videos and WebGL demos are an excellent way to develop a real *feel* for the *perceptual* quality of a particular approach to application remoting in real world scenarios.

To use a different application instead of Firefox, in most cases it is simply a case of using a different image name and readers are encouraged to build on what they have learned earlier in this book by running their own applications in Docker containers.

Before we begin our exploration of remote access servers, we shall first cover some general purpose client applications that support a wide range of different remote access protocols.

The following clients can be used with the majority of the server technologies we will be covering though we shall also investigate some additional, more protocol specific, clients as we cover each protocol.

6-4-1 vinagre

[Vinagre](#) is the default remote desktop client for the GNOME desktop environment and provides good support for the VNC, SSH, RDP, and SPICE protocols.

The [Dockerfile](#) may be found in the [vinagre](#) directory of [docker-gui](#) under [6-remote-applications/general-purpose-clients](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    vinagre openssh-client libvdpau1 mesa-vdpau-drivers \
    libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["vinagre"]
```

and may be built with:

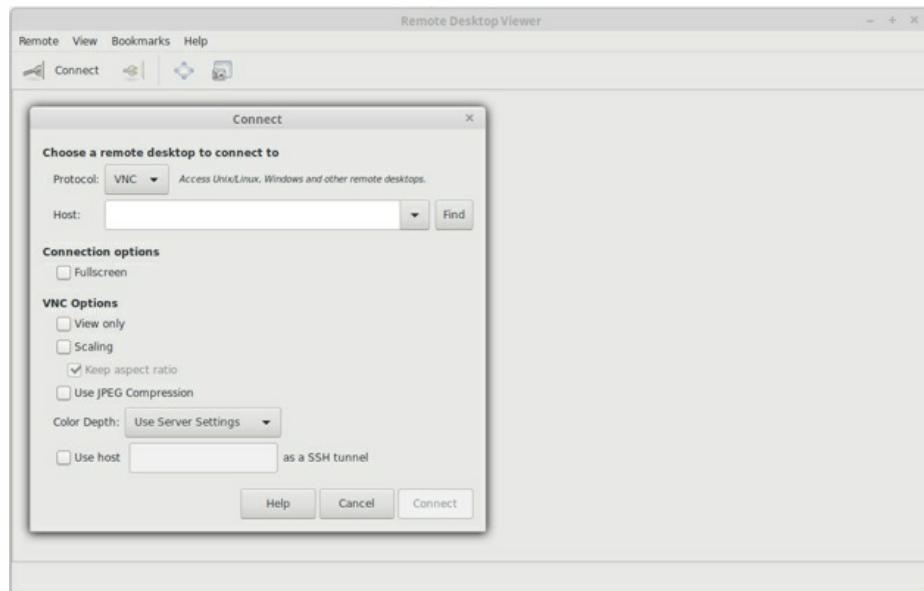
```
docker build -t vinagre .
```

This Dockerfile includes the libvdpau1 and mesa-vdpau-drivers packages, however it's not totally clear whether Vinagre actually supports VDPAU so including them is slightly speculative.

The [vinagre.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
$GPU_FLAGS \
vinagre
```



6-4-2 remmina

[Remmina](#) is a remote desktop client with a wide range of plugins providing support for the RDP, VNC, NX, XDMCP, SPICE, and SSH protocols.

The [Dockerfile](#) may be found in the [remmina](#) directory of [docker-gui](#) under [6-remote-applications/general-purpose-clients](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN echo "deb http://ftp.debian.org/debian stretch-backports main" > /etc/apt/sources.list.d/stretch-backports.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
        -t stretch-backports \
    remmina remmina-plugin-vnc remmina-plugin-rdp \
    remmina-plugin-nx remmina-plugin-spice \
    remmina-plugin-exec remmina-plugin-xdmcp \
    remmina-plugin-secret remmina-plugin-telepathy \
    libvdpau1 mesa-vdpau-drivers \
    libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/* && \
# remmina-plugin-rdp and remmina-plugin-spice
# have a PulseAudio dependency, so enable PulseAudio.
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["remmina"]
```

The remmina image may be built with:

```
docker build -t remmina .
```

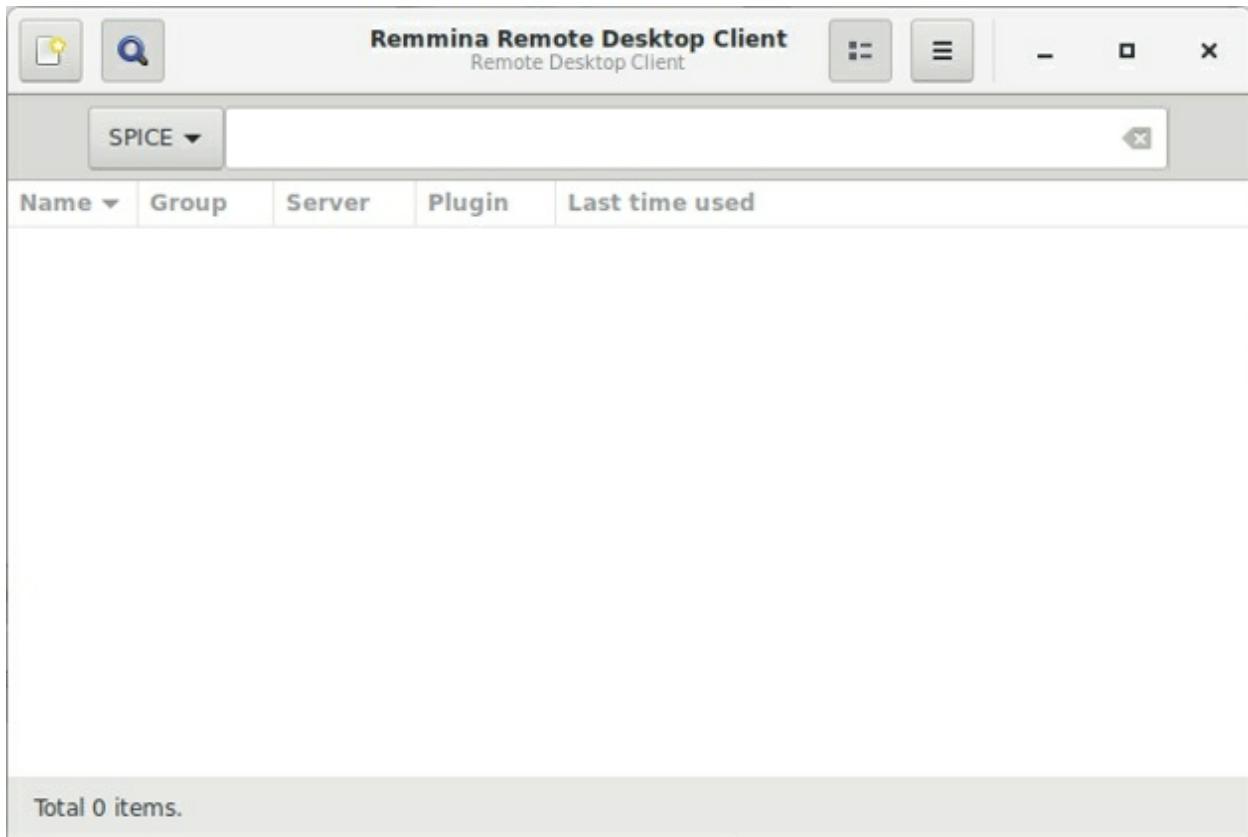
Note that Remmina is only available for Debian stretch from the backports repository, so the first step in the Dockerfile is to enable backports.

With backports enabled installing the remmina package is as simple as installing any other package, though note that the packages for each required plugin must be explicitly installed. As with Vinagre this Dockerfile speculatively includes the libvdpau1 and mesa-vdpau-drivers packages.

The [remmina.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/$(id -un):/home/$(id -un) \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    remmina
```



For protocols that natively support it, such as RDP and SPICE, Remmina and Vinagre both support streaming audio from the remote application server, so PulseAudio support has been enabled for both of these clients.

For protocols that do not natively support remote audio, such as VNC, it is necessary to use the approach described earlier in the [Remote Audio](#) section. The advantage of using a protocol's native audio support over PulseAudio streaming is that native audio generally supports compression, which can significantly reduce audio stutter over lower bandwidth connections.

6-5 VNC

[VNC](#) is a technology that enables remote desktop sharing using the [Remote Frame Buffer](#) protocol (RFB) to remotely control another computer. VNC and RFB are platform agnostic and the original source code and many modern derivatives are free and open-source.

The RFB protocol is designed to place very few requirements on the client and moreover the protocol makes the client stateless. The implication of client statelessness is that if a client disconnects from a given server then subsequently reconnects to that same server, the state of the user interface is preserved. RFB also supports multiple connections, so a different endpoint can be used to connect to the same RFB server and will see the same UI as the original endpoint.

The RFB display protocol is based on the simple primitive of a rectangle of pixel data placed at a given position, however it allows for a range of different encodings of the pixel data to trade network bandwidth, client drawing speed, and server processing speed. Most modern VNC servers support [Tight Encoding](#), which includes an option to send data rectangles encoded as [JPEG](#), and some servers also support [Tight PNG Encoding](#) which allows for lossless [PNG](#) encoding.

By using JPEG and PNG encodings clients can often benefit from offloading processing to efficient, often accelerated, libraries such as [libjpeg-turbo](#).

6-5-1 tightvnc

TightVNC was one of the first VNC servers to use the Tight Encoding scheme described previously.

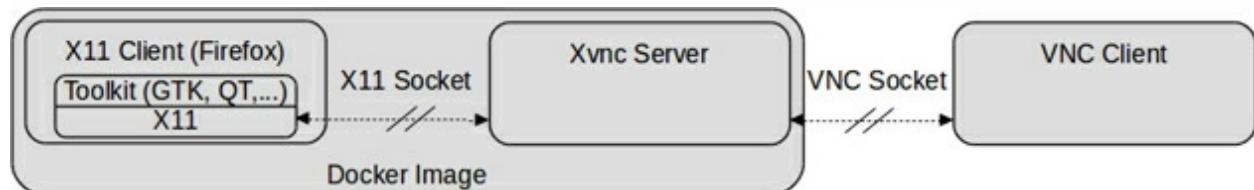
On Linux, the TightVNC vncserver application is a wrapper for Xvnc. Xvnc is based on a standard X Server, but it has a “virtual” screen rather than a physical one. Conceptually Xvnc behaves as two servers in one. From the perspective of applications Xvnc is simply an X Server and they display themselves on it as if it were a normal X display. To the remote user, however, it is a VNC server.

Architecturally this approach may be illustrated as follows:



tightvnc-firefox

One conceptually simple approach that we could take to containerise this system is to create an image that bundles the application, the client toolkit, the X11 client libraries, **and** the Xvnc server. We can then simply expose the VNC socket from the container and clients could connect quite simply.



Before describing our approach to containerising TightVNC it is worth noting that thus far the vast majority of our containers have comprised a single process, or have at least had any child processes well-managed by the application.

That is a very common model, but it allows us to gloss over something that many people forget about Docker. Each Docker container is a [PID namespace](#), which means that the processes in a container are isolated from other processes on the host. A PID namespace is a tree, which starts at PID 1, which is commonly called init.

When a Docker container is run therefore, because of PID namespaces, PID 1 is whatever has been set as ENTRYPOINT, or if there is none it's either the shell or another program, depending on the format of CMD.

This might not seem like a big deal, but the implications are profound because PID 1 has some unique responsibilities:

- When the PID 1 process dies for any reason, all other processes are killed with the KILL signal.
- When any process having children dies for any reason, its children are reparented to the process with PID 1.
- Many signals which have the default action of TERM do not have one for PID 1.

The implication of the second point above is that it is the PID 1 process that has a responsibility to reap [zombie](#) processes and in practice many applications do not do that correctly, so are unsuitable to run as PID 1 in a multi-process container.

There are a number of solutions to this in Docker. The most obvious is to bundle an init system like [supervisord](#), or even [systemd](#), in the container. Those are often overkill, however, and Docker provides the lighter and simpler option of “injecting” the [tini](#) init process that is suitable for many use cases. This can be achieved by simply including the **-init** option in the `docker run` command, which is the approach we have taken through much of this book.

For container orchestration systems like Kubernetes the approach taken is different. Kubernetes uses a [pause](#) container as PID 1 for each pod and uses PID namespace sharing so other containers in the pod have the pause process as their parent, but that is beyond the scope of this book.

The [Dockerfile](#) is in the [tightvnc-firefox](#) directory of [docker-gui](#) under [6-remote-applications/vnc/tightvnc](#).

```
FROM debian:stretch-slim

ENV FIREFOX_LANG=en-GB
ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

# Install Firefox plus tightvncserver
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    wget bzip2 ca-certificates gnupg dirmngr procps locales \
    fonts-symbola fonts-lmodern fonts-freefont-ttf \
    fonts-liberation fonts-dejavu gsffonts \
    libgtk-3-0 libgtk2.0-0 libnss3 libxt6 libavcodec57 \
    libvpx4 libdbus-glib-1-2 libcanberra-gtk3-module \
    libpulse0 libv4l-0 libgl1-mesa-glx libgl1-mesa-dri \
    tightvncserver x11-xserver-utils jwm xfonts-base \
    xfonts-75dpi xfonts-100dpi xfonts-scalable && \
```

```

wget -O firefox.tar.bz2 "https://download.mozilla.org/?product=firefox-latest&os=linux64&lang=${FIREFOX_LANG}" && \
tar xjf firefox.tar.bz2 -C /usr/lib/ && \
rm firefox.tar.bz2 && \
rm -rf /var/lib/apt/lists/* && \
sed -i "s/^# *($LANG)/\1/" /etc/locale.gen && \
locale-gen && \
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\${usr}/share/jwm/jwm-red.svg\">root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\${etc}/jwm/debian-menu</Include>//g" /etc/jwm/system.jwmrc && \
# The mode of /tmp/.X11-unix needs to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix && \
cp /etc/pulse/client.conf \
/etc/pulse/client-noshm.conf && \
sed -i "s/enable-shm = yes/enable-shm = no/g" \
/etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf
CMD vncserver $DISPLAY -ac -geometry $GEOMETRY -rfbport 5900 &&
/usr/lib/firefox/firefox --no-remote

```

The image may be built **on the remote host** with:

```
docker build -t tightvnc-firefox .
```

The starting point for the tightvnc-firefox Dockerfile was our earlier [Firefox](#) application, and most of the installed packages are Firefox dependencies. The packages specific to TightVNC are tightvncserver, x11-xserver-utils, locales, and jwm.

Whilst it is possible for Firefox to run directly as a child of the root X window that can make it behave in unexpected ways, so we include [jwm](#) as a very lightweight [window manager](#). The examples in this book use jwm simply because it is very lightweight, but behaves in a more intuitive way than [twm](#).

After installing the packages from the debian repository and downloading and installing Firefox, the Dockerfile next generates the language locale and sets up the timezone, following the pattern first described in the [Office](#) section.

After configuring the locale and timezone the Dockerfile next configures jwm by manipulating /etc/jwm/system.jwmrc via sed. This isn't strictly necessary, but for a single application the tweaks make the window manager look a little neater.

We next create the /tmp/.X11-unix directory and set its permissions. This is the default location for the X11 Unix domain socket that will be created by the X Server when it starts. Finally, we add the PulseAudio client-noshm script that was first described in the [PulseAudio](#) section.

The CMD launches the VNC server using the specified X DISPLAY and GEOMETRY then launches Firefox.

```
CMD vncserver $DISPLAY -ac -geometry $GEOMETRY -rfbport 5900 &&
/usr/lib/firefox/firefox --no-remote
```

VNC supports simple password based authentication. The [tightvnc-storepasswd.sh](#) script uses the tightvnc-firefox image, but runs the vncpasswd program as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
mkdir -p $(id -un)/.vnc  
$DOCKER_COMMAND run --rm -it \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-v $PWD/$(id -un):/home/$(id -un) \  
tightvnc-firefox vncpasswd
```

This will interactively create a VNC password file .vnc/passwd in the directory that we have bind-mounted.

The [tightvnc-firefox.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all  
. $BIN/docker-command.sh  
. $BIN/docker-pulseaudio-all.sh  
. $BIN/docker-dbus-all.sh  
  
# Create a directory on the host that we can mount as a  
# "home directory" in the container for the current user.  
mkdir -p $(id -un)/.config/pulse  
mkdir -p $(id -un)/.config/dconf  
  
# Create password if required.  
if ! test -f "$(id -un)/.vnc/passwd"; then  
    echo "creating password"  
    $(cd $(dirname $0); echo $PWD)/tightvnc-storepasswd.sh  
fi
```

```
$DOCKER_COMMAND run --rm -it \
--init \
--shm-size 2g \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
-e GEOMETRY=1280x720 \
-e USER=$(id -un) \
tightvnc-firefox
```

We can launch this remotely with the following command, noting that the actual user name and host need to be supplied and that the actual path depends on where the launch script was installed on the remote host.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/tightvnc/tightvnc-firefox; ./tightvnc-firefox.sh"
```

Note the syntax of the ssh command. Calling cd to change the working directory on the remote host means any files created by the application are relative to that directory. Note too the -t used to launch this application interactively, as the first time we launch this it will ask for the VNC password to be set.

Calling the launch script remotely in this way, or even simply launching a remote shell via ssh and then calling the launch script, are the preferred approaches and this approach is required for audio forwarding to work correctly.

If physical access is available to the remote machine it is clearly possible to run the launch script directly, however that will cause the application's audio to play directly on the remote machine. The [docker-pulseaudio-all.sh](#) script sourced by the main launch script uses the `SSH_CLIENT` environment variable to decide whether the PulseAudio daemon is local or remote to the application, as explained in [Remote Audio](#).

The Docker CMD launches the VNC server using the `X DISPLAY` and `GEOMETRY` environment variables set in the launch script. The server is also explicitly started using `-rfbport 5900` so we use the `-p` option to export that port from the container.

The initial `GEOMETRY` of `1280x720` is somewhat arbitrary and can easily be changed. Some remote servers respond automatically to client resize events, however TightVNC/Xvnc doesn't seem to.

It was mentioned earlier that [vncserver](#) is a wrapper round [Xvnc](#) and one of the things it does is create a `~/.vnc/xstartup` script that launches `/etc/X11/Xsession`, hence why jwm gets started without it *explicitly* being launched.

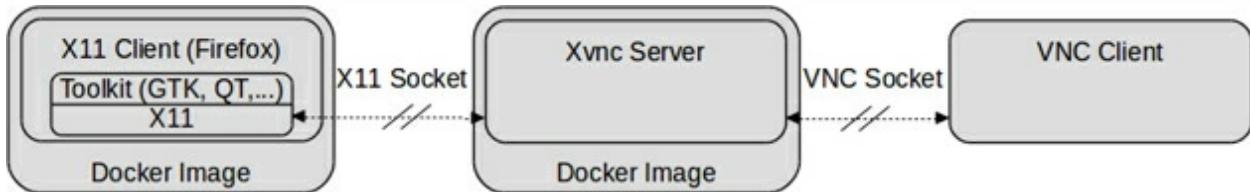
VNC clients like Vinagre or Remmina can connect via `<remote-host>:5900` and will be authenticated via the VNC password set at launch time. For additional security both Vinagre and Remmina can connect through an SSH tunnel, most easily and transparently created via the remote host's SSH server.

tightvnc

The approach taken in our `tightvnc-firefox` example of bundling the application and VNC server into a single container is perfectly reasonable, however it doesn't *really* sit well with the idea that containers should serve distinct purposes and provide separation of concerns.

Back when we first covered [Simple X11 Applications](#) and discussed sharing the X11 socket with the application, we noted that the X11 socket being passed to the application container does not *necessarily* have to be the *host's* X11 socket. We briefly mentioned the idea of a modular solution whereby a second container could bundle a headless X Server, VNC server etc. and we could instead expose that X Server's X11 socket to the application's container.

Architecturally this approach may be illustrated as follows:



Here we have one container whose purpose is to act as the application's framebuffer and export that to remote clients, and other containers representing the applications. Depending on our requirements we could have a single “framebuffer server” instance serving multiple applications or even one per application, serving each instance on a different port.

We shall adopt this pattern for the rest of the chapter.

The [Dockerfile](#) is in the [tightvnc](#) directory of [docker-gui](#) under [6-remote-applications/vnc/tightvnc](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    tightvncserver libgl1-mesa-glx libgl1-mesa-dri \
    x11-xserver-utils locales jwm xfonts-base \
    xfonts-75dpi xfonts-100dpi xfonts-scalable && \
    rm -rf /var/lib/apt/lists/* && \
    sed -i "s/^# *\$LANG\$/\1/" /etc/locale.gen && \
    locale-gen && \
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
    # Tidy up JWM for single app use case
    sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
    sed -i "s/<TrayButton icon=\"\${usr}\${share}\${jwm}\${jwm-red.svg}\">root:1</TrayButton>/\g" /etc/jwm/system.jwmrc && \
    sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>/\g" /etc/jwm/system.jwmrc && \
    sed -i "s/<Include>\${etc}\${jwm}\${debian-menu}</Include>/\g" /etc/jwm/system.jwmrc && \
    # We export /tmp/.X11-unix as a volume and we need
    # the mode of /tmp/.X11-unix to be set to 1777
    mkdir /tmp/.X11-unix && \
    chmod 1777 /tmp/.X11-unix

VOLUME /tmp/.X11-unix
CMD vncserver $DISPLAY -ac -geometry $GEOMETRY -rfbport 5900 &&
tail -F ~/.vnc/*.log
```

The tightvnc image may be built **on the remote host** with:

```
docker build -t tightvnc .
```

We see that this Dockerfile is significantly simpler than the one for tightvnc-firefox, as we have removed all of the Firefox dependencies and retained only what is required for the TightVNC server, which includes the locale, timezone and window manager configuration.

In addition to creating the X11 socket directory and setting its permissions as before, one important difference with this Dockerfile is exporting the X11 socket directory as a volume:

```
VOLUME /tmp/.X11-unix
```

This is important, as our application containers shall be connecting to the X Server in the tightvnc containers rather than the host's display server as we have previously done.

The [tightvnc-storepasswd.sh](#) script that interactively sets the VNC password is as before:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh

mkdir -p $(id -un)/.vnc
$DOCKER_COMMAND run --rm -it \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
tightvnc vncpasswd
```

The [tightvnc.sh](#) launch script is as follows, noting that this script also expects the [firefox](#) image that we developed previously to be available on the **remote host** :

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio-all.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
mkdir -p $(id -un)\\.config/dconf

# Create password if required.
if ! test -f "$(id -un)\\.vnc/passwd"; then
    echo "creating password"
    $(cd $(dirname $0); echo $PWD)/tightvnc-storepasswd.sh
fi

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=tightvnc \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=:1 \
-e GEOMETRY=1280x720 \
-e USER=$(id -un) \
tightvnc
```

```
$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:tightvnc \
--volumes-from tightvnc \
firefox

$DOCKER_COMMAND stop tightvnc
```

This script first checks if the VNC password has been set, and if not calls [tightvnc-storepasswd.sh](#).

Next it launches tightvnc, which is similar to how tightvnc-firefox was launched. Again we have included `--init` to run `tini` as PID 1, but this time we have launched the container with a name “tightvnc” and have daemonised it via `-d`.

The next part of the script launches Firefox. This is somewhat similar to how we launched Firefox locally, however we now don’t need the `X11_FLAGS` used to connect to the host’s display server, but we must add the following important detail:

```
--ipc=container:tightvnc \
--volumes-from tightvnc \
```

These two lines are very important and are what enables us to break apart our previous, slightly monolithic, tightvnc-firefox into two containers to give a more modular architecture. The first line tells the Firefox application container to share the tightvnc container's IPC, which is needed to enable shared memory between the containers and is very important for efficiency. The second line tells the application container to share the tightvnc container's volumes, which is needed so the application container can see the X11 Unix domain socket.

As before we can launch this remotely with the following command, noting that the actual user name and host need to be supplied and that the actual path depends on where the launch script was installed on the remote host.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/tightvnc/tightvnc; ./tightvnc.sh"
```

Again the DISPLAY and GEOMETRY environment variables used to configure the VNC server are set in the launch script.

Despite the significant changes to the overall architecture. the user experience is largely the same as it was with our more monolithic version as the VNC server is first launched then the application container is launched and connects to the VNC server's X Server.

To keep things simple and tidy, the Firefox container is not daemonised and so blocks until the user quits out of Firefox, the last line of the script then stops the tightvnc container.

6-5-2 x11vnc

Whilst TightVNC uses Xvnc, which acts as both an X Server and a VNC server, [x11vnc](#) takes a somewhat different approach. With x11vnc it is possible to connect to an existing **real** X11 display, as x11vnc polls using [XShmGetImage](#) to retrieve a copy of the X Server's framebuffer. On each poll the copied frame buffers are compared to see which regions have changed and thus which regions need to be sent via VNC.

Real X11 Display

The first x11vnc example we shall cover simply connects to a running X Server and exports the framebuffer via VNC.

The [Dockerfile](#) is in the [x11vnc](#) directory of [docker-gui](#) under [6-remote-applications/vnc/x11vnc](#).

```
FROM debian:stretch-slim

# Install x11vnc
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    x11vnc && \
    rm -rf /var/lib/apt/lists/*

CMD x11vnc -forever -usepw
```

The x11vnc image may be built **on the remote host** with:

```
docker build -t x11vnc .
```

The [x11vnc-storepasswd.sh](#) script that interactively sets the VNC password is very similar to the one we used for TightVNC, though here we set the password with `x11vnc -storepasswd` not `vncpasswd`:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
mkdir -p $(id -un)/.vnc  
$DOCKER_COMMAND run --rm -it \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-v $PWD/$(id -un):/home/$(id -un) \  
x11vnc x11vnc -storepasswd
```

The [x11vnc.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-xauth.sh  
. $BIN/docker-command.sh  
  
mkdir -p $(id -un)  
if ! test -f "$(id -un)/.vnc/passwd"; then  
    echo "creating password"  
    $(cd $(dirname $0); echo $PWD)/x11vnc-storepasswd.sh  
fi  
  
$DOCKER_COMMAND run --rm -it \  
--ipc=host \  
-p 5900:5900 \  
-u $(id -u):$(id -g) \  
-v $PWD/$(id -un):/home/$(id -un) \  
-v /etc/passwd:/etc/passwd:ro \  
$X11_FLAGS \  
-e DISPLAY=:0 x11vnc
```

This script first checks if the VNC password has been set, and if not calls [x11vnc-storepasswd.sh](#).

Next the script launches x11vnc, which follows a very similar process to that used to launch local X11 applications.

There are a couple of subtle details to be aware of however. Firstly, we have set `--ipc=host`. This shares the host's IPC, including SysV shared memory, with the container and is needed by x11vnc in order to run XShmGetImage. Secondly, we have explicitly set `DISPLAY=:0`. Normally when we have used `X11_FLAGS` the `DISPLAY` is set to the user's `DISPLAY`, but as we saw when we covered [X11 Forwarding](#) when we use ssh the user's `DISPLAY` variable is not set.

As before we can launch this remotely with the following command, noting that the actual user name and host need to be supplied and that the actual path depends on where the launch script was installed on the remote host.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/x11vnc/x11vnc; ./x11vnc.sh"
```

When connecting to a running X11 display server using this method the most useful way to forward audio is to set up a [tunnel](#) between the local and remote PulseAudio daemons. The easiest way to do this is to install **paprefs** on both machines and enable zeroconf discovery.

Xvfb

Although x11vnc's primary use case is to allow VNC connections to be made to running X Servers associated with physical graphics hardware, it can also attach to virtual X servers. This allows us to create a container running x11vnc with a headless X Server that behaves in a similar way to our earlier TightVNC example.

The first headless X Server we shall cover is the X virtual framebuffer [Xvfb](#)

The [Dockerfile](#) is in the [x11vnc-xvfb](#) directory of [docker-gui](#) under [6-remote-applications/vnc/x11vnc](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

# Install x11vnc and xvfb
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libgl1-mesa-glx libgl1-mesa-dri \
    x11vnc xvfb locales jwm && \
    rm -rf /var/lib/apt/lists/* && \
# Create script to start Xvfb, jwm and x11vnc
echo '#!/bin/bash\nXvfb $DISPLAY -screen 0 $GEOMETRY -cc 4 & \
nsleep 0.5\njwm &\nx11vnc -forever -usepw' > /usr/local/bin/sta
rtup && \
    chmod +x /usr/local/bin/startup && \
# Generate locales
sed -i "s/^# *`($LANG)`/\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
```

```

DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\usr\share\jwm\jwm-red.svg\">root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">>showdesktop</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\etc\jwm\debian-menu</Include>//g" /etc/jwm/system.jwmrc && \
# We export /tmp/.X11-unix as a volume and we need
# the mode of /tmp/.X11-unix to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix

VOLUME /tmp/.X11-unix

CMD ["/usr/local/bin/startup"]

```

The image may be built **on the remote host** with:

```
docker build -t x11vnc-xvfb .
```

This Dockerfile is very similar to the one we created for TightVNC, but as the startup is little more involved a simple script is created in /usr/local/bin/startup as follows:

```

#!/bin/bash
Xvfb $DISPLAY -screen 0 $GEOMETRY -cc 4 &
sleep 0.5
jwm &
x11vnc -forever -usepw

```

This script launches Xvfb on the specified DISPLAY with the required GEOMETRY, then briefly sleeps to ensure X has started before starting the window manager and x11vnc.

The [x11vnc-storepasswd.sh](#) script is similar to the one for x11vnc but with a different image name:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
mkdir -p $(id -un)/.vnc  
$DOCKER_COMMAND run --rm -it \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-v $PWD/$(id -un):/home/$(id -un) \  
x11vnc-xvfb x11vnc -storepasswd
```

The [x11vnc-xvfb.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all  
. $BIN/docker-command.sh  
. $BIN/docker-pulseaudio-all.sh  
. $BIN/docker-dbus-all.sh  
  
# Create a directory on the host that we can mount as a  
# "home directory" in the container for the current user.  
mkdir -p $(id -un)/.config/pulse  
mkdir -p $(id -un)/.config/dconf  
  
# Create password if required.  
if ! test -f "$(id -un)/.vnc/passwd"; then  
    echo "creating password"  
    $(cd $(dirname $0); echo $PWD)/x11vnc-storepasswd.sh  
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=x11vnc-xvfb \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=:1 \
-e GEOMETRY=1280x720x24 \
x11vnc-xvfb

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:x11vnc-xvfb \
--volumes-from x11vnc-xvfb \
firefox

$DOCKER_COMMAND stop x11vnc-xvfb

```

This is similar to our earlier [tightvnc.sh](#) launch script, with the application again sharing the server's IPC and volumes.

As before, we can launch this remotely with a command based on the following with the appropriate user, host, and path:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/x11vnc/x11vnc-xvfb; ./x11vnc-xvfb.sh"
```

Xdummy

Although Xvfb works well and is simple to use it does not support some useful features like [RandR](#). A more modern alternative is Xdummy, which was originally a script written by the author of x11vnc to allow the [xorg-video-dummy](#) dummy video driver to be used by non-root users, but now simply requires the appropriate Xorg configuration.

The [Dockerfile](#) is in the [x11vnc-xdummy](#) directory of [docker-gui](#) under [6-remote-applications/vnc/x11vnc](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

# Install x11vnc and xserver-xorg-video-dummy
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libgl1-mesa-glx libgl1-mesa-dri \
    x11vnc xserver-xorg-video-dummy locales jwm && \
    rm -rf /var/lib/apt/lists/* && \
    # Create script to start Xorg, jwm and x11vnc
    echo '#!/bin/bash\nXorg $DISPLAY -cc 4 &\nsleep 0.5\njwm &\nx11vnc -forever -usepw' > /usr/local/bin/startup && \
    chmod +x /usr/local/bin/startup && \
    # Generate locales
    sed -i "s/^# *\"$LANG\"/\1/" /etc/locale.gen && \
    locale-gen && \
    # Set up the timezone
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
```

```

# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\usr\share\jwm\jwm-red.svg\"
>root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\etc\jwm\debian-menu</Include>//g" /etc/jwm/system.jwmrc && \
# We export /tmp/.X11-unix as a volume and we need
# the mode of /tmp/.X11-unix to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix

VOLUME /tmp/.X11-unix
COPY xorg.conf /etc/X11/xorg.conf
CMD ["/usr/local/bin/startup"]

```

The image may be built **on the remote host** with:

```
docker build -t x11vnc-xdummy .
```

Like the Dockerfile for x11vnc-xvfb we create a simple startup script in /usr/local/bin/startup:

```

#!/bin/bash
Xorg $DISPLAY -cc 4 &
sleep 0.5
jwm &
x11vnc -forever -usepw

```

In this case though we start Xorg rather than Xvfb and the display size is set in the Xorg configuration file rather than the GEOMETRY environment variable.

We COPY our [xorg.conf](#) to /etc/X11/xorg.conf, though bind-mounting it would allow for easier changes. This configuration originally came from <https://xpra.org/xorg.conf>.

The [x11vnc-storepasswd.sh](#) script is identical to the one for x11vnc-xvfb apart from the image name:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
mkdir -p $(id -un)/.vnc  
$DOCKER_COMMAND run --rm -it \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-v $PWD/$(id -un):/home/$(id -un) \  
x11vnc-xdummy x11vnc -storepasswd
```

The [x11vnc-xdummy.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all  
. $BIN/docker-command.sh  
. $BIN/docker-pulseaudio-all.sh  
. $BIN/docker-dbus-all.sh  
  
# Create a directory on the host that we can mount as a  
# "home directory" in the container for the current user.  
mkdir -p $(id -un)/.config/pulse  
mkdir -p $(id -un)/.config/dconf  
  
# Create password if required.  
if ! test -f "$(id -un)/.vnc/passwd"; then  
    echo "creating password"  
    $(cd $(dirname $0); echo $PWD)/x11vnc-storepasswd.sh  
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=x11vnc-xdummy \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=:1 \
x11vnc-xdummy

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:x11vnc-xdummy \
--volumes-from x11vnc-xdummy \
firefox

$DOCKER_COMMAND stop x11vnc-xdummy

```

This is similar to our x11vnc-xvfb launch script, again the application container shares the server's IPC and volumes.

As before, we can launch this remotely with a command based on the following with the appropriate user, host and path:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/x11vnc/x11vnc-xdummy; ./x11vnc-xdummy.sh"
```

6-5-3 tigervnc

[TigerVNC](#) started as a next-generation development effort based on the VNC 4 branch of TightVNC, but subsequently forked. TigerVNC supports a variant of Tight Encoding that is greatly accelerated by the use of the libjpeg-turbo JPEG codec.

tigervnc

As TigerVNC was originally based on TightVNC creating a containerised version follows a similar pattern.

The [Dockerfile](#) is in the [tigervnc](#) directory of [docker-gui](#) under [6-remote-applications/vnc/tigervnc](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

# Install tigervnc-standalone-server
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libgl1-mesa-glx libgl1-mesa-dri tigervnc-common \
    tigervnc-standalone-server locales jwm xfonts-base \
    xfonts-75dpi xfonts-100dpi xfonts-scalable && \
    rm -rf /var/lib/apt/lists/* && \
# Create simple script to start Xtigervnc and jwm
echo '#!/bin/bash\nXtigervnc $DISPLAY -ac -shmem -geometry $G
EOMETRY -rfbport 5900 -rfbauth ~/.vnc/passwd &\nsleep 0.5\njwm'
> /usr/local/bin/startup && \
chmod +x /usr/local/bin/startup && \
# Generate locales
sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
locale-gen && \
```

```

# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\\"/usr\share\jwm\jwm-red.svg\\">root:1<\TrayButton>/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\\">showdesktop<\TrayButton>/g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\etc\jwm\debian-menu<\Include>/g" /etc/jwm/system.jwmrc && \
# We export /tmp/.X11-unix as a volume and we need
# the mode of /tmp/.X11-unix to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix

VOLUME /tmp/.X11-unix

CMD ["/usr/local/bin/startup"]

```

The tigervnc image may be built **on the remote host** with:

```
docker build -t tigervnc .
```

In this Dockerfile we create a script to launch Xtigervnc at startup. Xtigervnc is the TigerVNC equivalent of Xvnc, and like Xvnc acts as both an X Server and a VNC server.

In the TigerVNC [documentation](#) the programs are referred to as vncserver, Xvnc etc., however some distributions like Debian name them tigervncserver and Xtigervnc presumably to avoid conflict with TightVNC installations.

As with TightVNC the DISPLAY and GEOMETRY of the X Server are configurable via environment variables passed to the container at startup.

The [tigervnc-storepasswd.sh](#) script is identical to the one for TightVNC apart from the image name:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
mkdir -p $(id -un)/.vnc  
$DOCKER_COMMAND run --rm -it \  
-u $(id -u):$(id -g) \  
-v /etc/passwd:/etc/passwd:ro \  
-v $PWD/$(id -un):/home/$(id -un) \  
tigervnc vncpasswd
```

The [tigervnc.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all  
. $BIN/docker-command.sh  
. $BIN/docker-pulseaudio-all.sh  
. $BIN/docker-dbus-all.sh  
  
# Create a directory on the host that we can mount as a  
# "home directory" in the container for the current user.  
mkdir -p $(id -un)/.config/pulse  
mkdir -p $(id -un)/.config/dconf  
  
# Create password if required.  
if ! test -f "$(id -un)/.vnc/passwd"; then  
    echo "creating password"  
    $(cd $(dirname $0); echo $PWD)/tigervnc-storepasswd.sh  
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=tigervnc \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=:1 \
-e GEOMETRY=1280x720 \
tigervnc

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:tigervnc \
--volumes-from tigervnc \
firefox

$DOCKER_COMMAND stop tigervnc

```

This is similar to the TightVNC launch script, with the application container again sharing the server's IPC and volumes.

We can launch this remotely with a command based on the following with appropriate user, host, and path:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/tigervnc/tigervnc; ./tigervnc.sh"
```

Xdummy

In addition to the standalone server described in the previous section TigerVNC also provides an [undocumented](#) Xorg extension that may be used with a standard Xorg X Server.

The [Dockerfile](#) may be found in the [tigervnc-xdummy](#) directory of the [docker-gui](#) repository under [6-remote-applications/vnc/tigervnc](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libgl1-mesa-glx libgl1-mesa-dri \
    xserver-xorg-video-dummy tigervnc-xorg-extension \
    tigervnc-common locales jwm && \
    rm -rf /var/lib/apt/lists/* && \
    # Create simple launch script to start Xorg and jwm. Need
    # to create a symlink from real password file at run-time
    # as xorg.conf options seem to need absolute paths.
    mkdir /usr/local/etc/.vnc && \
    chmod 777 /usr/local/etc/.vnc && \
    echo '#!/bin/bash\nln -s ~/.vnc/passwd /usr/local/etc/.vnc/pa
sswd\nXorg $DISPLAY -cc 4 &\nsleep 0.5\njwm' > /usr/local/bin/s
tartup && \
    chmod +x /usr/local/bin/startup && \
    # Generate locales
    sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
    locale-gen && \
    # Set up the timezone
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
```

```

# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\usr\share\jwm\jwm-red.svg\"
>root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\etc\jwm\debian-menu</Include>//g" /etc/jwm/system.jwmrc && \
# We export /tmp/.X11-unix as a volume and we need
# the mode of /tmp/.X11-unix to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix

VOLUME /tmp/.X11-unix

COPY xorg.conf /etc/X11/xorg.conf

CMD ["/usr/local/bin/startup"]

```

The image may be built **on the remote host** with:

```
docker build -t tigervnc-xdummy .
```

This Dockerfile creates a simple startup script in /usr/local/bin/startup to launch Xorg and jwm:

```

#!/bin/bash
ln -s ~/.vnc/passwd /usr/local/etc/.vnc/passwd
Xorg $DISPLAY -cc 4 &
sleep 0.5
jwm

```

The first line is because we want /.vnc/passwd to be relative to ~ but xorg.conf options seems to need absolute paths.

Again we COPY our [xorg.conf](#) to /etc/X11/xorg.conf, though we could bind-mount it instead to make changes easier.

One issue encountered by the author when trying to use the TigerVNC Xorg extension is that it is, unfortunately, largely [undocumented](#). The solution was *eventually* found when an Internet search of “libvnc.so configuration” yielded the links:

<https://ubuntuforums.org/archive/index.php/t-279069.html>

<https://github.com/TigerVNC/tigervnc/blob/master/contrib/packages/libvnc.conf>

The following xorg.conf fields are the relevant ones:

```
Section "Module"
    Load "vnc"
EndSection

Section "Screen"
    ...
    Option "SecurityTypes" "VncAuth"
    Option "UserPasswdVerifier" "VncAuth"
    Option "PasswordFile" "/usr/local/etc/.vnc/passwd"
    Option "rfbport" "5900"
EndSection
```

It is also worth mentioning that the xorg.conf files presented in this book have a few other changes from the original

<https://xpra.org/xorg.conf>. In particular our ones simply set

Modes "1280x720" to configure the geometry, and we comment out the line Virtual 8192 4096, the latter fixes what appears to be a DPI issue that causes huge mouse pointers to be displayed.

The [tigervnc-storepasswd.sh](#) script is identical to the one for TightVNC apart from the image name:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh

mkdir -p $(id -un)/.vnc
$DOCKER_COMMAND run --rm -it \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
tigervnc-xdummy vncpasswd
```

The [tigervnc-xdummy.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio-all.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -f "$(id -un)/.vnc/passwd"; then
    echo "creating password"
    $(cd $(dirname $0); echo $PWD)/tigervnc-storepasswd.sh
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=tigervnc-xdummy \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=:1 \
tigervnc-xdummy

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:tigervnc-xdummy \
--volumes-from tigervnc-xdummy \
firefox

$DOCKER_COMMAND stop tigervnc-xdummy

```

This is similar to the TightVNC launch script, with the application container again sharing the server container's IPC and volumes.

We can launch this remotely with a command based on the following, with the appropriate user, host and path:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/tigervnc/tigervnc-xdummy; ./tigervnc-xdummy.sh"
```

tigervnc viewer

TigerVNC provides its own VNC viewer as an alternative client to Vinagre or Remmina for VNC connections.

The [Dockerfile](#) may be found in the [tigervnc-viewer](#) directory of [docker-gui](#) under [6-remote-applications/vnc/tigervnc](#).

```
FROM debian:stretch-slim

# Install tigervnc-viewer
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    tigervnc-viewer && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["xtigervncviewer"]
```

The image may be built with:

```
docker build -t tigervnc-viewer .
```

The [tigervnc-viewer.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh

mkdir -p $(id -un)
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/$(id -un):/home/$(id -un) \
    $X11_FLAGS \
    tigervnc-viewer
```

6-5-4 turbovnc

[TurboVNC](#) was originally a fork of TightVNC 1.3 and also shares some heritage with TigerVNC. The primary focus of TurboVNC, however, is on 3D and video workloads so the developers have made several codec optimisations and enabled client double buffering to enhance the performance and hide latency for such applications.

The [Dockerfile](#) is in the [turbovnc](#) directory of [docker-gui](#) under [6-remote-applications/vnc/turbovnc](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London
ENV TVNC_WM=/usr/bin/jwm

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl ca-certificates locales libssl1.0.2 jwm \
    libgl1-mesa-glx libgl1-mesa-dri xauth \
    x11-xkb-utils xkb-data x11-xserver-utils xfonts-base \
    xfonts-75dpi xfonts-100dpi xfonts-scalable && \
    # Attempt to work out the latest turbovnc version from
    # https://sourceforge.net/projects/turbovnc/files/
    TVNC_VERSION=$(curl -sSL https://sourceforge.net/projects/turbovnc/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 |
    cut -d \> -f2 | cut -d \< -f1) && \
    echo "turbovnc version: ${TVNC_VERSION}" && \
    curl -sSL https://sourceforge.net/projects/turbovnc/files/${TVNC_VERSION}/turbovnc_${TVNC_VERSION}_amd64.deb -o turbovnc_${TVNC_VERSION}_amd64.deb && \
    dpkg -i turbovnc_*_amd64.deb && \
```

```

# Tidy up packages only used for installing turbovnc.
rm turbovnc_*_amd64.deb && \
apt-get clean && \
apt-get purge -y curl ca-certificates && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/* && \
# Generate locales
sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\\usr\\share\\jwm\\jwm-red.svg\\">>root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\\etc\\jwm\\debian-menu</Include>//g" /etc/jwm/system.jwmrc && \
# We export /tmp/.X11-unix as a volume and we need
# the mode of /tmp/.X11-unix to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix

VOLUME /tmp/.X11-unix

CMD /opt/TurboVNC/bin/vncserver $DISPLAY -ac -fg -geometry $GEO
METRY -nohttpd -rfbport 5900

```

The turbovnc image may be built **on the remote host** with:

```
docker build -t turbovnc .
```

In most respects the TurboVNC Dockerfile is very similar to the Dockerfiles we created for TightVNC, TigerVNC and x11vnc, where we generate the locale and timezone then configure jwm and finally export /tmp/.X11-unix as a volume.

The main difference is that TurboVNC is not available in the Debian package repositories, so the TurboVNC deb package must be directly downloaded from <https://sourceforge.net/projects/turbovnc/files/>

The Dockerfile attempts to “scrape” the most recent TurboVNC version from the download page via:

```
TVNC_VERSION=$(curl -SSL https://sourceforge.net/projects/turbo  
vnc/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 | cu  
t -d \> -f2 | cut -d \< -f1)
```

This works, however it is fragile and changes to the format of the TurboVNC sourceforge download page could cause it to fail. If that happens then simply replacing that line with a specific TurboVNC version should fix it, e.g.

```
TVNC_VERSION=2.2.3
```

After downloading and installing the TurboVNC deb package the Dockerfile then removes the downloaded deb file and other packages that were only required to download and install TurboVNC.

The [turbovnc-storepasswd.sh](#) script is similar to the TightVNC one apart from the image name and vncpasswd path:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh

mkdir -p $(id -un)/.vnc
$DOCKER_COMMAND run --rm -it \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v ${PWD}/$(id -un):/home/$(id -un) \
turbovnc /opt/TurboVNC/bin/vncpasswd
```

The [turbovnc.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio-all.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -f "$(id -un)/.vnc/passwd"; then
    echo "creating password"
    $(cd $(dirname $0); echo $PWD)/turbovnc-storepasswd.sh
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=turbovnc \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e DISPLAY=:1 \
-e GEOMETRY=1280x720 \
turbovnc

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:turbovnc \
--volumes-from turbovnc \
firefox

$DOCKER_COMMAND stop turbovnc

```

This is similar to the TightVNC launch script, with the application container again sharing the server container's IPC and volumes.

We can launch this remotely with a command based on the following, with the appropriate user, host and path:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/turbovnc/turbovnc; ./turbovnc.sh"
```

turbovnc viewer

The TurboVNC installation comes with a Java based VNC viewer, which we can deploy to a container fairly easily.

The [Dockerfile](#) is in the [turbovnc-viewer](#) directory of [docker-gui](#) under [6-remote-applications/vnc/turbovnc](#).

```
FROM debian:stretch-slim

RUN mkdir -p /usr/share/man/man1 && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl ca-certificates default-jre && \
    # Attempt to work out the latest turbovnc version from
    # https://sourceforge.net/projects/turbovnc/files/
    TVNC_VERSION=$(curl -sSL https://sourceforge.net/projects/turbovnc/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 | \
    cut -d \> -f2 | cut -d \<; -f1) && \
    echo "turbovnc version: ${TVNC_VERSION}" && \
    curl -sSL https://sourceforge.net/projects/turbovnc/files/${TVNC_VERSION}/turbovnc_${TVNC_VERSION}_amd64.deb -o turbovnc_${TVNC_VERSION}_amd64.deb && \
    dpkg -i turbovnc_*_amd64.deb && \
    # Tidy up packages only used for installing turbovnc.
    rm turbovnc_*_amd64.deb && \
    apt-get clean && \
    apt-get purge -y curl && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["/opt/TurboVNC/bin/vncviewer"]
```

The image may be built with:

```
docker build -t turbovnc-viewer .
```

As the TurboVNC viewer is actually deployed as part of the the main TurboVNC deb package, the majority of this Dockerfile simply repeats what was done in the TurboVNC server. Again we “scrape” the most recent version from the download page then download the package and install it via dpkg, before removing the downloaded deb file and other packages that were only required to download and install TurboVNC.

As this is a Java application we need to install default-jre so, as previously mentioned in our [libreoffice](#) example, because we are using a slim base image we have to add:

```
mkdir -p /usr/share/man/man1
```

or the docker build will fail, see

<https://github.com/debuerreotype/debuerreotype/issues/10>

The [turbovnc-viewer.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm -it \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$X11_FLAGS \
turbovnc-viewer
```

6-6 RDP

The Remote Desktop Protocol ([RDP](#)) is a proprietary Microsoft extension of the ITU-T [T.128](#) multipoint application sharing protocol that was also known as T.Share.

RDP is an extensible protocol and can support up to 64,000 independent data transmission channels allowing separate virtual channels for carrying device communication and presentation data from the server.

RDP supports several mechanisms to reduce network bandwidth including data compression, persistent caching of bitmaps, and caching of glyphs and fragments in RAM. The persistent bitmap cache can provide a significant improvement in performance over low-bandwidth connections, especially when running applications that make extensive use of large bitmaps.

The RDP protocol supports many useful features like persistent sessions, clipboard mapping, print redirection, audio forwarding and even USB forwarding, though not all implementations of the protocol support all available features.

Whilst VNC connects a remote user to the computer itself by sharing its screen, keyboard, and mouse, by contrast RDP logs in a remote user by effectively creating a real desktop session on the server. This difference is subtle, but has implications for *application* remoting as we shall shortly discover.

6-6-1 xrdp

Although RDP is a proprietary Microsoft protocol [xrdp](#) is a free and open-source RDP server implementation that enables operating systems other than Microsoft Windows to provide an RDP compatible remote desktop. Whilst older versions of xrdp used a VNC server as a display relay, newer versions use the [xorgxrdp](#) Xorg driver to map X11 to RDP more natively.

xrdp accepts connections from a variety of RDP clients including Vinagre, Remmina, FreeRDP, rdesktop, NeutrinoRDP and Microsoft Remote Desktop Client (for Windows, Mac OS, iOS and Android).

The [Dockerfile](#) is in the [xrdp](#) directory of [docker-gui](#) under [6-remote-applications/rdp](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN \
    sed -i 's/main/main non-free/' /etc/apt/sources.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    wget apt-utils git autoconf automake libtool x11-utils \
    software-properties-common libssl-dev libpam0g-dev \
    nasm xsllibc flex bison pkg-config locales libcap-dev \
    libx11-dev xserver-xorg-dev xserver-xorg-core \
    libxf86-drv libxrandr-dev libxml2-dev dpkg-dev \
    libmp3lame-dev libopus-dev libfdk-aac-dev libjpeg-dev \
    libturbojpeg0-dev libpixman-1-dev libfuse-dev \
    libgl1-mesa-glx libgl1-mesa-dri jwm xauth xfonts-base \
    xfonts-75dpi xfonts-100dpi xfonts-scalable && \
```

```
# Install tini as we need tini's -g option so every
# process in the group gets the signal
wget -O /sbin/init https://github.com/krallin/tini/releases/d
ownload/v0.18.0/tini && \
chmod +x /sbin/init && \
# Generate locales
sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/j
wm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\\usr\\share\\jwm\\jwm-red.svg\\"
>root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>//
g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\\etc\\jwm\\debian-menu</Include>//g" /et
c/jwm/system.jwmrc && \
# Clone xrdp and xorgxrdp source from GitHub and build.
cd /usr/src && \
git clone --recursive \
https://github.com/neutrinoLabs/xrdp.git && \
git clone \
https://github.com/neutrinoLabs/xorgxrdp.git && \
cd xrdp && git checkout v0.9.12 -b build && \
# The default xrdp behaviour is to defer starting Xorg
# until session start, so we patch session.c to prevent
# it skipping the display we will be pre-starting.
sed -i 's/!x_server_running_check_ports(display)/1/' \
/usr/src/xrdp/sesman/session.c && \
./bootstrap && \
```

```

./configure --enable-opus --enable-fuse --enable-jpeg \
--enable-tjpeg --enable-ipv6 --enable-vsock \
--enable-pixman --enable-mp3lame \
--enable-fdkaac && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
cd ../xorgxrdp && git checkout v0.2.12 -b build && \
./bootstrap && ./configure && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
sed -i 's/ssl_protocols=TLSv1.2, TLSv1.3/ssl_protocols=TLSv1. \
2/' /etc/xrdp/xrdp.ini && \
# Make the log files writeable so xrdp needn't be root.
touch /var/log/xrdp.log && \
chmod 1666 /var/log/xrdp.log && \
touch /var/log/xrdp-sesman.log && \
chmod 1666 /var/log/xrdp-sesman.log && \
chmod 644 /etc/xrdp/rsakeys.ini && \
chmod 644 /etc/xrdp/cert.pem && \
chmod 644 /etc/xrdp/key.pem && \
chmod 666 /etc/xrdp/sesman.ini && \
# We export /tmp/.X11-unix as a volume.
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix && \
# Create Fake Xorg for xrdp to use on session start.
echo '#!/bin/bash\nsleep infinity' > \
/usr/lib/xorg/XorgFake && \
chmod +x /usr/lib/xorg/XorgFake && \
sed -i 's/param=Xorg/param=\\"/usr\\lib\\xorg\\XorgFake\\"' \
/etc/xrdp/sesman.ini && \
# Script to start Xorg, jwm and xrdp. The sed replaces
# X11DisplayOffset with $DISPLAY using a temp file in
# /tmp as /etc/xrdp isn't writable so can't use sed -i
echo '#!/bin/bash\nsed \"\'s/X11DisplayOffset=10/X11DisplayOf
fset=${DISPLAY}:1\'' /etc/xrdp/sesman.ini > /tmp/sesman.ini &
& cat /tmp/sesman.ini > /etc/xrdp/sesman.ini && rm -f /tmp/sesm
an.ini\nXorg $DISPLAY -ac -noreset -nolisten tcp -config /etc/X
11/xrdp/xorg.conf &\nxrdp-sesman --nodaemon &\nexec xrdp --noda
emon' > /usr/local/bin/startup && \

```

```
chmod +x /usr/local/bin/startup && \
# Remove packages used only for installation and build.
# Need to keep pkg-config or xrdp-chansrv behaves oddly.
rm -rf /usr/src/xrdp && \
rm -rf /usr/src/xorgxrdp && \
apt-get clean && \
apt-get purge -y software-properties-common bzip2 \
wget apt-utils git autoconf automake libtool xsltproc \
nasm flex bison gcc-6 libgcc-6-dev binutils xz-utils \
libpython3.5-minimal libx11-dev xserver-xorg-dev && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

VOLUME /tmp/.X11-unix

# The -g option is important for clean shut down.
# See https://github.com/krallin/tini#process-group-killing
ENTRYPOINT ["/sbin/init", "-g", "--"]
CMD ["/usr/local/bin/startup"]
```

The xrdp image may be built **on the remote host** with:

```
docker build -t xrdp .
```

As mentioned earlier, RDP logs-in a remote user by effectively creating a real desktop session on the server. An implication of this is that the normal mode of operation for xrdp is to start the X Server when the remote user logs in.

This way of working makes sense when the login equates to starting a remote desktop session, but if we wish to do *application* vice desktop remoting it causes us a problem as our applications expect to connect directly to a running X Server and will fail to start if an X Server is not available.

Our requirement to connect to a *running* X Server means that we cannot simply install an xrdp binary and use it out-of-the-box, as we have to perform some patching to “persuade” xrdp to behave in a way for which it wasn’t really designed. Although xrdp is available in the package repositories for most Linux distributions we must instead build from source to perform the patching, which makes the Dockerfile rather complicated but also perhaps quite interesting.

The first thing to note in the Dockerfile is the addition of the Debian “non-free” package repository. We add this because we wish to support the widest range of codecs to give the best possible RDP user experience. We can see the range of codecs in the packages that are installed over the next few lines of the Dockerfile, noting that we install many as `-dev` packages as we shall be building xrdp from source.

After installing the main packages we next install the [tini](#) init process to /sbin/init.

```
wget -O /sbin/init https://github.com/krallin/tini/releases/download/v0.18.0/tini  
chmod +x /sbin/init
```

We have encountered tini several times before as it is “injected” automatically by Docker when we use the `--init` option, however in this instance we wish to start tini using its `-g` option to kill the child process *group*, so that every process in the group gets the signal. This corresponds more closely to what happens when you do ctrl-C etc. in a terminal.

After installing tini we generate the locale then set up the timezone and configure jwm, all of which follow the same pattern that we used for our various VNC servers.

To install the xrdp and xorgxrdp source code we simply git clone from their GitHub repositories, noting that we checkout specific versions which is particularly important for this application given our patching.

The actual patch is quite simple, but a little subtle:

```
sed -i 's/!x_server_running_check_ports(display)/1/' \
/usr/src/xrdp/sesman/session.c
```

With xrdp the default behaviour is to defer starting Xorg until the session starts on RDP login, however we wish to start Xorg when the *server* starts. We can certainly start Xorg ourselves, but rather than use that X Server xrdp attempts to start its own. The `x_server_running_check_ports` function referred to above checks if there is already a server running on the given display, and if there is xrdp will skip it and launch its X Server on the next available display and connect to that.

In other words, if we were to start Xorg on `DISPLAY :1` then, when an RDP login occurs, xrdp will see `:1` is in use and so create its session Xorg on `DISPLAY :2` and connect to that. The patch above disables that check, so xrdp will create the session Xorg on `DISPLAY :1` (which we need to prevent) and then connect to the Xorg on `DISPLAY :1`, which we want.

We shall explain how we prevent xrdp from creating the unwanted session Xorg a little later.

After applying our patch to session.c we next compile and install both xrdp and xorgxrdp as described here:

<https://github.com/neutrino-labs/xrdp/wiki/Building-on-Debian-8>

Once xrdp and xorgxrdp have been built and installed the Dockerfile next makes some adjustments to various configuration files. We first remove TLSv1.3 from the ssl_protocols list in xrdp.ini to allow older clients to connect. Note that this reduces security, so remove this line if support for older clients is not required. We next change the permissions of a number of configuration files as we don't want our xrdp to have to run as root in the container.

It was mentioned earlier that xrdp will start Xorg when a session starts due to an RDP login. To work around this we create a fake Xorg as follows:

```
echo '#!/bin/bash\nsleep infinity' > \
      /usr/lib/xorg/XorgFake
chmod +x /usr/lib/xorg/XorgFake
```

and configure xrdp to use that with:

```
sed -i 's/param=Xorg/param=/usr/lib/xorg/XorgFake/' \
      /etc/xrdp/sesman.ini
```

With sesman.ini configured to use our XorgFake now, when an RDP login occurs, xrdp will start a new session as before, but the "X Server" it launches is simply a sleep. Our earlier patch of session.c stops xrdp skipping our running Xorg so the session will connect to that, which is what we want.

This Dockerfile creates a simple startup script in /usr/local/bin/startup to launch Xorg, jwm and xrdp:

```
#!/bin/bash
sed '\"s/X11DisplayOffset=10/X11DisplayOffset=${DISPLAY}:1}/\"'
' /etc/xrdp/sesman.ini > /tmp/sesman.ini && cat /tmp/sesman.ini
> /etc/xrdp/sesman.ini && rm -f /tmp/sesman.ini
Xorg $DISPLAY -ac -noreset -nolisten tcp -config /etc/X11/xrdp/
xorg.conf &
xrdp-sesman --nodaemon &
exec xrdp --nodaemon
```

The first line replaces the X11DisplayOffset with the value of the DISPLAY environment variable, so that the xrdp session will connect to the Xorg we'll be starting. That line is a little convoluted and has to use a temporary file because /etc/xrdp isn't writeable, so we can't simply use `sed -i`

The next line actually starts Xorg, note that we are using /etc/X11/xrdp/xorg.conf, which configures Xorg to use the xorgxrdp driver.

After starting Xorg we next start xrdp-sesman and finally we start xrdp itself, both as foreground processes.

The remainder of the Dockerfile removes the xrdp source code and intermediate build directories, then removes all of the packages that were only required to download and build the application. We retain pkg-config because one of its dependencies is required by xrdp-chansrv, which behaves oddly if we remove pkg-config.

[xrdp-storepasswd.sh](#) serves a similar purpose to the TightVNC `storepasswd.sh`, though the detail is rather different:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh

$DOCKER_COMMAND run --rm -it \
    -v $PWD:/mnt \
    xrdp sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'

tar zxf etc.tar.gz
rm -f etc.tar.gz
```

Because RDP uses a real user login this script runs the `adduser` command in a container and exports `/etc/passwd`, `/etc/shadow` and `/etc/group` to bind-mount later.

The [xrdp.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio-all.sh
. $BIN/docker-dbus-all.sh

# Create "home directory" on the host.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -d "etc"; then
    $(cd $(dirname $0); echo $PWD)/xrdp-storepasswd.sh
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--shm-size 2g \
--name=xrdp \
-p 3389:3389 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v $PWD/etc/passwd:/etc/passwd:ro \
-v $PWD/etc/shadow:/etc/shadow:ro \
-v $PWD/etc/group:/etc/group:ro \
-e DISPLAY=:1 \
xrdp

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:xrdp \
--volumes-from xrdp \
firefox

$DOCKER_COMMAND stop xrdp

```

This is similar to the TightVNC launch script, with the application container again sharing the server container's IPC and volumes. Here we don't use `--init`, as our ENTRYPPOINT already is /sbin/init.

We can launch this remotely with a command based on the following, with the appropriate user, host and path:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/rdp/xrdp; ./xrdp.sh"
```

6-6-2 xrdp-native-audio

The previous section illustrated xrdp with basic PulseAudio forwarding so that we could focus on the complexities of forwarding applications rather than desktops using RDP. In this section we shall extend what we have learned about xrdp to support native RDP audio forwarding, which can reduce required network bandwidth by compressing the audio, and in addition will forward audio to non-Linux RDP clients.

The [Dockerfile](#) is in the [xrdp-native-audio](#) directory of [docker-gui](#) under [6-remote-applications/rdp](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN \
    sed -i 's/main/main non-free/' /etc/apt/sources.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    wget apt-utils git autoconf automake libtool x11-utils \
    software-properties-common libssl-dev libpam0g-dev \
    nasm xsllibc proc flex bison pkg-config locales libcap-dev \
    libpulse-dev pulseaudio libudev-dev \
    intltool libltdl-dev libsndfile-dev bash-completion \
    libsystemd-dev libdbus-1-dev libspeexdsp-dev autopoint \
    libx11-dev xserver-xorg-dev xserver-xorg-core \
    libxfixes-dev libxrandr-dev libxml2-dev dpkg-dev \
    libmp3lame-dev libopus-dev libfdk-aac-dev libjpeg-dev \
    libturbojpeg0-dev libpixman-1-dev libfuse-dev \
    libgl1-mesa-glx libgl1-mesa-dri jwm xauth xfonts-base \
    xfonts-75dpi xfonts-100dpi xfonts-scalable && \
```

```
# Install tini as we need tini's -g option so every
# process in the group gets the signal
wget -O /sbin/init https://github.com/krallin/tini/releases/d
ownload/v0.18.0/tini && \
chmod +x /sbin/init && \
# Generate locales
sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/j
wm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\\usr\\share\\jwm\\jwm-red.svg\\"
>root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>//
g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\\etc\\jwm\\debian-menu</Include>//g" /et
c/jwm/system.jwmrc && \
# Clone xrdp and xorgxrdp source from GitHub and build.
cd /usr/src && \
git clone --recursive \
https://github.com/neutrinoLabs/xrdp.git && \
git clone \
https://github.com/neutrinoLabs/xorgxrdp.git && \
git clone \
https://github.com/pulseaudio/pulseaudio.git && \
git clone \
https://github.com/neutrinoLabs/pulseaudio-module-xrdp.gi
t && \
cd xrdp && git checkout v0.9.12 -b build && \
# The default xrdp behaviour is to defer starting Xorg
# until session start, so we patch session.c to prevent
```

```

# it skipping the display we will be pre-starting.
sed -i 's/!x_server_running_check_ports(display)/1/' \
    /usr/src/xrdp/sesman/session.c && \
./bootstrap && \
./configure --enable-opus --enable-fuse --enable-jpeg \
    --enable-tjpeg --enable-ipv6 --enable-vsock \
    --enable-pixman --enable-mp3lame \
    --enable-fdkaac && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
cd ../xorgxrdp && git checkout v0.2.12 -b build && \
./bootstrap && ./configure && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
sed -i 's/ssl_protocols=TLSv1.2, TLSv1.3/ssl_protocols=TLSv1. \
2/' \
    /etc/xrdp/xrdp.ini && \
# Build xrdp source / sink modules
cd ../pulseaudio && git checkout v10.0 -b build && \
./autogen.sh && \
cd ../pulseaudio-module-xrdp && \
git checkout v0.4 -b build && \
./bootstrap && \
./configure PULSE_DIR=/usr/src/pulseaudio && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
# Modify PulseAudio daemon config to support xrdp.
sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
    /etc/pulse/daemon.conf && \
sed -i "s/load-module module-console-kit/#load-module module- \
console-kit/g" \
    /etc/pulse/default.pa && \
echo "load-module module-xrdp-sink" >> \
    /etc/pulse/default.pa && \
echo "load-module module-xrdp-source" >> \
    /etc/pulse/default.pa && \
# Make the log files writeable so xrdp needn't be root.
touch /var/log/xrdp.log && \
chmod 1666 /var/log/xrdp.log && \
touch /var/log/xrdp-sesman.log && \
chmod 1666 /var/log/xrdp-sesman.log && \

```

```
chmod 644 /etc/xrdp/rsakeys.ini && \
chmod 644 /etc/xrdp/cert.pem && \
chmod 644 /etc/xrdp/key.pem && \
chmod 666 /etc/xrdp/sesman.ini && \
# We export /tmp/.X11-unix as a volume.
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix && \
mkdir /run/user && \
chmod 1777 /run/user && \
# Create Fake Xorg for xrdp to use on session start.
echo '#!/bin/bash\nsleep infinity' > \
/usr/lib/xorg/XorgFake && \
chmod +x /usr/lib/xorg/XorgFake && \
sed -i 's/param=Xorg/param=/usr/lib/xorg/XorgFake/' \
/etc/xrdp/sesman.ini && \
# Script to start Xorg, jwm and xrdp. The sed replaces
# X11DisplayOffset with $DISPLAY using a temp file.
echo '#!/bin/bash\nsed '\''\''s/X11DisplayOffset=10/X11Display0
ffset=${DISPLAY:1}\'\'' /etc/xrdp/sesman.ini > /tmp/sesman.ini
&& cat /tmp/sesman.ini > /etc/xrdp/sesman.ini && rm -f /tmp/ses
man.ini\npulseaudio &\nXorg $DISPLAY -ac -noreset -nolisten tcp
-config /etc/X11/xrdp/xorg.conf &\nxrdp-sesman --nodaemon &\nex
ec xrdp --nodaemon' > /usr/local/bin/startup && \
chmod +x /usr/local/bin/startup && \
# Remove packages used only for installation and build.
rm -rf /usr/src/xrdp && \
rm -rf /usr/src/xorgxrdp && \
rm -rf /usr/src/pulseaudio && \
rm -rf /usr/src/pulseaudio-module-xrdp && \
apt-get clean && \
apt-get purge -y software-properties-common bzip2 \
wget apt-utils git autoconf automake libtool xsltproc \
nasm flex bison gcc-6 libgcc-6-dev binutils xz-utils \
intltool libltdl-dev libsndfile-dev bash-completion \
libsystemd-dev libdbus-1-dev libspeexdsp-dev \
libpython3.5-minimal libx11-dev xserver-xorg-dev \
libpulse-dev libudev-dev autopoint && \
```

```
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

VOLUME /tmp/.X11-unix
VOLUME /run/user

# The -g option is important for clean shut down.
# See https://github.com/krallin/tini#process-group-killing
ENTRYPOINT ["/sbin/init", "-g", "--"]
CMD ["/usr/local/bin/startup"]
```

The image may be built **on the remote host** with:

```
docker build -t xrdp-native-audio .
```

This Dockerfile is very similar to the basic xrdp Dockerfile, but in addition it adds support for audio redirection by building the xrdp-source and xrdp-sink PulseAudio modules based on the instructions found in the xrdp Wiki: <https://github.com/neutrinoLabs/pulseaudio-module-xrdp/wiki/README>.

Once the PulseAudio modules have been built, the Dockerfile modifies the PulseAudio configuration to support xrdp as follows:

```
sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
/etc/pulse/daemon.conf && \
sed -i "s/load-module module-console-kit/#load-module module-co
nsole-kit/g" \
/etc/pulse/default.pa && \
echo "load-module module-xrdp-sink" >> \
/etc/pulse/default.pa && \
echo "load-module module-xrdp-source" >> \
/etc/pulse/default.pa && \
```

Configuring the PulseAudio daemon in this way was first encountered in the [Bluetooth and PulseAudio](#) section. The first line above stops PulseAudio exiting prematurely by setting the exit-idle-time to -1. The second line is rather more obtuse, but the gist is that the default configuration attempts to load the console-kit module if

`module-console-kit.so` is present (which it is). As we are running in a container, however, rather than on a real XDG desktop we don't *actually* have ConsoleKit running, so trying to use that module will cause the PulseAudio daemon startup to fail as module-console-kit will fail to initialise if ConsoleKit isn't running.

This Dockerfile also creates a simple startup script in `/usr/local/bin/startup` to launch PulseAudio, Xorg, jwm and xrdp similar to the one created in the basic xrdp Dockerfile:

```
#!/bin/bash
sed '\\"'s/X11DisplayOffset=10/X11DisplayOffset=${DISPLAY:1}\/\\'
' /etc/xrdp/sesman.ini > /tmp/sesman.ini && cat /tmp/sesman.ini
> /etc/xrdp/sesman.ini && rm -f /tmp/sesman.ini
pulseaudio &
Xorg $DISPLAY -ac -noreset -nolisten tcp -config /etc/X11/xrdp/
xorg.conf &
xrdp-sesman --nodaemon &
exec xrdp --nodaemon
```

Again this replaces the `X11DisplayOffset` with the value of the `DISPLAY` environment variable then starts pulseaudio, Xorg, xrdp-sesman and finally xrdp, using `/etc/X11/xrdp/xorg.conf` to configure Xorg to use the `xorgxrdp` driver.

The [xrdp-storepasswd.sh](#) script is identical to the earlier xrdp one apart from the image name:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh

$DOCKER_COMMAND run --rm -it \
-v $PWD:/mnt \
xrdp-native-audio sh -c 'adduser --uid "$(id -u)" --no-create-home "$(id -un)"; usermod -aG sudo "$(id -un)"; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'

tar zxf etc.tar.gz
rm -f etc.tar.gz
```

The [xrdp.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -d "etc"; then
    $(cd $(dirname $0); echo $PWD)/xrdp-storepasswd.sh
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--shm-size 2g \
--name=xrdp-native-audio \
-p 3389:3389 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v $PWD/etc/passwd:/etc/passwd:ro \
-v $PWD/etc/shadow:/etc/shadow:ro \
-v $PWD/etc/group:/etc/group:ro \
-e XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR \
-e DISPLAY=:1 \
xrdp-native-audio

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
-e DISPLAY=:1 \
--ipc=container:xrdp-native-audio \
--volumes-from xrdp-native-audio \
firefox

$DOCKER_COMMAND stop xrdp-native-audio

```

This is similar to the xrdp launch script, but here we set XDG_RUNTIME_DIR in the server. In the application we connect *directly* to the server's PulseAudio socket, as it is RDP **not** PulseAudio that is forwarding the audio.

We can launch this remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/rdp/xrdp-native-audio; ./xrdp.sh"
```

6-6-3 xfreerdp

xfreerdp is a stand-alone X11 RDP client which is part of the [FreeRDP](#) project.

The [Dockerfile](#) may be found in the [xfreerdp](#) directory of [docker-gui](#) under [6-remote-applications/rdp](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    freerdp-x11 libfreerdp-plugins-standard locales && \
    # Generate locales
    sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
    locale-gen && \
    # Set up the timezone
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["xfreerdp"]
```

The image may be built with:

```
docker build -t xfreerdp .
```

The Dockerfile is mostly straightforward aside from noting that in Debian the package name is freerdp-x11 rather than xfreerdp and that it is important to remember to include the libfreerdp-plugins-standard package to ensure that all of the available plugins are included in the image.

The [xfreerdp.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm -it \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/$(id -un):/home/$(id -un) \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    xfreerdp $@
```

Note that it is necessary to start xfreerdp with the required command line options, for example:

```
./xfreerdp.sh -sound -size:1280x720 -v:192.168.0.12 -rfx
```

Simply running `./xfreerdp.sh` with no options will display the available options then exit.

6-7 Xpra

X Persistent Remote Applications ([xpra](#)) is a remote display server and client for forwarding applications and desktop screens. Unlike [X11 Forwarding](#) xpra allows disconnection and reconnection without losing any state and unlike [VNC](#) xpra supports seamless windows, so forwarded applications appear on the local desktop as normal windows managed by the local window manager.

Xpra Server

The [Dockerfile](#) is in the [xpra](#) directory of [docker-gui](#) under [6-remote-applications/xpra](#).

```
FROM debian:stretch-slim

# Install xpra and xserver-xorg-video-dummy
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl xvfb cups-client libgl1-mesa-glx libgl1-mesa-dri \
    xserver-xorg-core x11-xserver-utils gir1.2-gtk-3.0 \
    gir1.2-notify-0.7 libturbojpeg0 liblzo2-dev \
    python-minimal python3 python3-rencode \
    python3-pyinotify python3-pil python3-lz4 \
    python3-dbus python3-cups python3-netifaces \
    python3-gi-cairo python3-brotli python3-gst-1.0 \
    python3-opengl python3-numpy ca-certificates \
    pulseaudio gstreamer1.0-pulseaudio xffonts-base \
    gstreamer1.0-plugins-good gstreamer1.0-plugins-bad \
    gstreamer1.0-plugins-ugly gstreamer1.0-libav \
    xffonts-75dpi xffonts-100dpi xffonts-scalable \
    python3-pip python3-setuptools python3-dev gcc && \
# Install packages not available in debian repo via
# pip. Although crypto and paramiko are in repo the
# versions are old and don't work correctly with xpra.
```

```

pip3 --no-cache-dir install wheel && \
pip3 --no-cache-dir install PyOpenGL-accelerate==3.1.0 && \
opencv-python python-lzo cryptography paramiko-ng \
python-uinput && \
# Install xpra binary and dependencies.
XPRA_VERSION=3.0.5-r24939-1 && \
XPRA=https://xpra.org/dists/stretch/main/binary-amd64 && \
echo "XPRA_VERSION version: ${XPRA_VERSION}" && \
curl -sSL ${XPRA}/ffmpeg-xpra_4.0-1_amd64.deb \
-o ffmpeg-xpra_4.0-1_amd64.deb && \
curl -sSL \
${XPRA}/xserver-xorg-video-dummy_0.3.8-5_amd64.deb \
-o xserver-xorg-video-dummy_0.3.8-5_amd64.deb && \
curl -sSL ${XPRA}/python3-xpra_${XPRA_VERSION}_amd64.deb \
-o python3-xpra_${XPRA_VERSION}_amd64.deb && \
curl -sSL ${XPRA}/xpra_${XPRA_VERSION}_amd64.deb \
-o xpra_${XPRA_VERSION}_amd64.deb && \
dpkg -i ffmpeg-xpra_4.0-1_amd64.deb \
xserver-xorg-video-dummy_0.3.8-5_amd64.deb && \
dpkg -i python3-xpra_${XPRA_VERSION}_amd64.deb \
xpra_${XPRA_VERSION}_amd64.deb && \
# Force $XDG_RUNTIME_DIR/pulse directory.
sed -i 's/self.pulseaudio_private_dir = osexpand(os.path.join(
(xpra_rd, "pulse-%s" % display))/self.pulseaudio_private_dir =
os.environ.get("XDG_RUNTIME_DIR", "")/g' /usr/lib/python3/dist-
packages/xpra/server/mixins/audio_server.py && \
# Create simple launch scripts to start xpra.
echo '#!/bin/bash\nmkdir -p $XDG_RUNTIME_DIR/pulse\nmkdir -p
~/.xpra\nexec xpra start --daemon=no --notifications=no $DISPLAY
--dbus-launch=$@\n' > /usr/local/bin/start && \
echo '#!/bin/bash\nxpra attach $@\n' > \
/usr/local/bin/attach && \
chmod +x /usr/local/bin/start && \
chmod +x /usr/local/bin/attach && \
# Modify xpra config.
sed -i "s/log-dir = auto/log-dir = ~/.xpra/g" \
/etc/xpra/conf.d/60_server.conf && \

```

```

sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
/etc/pulse/daemon.conf && \
sed -i "s/load-module module-console-kit/#load-module module-\
console-kit/g" /etc/pulse/default.pa && \
# We export /tmp/.X11-unix as a volume.
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix && \
mkdir /run/user && \
chmod 1777 /run/user && \
rm ffmpeg-xpra_4.0-1_amd64.deb \
xserver-xorg-video-dummy_0.3.8-5_amd64.deb \
python3-xpra_${XPRA_VERSION}_amd64.deb \
xpra_${XPRA_VERSION}_amd64.deb && \
apt-get clean && \
apt-get purge -y curl python3-pip python3-setuptools \
python3-dev gcc && \
apt-get autoremove -y && \
rm -rf /root/.cache/pip && \
rm -rf /var/lib/apt/lists/* && \
cp /etc/pulse/client.conf \
/etc/pulse/client-noshm.conf && \
sed -i "s/; enable-shm = yes/enable-shm = no/g" \
/etc/pulse/client-noshm.conf

VOLUME /tmp/.X11-unix
VOLUME /run/user

```

The image may be built **on local and remote hosts** with:

```
docker build -t xpra .
```

There are rather a lot of packages installed in the first part of this Dockerfile. In part this is down to xpra being quite feature-rich and so there are lots of Python dependencies, but xpra also supports a large number of audio and video codecs, so we include the full range of gstreamer1.0 plugins.

Not all of the required Python dependencies are available in the Debian repositories, and although crypto and paramiko actually *are* in Debian repository the versions there are old and don't work correctly with xpra, so next we use pip3 to install the remaining Python packages.

Note that the version of PyOpenGL-accelerate installed via pip is pinned to 3.1.0 to match the version of python3-opengl installed from the Debian repository. Note too that we can't simply install PyOpenGL via pip as the xpra .deb package has a dependency on the python3-opengl package.

Once all of the dependencies have been installed we next install the main xpra packages by first downloading their .deb packages from the <https://xpra.org/dists> repository then using dpkg to manually install them. We use this approach as the xpra dists repository has significantly more up-to-date versions of xpra than the main Debian repository.

Because the xpra server is normally run directly on a host machine rather than in a container xpra *really* wants to use a private PulseAudio session, however in a container we'd prefer it to use XDG_RUNTIME_DIR/pulse. Unfortunately there is no easy way to configure xpra to do this, so it is necessary to do some patching of audio_server.py:

```
sed -i 's/self.pulseaudio_private_dir = osexpand(os.path.join(x  
pra_rd, "pulse-%s" % display))/self.pulseaudio_private_dir = os  
.environ.get("XDG_RUNTIME_DIR", "")/g' /usr/lib/python3/dist-pa  
ckages/xpra/server/mixins/audio_server.py
```

As we would like to use the run-time value of XDG_RUNTIME_DIR that will be passed to the container as an environment variable at startup, we create a simple xpra server startup script in /usr/local/bin/start.

```
#!/bin/bash
mkdir -p $XDG_RUNTIME_DIR/pulse
mkdir -p ~/.xpra
exec xpra start --daemon=no --notifications=no $DISPLAY --dbus-launch= $@
```

This script simply creates the PulseAudio socket directory in XDG_RUNTIME_DIR/pulse and the xpra socket directory in ~/.xpra then calls `xpra start`. Because xpra made D-bus launch the default in v2.5 this wrapper script sets `--dbus-launch=` to disable it, as there may be xpra startup issues if D-bus isn't available.

We also modify the xpra server configuration so that the logs are visible in the bind-mounted `~/.xpra` directory.

```
sed -i "s/log-dir = auto/log-dir = ~\/.xpra/g" \
/etc/xpra/conf.d/60_server.conf
```

For symmetry with the server we next create a simple xpra client attach script in /usr/local/bin/attach.

```
#!/bin/bash
xpra attach $@
```

As with our [xrdp-native-audio](#) and [Bluetooth and PulseAudio](#) examples the Dockerfile modifies the default PulseAudio configuration to stop the daemon exiting prematurely.

```
sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
    /etc/pulse/daemon.conf
sed -i "s/load-module module-console-kit/#load-module module-co
nsole-kit/g" /etc/pulse/default.pa
```

The xpra server launch script [x11xpra.sh](#) is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

if ! test -f "$(id -un)/machine-id"; then
    echo $(dbus-uuidgen) > $(id -un)/machine-id
fi

$DOCKER_COMMAND run --rm -it -d \
    --init \
    --shm-size 2g \
    --name=x11xpra \
    -p 10000:10000 \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $PWD/$(id -un)/machine-id:/etc/machine-id \
    -e XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR \
    -e DISPLAY=:1 \
    xpra start --bind=$HOME/.xpra/xpra-socket --bind-tcp=0.0.0.
0:10000
```

```

# Test for the presence of $XDG_RUNTIME_DIR/pulse/pid to
# wait until xpra has launched Pulseaudio as if we launch
# application containers before the Pulseaudio socket is
# available then their audio will fail.
$DOCKER_COMMAND run --rm -it \
    --volumes-from x11xpra \
    xpra bash -c "while [ ! -f $XDG_RUNTIME_DIR/pulse/pid ]; do
echo 'Waiting for xpra Pulseaudio daemon';sleep 1; done; echo 'xpra Pulseaudio daemon running'"

$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:x11xpra \
    --volumes-from x11xpra \
    firefox

$DOCKER_COMMAND stop x11xpra

```

This launch script follows a similar pattern to many of our other remote server examples, but there are a number of complications with xpra that are rather subtle and took the author of this book quite some time to figure out when developing this section.

Our first addition is the creation of a UUID that we shall use as a fake machine-id in our xpra containers.

```

if ! test -f "$(id -un)/machine-id"; then
    echo $(dbus-uuidgen) > $(id -un)/machine-id
fi

```

Our fake machine-id is bind-mounted to the container by the following line in the `docker run` command:

```
-v $PWD/$(id -un)/machine-id:/etc/machine-id
```

The /etc/machine-id file contains the unique machine ID of the local system normally set during installation or boot:

<https://www.freedesktop.org/software/systemd/man/machine-id.html>.

The issue we have is that containers derived from the same base image will have the same default machine-id (which is normally set by systemd on first boot). This ID is used by xpra to detect if the client and server environment are identical, and if so xpra refuses to allow speaker forwarding. An alternative is to set `XPRA_ALLOW_SOUND_LOOP=1` on the environment, but setting machine-id can be useful in other situations too, so it's a useful approach to illustrate.

The next issue that caused problems during the development of the xpra containers was the delay between the container starting and xpra launching the PulseAudio daemon, which resulted in intermittent audio failures. To resolve this we add a polling loop to test for `$XDG_RUNTIME_DIR/pulse/pid`, so we can defer launching the application container until the Pulseaudio socket is actually available.

```
$DOCKER_COMMAND run --rm -it \
  --volumes-from x11xpra \
  xpra bash -c "while [ ! -f $XDG_RUNTIME_DIR/pulse/pid ]; do
    echo 'Waiting for xpra Pulseaudio daemon'; sleep 1; done;
    echo 'xpra Pulseaudio daemon running'"
```

Our wait for the PulseAudio daemon works by launching a temporary container that mounts the volumes from our x11xpra container to enable it to see the server's /run/user directory. The temporary container simply launches a shell that executes a script which checks for the existence of the PulseAudio PID file

`$XDG_RUNTIME_DIR/pulse/pid` in a loop and returns when the PID file has been created.

The final part of the xpra server launch script launches our Firefox application and, as with our [xrdp-native-audio](#) example, we connect directly to the server's PulseAudio socket, as it is xpra not PulseAudio that forwards the audio.

We can launch this remotely with a command based on the following, with the appropriate user, host and path:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/xpra/xpra; ./x11xpra.sh"
```

Xpra Client

Whilst most of the applications we have covered “just work” with the client host’s PulseAudio server, audio for the the xpra client sometimes failed inexplicably for the author.

The issue turned out to be due to having two GPUs, each with HDMI output. Whilst PulseAudio is *generally* good at device enumeration PulseAudio and HDMI often don’t play nicely, especially where multiple cards and devices are involved. It is possible for the default/selected sink to *not* be the first enumerated HDMI sink, if this happens audio won’t be played by the client because xpra uses the first enumerated sink.

The [enumerate-hdmi.sh](#) script attempts to work around this issue and should be run on the xpra client side host if there are multiple HDMI cards present.

```
DEFAULT_SINK=$(pactl info | sed -n -e 's/Default Sink: //p')
if [[ $DEFAULT_SINK == *"hdmi"* ]]; then
    arr=()
    IFS=$'\n'; for i in $(pactl list short sinks | grep hdmi); do
        if [[ $i == *"$DEFAULT_SINK"* ]]; then
            break
        else
            arr+=($i)
        fi
    done

    for i in ${arr[@]}; do
        ALSA_CARD=$(echo $i | cut -d. -f2,3)
        for j in $(pactl list short modules | grep module-alsa-card
); do
            if [[ $j == *"$ALSA_CARD"* ]]; then
                MODULE_INDEX=$(echo $j | cut -f1)
                MODULE_PROPERTIES="pactl load-module $(echo $j | cut -f
2- | sed -n -e 's/\t/ /p')"
                pactl unload-module $MODULE_INDEX
                eval $MODULE_PROPERTIES
            fi
        done
    done
else
    echo "$DEFAULT_SINK is not HDMI, exiting!"
fi
```

This script attempts to work around the problem by unloading the module(s) that relate to the non-default HDMI sink so that the one we want becomes the first to be enumerated.

The first part of the script Iterates through all of the available HDMI sinks storing them in an array, until we reach the default. This array represents all of the HDMI sinks enumerated *before* our default/selected one.

The second part loops through the array, finds the associated module, and unloads then reloads the module using the same properties. This should ensure that the default/selected HDMI sink gets enumerated first.

The xpra client launch script [xpra-attach.sh](#) is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm -it \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/$(id -un):/home/$(id -un) \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    $X11_FLAGS \
    $GPU_FLAGS \
    xpra attach $@
```

If the xpra client and server are co-located on the same host the most efficient way to launch the client is to use the xpra Unix domain socket and enable memory-mapped I/O.

```
./xpra-attach.sh --desktop-scaling=1 --mmap=$HOME/.xpra/xpra-mm  
ap socket:$HOME/.xpra/xpra-socket
```

If the client and server are on different hosts then we must instead connect using the xpra TCP socket.

```
./xpra-attach.sh --desktop-scaling=1 --mmap=off tcp://<host-ip>  
:10000
```

Whilst it is often best to simply use the default codecs chosen by the client and server, it is also possible to choose specific codecs as follows.

```
./xpra-attach.sh --desktop-scaling=1 --speaker-codec=flac --mma  
p=$HOME/.xpra/xpra-mmap socket:$HOME/.xpra/xpra-socket
```

As an alternative to using the enumerate-hdmi.sh script described earlier, it is possible to use the XPRA_PULSE_SINK_DEVICE_NAME environment variable by adding something like the following to the launch script:

```
-e XPRA_PULSE_SINK_DEVICE_NAME=alsa_output.pci-0000_01_00.1.hdm  
i-stereo-extra1
```

To discover the full range of options:

```
./xpra-attach.sh --help
```

6-8 NX

NX Technology (NX) is a compression and transport protocol that was initially designed as a way to improve some of the shortcomings of [X11 Forwarding](#) by compressing the X11 transport, caching data to minimise latency, and minimising many of the round-trips inherent in the X11 protocol.

The origins of NX Technology date back to the earlier Differential X Protocol Compression ([DXPC](#)) system. [NX X Protocol Compression](#) has however evolved significantly to include features such as image compression and streaming and improved caching. The protocol will also perform ZLIB compression of the data parts of X11 messages which don't have a specific differential encoding.

The NX architecture comprises two key components, the nxproxy and nxagent. The nxproxy runs on both the remote and local machines, forwarding the compressed and cached remote X11 protocol requests to the local X Server. The nxagent runs on the remote machine and acts as a local (from the perspective of the remote client) X Server, so is therefore able to avoid the majority of the X11 protocol round trips. Unlike other remoting systems NX relies on SSH as a transport and NX clients use SSH tunnelling by default.

From version 3.5 NX's core compression technology has been available under the GPLv2 licence as [nx-libs](#), however from version 4.0 NX added a number of significant enhancements including video and audio encoding and became closed source.

6-8-1 x2goserver

X2Go uses a modified version of the NX v3 protocol that, in common with other NX versions, is tunnelled over SSH.

The Dockerfile is in the x2goserver directory of docker-gui under 6-remote-applications/nx.

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ca-certificates libgl1-mesa-glx libgl1-mesa-dri jwm \
    gnupg dirmngr locales xserver-xorg-video-dummy && \
    echo "deb http://packages.x2go.org/debian stretch extras main
\n" > /etc/apt/sources.list.d/x2go.list && \
    apt-key adv --recv-keys --keyserver hkp://keyserver.ubuntu.co
m:80 E1F958385BFE2B6E && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    x2goserver && \
    echo '#!/bin/bash\nif ! test -d "${HOME}/.ssh"; then\nmkdir $
{HOME}/.ssh\nssh-keygen -f ${HOME}/.ssh/ssh_host_rsa_key -N ""
-t rsa -b 4096\nssh-keygen -f ${HOME}/.ssh/ssh_host_ecdsa_key -
N "" -t ecdsa -b 521\nssh-keygen -f ${HOME}/.ssh/ssh_host_ed255
19_key -N "" -t ed25519\nfi\nXorg $DISPLAY -ac -cc 4 &\nsleep 0
.5\njwm &\necho "$(id -un)@$DISPLAY" > ${HOME}/.session_id\n\
n/usr/sbin/sshd -p 2222 -h ${HOME}/.ssh/ssh_host_rsa_key -h ${H
OME}/.ssh/ssh_host_ecdsa_key -h ${HOME}/.ssh/ssh_host_ed25519_k
ey -D' > /usr/local/bin/start-server && \
    chmod +x /usr/local/bin/start-server && \
```

```

# Overwrite x2golistdesktops with simple script to
# return running desktop.
echo '#!/bin/bash\nncat ${HOME}/.session_id' > \
    /usr/bin/x2golistdesktops && \
# Generate locales
sed -i "s/^# *($LANG)/\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\usr\share\jwm\jwm-red.svg\"
>root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">>showdesktop</TrayButton>//
g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\etc\jwm\debian-menu</Include>//g" /et
c/jwm/system.jwmrc && \
# We export /tmp/.X11-unix as a volume and we need
# the mode of /tmp/.X11-unix to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix && \
# sshd privilege separation directory
mkdir /run/sshd && \
rm -rf /var/lib/apt/lists/*

VOLUME /tmp/.X11-unix
COPY xorg.conf /etc/X11/xorg.conf

```

The x2goserver image may be built **on the remote host** with:

```
docker build -t x2goserver .
```

Because the x2goserver package is not available in the main Debian repositories, after installing the main dependencies the Dockerfile adds the x2go repository following the instructions in the [x2go documentation](#). We use [hkp://keyserver.ubuntu.com](https://keyserver.ubuntu.com):80 as the keyserver, as the author has found that to be more reliable than keys.gnupg.net, especially running behind a corporate proxy.

After adding the x2go repository it is simply a case of using apt-get install to install x2goserver like any other package.

Much of the remainder of the Dockerfile is similar to our [VNC](#) examples, where we generate the locale then set up the timezone and configure jwm, however the startup script in /usr/local/bin/start-server requires a little explanation.

```
#!/bin/bash
if ! test -d "${HOME}/.ssh"; then
    mkdir ${HOME}/.ssh
    ssh-keygen -f ${HOME}/.ssh/ssh_host_rsa_key -N "" \
        -t rsa -b 4096
    ssh-keygen -f ${HOME}/.ssh/ssh_host_ecdsa_key -N "" \
        -t ecdsa -b 521
    ssh-keygen -f ${HOME}/.ssh/ssh_host_ed25519_key -N "" \
        -t ed25519
fi

Xorg $DISPLAY -ac -cc 4 &
sleep 0.5
jwm &
echo "$(id -un)@$DISPLAY" > ${HOME}/.session_id

/usr/sbin/sshd -p 2222 -D \
    -h ${HOME}/.ssh/ssh_host_rsa_key \
    -h ${HOME}/.ssh/ssh_host_ecdsa_key \
    -h ${HOME}/.ssh/ssh_host_ed25519_key
```

As mentioned previously, NX is tunneled over SSH and the majority of the start-server script is actually for setting up sshd. Recall that we have used SSH previously for [X11 Forwarding](#) and for tunnelling [VNC](#), however in those cases we were using the **server host's** sshd as we were already using that to launch the remote server. With X2Go however the X2Go client is using ssh to directly launch several commands on the X2Go server and the whole process is a little more coupled, so it makes sense to launch sshd in the container.

The start-server script first checks for the presence of the `~/.ssh` directory, and if that doesn't exist the script creates it then generates rsa, ecdsa, and ed25519 keys.

The script next launches Xorg and jwm and adds

`<username>@<DISPLAY>` to `~/.session_id`.

Finally, the script starts sshd on port 2222, using -D to ensure that it is **not** daemonised and using -h to select the host keys that were generated earlier.

In addition to the `/usr/local/bin/start-server` script the Dockerfile also replaces the X2Go server script `/usr/bin/x2golistdesktops` with a trivial script to cat the `~/.session_id` file generated earlier by start-server.

```
#!/bin/bash
cat ${HOME}/.session_id'
```

The `x2golistdesktops` script is launched on the server by the X2Go client via SSH.

The [x2goserver-storepasswd.sh](#) script is identical to the earlier xrdp one apart from the image name:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
$DOCKER_COMMAND run --rm -it \  
-v $PWD:/mnt \  
x2goserver sh -c 'adduser --uid '$(id -u)' --no-create-home  
'$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar  
.gz -C / ./etc/passwd ./etc/shadow ./etc/group'  
  
tar zxf etc.tar.gz  
rm -f etc.tar.gz
```

The [x2goserver.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all  
. $BIN/docker-command.sh  
. $BIN/docker-pulseaudio-all.sh  
. $BIN/docker-dbus-all.sh  
  
# Create a directory on the host that we can mount as a  
# "home directory" in the container for the current user.  
mkdir -p $(id -un)/.config/pulse  
mkdir -p $(id -un)/.config/dconf  
  
# Create password if required.  
if ! test -d "etc"; then  
    echo "creating password"  
    $(cd $(dirname $0); echo $PWD)/x2goserver-storepasswd.sh  
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=x2goserver \
-p 2222:2222 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v $PWD/etc/passwd:/etc/passwd:ro \
-v $PWD/etc/shadow:/etc/shadow:ro \
-v $PWD/etc/group:/etc/group:ro \
-e DISPLAY=:1 \
-e GEOMETRY=1280x720 \
x2goserver start-server

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
-e DISPLAY=:1 \
--ipc=container:x2goserver \
--volumes-from x2goserver \
firefox

$DOCKER_COMMAND stop x2goserver

```

This launch script follows a similar pattern to many of our other remote server examples.

We can launch this remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/nx/x2goserver; ./x2goserver.sh"
```

6-8-2 x2goclient

The [Dockerfile](#) is in the [x2goclient](#) directory of [docker-gui](#) under [6-remote-applications/nx](#).

```
FROM debian:stretch-slim

# Note that x2goclient needs xauth and fails to connect
# in a fairly hard to diagnose way without it.
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    ca-certificates gnupg dirmngr xauth \
    openssh-sftp-server openssh-server \
    libgl1-mesa-glx libgl1-mesa-dri && \
    # Set up the repositories for x2go
    echo "deb http://packages.x2go.org/debian stretch extras main
\n" > /etc/apt/sources.list.d/x2go.list && \
    apt-key adv --recv-keys --keyserver hkp://keyserver.ubuntu.co
m:80 E1F958385BFE2B6E && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    x2goclient && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["x2goclient"]
```

The x2goclient image may be built with:

```
docker build -t x2goclient .
```

Unlike [x2goserver](#), x2goclient is actually available in the main Debian repository, however here we use the x2go repository as we did with the server because that has a newer version of the client. Note too the importance of installing xauth, or the client will fail in a rather difficult to diagnose way.

Perhaps surprisingly this client Dockerfile also installs the openssh-sftp-server and openssh-server packages. This is because in some modes of operation the client will launch its own SSH/SFTP server for the purpose of local filesystem export.

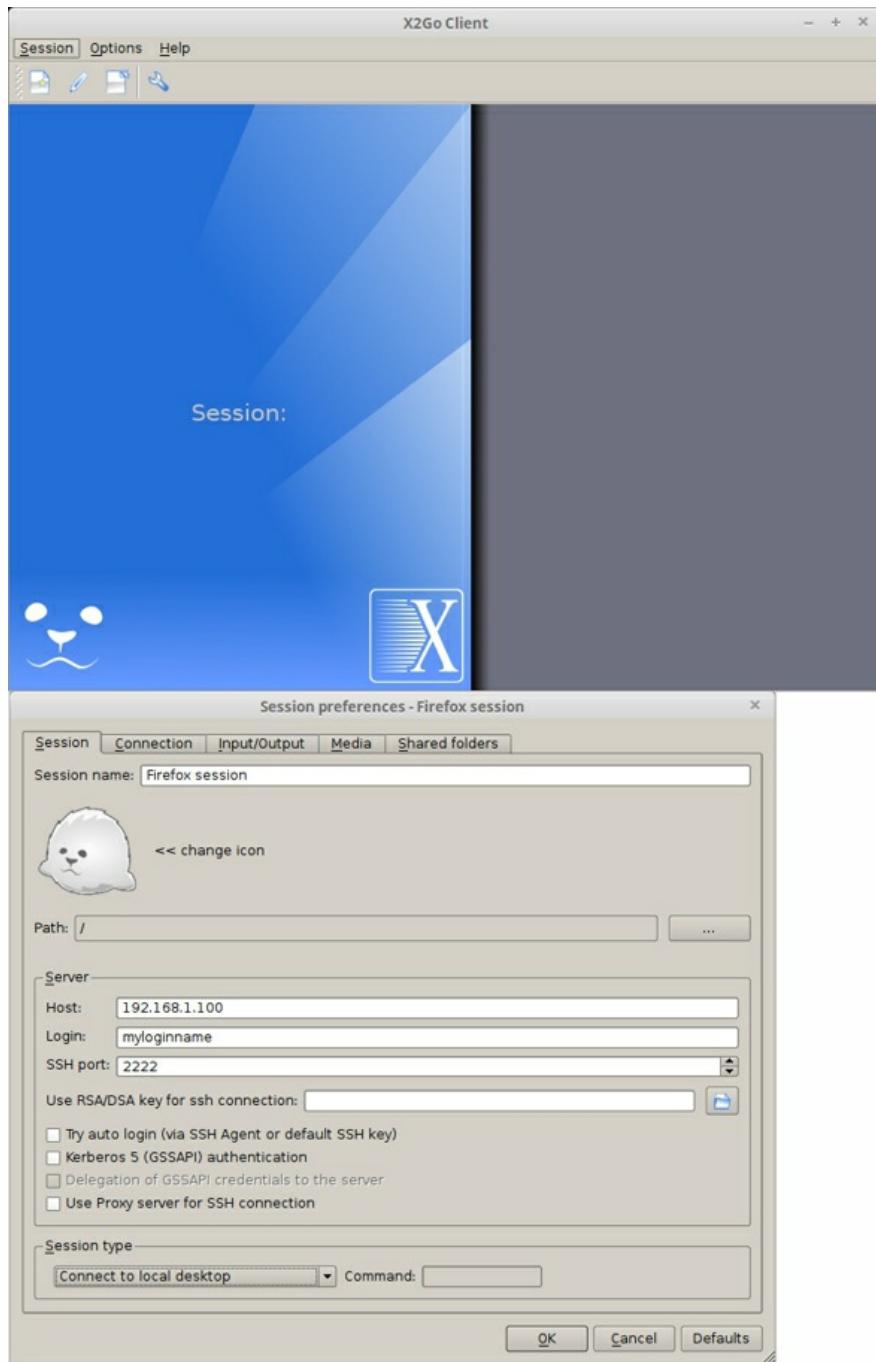
The [x2goclient.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/dconf
$DOCKER_COMMAND run --rm \
    --init \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/$(id -un):/home/$(id -un) \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $X11_FLAGS \
    x2goclient
```

This script is mostly just following the pattern that we have seen many times before when launching applications that connect to the local X Server and integrate with D-bus, however in this case note that we have also included the `--init` flag. This is because, like most of our remote server applications, the X2Go client launches several different processes and it is important to ensure that our client has the correct PID 1 signal handling behaviour.

The X2Go client launches two windows at startup:



To use the X2Go client the *Session Preferences* window should be populated as follows:

- **Session name:** An arbitrary session name.
- **Host:** The server's host name or IP address.
- **Login:** The user name to use when logging in on to the remote server.
- **SSH port:** The port to use to connect to the remote server. Port 2222 should be used here unless the x2goserver.sh launch script has been modified to expose a different port.
- **RSA/DSA key:** Specify the key, if created. The key decryption will prompt for a password.
- **Session type:** Selecting *Connect to local desktop* here will connect to the jwm session that should already be running on the X2Go server.

After successful configuration of the session a new session window will appear on the top right of the main window and clicking on this will open the login window.

X2Go offers a wide range of session types that depend on how the server has been configured. For our example *Connect to local desktop* is the most appropriate session type, however if the x2goserver image is extended with additional applications then it is possible to use X2Go's *Single application* or *Published applications* session type to forward the application in a seamless window. These seamless window modes appear, however, to rely on X2Go starting the application itself and it is not clear if it is possible to forward already running applications like our Firefox.

6-8-3 nomachine

As of version 4.0 NX became the proprietary and closed source product [NoMachine](#). This section illustrates how to install the free version of NoMachine in a Docker container.

Important note: unlike the majority of applications illustrated in this book NoMachine is a commercial product. NoMachine is free to download and install for personal or individual use, including access to office computers, provided it is not to carry out activities that would be considered “commercial usage” as per the NoMachine [FAQ](#).

NoMachine Server

The [Dockerfile](#) is in the [nomachine](#) directory of [docker-gui](#) under [6-remote-applications/nx](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

# Install NoMachine and xserver-xorg-video-dummy.
# Note net-tools as NoMachine uses netstat to detect
# if there is a currently running X Server.
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl sudo ca-certificates net-tools xauth libvdpau1 \
    mesa-vdpau-drivers libgl1-mesa-glx libgl1-mesa-dri \
    desktop-base xdg-user-dirs xserver-xorg-video-dummy \
    xserver-xorg-input-libinput locales jwm cups \
    xfonts-base xfonts-75dpi xfonts-100dpi \
    xfonts-scalable pulseaudio && \
```

```

# Ensure the nx user ID is not in the normal user range.
groupadd -r -g 2000 nx && \
useradd -u 2000 -r -g nx nx && \
# Try to work out the latest version from the NoMachine
# Linux download page. If this fails set the following
# variables manually instead.
NOMACHINE_VERSION=$(curl -sSL \
  "https://www.nomachine.com/download/download&id=3" | \
  grep "Linux/nomachine" | \
  cut -d \' -f2 | cut -d \_ -f2-3) && \
NOMACHINE_MAJOR_VERSION=$(echo $NOMACHINE_VERSION | \
  cut -d \. -f1-2) && \
echo "VERSION: ${NOMACHINE_VERSION}" && \
echo "MAJOR_VERSION: ${NOMACHINE_MAJOR_VERSION}" && \
curl -sSL https://download.nomachine.com/download/${NOMACHINE
_MAJOR_VERSION}/Linux/nomachine_${NOMACHINE_VERSION}_amd64.deb
-o nomachine_${NOMACHINE_VERSION}_amd64.deb && \
dpkg -i nomachine_${NOMACHINE_VERSION}_amd64.deb && \
# Create simple script to start NoMachine server.
# Setting XDG_RUNTIME_DIR in /etc/environment
# ensures the correct PulseAudio socket path
# will be used even when PulseAudio is started
# via sudo /etc/NX/nxserver --startup
echo '#!/bin/bash\necho \"'"export XDG_RUNTIME_DIR=$XDG_RUNTI
ME_DIR'\"' | sudo tee --append /etc/environment > /dev/null\nUU
ID=$(cat /proc/sys/kernel/random/uuid)\nsudo sed -i '\"'s/$(cat
/usr/NX/etc/uuid)/$UUID/g'\"' /usr/NX/etc/nodes.db\necho $UUID
| sudo tee /usr/NX/etc/uuid > /dev/null\n\nGEOMETRY=${GEOMETRY:
-1280x720}\nsudo sed -i '\"'s/Modes \\\'\\''1280x720\\\'\\'/Modes
\\\'\\'$GEOMETRY\\\'\\'/g'\"' /etc/X11/xorg.conf\nXorg $DISPLAY -
ac -cc 4 &\nsleep 0.5\njwm &\nwhile echo $(sudo /etc/NX/nxserve
r --startup) | grep WARNING; do sleep 1; done\nsudo tail -f /us
r/NX/var/log/nxserver.log' > /usr/local/bin/start-server && \
chmod +x /usr/local/bin/start-server && \

```

```
# Modify PulseAudio daemon config.
sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
/etc/pulse/daemon.conf && \
sed -i "s/load-module module-console-kit/#load-module module-
console-kit/g" /etc/pulse/default.pa && \
# Allow users in sudo group to run commands
# without password.
sed -i "s/%sudo\!tALL=(ALL\:ALL) ALL/%sudo\!tALL=(ALL\:ALL) NOP
ASSWD\:ALL/g" /etc/sudoers && \
# Generate locales
sed -i "s/^# *`($LANG)`/\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/j
wm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\`/usr\`/share\`/jwm\`/jwm-red.svg\`>root:1<\`/TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\`>showdesktop<\`/TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\`/etc\`/jwm\`/debian-menu<\`/Include>//g" /et
c/jwm/system.jwmrc && \
# We'll be exporting /run/user as a volume
# and we need the mode to be set to 1777
mkdir /run/user && \
chmod 1777 /run/user && \
# Tidy up
rm nomachine_${NOMACHINE_VERSION}_amd64.deb && \
apt-get clean && \
apt-get purge -y curl && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/* && \
```

```
cp /etc/pulse/client.conf \
    /etc/pulse/client-noshm.conf && \
sed -i "s/; enable-shm = yes/enable-shm = no/g" \
    /etc/pulse/client-noshm.conf

VOLUME /tmp/.X11-unix
VOLUME /run/user

COPY xorg.conf /etc/X11/xorg.conf
```

The image may be built **on local and remote hosts** with:

```
docker build -t nomachine .
```

As usual, the Dockerfile begins by installing the package dependencies. In addition to the more obvious ones, we have speculatively included the libvdpau1 and mesa-vdpau-drivers packages on the assumption that the NoMachine client supports VDPAU accelerated video decoding.

One slightly surprising package included in the dependencies is net-tools. This package is required as the NoMachine server uses the [netstat](#) command from that package to detect whether there is a currently running X Server.

After installing the dependencies the Dockerfile next adds the nx user and group as the NoMachine server expects to first be started as root then switch to the nx user.

```
groupadd -r -g 2000 nx
useradd -u 2000 -r -g nx nx
```

Next, the Dockerfile attempts to “scrape” the most recent NoMachine version from the download page via:

```
NOMACHINE_VERSION=$(curl -SSL \  
"https://www.nomachine.com/download/download&id=3" |\  
grep "Linux/nomachine" |\  
cut -d \' -f2 | cut -d \_ -f2-3)
```

and

```
NOMACHINE_MAJOR_VERSION=$(echo $NOMACHINE_VERSION |\  
cut -d \. -f1-2)
```

This works, however it is fragile and any changes to the format of the NoMachine download page could cause it to fail. If that happens then simply replacing those lines with a specific NoMachine version should fix it, e.g.

```
NOMACHINE_VERSION=6.9.2_1  
NOMACHINE_MAJOR_VERSION=6.9
```

Note however that NoMachine has a [policy](#) of not offering old versions for download. If it is important to be able to restore a specific version, readers should manually download the package and make sure that a backup copy of the original package is made, then modify the Dockerfile to COPY the saved .deb instead of automatically downloading it.

After downloading and installing the nomachine .deb package the Dockerfile next creates a small script in /usr/local/bin/start-server to configure and start nxserver.

```

#!/bin/bash
echo "export XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR" | sudo tee --append
> /etc/environment > /dev/null

UUID=$(cat /proc/sys/kernel/random/uuid)
sudo sed -i "s/$(cat /usr/NX/etc/uuid)/$UUID/g" /usr/NX/etc/nodes.db
echo $UUID | sudo tee /usr/NX/etc/uuid > /dev/null

GEOMETRY=${GEOMETRY:-1280x720}
sudo sed -i "s/Modes \"\"\"1280x720\"\"\"/Modes \"$GEOMETRY\"/g" /etc/X11/xorg.conf

Xorg $DISPLAY -ac -cc 4 &
sleep 0.5
jwm &

while echo $(sudo /etc/NX/nxserver --startup) | grep WARNING; do
  sleep 1; done
sudo tail -f /usr/NX/var/log/nxserver.log

```

The first line sets `XDG_RUNTIME_DIR` in `/etc/environment` with the value passed to the container as an environment variable. This is slightly odd, but as NoMachine is started via

`sudo /etc/NX/nxserver --startup` it won't see the environment variable directly and therefore has to rely on `/etc/environment`. This environment variable is used so that the PulseAudio daemon started by NoMachine creates its Unix domain socket in the expected well-known location.

The next few lines generate a UUID and set its value in `/usr/NX/etc/nodes.db` and `/usr/NX/etc/uuid`. The reason for this is because NoMachine uses this UUID like a `machine_id` and normally initialises the value of `/usr/NX/etc/uuid` at installation time.

The implication of this is that if we were to launch multiple containers from the same image they would all be viewed by NoMachine as the same machine. One particular problem is that if a client and server were launched using the same image this issue would prevent audio forwarding between them, so here we generate a new UUID on container startup.

If generating a new NoMachine server ID each startup is undesirable, an alternative approach is to generate the UUID on the host. Place it in `~/.config/uuid`, then bind-mount that to `/usr/NX/etc/uuid`. For this to work however, the Dockerfile would need to be modified to remove the UUID lines above from the `/usr/local/bin/start-server` script.

The next line in the start-server script uses the value passed in the `GEOMETRY` environment variable to modify the `Modes` line of `/etc/X11/xorg.conf`. An alternative approach would be to remove this line from the script and bind-mount, rather than `COPY`, `xorg.conf` from the host.

After setting the display's `GEOMETRY` value the next few lines of the start-server script start Xorg and then jwm.

When the X Server and window manager have started, the start-server script then attempts to start the NoMachine server. Starting `nxserver` is done in a loop as it doesn't always start immediately, so we check for any `WARNING` message and retry. Once the server has started, we prevent the container from exiting immediately by running

```
tail -f on /usr/NX/var/log/nxserver.log.
```

After creating the /usr/local/bin/start-server script the remainder of the Dockerfile is relatively straightforward.

We modify the PulseAudio daemon configuration as we did previously in the xrdp and xpra Dockerfiles. This prevents the PulseAudio daemon from exiting prematurely on idle and removes the ConsoleKit dependency.

We next edit /etc/sudoers to allow users in the sudo group to run commands without a password. We do this because we would like to launch the container as a normal desktop user, but nxserver expects to be launched as root then change itself to the nx user, so we *actually* start the server using `sudo /etc/NX/nxserver --startup`.

Finally, we generate the locales and timezone, configure jwm and perform some tidy up.

The [nomachine-storepasswd.sh](#) script is identical to the earlier x2goserver one apart from the image name:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
$DOCKER_COMMAND run --rm -it \  
-v $PWD:/mnt \  
nomachine sh -c 'adduser --uid '$(id -u)' --no-create-home  
'$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'  
  
tar zxf etc.tar.gz  
rm -f etc.tar.gz
```

The [nomachine.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -d "etc"; then
    echo "creating password"
    $(cd $(dirname $0); echo $PWD)/nomachine-storepasswd.sh
fi

# Set XDG_RUNTIME_DIR on env for PulseAudio daemon.
# nomachine server requires --cap-add=SYS_PTRACE
$DOCKER_COMMAND run --rm -it -d \
    --init \
    --cap-add=SYS_PTRACE \
    --shm-size 2g \
    --name=nomachine \
    -p 4000:4000 \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $PWD/etc/passwd:/etc/passwd:ro \
    -v $PWD/etc/shadow:/etc/shadow:ro \
    -v $PWD/etc/group:/etc/group:ro \
    -e XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR \
    -e DISPLAY=:1 \
    -e GEOMETRY=1280x720 \
    nomachine start-server
```

```

# Use --volumes-from to mount /tmp/.X11-unix
# from nomachine and also use that container's IPC
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:nomachine \
    --volumes-from nomachine \
    firefox

$DOCKER_COMMAND stop nomachine

```

This is very similar to most of our other server launch scripts, however here we add `--cap-add=SYS_PTRACE` because the netstat call that is invoked by nxserver at startup uses the ptrace system call.

We can launch nomachine.sh remotely with:

```

ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-a
pplications/nx/nomachine; ./nomachine.sh"

```

NoMachine Client

As with the xpra client the NoMachine client uses the same image as the server. Also like the xpra client the NoMachine client has issues if multiple HDMI audio cards are present, because NoMachine uses the first *enumerated* HDMI sink rather than the one marked as default. To mitigate this we again use a script to enumerate the available HDMI devices.

The [enumerate-hdmi.sh](#) script for NoMachine is identical to the one we used for xpra.

```
DEFAULT_SINK=$(pactl info | sed -n -e 's/Default Sink: //p')
if [[ $DEFAULT_SINK == *"hdmi"* ]]; then
    arr=()
    IFS=$'\n'; for i in $(pactl list short sinks | grep hdmi); do
        if [[ $i == *"$DEFAULT_SINK"* ]]; then
            break
        else
            arr+=($i)
        fi
    done

    for i in ${arr[@]}; do
        ALSA_CARD=$(echo $i | cut -d. -f2,3)
        for j in $(pactl list short modules | grep module-alsa-card
); do
            if [[ $j == *"$ALSA_CARD"* ]]; then
                MODULE_INDEX=$(echo $j | cut -f1)
                MODULE_PROPERTIES="pactl load-module $(echo $j | cut -f
2- | sed -n -e 's/\t/ /p')"
                pactl unload-module $MODULE_INDEX
                eval $MODULE_PROPERTIES
            fi
        done
    done
else
    echo "$DEFAULT_SINK is not HDMI, exiting!"
fi
```

This script attempts to work around the problem by unloading the module(s) that relate to the non-default HDMI sink so that the one we want becomes the first to be enumerated.

The [nomachine-client.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-pulseaudio.sh

mkdir -p $(id -un)-client/.config/pulse
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un)-client:/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-v /etc/group:/etc/group:ro \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
$GPU_FLAGS \
nomachine /usr/NX/bin/nxplayer
```



6-9 SPICE

The Simple Protocol for Independent Computing Environments ([SPICE](#)) aims to provide a complete open remote computing solution, providing client access to remote displays and devices (e.g. keyboard, mouse, audio, and USB).

SPICE was originally developed by Qumranet, the company behind [KVM](#), and at the moment it is most commonly used to provide remote access to virtual machines. Despite SPICE being primarily aimed at virtual machines, it can however also be used to access remote X11 displays.

SPICE comprises four principal components: Protocol, Client, Server, and Guest. The [SPICE protocol](#) is the specification for the communication messages between the other components. A SPICE client such as [vinagre](#), [remmina](#), [remote-viewer](#) or [spicy](#) allows users to interact with the VM or remote machine. The SPICE server is the library used to share the VM or remote display using the SPICE protocol. The guest is the additional software that must be running to make SPICE fully functional, such as the [QXL driver](#) and SPICE [VDAgent](#).

The [Spice for Newbies](#) guide provides comprehensive low-level technical documentation of the SPICE architecture, whilst the [Spice User Manual](#) provides higher-level documentation aimed at configuring SPICE. Although this documentation is primarily intended to be used for configuring SPICE in KVM/QEMU virtual machines some of the information is also useful for configuring remote displays.

6-9-1 xserver-xspice

Xspice is an X Server and SPICE server in one. It comprises a wrapper script for executing Xorg with the correct parameters and environment variables and an Xorg module named spiceqxl_drv.so. The Xorg module implements three drivers: a video driver that is mostly identical to the SPICE guest QXL Xorg display driver, and keyboard and mouse drivers reading from the SPICE inputs channel.

Xspice allows regular X11 client connections to the X Server, while a SPICE client provides the keyboard and mouse input and video output. Like VNC and RDP, SPICE client disconnections don't impact X11 client connections.

The [Dockerfile](#) is in the [xserver-xspice](#) directory of [docker-gui](#) under [6-remote-applications/spice](#).

```
FROM debian:stretch-slim

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

# Install from testing as it's not in stretch stable.
RUN \
    echo "deb http://ftp.debian.org/debian testing main" > \
        /etc/apt/sources.list.d/stretch-testing.list && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    pulseaudio ca-certificates locales jwm \
    libgl1-mesa-glx libgl1-mesa-dri spice-vdagent \
    gstreamer1.0-plugins-good gstreamer1.0-plugins-ugly && \
```

```
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends -t testing \
xserver-xspice && \
# Modify PulseAudio daemon config.
sed -i "s/; exit-idle-time = 20/exit-idle-time = -1/g" \
/etc/pulse/daemon.conf && \
sed -i "s/load-module module-console-kit/#load-module module- \
console-kit/g" /etc/pulse/default.pa && \
# Create simple launch script to start pulseaudio,
# xserver-xspice and the spice vdagent used to support
# server resizing and copy/paste. Note that the exec
# before /usr/bin/Xspice is important to ensure that
# signals get correctly handled - without it the
# container may hang on exit.
echo '#!/bin/bash\npulseaudio &\nexec /usr/bin/Xspice --auto
$DISPLAY --password $(cat ${HOME}/.xserver-xspice-passwd) --xse
ssion /usr/local/bin/xsession --vdagent --video-codecs ${SPICE_
VIDEO_CODECS:-gstreamer:h264;gstreamer:vp8;gstreamer:mjpeg;spic
e:mjpeg} --audio-fifo-dir $XDG_RUNTIME_DIR/pulse > /dev/null' >
/usr/local/bin/start-server && \
echo '#!/bin/bash\npacmd load-module module-pipe-sink format=
s16 rate=48000 channels=2\njwm' > /usr/local/bin/xsession && \
chmod +x /usr/local/bin/start-server && \
chmod +x /usr/local/bin/xsession && \
# Generate locales
sed -i "s/^# *`($LANG)`/\1/" /etc/locale.gen && \
locale-gen && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
```

```

# Tidy up JWM for single app use case
sed -i "s/Desktops width=\"4\"/Desktops width=\"1\"/g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton icon=\"\usr\share\jwm\jwm-red.svg\"
>root:1</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<TrayButton label=\"_\">showdesktop</TrayButton>//g" /etc/jwm/system.jwmrc && \
sed -i "s/<Include>\etc\jwm\debian-menu</Include>//g" /etc/jwm/system.jwmrc && \
# We'll be exporting /run/user as a volume
# and we need the mode to be set to 1777
mkdir /tmp/.X11-unix && \
chmod 1777 /tmp/.X11-unix && \
mkdir /run/user && \
chmod 1777 /run/user && \
rm -rf /var/lib/apt/lists/*

VOLUME /tmp/.X11-unix
VOLUME /run/user

CMD ["/usr/local/bin/start-server"]

```

The image may be built **on the remote host** with:

```
docker build -t xserver-xspice .
```

As the xserver-xspice package is only available for Debian stretch in the testing repository, the first step in the Dockerfile is to enable testing in /etc/apt/sources.list.

Most of the packages installed by the Dockerfile are fairly obvious, though we explicitly install gstreamer1.0-plugins-good, gstreamer1.0-plugins-ugly and spice-vdagent as they are not implicitly installed as xserver-xspice dependencies.

We next modify the PulseAudio daemon configuration as we did previously in the xrdp and xpra Dockerfiles. This prevents the PulseAudio daemon from exiting prematurely on idle and removes the ConsoleKit dependency.

After configuring PulseAudio, the Dockerfile next creates a script in /usr/local/bin/start-server to start PulseAudio, Xspice, and the SPICE VDAgent that is used to support server resizing and copy/paste.

```
#!/bin/bash
pulseaudio &
exec /usr/bin/Xspice \
  --auto $DISPLAY \
  --password $(cat ${HOME}/.xserver-xspice-passwd) \
  --xsession /usr/local/bin/xsession \
  --vdagent \
  --video-codecs ${SPICE_VIDEO_CODECS:-gstreamer:h264;gstreamer
:vp8;gstreamer:mjpeg;spice:mjpeg} \
  --audio-fifo-dir $XDG_RUNTIME_DIR/pulse > /dev/null
```

Note that the exec before /usr/bin/Xspice is important to ensure that signals get correctly handled, without it the container may hang on exit.

We also create a script in /usr/local/bin/xsession that is called by start-server. The xsession script first loads the PulseAudio module-pipe-sink module that SPICE uses to forward the audio and then starts jwm.

```
#!/bin/bash
pacmd load-module module-pipe-sink format=s16 rate=48000 channel
ls=2
jwm
```

Finally, as with many of our previous remote server Dockerfiles, we generate the locales and timezone, configure jwm, and perform some tidy up.

The [xserver-xspice.sh](#) launch script is as follows. **Warning** this script simply reads the password from stdin and stores it to `~/.xserver-xspice-passwd` in the clear. For better security the SPICE ticketing or [SASL](#) authentication schemes should be used, though the SPICE documentation is unfortunately rather sparse on how to configure those:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -f "$(id -un)/.xserver-xspice-passwd"; then
    echo "creating password"
    read -s -p "Enter password: " password
    echo
    read -s -p "Confirm password: " confirmation
    echo
    if [ $confirmation != $password ]; then
        echo "Confirmation doesn't match password, exiting!"
        exit 1
    fi
    echo $password > $(id -un)/.xserver-xspice-passwd
fi
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=xserver-xspice \
-p 5900:5900 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
-e XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR \
-e DISPLAY=:1 \
xserver-xspice

# Use --volumes-from to mount /tmp/.X11-unix
# from xserver-xspice and also use that container's IPC
$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
-e DISPLAY=:1 \
--ipc=container:xserver-xspice \
--volumes-from xserver-xspice \
firefox

$DOCKER_COMMAND stop xserver-xspice

```

We can launch xserver-xspice.sh remotely with:

```

ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-a
pplications/spice/xserver-xspice; ./xserver-xspice.sh"

```

6-9-2 remote-viewer

[remote-viewer](#) is a simple remote desktop display client that supports SPICE and VNC. It shares most of its features and limitations with [virt-viewer](#).

The [Dockerfile](#) is in the [remote-viewer](#) directory of [docker-gui](#) under [6-remote-applications/spice](#).

```
FROM debian:stretch-slim

# Install virt-viewer
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    virt-viewer libvdpau1 mesa-vdpau-drivers \
    libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["remote-viewer"]
```

The remote-viewer image may be built with:

```
docker build -t remote-viewer .
```

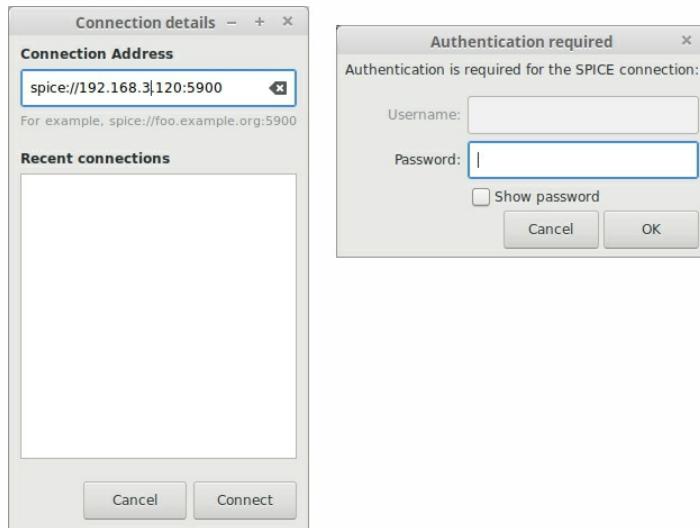
Note that remote-viewer is installed from the Debian virt-viewer package and as with some of our other applications we have speculatively included the libvdpau1 and mesa-vdpau-drivers packages on the assumption that remote-viewer supports VDPAU accelerated video decoding.

The [remote-viewer.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
remote-viewer
```

Starting remote-viewer without a URI will open a simple dialogue with an entry and a list of previously accessed URIs.



6-9-3 spicy

spicy is a simple GTK+ SPICE client which, like [vinagre](#) and [remmina](#), has the useful option to resize the remote display to match the client window size.

The [Dockerfile](#) is in the [spicy](#) directory of [docker-gui](#) under [6-remote-applications/spice](#).

```
FROM debian:stretch-slim

# Install spice-client-gtk
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    spice-client-gtk libvdpau1 mesa-vdpau-drivers \
    libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["spicy"]
```

The spicy image may be built with:

```
docker build -t spicy .
```

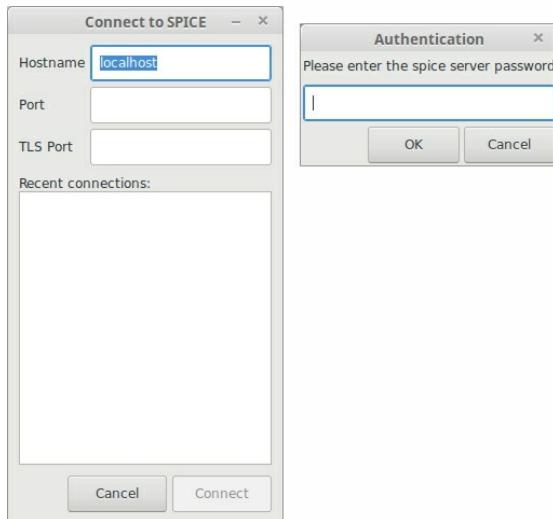
Note that spicy is installed from the Debian spice-client-gtk package and as with some of our other applications we have speculatively included the libvdpau1 and mesa-vdpau-drivers packages on the assumption that spicy supports VDPAU accelerated video decoding.

The [spicy.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
$X11_FLAGS \
spicy
```

Starting spicy without a URI will open a simple dialogue with an entry and a list of previously accessed URIs.



7 Remote 3D Accelerated Applications

The [Remote Applications](#) chapter should have provided readers of this book with a comprehensive suite of tools that may be expanded upon to provide remote access to wide range of applications, however there remains a serious problem. The performance of 3D applications leaves much to be desired, as may be seen by visiting the web site <http://madebyevan.com/webgl-water/> from any of the remote Firefox examples.

The issue is that so far the only OpenGL support available to our remote applications is software rendering, which may be seen by navigating to <https://webglreport.com/?v=2> from a remote Firefox. The available renderer should be similar to:

```
Gallium 0.4 on llvmpipe (LLVM 3.9, 256 bits)
```

The *traditional* approach to the problem of remotely displaying a 3D application with hardware-accelerated rendering is to use GLX “indirect rendering”. Indirect rendering uses the X11 GLX extension to encapsulate OpenGL commands inside the X11 protocol stream and export them from the application to an X Server. Indirect rendering however suffers from many of the same problems as basic [X11 Forwarding](#) and, moreover, many 3D applications use texture mapping which may require prohibitive network bandwidth.

To mitigate these issues and also to enable far greater utilisation of shared GPU resources **server side** hardware-accelerated 3D rendering becomes an appealing option.

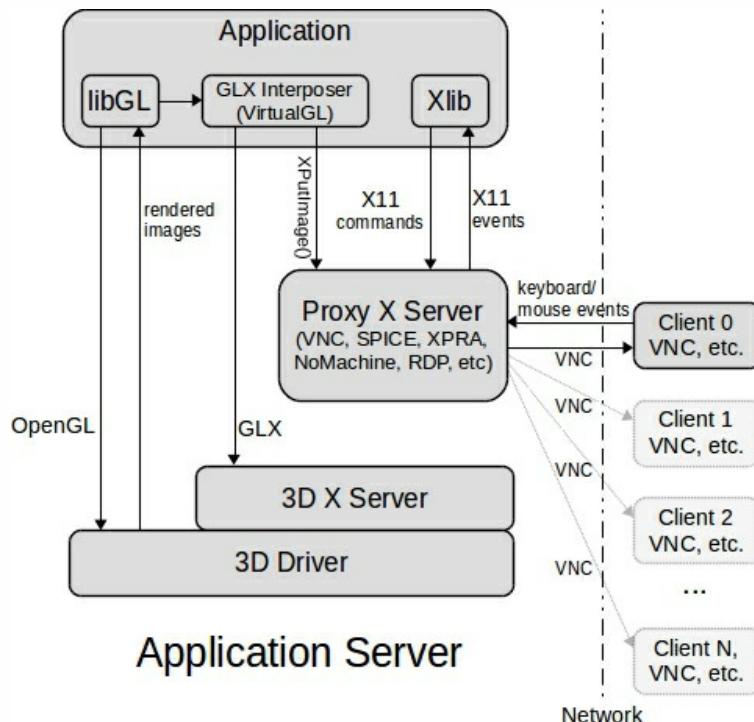
7-1 virtualgl

Performing OpenGL rendering on a server, local to the application, means that the application now has a fast and direct path to the 3D rendering hardware. Moreover, only the rendered 2D images need be sent to the user's desktop however large the 3D data might be. In other words, performing 3D rendering on the server converts a 3D performance problem into a 2D performance problem. The problem is thus how to stream a few megapixels of image data over a network at usable frame rates, which is a problem already addressed by modern video streaming technologies.

One effective approach to server side OpenGL rendering is to use [EGL](#), as described in this Nvidia article: <https://devblogs.nvidia.com/egl-eye-opengl-visualization-without-x-server>. That approach works well for applications designed from the ground-up to be rendered in this way, however many applications use GLX and others use a mixture of OpenGL and toolkits like GTK+, so a more generic approach is required to support these.

The [VirtualGL](#) project provides such a generic solution to the problem of rendering unmodified GLX applications by using [GLX Forking](#), which is also known as split rendering. With GLX Forking, rather than being delivered to their intended X Server the GLX commands are instead intercepted and rerouted to a hardware accelerated 3D X Server whilst the rest of the X11 commands continue to their original target.

VirtualGL intercepts the GLX commands by GLX Interposing, which exploits functionality provided by the dynamic linker on Unix-like systems to allow the linker to bind symbols provided by shared libraries specified by **LD_PRELOAD** before other libraries. The VirtualGL Interposer also enables GPU resource sharing by initialising the OpenGL contexts such that rendering occurs in invisible pixel buffers (**Pbuffers**) rather than in X windows.



At the appropriate time, determined by monitoring [**glXSwapBuffers\(\)**](#) or [**glFinish\(\)**](#), the GLX interposer reads the rendered pixels from the Pbuffers and composites them using `XPutImage()` (or preferably `XShmPutImage()`) into the X window that has been mapped to the appropriate Pbuffer.

The [Dockerfile](#) is in the [virtualgl](#) directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM debian:stretch-slim

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    curl ca-certificates \
    libgl1-mesa-glx libgl1-mesa-dri \
    libglu1-mesa libxv1 libxtst6 && \
    # Attempt to work out the latest VirtualGL version from
    # https://sourceforge.net/projects/virtualgl/files/
    VGL_VERSION=$(curl -sSL https://sourceforge.net/projects/virt
ualgl/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 | \
cut -d \> -f2 | cut -d \<; -f1) && \
    echo "VirtualGL version: ${VGL_VERSION}" && \
    # Given the version download and install VirtualGL
    curl -sSL https://sourceforge.net/projects/virtualgl/files/${
VGL_VERSION}/virtualgl_${VGL_VERSION}_amd64.deb -o virtualgl_${{
VGL_VERSION}}_amd64.deb && \
    dpkg -i virtualgl_*_amd64.deb && \
    # Tidy up packages only used for installing VirtualGL.
    rm virtualgl_*_amd64.deb && \
    apt-get clean && \
    apt-get purge -y curl ca-certificates && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*
```

The virtualgl image may be built **on both the remote and local host** with:

```
docker build -t virtualgl .
```

In common with every other OpenGL enabled Dockerfile we first install libgl1-mesa-glx and libgl1-mesa-dri, but we also need to install libglu1-mesa, libxv1 and libxtst6. Unfortunately those packages aren't included in the package dependencies for the VirtualGL deb and after some strange failures, the author had to resort to [ldd](#) to work out the missing libraries.

In a similar way to our earlier [turbovnc](#) Dockerfile, this Dockerfile attempts to "scrape" the most recent VirtualGL version from the download page via:

```
VGL_VERSION=$(curl -sSL https://sourceforge.net/projects/virtualgl/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 | cut -d \> -f2 | cut -d \< -f1)
```

As with TurboVNC, this scraping is fragile and changes to the format of the VirtualGL sourceforge download page could cause it to fail. If that happens then simply replacing that line with a specific VirtualGL version should fix it, e.g.

```
VGL_VERSION=2.6.3
```

After downloading and installing the VirtualGL deb package the Dockerfile then removes the downloaded deb file.

glxspheres

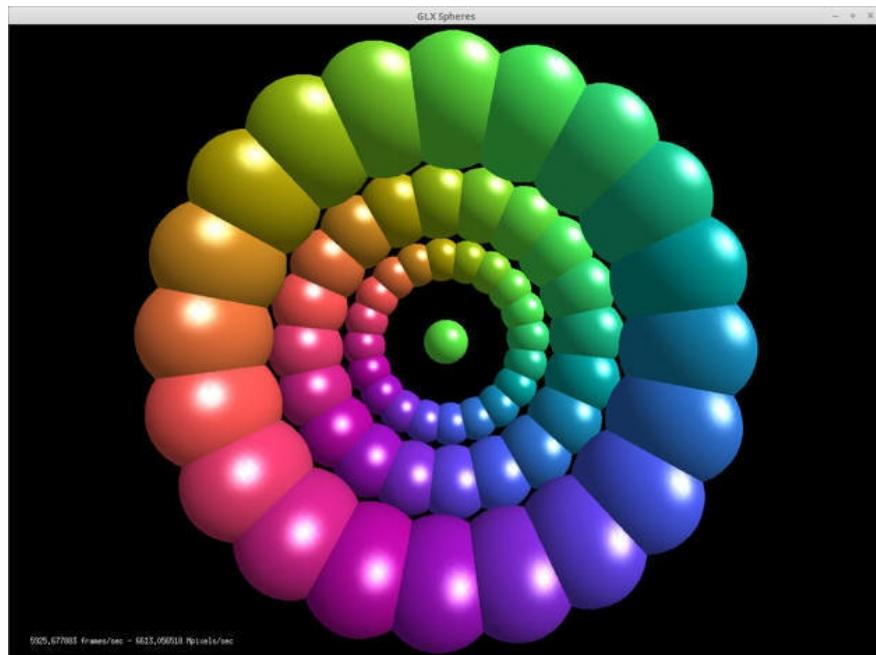
In addition to the main VirtualGL binaries and libraries the VirtualGL deb package ships with a few additional binaries, one of which is `glxspheres64`. This is a benchmark to illustrate VirtualGL performance rather better than `glxgears`, which is not a particularly representative OpenGL application.

The [glxspheres.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm -it \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
$GPU_FLAGS \
-e vblank_mode=0 \
-e __GL_SYNC_TO_VBLANK=0 \
virtualgl /opt/VirtualGL/bin/glxspheres64 $@
```

The vblank_mode and __GL_SYNC_TO_VBLANK environment variables disable vertical sync for Mesa and Nvidia respectively so that the rate isn't stuck at 60fps.



The `glxspheres.sh` script is simply intended to establish a baseline performance, as it is using the host GLX and GPU directly and doesn't *actually* use VirtualGL at all.

The following [`glxspheres-vgl.sh`](#) launch script, however, does run `glxspheres64` through VirtualGL. `DISPLAY` can be any X11 display on the host, for example either the main display or a Xephyr window:

```
TARGET_DISPLAY=${DISPLAY}:0
DISPLAY=${VGL_DISPLAY}:0 # The 3D Display

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm -it \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_FLAGS \
$GPU_FLAGS \
-e DISPLAY=$TARGET_DISPLAY \
virtualgl vglrun /opt/VirtualGL/bin/glxspheres64 $@
```

This script should be run by specifying the target 2D `DISPLAY` on the environment e.g.:

```
DISPLAY=:0 ./glxspheres-vgl.sh
```

Note that `glxspheres-vgl.sh` assumes the 3D X Server is on display :0. If the 3D X Server is on a different display then the `VGL_DISPLAY` environment variable should be set to the required value and passed to the container.

To actually run glxspheres64 remotely, the [glxspheres-remote.sh](#) launch script is as follows:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./glxspheres
-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./glxspheres-remote.sh"
    exit 1
fi

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_XAUTH \
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
-e DISPLAY=:1 \
--ipc=container:$REMOTE \
--volumes-from $REMOTE \
virtualgl vglrun /opt/VirtualGL/bin/glxspheres64 $@
```

This script depends on one of the servers from the [Remote Applications](#) chapter like [xserver-xspice](#), [nomachine](#), [turbovnc](#). etc. already being up and running on the remote host and we pass the container name to the script in the REMOTE environment variable as follows:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3
D-accelerated-applications/virtualgl; REMOTE=xserver-xspice ./g
lxospheres-remote.sh"
```

7-2 glxgears-vgl

To show that VirtualGL really does provide a general solution for rendering unmodified remote GLX applications the remainder of this chapter illustrates remote versions of some of the applications that we have previously covered.

The majority of applications simply need to be launched using vglrun, so the easiest approach is to replace the base image in their Dockerfiles with our virtualgl image and add vglrun to the application ENTRYPOINT.

The [Dockerfile](#) is in the `glxgears-vgl` directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl

# nvidia-docker hooks (only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    mesa-utils && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["vglrun", "glxgears"]
```

The image may be built **on the remote host** with:

```
docker build -t glxgears-vgl .
```

The [glxgears-remote.sh](#) launch script is as follows:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./glxgears-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./glxgears-remote.sh"
    exit 1
fi

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

# Use --volumes-from to mount /tmp/.X11-unix
# from REMOTE container and also use that container's IPC
$DOCKER_COMMAND run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_XAUTH \
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
-e DISPLAY=:1 \
--ipc=container:$REMOTE \
--volumes-from $REMOTE \
glxgears-vgl
```

As with our glxspheres example this script depends on a server like xserver-xspice, nomachine, etc. already being available on the remote host and again we can select which of our servers to use by specifying the container name in the REMOTE environment variable.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3D-accelerated-applications/glxgears-vgl; REMOTE=nomachine ./glxgears-remote.sh"
```

7-3 firefox-vgl

The [Dockerfile](#) is in the `firefox-vgl` directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl

ENV FIREFOX_LANG=en-GB

LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    wget bzip2 ca-certificates gnupg dirmngr procps \
    fonts-symbola fonts-lmodern fonts-freefont-ttf \
    fonts-liberation fonts-dejavu gsfonts \
    libgtk-3-0 libgtk2.0-0 libnss3 libxt6 libavcodec57 \
    libvpx4 libdbus-glib-1-2 libcanberra-gtk3-module \
    libpulse0 libv4l-0 libgl1-mesa-glx libgl1-mesa-dri && \
    # Debian ships with esr, so install latest Firefox.
    wget -qO - "https://download.mozilla.org/?product=firefox-latest&os=linux64&lang=${FIREFOX_LANG}" | tar -xjv -C /usr/lib &&
\
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

COPY fonts.conf /etc/fonts/local.conf
ENTRYPOINT ["vglrun", "/usr/lib/firefox/firefox", "--no-remote"]
]
```

The image may be built **on the remote host** with:

```
docker build -t firefox-vgl .
```

The [firefox-remote.sh](#) launch script is as follows:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./firefox-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./firefox-remote.sh"
    exit 1
fi

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $X11_XAUTH \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:$REMOTE \
    --volumes-from $REMOTE firefox-vgl
```

Like glxspheres this script depends on a server like xserver-xspice, nomachine, etc. already being available on the remote host and again we can select which of our servers to use by specifying the container name in the REMOTE environment variable.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3  
D-accelerated-applications/firefox-vgl; REMOTE=nomachine ./fire  
fox-remote.sh"
```

By using Firefox we can perhaps see one of the clearest illustrations yet of how using VirtualGL can improve the performance of some remote applications. By visiting <http://madebyevan.com/webgl-water/> first from the Firefox bundled with the remote server and then from the Firefox accelerated by VirtualGL, we should hopefully see significantly improved rendering performance of the WebGL.

By navigating to <https://webglreport.com/?v=2> from the Firefox accelerated by VirtualGL we should now see the native GPU renderer rather than llvmpipe e.g.:

```
GeForce GTX TITAN X/PCIe/SSE2
```

firefox-remote.sh is configured to use the native audio support of servers like xserver-xspice, nomachine or xrdp-native-audio, alternatively to use PulseAudio forwarding add

```
. $BIN/docker-pulseaudio-all.sh
```

to the start of the launch script and replace

```
-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native
```

with

```
$PULSEAUDIO_FLAGS
```

7-4 cuda-smoke-vgl

The [Dockerfile](#) is in the [cuda-smoke-vgl](#) directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl

ENV CUDA_VERSION 9.0
ENV NVIDIA_REQUIRE_CUDA "cuda>=${CUDA_VERSION}"
ENV CUDA_FULL_VERSION ${CUDA_VERSION}.176

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver
LABEL com.nvidia.cuda.version="${CUDA_FULL_VERSION}"

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used to install CUDA
    apt-get install -y --no-install-recommends \
    apt-transport-https wget gnupg ca-certificates && \
    # Add the CUDA packages and signing key
    wget -O - https://developer.download.nvidia.com/compute/cuda/
    repos/ubuntu1604/x86_64/7fa2af80.pub | apt-key add - && \
    echo "deb https://developer.download.nvidia.com/compute/cuda/
    repos/ubuntu1604/x86_64 /" > /etc/apt/sources.list.d/cuda.list
    && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libgl1-mesa-glx libgl1-mesa-dri libglu1-mesa freeglut3 \
    cuda-cudart-${CUDA_VERSION} \
    cuda-samples-${CUDA_VERSION} && \
    ln -s cuda-${CUDA_VERSION} /usr/local/cuda && \
    # Compile the sample. Note the tweak of findglllib.mk
    # that is needed to find the OpenGL libraries on debian.
    cd /usr/local/cuda/samples/5_Simulations && \
    cd smokeParticles && \
```

```

sed -i 's/grep -i ubuntu/grep -i "ubuntu\\|debian"/' \
    findgllib.mk && \
make && cp smokeParticles /usr/local/cuda/bin/. && \
cp -R data /usr/local/cuda/bin/. && make clean && \
# Remove the packages used for installation and build
apt-get clean && \
apt-get purge -y apt-transport-https wget gnupg \
    ca-certificates cuda-samples-${CUDA_VERSION} && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

# Set WORKDIR and use relative path as this example uses
# Nvidia's sdkFindFilePath to find the data directory and
# using the absolute path doesn't work correctly.
WORKDIR /usr/local/cuda/bin

```

The image may be built **on the remote host** with:

```
docker build -t cuda-smoke-vgl .
```

The [cuda-smoke-remote.sh](#) launch script is as follows:

```

if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./cuda-smoke
-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./cuda-smoke-remote.sh"
    exit 1
fi

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

```

```
# Use --volumes-from to mount /tmp/.X11-unix
# from REMOTE container and also use that container's IPC
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    $X11_XAUTH \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    -e DISPLAY=:1 \
    --ipc=container:$REMOTE \
    --volumes-from $REMOTE \
    cuda-smoke-vgl vglrun ./smokeParticles $@
```

Again this script depends on a server like xserver-xspice, nomachine, etc. already being available on the remote host and again we can select which of our servers to use by specifying the container name in the REMOTE environment variable.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3D-accelerated-applications/cuda-smoke-vgl; REMOTE=nomachine ./cuda-smoke-remote.sh"
```

This example nicely illustrates that we can use the GPUs available on the remote server both for 3D acceleration **and** compute acceleration.

If we use NoMachine as our remote server we also have the option of forwarding the display to a mobile device, as NoMachine has both iOS and Android clients available. This is quite a neat way to illustrate the potential of what we have learned so far as the NoMachine client works quite well even on fairly low-end mobile devices.

7-5 frameworks-graphics-samples-vgl

The [Dockerfile](#) is in the [frameworks-graphics-samples-vgl](#) directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    git ca-certificates build-essential g++ \
    libxinerama-dev libxext-dev libxrandr-dev libxi-dev \
    libxcursor-dev libxxf86vm-dev libvulkan-dev \
    libgl1-mesa-dri libgl1-mesa-dev && \
    cd /usr/src && \
    git clone https://github.com/NVIDIAAGameWorks/GraphicsSamples.
git && \
    cd GraphicsSamples && \
    # debian:stretch Vulkan version is 1.0.39.0 so later
    # versions of GameWorks won't compile with stretch.
    git checkout 61350521f6b183d9f694f5c72c11efcf1a0cc665 -b build
d && \
    cd samples/build/linux64 && \
    make -j$(getconf _NPROCESSORS_ONLN) release && \
    rm -rf build && \
    apt-get clean && \
    apt-get purge -y \
    git ca-certificates build-essential g++ && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /usr/src/GraphicsSamples/samples/bin/linux64
```

The image may be built **on the remote host** with:

```
docker build -t frameworks-graphics-samples-vgl .
```

The [blended-anti-aliasing-remote.sh](#) launch script is:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./blended-anti-aliasing-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./blended-anti-aliasing-remote.sh"
    exit 1
fi

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

# Use --volumes-from to mount /tmp/.X11-unix
# from REMOTE container and also use that container's IPC
docker run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_XAUTH \
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
-e DISPLAY=:1 \
--ipc=container:$REMOTE \
--volumes-from $REMOTE \
frameworks-graphics-samples-vgl vglrun ./BlendedAA
```

As with the other VirtualGL applications this script depends on a server already being available on the remote host.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3D-accelerated-applications/frameworks-graphics-samples-vgl; REMOTE=nomachine ./blended-anti-aliasing-remote.sh"
```

The [threaded-rendering-remote.sh](#) launch script is:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./threaded-r
endering-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./threaded-rendering-remot
e.sh"
    exit 1
fi

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

# Use --volumes-from to mount /tmp/.X11-unix
# from REMOTE container and also use that container's IPC
docker run --rm \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
$X11_XAUTH \
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
-e DISPLAY=:1 \
--ipc=container:$REMOTE \
--volumes-from $REMOTE \
frameworks-graphics-samples-vgl vglrun ./ThreadedRenderingGL
```

As before this script depends on a remote server being available.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3
D-accelerated-applications/frameworks-graphics-samples-vgl; REMO
TE=nomachine ./threaded-rendering-remote.sh"
```

7-6 blender-vgl

The [Dockerfile](#) is in the [blender-vgl](#) directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl

# nvidia-docker hooks (Only needed for Nvidia Docker V1)
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    # Add the packages used
    apt-get install -y --no-install-recommends \
    blender libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["vglrun", "blender"]
```

The image may be built **on the remote host** with:

```
docker build -t blender-vgl .
```

The [blender-remote.sh](#) launch script is as follows:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./blender-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./blender-remote.sh"
    exit 1
fi
```

```

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $X11_XAUTH \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:$REMOTE \
    --volumes-from $REMOTE \
    blender-vgl @@

```

Again we can select which of our remote servers to use by specifying the container name in the REMOTE environment variable.

```

ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3
D-accelerated-applications/blender-vgl; REMOTE=nomachine ./blen
der-remote.sh"

```

CUDA

To make things a little more interesting we can also follow the example of [cuda-smoke-vgl](#) and add CUDA accelerated rendering to our remote Blender application.

The [Dockerfile](#) is in the `blender-vgl` directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl

ENV CUDA_VERSION 9.0
ENV NVIDIA_REQUIRE_CUDA "cuda>=${CUDA_VERSION}"
ENV CUDA_FULL_VERSION ${CUDA_VERSION}.176
LABEL com.nvidia.volumes.needed=nvidia_driver
LABEL com.nvidia.cuda.version="${CUDA_FULL_VERSION}"

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    blender libgl1-mesa-glx libgl1-mesa-dri \
    apt-transport-https wget gnupg ca-certificates && \
    wget -O - https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub | apt-key add - && \
    echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64 /" > /etc/apt/sources.list.d/cuda.list \
&& \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    cuda-cudart-${CUDA_VERSION} \
    cuda-command-line-tools-${CUDA_VERSION} && \
    ln -s cuda-${CUDA_VERSION} /usr/local/cuda && \
    apt-get clean && \
    apt-get purge -y apt-transport-https wget gnupg \
        ca-certificates && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["vglrun", "blender"]
```

The image may be built **on the remote host** with:

```
docker build -t blender-cuda-vgl -f Dockerfile-cuda .
```

The [blender-cuda-remote.sh](#) launch script is as follows:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./blender-cuda-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./blender-cuda-remote.sh"
    exit 1
fi
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $X11_XAUTH \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:$REMOTE \
    --volumes-from $REMOTE blender-cuda-vgl $@
```

We can launch `blender-cuda-remote.sh` remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3D-accelerated-applications/blender-vgl; REMOTE=nomachine ./blender-cuda-remote.sh"
```

7-7 extemetuxracer-vgl

The [Dockerfile](#) may be found in the [extemetuxracer-vgl](#) directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl
LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    extemetuxracer libpulse0 \
    libgl1-mesa-glx libgl1-mesa-dri && \
    apt-get -y autoremove && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["vglrun", "/usr/games/etr"]
```

The image may be built **on the remote host** with:

```
docker build -t extemetuxracer-vgl .
```

The [extemetuxracer-remote.sh](#) launch script is as follows:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./extemetux
racer-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./extemetuxracer-remote.s
h"
    exit 1
fi
```

```

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)\\.config/pulse
$DOCKER_COMMAND run --rm \
    --device=/dev/input -v /dev/input:/dev/input:ro \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $X11_XAUTH \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:$REMOTE \
    --volumes-from $REMOTE \
    extremetuxracer-vgl $@

```

We can launch `extremetuxracer-remote.sh` remotely with:

```

ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3
D-accelerated-applications/extremetuxracer-vgl; REMOTE=nomachine
e ./extremetuxracer-remote.sh"

```

As with [firefox-vgl](#), our `extremetuxracer` application is configured to use the server's native audio forwarding support and would need to be modified as described in `firefox-vgl` to use PulseAudio forwarding.

Another point to be aware of is that game controller forwarding is not available in any of our example servers, so unfortunately we have to rely on basic keyboard controls.

7-8 dolphin-emu-vgl

The [Dockerfile](#) may be found in the [dolphin-emu-vgl](#) directory of [docker-gui](#) under [7-remote-3D-accelerated-applications](#).

```
FROM virtualgl

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

LABEL com.nvidia.volumes.needed=nvidia_driver

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    locales tzdata dolphin-emu pulseaudio \
    libgl1-mesa-glx libgl1-mesa-dri && \
    apt-get -y autoremove && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
    locale-gen && \
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    DEBIAN_FRONTEND=noninteractive \
    dpkg-reconfigure tzdata && \
    cp /etc/pulse/client.conf \
        /etc/pulse/client-noshm.conf && \
    sed -i "s/; enable-shm = yes/enable-shm = no/g" \
        /etc/pulse/client-noshm.conf

ENTRYPOINT ["vglrun", "/usr/games/dolphin-emu"]
```

The image may be built **on the remote host** with:

```
docker build -t dolphin-emu-vgl .
```

The [dolphin-emu-remote.sh](#) launch script is as follows:

```
if [ -z ${REMOTE+x} ]; then
    echo "Usage: REMOTE=<display server container> ./dolphin-emu-remote.sh"
    echo "e.g. REMOTE=xserver-xspice ./dolphin-emu-remote.sh"
    exit 1
fi

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

mkdir -p $(id -un)/.config/pulse
$DOCKER_COMMAND run --rm \
    --device=/dev/input -v /dev/input:/dev/input:ro \
    --device=/dev/cdrom \
    --group-add $(cut -d: -f3 < <(getent group cdrom)) \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $X11_XAUTH \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:$REMOTE \
    --volumes-from $REMOTE \
    dolphin-emu-vgl $@
```

We can launch dolphin-emu-remote.sh remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/7-remote-3D-accelerated-applications/dolphin-emu-vgl; REMOTE=nomachine ./dolphin-emu-remote.sh"
```

8 Web Applications

The [Remote Applications](#) chapter provided us with a broad range of options for application remoting. Unfortunately however, all the options presented require some form of protocol specific client application to be installed.

In many cases, particularly in tightly controlled corporate environments, the range of available clients might be limited and often installing a client simply isn't an option at all. Fortunately, HTML5 browsers are becoming relatively ubiquitous, even on locked-down corporate IT systems, which gives us another option for remoting our applications.

With a browser-based approach, administrators don't need to install client software locally on users' devices. HTML5 remote desktops also don't store any data on the client itself. Instead, data reaches the client through the WebSocket protocol. This model centralises processing and storage to just that running on the server, simplifying tasks like patch management, so as well as being convenient for users it can also reduce administration effort.

By moving most of the processing and deployment burden from the endpoint to the data centre, client hardware requirements are minimal compared to those of traditional desktops. With many of the solutions presented in this chapter even relatively low-end devices can access applications that make use of 3D and compute acceleration.

8-1 GTK 3 Broadway

Broadway is a back-end for GTK 3 that allows GTK+ applications to be displayed natively within modern web browsers using HTML5 and WebSockets.

8-1-1 broadwayd-gnome-calculator

broadwayd is a display server for the Broadway back-end that we shall illustrate here using [GNOME Calculator](#).

The [Dockerfile](#) is in the [broadwayd-gnome-calculator](#) directory of [docker-gui](#) under [8-web-applications/gtk3-broadway](#).

```
FROM debian:stretch-slim

# Install broadwayd and gnome-calculator
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libcanberra-gtk3-module packagekit-gtk3-module \
    ca-certificates gnome-calculator libgtk-3-bin && \
    echo '#!/bin/bash\nbroadwayd :0 &\nexec gnome-calculator' > / \
    usr/local/bin/start-server && \
    chmod +x /usr/local/bin/start-server && \
    rm -rf /var/lib/apt/lists/*

CMD ["/usr/local/bin/start-server"]
```

The image may be built **on the remote host** with:

```
docker build -t broadwayd-gnome-calculator .
```

This Dockerfile is similar to our original [gnome-calculator](#), but adds the libgtk-3-bin package which contains broadwayd and also includes a simple script to launch broadwayd and gnome-calculator when the container starts.

The [broadwayd-gnome-calculator.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un) /.config/dconf

$DOCKER_COMMAND run --rm -it \
--init \
-p 8080:8080 \
-e GDK_BACKEND=broadway \
-e BROADWAY_DISPLAY=:0 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
broadwayd-gnome-calculator
```

To use Broadway this script replaces the X11_FLAGS with GDK_BACKEND=broadway and BROADWAY_DISPLAY=:0 and may be launched remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/gtk3-broadway/broadwayd-gnome-calculator; ./broadwayd-gnome-calculator.sh"
```

To use, simply point a browser to <http://remote-host:8080>

8-1-2 broadwayd

broadwayd allows multiple GTK+ applications to display their windows in the same web browser by connecting to the same broadwayd instance.

To illustrate this we adopt a pattern similar to our other remote servers, where we have one container whose purpose is to act as the application's display server and export that to clients and other containers representing the applications.

The [Dockerfile](#) for broadwayd is in the [broadwayd](#) directory of [docker-gui](#) under [8-web-applications/gtk3-broadway](#).

```
FROM debian:stretch-slim

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libgtk-3-bin && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["broadwayd"]
```

The image may be built **on the remote host** with:

```
docker build -t broadwayd .
```

If we build our original [gnome-calculator](#) from [4-local-applications/gnome-dbus-and-apparmor](#) **on the remote host** we can see this approach achieves the same result as our earlier [broadwayd-gnome-calculator](#) application, but in a rather more modular way.

The [broadwayd-gnome-calculator.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/dconf
$DOCKER_COMMAND run --rm -it -d \
    --init \
    --shm-size 2g \
    --name=broadwayd \
    -p 8080:8080 \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    broadwayd :0

$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -e GDK_BACKEND=broadway \
    -e BROADWAY_DISPLAY=:0 \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    --network=container:broadwayd \
    --ipc=container:broadwayd \
    gnome-calculator
$DOCKER_COMMAND stop broadwayd
```

This may be launched remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/gtk3-broadway/broadwayd; ./broadwayd-gnome-calculator.sh"
```

As before, to use we can simply navigate a browser to <http://remote-host:8080>.

Although this pattern is similar to our other remote servers, note the subtle difference where we use:

```
--network=container:broadwayd
```

whereas with our other remote display servers we have used `--volumes-from`. This difference is because the broadway socket is an [abstract socket](#), whereas the X11 socket used elsewhere is a standard Unix domain socket.

Epiphany

Because Broadway is a back-end for GTK 3 it will only work with *native* GTK 3 applications, which means that [Epiphany](#) is the only browser that we can currently remotely serve via Broadway. To try this, first build [epiphany](#) from [5-more-complex-applications/browsers on the remote host](#).

The [broadwayd-epiphany.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio-all.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un) /.config/pulse
mkdir -p $(id -un) /.config/dconf
```

```

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=broadwayd \
-p 8080:8080 \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
broadwayd :0

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-e GDK_BACKEND=broadway \
-e BROADWAY_DISPLAY=:0 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
--network=container:broadwayd \
--ipc=container:broadwayd \
epiphany

$DOCKER_COMMAND stop broadwayd

```

and may be launched remotely with:

```

ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/gtk3-broadway/broadwayd; ./broadwayd-epiphany.sh"

```

Whilst Epiphany *should* work with Broadway, the epiphany image needs to be built from the Debian testing repository to play YouTube videos correctly. The instability of testing means that updates sometimes break Broadway support, so it may be necessary to experiment building with other base images.

remmina

Although Broadway only has *native* support for GTK 3 applications, by being creative it is actually possible to extend this to a more complete set of applications because one of the applications that Broadway *does* support is [remmina](#).

If we build remmina and any of the servers that it supports, like [xserver-xspice](#), **on the remote host** we can then use our Broadway enabled remmina as an HTML5 proxy to the other remote server.

The [broadwayd-remmina.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio-all.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=broadwayd \
-p 8080:8080 \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
broadwayd :0
```

```
$DOCKER_COMMAND run --rm \
--shm-size 2g \
-e GDK_BACKEND=broadway \
-e BROADWAY_DISPLAY=:0 \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
$PULSEAUDIO_FLAGS \
--network=container:broadwayd \
--ipc=container:broadwayd \
remmina

$DOCKER_COMMAND stop broadwayd
```

We can launch xserver-xspice.sh remotely exactly as we did before:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/spice/xserver-xspice; ./xserver-xspice.sh"
```

and then launch broadwayd-remmina.sh.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/gtk3-broadway/broadwayd; ./broadwayd-remmina.sh"
```

and if we navigate to <http://remote-host:8080> we can then configure remmina to connect to xserver-xspice.

This approach works surprisingly well, as the remote server is actually local to broadwayd, so only the Broadway HTML5 data is actually being sent over the network.

8-2 Guacamole

With [broadwayd](#) we were able to use remmina as a proxy to a range of other remote display servers so that only Broadway's HTML5 WebSocket protocol actually traverses the network.

[Apache Guacamole](#) follows a similar proxy-based approach. The Guacamole architecture uses a pluggable middleware layer, which translates between the particular remote desktop protocol and the Guacamole protocol. The Guacamole protocol is a protocol for remote display rendering and event transport and as such in many ways is a superset of existing remote desktop protocols, however it is highly optimised for efficient browser parsing and rendering.

The [Dockerfile](#) is in the [guacamole](#) directory of [docker-gui](#) under [8-web-applications](#).

```
FROM debian:stretch-slim

RUN mkdir -p /usr/share/man/man1 && \
    mkdir -p /usr/share/man/man7 && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    autoconf automake gcc libcairo2-dev libfreerdp-dev \
    libjpeg62-turbo-dev libossp-uuid-dev libpango1.0-dev \
    libpulse-dev libssh2-1-dev libssl-dev libtelnet-dev \
    libtool libvncserver-dev libwebsockets-dev libwebrtc-dev \
    make libavcodec-dev libwscale-dev libvorbis-dev curl \
    ca-certificates libfreerdp-plugins-standard \
    ghostscript fonts-liberation fonts-dejavu default-jre \
    xfonts-terminus libossp-uuid16 libpangocairo-1.0-0 \
    libpango-1.0-0 libwebsockets8 libavcodec57 \
```

```
libswscale4 libvorbisenc2 libvorbisfile3 libssh2-1 \
libtelnet2 libvncclient1 libfreerdp-client1.1 \
libfreerdp-cache1.1 && \
GUACAMOLE_VERSION=1.0.0 && \
cd /usr/local/src && \
curl -sSL "http://apache.org/dyn/closer.cgi?action=download&filename=guacamole/${GUACAMOLE_VERSION}/source/guacamole-server-${GUACAMOLE_VERSION}.tar.gz" | tar -xvz -C /usr/local/src && \
mv /usr/local/src/guacamole-server-${GUACAMOLE_VERSION} \
/usr/local/src/guacamole-server && \
cd guacamole-server && \
./configure && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
ldconfig && \
# Download Tomcat. If build fails try updating version
TOMCAT_VERSION=9.0.31 && \
CATALINA_HOME=/usr/local/tomcat && \
curl -sSL https://apache.mirrors.nublue.co.uk/tomcat/tomcat-9 \
/v${TOMCAT_VERSION}/bin/apache-tomcat-${TOMCAT_VERSION}.tar.gz \
| tar -xvz -C /usr/local && \
mv /usr/local/apache-tomcat-${TOMCAT_VERSION} \
/usr/local/tomcat && \
chmod -R ugo=rwX ${CATALINA_HOME} && \
# Download and install guacamole-client war
curl -sSL "http://apache.org/dyn/closer.cgi?action=download&filename=guacamole/${GUACAMOLE_VERSION}/binary/guacamole-${GUACAMOLE_VERSION}.war" -o ${CATALINA_HOME}/webapps/ROOT.war && \
rm -rf ${CATALINA_HOME}/webapps/ROOT && \
# Create simple launch script to start guacd and Tomcat
echo '#!/bin/bash\n/usr/local/sbin/guacd -f &\nexec /usr/local/tomcat/bin/catalina.sh run' > /usr/local/bin/startup && \
chmod +x /usr/local/bin/startup && \
```

```
# Tidy up
rm -rf /usr/local/src/guacamole-server && \
apt-get clean && \
apt-get purge -y \
autoconf automake gcc libcairo2-dev libfreerdp-dev \
libjpeg62-turbo-dev libossp-uuid-dev libpango1.0-dev \
libpulse-dev libssh2-1-dev libssl-dev libtelnet-dev \
libtool libvncserver-dev libwebsockets-dev \
libwebrtc-dev make libavcodec-dev libswscale-dev \
libvorbis-dev curl && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*
CMD ["/usr/local/bin/startup"]
```

The image may be built **on the remote host** with:

```
docker build -t guacamole .
```

As Guacamole is a Java application and we are using a slim base image, as previously mentioned in our [libreoffice](#) example we have to add some man page directories or the default-jre installation will fail:

```
mkdir -p /usr/share/man/man1 && \
mkdir -p /usr/share/man/man7
```

We can also see that Guacamole clearly has a large number of package dependencies and, unfortunately, most of the packages included in the Dockerfile were identified by trial and error as the Dockerfile was being developed by noting the output of the `guacamole-server ./configure` command.

In a nutshell, the Dockerfile downloads guacamole-server then builds it from source as there is no binary version available. It then downloads and installs Apache Tomcat and then the Guacamole .war file. The Dockerfile next creates a simple startup script to launch guacd and Tomcat:

```
#!/bin/bash  
/usr/local/sbin/guacd -f &  
exec /usr/local/tomcat/bin/catalina.sh run
```

Finally, the image is tidied up by removing the packages that were only required to build and install Guacamole.

It is important to note that Guacamole is a relatively complex application with a wide range of deployment options and this implementation has deliberately kept things relatively simple so that it will work out of the box. Readers are encouraged to visit the [Apache Guacamole](#) web site for themselves and experiment with this Dockerfile by adding any additional features that might be required.

The [guacamole.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
  
# Create a directory on the host that we can mount as a  
# "home directory" in the container for the current user.  
mkdir -p $(id -un)/.guacamole
```

```

if ! test -f "$(id -un)/.guacamole/user-mapping.xml"; then
    read -p "Enter remote VNC host name: " vnchost
    echo
    read -p "Enter remote RDP host name: " rdphost
    echo
    echo "creating password"
    read -s -p "Enter password: " password
    echo
    read -s -p "Confirm password: " confirmation
    echo
    if [ $confirmation != $password ]; then
        echo "Confirmation doesn't match password, exiting!"
        exit 1
    fi

    echo -e "<user-mapping>\n  <!-- Per-user authentication and c
onfig information -->\n  <authorize username=\"$(id -un)\" pass
word=\"$password\">\n    <connection name=\"VNC\">\n      <prot
ocol>vnc</protocol>\n      <param name=\"hostname\">$vnchost</p
aram>\n      <param name=\"port\">5900</param>\n      <param na
me=\"password\">$password</param>\n    </connection>\n  <co
nnexion name=\"RDP\">\n    <protocol>rdp</protocol>\n    <
param name=\"hostname\">$rdphost</param>\n    <param name=\"p
ort\">3389</param>\n    <param name=\"password\">$password</p
aram>\n  </connection>\n</authorize>\n</user-mapping>" >
$(id -un)/.guacamole/user-mapping.xml
fi

$DOCKER_COMMAND run --rm -it \
    --init \
    -p 8080:8080 \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    guacamole

```

This script starts by creating a simple `user-mapping.xml` file, as described in the [configuring-guacamole](#) document. This is Guacamole's basic default authentication module, which is simple and comprises a mapping of usernames to configurations that we populate with the VNC and RDP server hostnames, and a username and password.

For simplicity this script assumes that all of the usernames and passwords are the same and obviously for a production environment readers are encouraged to modify the Dockerfile and launch script to add additional security modules as described in the Guacamole documentation.

We can launch `turbovnc.sh` remotely exactly as we did before:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/6-remote-applications/vnc/turbovnc/turbovnc; ./turbovnc.sh"
```

and then launch `guacamole.sh`.

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/guacamole; ./guacamole.sh"
```

and if we navigate to <http://remote-host:8080> we can then configure guacamole to connect to turbovnc.

Because the VNC and RDP servers are local to the Guacamole server the performance is surprisingly good, as only the Guacamole protocol is actually traversing the network.

8-3 noVNC

Whilst Guacamole and GTK 3 Broadway (with remmina) can provide an HTML5 proxy to VNC, [noVNC](#) on the other hand is a true VNC client that natively implements the [Remote Frame Buffer](#) protocol in JavaScript.

Despite being a complete VNC implementation, as an HTML5 based application noVNC is constrained to use WebSockets as a network transport. For VNC server implementations that only accept TCP connections, a workaround is to use a WebSocket to TCP proxy like [Websockify](#). However some VNC servers like [x11vnc](#) also support native WebSocket connections, so we shall use that to illustrate noVNC.

The main [Dockerfile](#) is in the [x11novnc-xdummy](#) directory of [docker-gui](#) under [8-web-applications](#). This Dockerfile uses [x11vnc-xdummy](#) as a base image, as described in the [x11vnc](#) section, however a [Dockerfile-standalone](#) is also provided.

```
FROM x11vnc-xdummy

# Install noVNC
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl xz-utils ca-certificates && \
# Download noVNC.
NOVNC_VERSION=1.1.0 && \
NOVNC=/usr/local/bin/noVNC-$NOVNC_VERSION && \
APP=/usr/local/bin/noVNC/app && \
curl -ssl https://github.com/novnc/noVNC/archive/v$NOVNC_VERSION.tar.gz | tar -xzv -C /usr/local/bin && \
```

```

# Temporarily install Node.js and npm to transpile
# ECMAScript 6 modules. As well as speeding up load times
# on browsers that don't support modules this circumvents
# an issue with pre 0.9.12 LibVNCServer where the MIME
# type for Javascript was set incorrectly causing Chrome
# to reject them due to strict MIME type checking being
# enabled for modules.
curl -sSL https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-x64.tar.xz | tar -xJv -C /usr/local/lib && \
NODE=/usr/local/lib/node-v10.16.3-linux-x64/bin && \
ln -s ${NODE}/node /usr/local/bin/node && \
ln -s ${NODE}/npm /usr/local/bin/npm && \
ln -s ${NODE}/npx /usr/local/bin/npx && \
npm install -g es6-module-transpiler && \
npm install -g @babel/cli && \
npm install --save-dev @babel/core @babel/preset-env && \
ln -s ${NODE}/babel /usr/local/bin/babel && \
ln -s ${NODE}/compile-modules \
    /usr/local/bin/compile-modules && \
cd ${NOVNC} && \
# Tweak novNC configuration as the defaults point to
# websockify not the built-in x11vnc websocket support.
sed -i "s/UI.initSetting('port', port)/UI.initSetting('port',
5900)/g" app/ui.js && \
sed -i "s/UI.initSetting('path', 'websockify')/UI.initSetting(
'path', '')/g" app/ui.js && \
# Tweak the vnc.html to use the transpiled app.js
sed -i 's/type="module" crossorigin="anonymous" src="app\ui.
js"/src="app.js"/g' vnc.html && \
sed -i 's/<script src="vendor\promise.js"></script>//g' vnc
.html && \
sed -i 's/if (window._noVNC_has_module_support) //g' vnc.html
&& \
# Transpile the Javascript to speed up loading and
# allow it to work on a wider variety of browsers.
echo '{"presets": ["@babel/preset-env"]}' > .babelrc && \
compile-modules convert app/ui.js > app.js && \
babel app.js --out-file app.js && \

```

```

mkdir -p ${APP} && \
mv ${NOVNC}/app/images ${APP}/images && \
mv ${NOVNC}/app/locale ${APP}/locale && \
mv ${NOVNC}/app/sounds ${APP}/sounds && \
mv ${NOVNC}/app/styles ${APP}/styles && \
mv ${NOVNC}/app/error-handler.js \
    ${APP}/error-handler.js && \
mv ${NOVNC}/app.js ${APP}.js && \
# Renaming vnc.html to index.vnc seems necessary as the
# x11vnc built-in web server seems to serve index.vnc
# by default and there's no obvious configuration
# option to change this.
mv ${NOVNC}/vnc.html \
    /usr/local/bin/noVNC/index.vnc && \
# Create script to start Xorg, jwm and x11vnc
echo '#!/bin/bash\nXorg $DISPLAY -cc 4 &\nsleep 0.5\njwm &\nx
11vnc -forever -usepw -httpdir /usr/local/bin/noVNC' > /usr/loc
al/bin/startup && \
chmod +x /usr/local/bin/startup && \
# Tidy up
rm -rf ${NOVNC} && \
rm -rf /usr/local/lib/node-v10.16.3-linux-x64 && \
rm -rf /root/.npm && \
rm /usr/local/bin/node && \
rm /usr/local/bin/npm && \
rm /usr/local/bin/npx && \
apt-get clean && \
apt-get purge -y curl xz-utils && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

```

CMD ["/usr/local/bin/startup"]

The image may be built **on the remote host** with:

```
docker build -t x11novnc-xdummy .
```

As of version 1.0.0 noVNC began using [ES6 modules](#). Whilst this is, in theory, a good thing it brings with it two significant disadvantages. The first issue is that modules are only natively supported on very modern browsers. The second issue is that on some browsers, like Chrome, strict MIME type checking is enabled for modules. Strict MIME checking is a problem for us, because the MIME type for Javascript is set incorrectly in pre 0.9.12 LibVNCServer, so noVNC fails to load.

To resolve these issues and improve load times, after downloading noVNC we temporarily install [Node.js](#) and use [es6-module-transpiler](#) and [babel](#) to transpile noVNC into a single JavaScript bundle app.js.

Node.js, npm, es6-module-transpiler and babel are installed in the Dockerfile as follows:

```
curl -ssl https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-x64.tar.xz | tar -xJv -C /usr/local/lib && \
NODE=/usr/local/lib/node-v10.16.3-linux-x64/bin && \
ln -s ${NODE}/node /usr/local/bin/node && \
ln -s ${NODE}/npm /usr/local/bin/npm && \
ln -s ${NODE}/npx /usr/local/bin/npx && \
npm install -g es6-module-transpiler && \
npm install -g @babel/cli && \
npm install --save-dev @babel/core @babel/preset-env && \
ln -s ${NODE}/babel /usr/local/bin/babel && \
ln -s ${NODE}/compile-modules \
/usr/local/bin/compile-modules && \
cd ${NOVNC} && \
```

The default noVNC configuration assumes that Websockify is being used, so we next have to modify the port and path in UI.initSetting to enable x11vnc native WebSocket support.

```
sed -i "s/UI.initSetting('port', port)/UI.initSetting('port', 5  
900)/g" app/ui.js && \  
sed -i "s/UI.initSetting('path', 'websockify')/UI.initSetting('  
path', '')/g" app/ui.js && \  
"
```

We next need to modify vnc.html to use the transpiled app.js file instead of loading ui.js.

```
sed -i 's/type="module" crossorigin="anonymous" src="app\ui.js  
"/src="app.js"/g' vnc.html && \  
sed -i 's/<script src="vendor\promise.js"><\/script>/g' vnc.h  
tml && \  
sed -i 's/if (window._noVNC_has_module_support) //g' vnc.html &  
& \  
"
```

After all of the preparatory work has been completed, we transpile noVNC. We can then create an app directory /usr/local/bin/noVNC/app and move app.js and the noVNC assets there.

```
echo '{"presets": ["@babel/preset-env"]}' > .babelrc && \  
compile-modules convert app/ui.js > app.js && \  
babel app.js --out-file app.js && \  
mkdir -p ${APP} && \  
mv ${NOVNC}/app/images ${APP}/images && \  
mv ${NOVNC}/app/locale ${APP}/locale && \  
mv ${NOVNC}/app/sounds ${APP}/sounds && \  
mv ${NOVNC}/app/styles ${APP}/styles && \  
mv ${NOVNC}/app/error-handler.js \  
    ${APP}/error-handler.js && \  
mv ${NOVNC}/app.js ${APP}.js && \  
"
```

Finally, we need to move vnc.html to index.vnc as that is the file the x11vnc built-in web server seems to serve.

```
mv ${NOVNC}/vnc.html \
/usr/local/bin/noVNC/index.vnc && \
```

After installing the transpiled noVNC, we create a simple startup script to launch Xorg, jwm and x11vnc.

```
#!/bin/bash
Xorg $DISPLAY -cc 4 &
sleep 0.5
jwm &
x11vnc -forever -usepw -httpdir /usr/local/bin/noVNC
```

The [x11vnc-storepasswd.sh](#) script that interactively sets the VNC password is identical to the one we used for [x11vnc](#) apart from the image name.

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.vnc
$DOCKER_COMMAND run --rm -it \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
x11novnc-xdummy x11vnc -storepasswd
```

The [x11novnc-xdummy.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-pulseaudio-all.sh
. $BIN/docker-dbus-all.sh

# Create a directory on the host that we can mount as a
# "home directory" in the container for the current user.
mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -f "$(id -un)/.vnc/passwd"; then
    echo "creating password"
    $(cd $(dirname $0); echo $PWD)/x11novnc-storepasswd.sh
fi

# Launch Xdummy and x11vnc exposing /tmp/.X11-unix
# as a volume. Use -d option to daemonise and --init to
# run tini as pid 1
$DOCKER_COMMAND run --rm -it -d \
    --init \
    --shm-size 2g \
    --name=x11novnc-xdummy \
    -p 5800:5800 \
    -p 5900:5900 \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e DISPLAY=:1 \
    x11novnc-xdummy
```

```

# Use --volumes-from to mount /tmp/.X11-unix
# from x11novnc-xdummy and also use that container's IPC
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    $PULSEAUDIO_FLAGS \
    -e DISPLAY=:1 \
    --ipc=container:x11novnc-xdummy \
    --volumes-from x11novnc-xdummy \
    firefox

$DOCKER_COMMAND stop x11novnc-xdummy

```

This is more or less identical to our earlier x11vnc-xdummy.sh launch script aside from the image name being used and that port 5800 is being exposed in addition to port 5900.

x11novnc-xdummy.sh may be launched remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/x11novnc-xdummy; ./x11novnc-xdummy.sh"
```

To use noVNC, simply navigate a browser to <http://remote-host:5800>.

Because noVNC supports Tight VNC encodings, including TightJPG and TightPNG, the performance of noVNC is generally very good. It is even possible to watch videos with passable quality, though as we rely on basic uncompressed PulseAudio forwarding for VNC audio support there may be audio stutter on lower bandwidth networks.

8-4 Xpra

[Xpra](#) was previously covered in some detail in the Remote Applications chapter. In that chapter we connected to the server using the native xpra attach command, however xpra also has an HTML5 client available.

The main [Dockerfile](#) is in the [xpra](#) directory of [docker-gui](#) under [8-web-applications](#). This Dockerfile uses [xpra](#) as a base image, as described in the [xpra](#) section, however a [Dockerfile-standalone](#) is also provided in the GitHub repository.

```
FROM xpra

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl libjs-jquery-ui && \
    XPRA_VERSION=3.0.5-r24939-1 && \
    XPRA=https://xpra.org/dists/stretch/main/binary-amd64 && \
    echo "XPRA_VERSION version: ${XPRA_VERSION}" && \
    curl -sSL ${XPRA}/xpra-html5_${XPRA_VERSION}_amd64.deb \
        -o xpra-html5_${XPRA_VERSION}_amd64.deb && \
    dpkg -i xpra-html5_${XPRA_VERSION}_amd64.deb && \
    # Create a simple launch script to start xpra server
    echo '#!/bin/bash\nmkdir -p $XDG_RUNTIME_DIR/pulse\nmkdir -p \
~/xpra\nexec xpra start --daemon=no --notifications=no $DISPLAY \
--ibus-launch= --html=on $@\n' > /usr/local/bin/start && \
    chmod +x /usr/local/bin/start && \
    # Tidy up
    rm xpra-html5_${XPRA_VERSION}_amd64.deb && \
    apt-get clean && \
    apt-get purge -y curl && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*
```

The image may be built **on the remote host** with:

```
docker build -t xpra-html5 .
```

Because most of the hard work has been done in the base image this Dockerfile is relatively simple, first installing the curl and libjs-jquery-ui packages then downloading and installing xpra-html5 via dpkg.

The server startup script in /usr/local/bin/start is very similar to the one used in the original xpra server.

```
#!/bin/bash
mkdir -p $XDG_RUNTIME_DIR/pulse
mkdir -p ~/.xpra
exec xpra start --daemon=no --notifications=no $DISPLAY --dbus-launch= --html=on $@
```

The main difference is the addition of `--html=on`.

The [x11xpra.sh](#) launch script is as follows:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

if ! test -f "$(id -un)/machine-id"; then
    echo $(dbus-uuidgen) > $(id -un)/machine-id
fi
```

```
$DOCKER_COMMAND run --rm -it -d \
--init \
--shm-size 2g \
--name=x11xpra-html15 \
-p 10000:10000 \
-u $(id -u):$(id -g) \
-v /etc/passwd:/etc/passwd:ro \
-v $PWD/$(id -un):/home/$(id -un) \
-v $PWD/$(id -un)/machine-id:/etc/machine-id \
-e XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR \
-e DISPLAY=:1 \
xpra-html15 start --bind=$HOME/.xpra/xpra-socket --bind-tcp=0.0.0.0:10000

$DOCKER_COMMAND run --rm -it \
--volumes-from x11xpra-html15 \
xpra-html15 bash -c "while [ ! -f $XDG_RUNTIME_DIR/pulse/pid ]; do echo 'Waiting for xpra Pulseaudio daemon';sleep 1; done; echo 'xpra Pulseaudio daemon running'"

$DOCKER_COMMAND run --rm \
--shm-size 2g \
-u $(id -u):$(id -g) \
-v $PWD/$(id -un):/home/$(id -un) \
-v /etc/passwd:/etc/passwd:ro \
$APPARMOR_FLAGS \
$DCONF_FLAGS \
-e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
-e DISPLAY=:1 \
--ipc=container:x11xpra-html15 \
--volumes-from x11xpra-html15 \
firefox

$DOCKER_COMMAND stop x11xpra-html15
```

This launch script is identical to the original x11xpra.sh apart from the image name and may be launched remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/xpra; ./x11xpra.sh"
```

To use xpra, simply navigate a browser to <http://remote-host:10000>.

The xpra HTML5 client generally works very well, however one issue is that audio support is not ideal.

With Google Chrome, the audio support largely works out of the box, however unfortunately lip sync does not currently appear to work correctly.

With Firefox, unfortunately audio support does not initially appear to work. However, if we return to the xpra connection window, by disconnecting the session, and then select `Advanced options` we can then change the `Audio forwarding, codec` drop down menu from the default mpeg4: aac to http stream: mp3.

If we connect again after selecting http stream: mp3 (and ensure that audio is unmuted) we should now hear audio, though again unfortunately lip sync does not currently appear to work correctly.

8-5 SPICE

Like Xpra, [SPICE](#) was previously covered in some detail in the Remote Applications chapter. In that chapter we connected to the server using a range of native SPICE clients such as [vinagre](#), [remmina](#), [remote-viewer](#), or [spicy](#), however SPICE also has two independent HTML5 clients available.

8-5-1 xserver-xspice-html5

The “official” SPICE HTML5 client is briefly mentioned in the [features](#) page of the SPICE documentation.

The main [Dockerfile](#) is in the [xserver-xspice-html5](#) directory of [docker-gui](#) under [8-web-applications/spice](#). This Dockerfile uses [xserver-xspice](#) as a base image, as described in the [xserver-xspice](#) section, however a [Dockerfile-standalone](#) is also provided in the GitHub repository.

```
FROM xserver-xspice

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl python-numpy python-setuptools && \
    WS_VERSION=0.9.0 && \
    SPICE_HTML5_VERSION=0.2.2 && \
    SPICE=spice-html5-${SPICE_HTML5_VERSION} && \
    curl -SSL https://github.com/novnc/websockify/archive/v${WS_V \
ERSION}.tar.gz | tar -xzv -C /usr/local/bin && \
    curl -SSL https://github.com/freedesktop/spice-html5/archive/ \
${SPICE}.tar.gz | tar -xzv -C /usr/local/bin && \
```

```
cd /usr/local/bin/websockify-${WS_VERSION} && \
python setup.py install && \
mv /usr/local/bin/spice-html5-${SPICE} \
    /usr/local/bin/spice-html5 && \
mv /usr/local/bin/spice-html5/spice.html \
    /usr/local/bin/spice-html5/index.html && \
# Tweak spice-html5 configuration as defaults to the
# wrong host and port.
sed -i "s/toggle_console;/toggle_console;\ndocument.getElementById('host').value = window.location.hostname;\ndocument.getElementById('port').value = window.location.port;/g" /usr/local/bin/spice-html5/index.html && \
# Create simple launch script to start pulseaudio,
# xserver-xspice and the spice vdagent used to support
# server resizing and copy/paste. Note that the exec
# before /usr/bin/Xspice is important to ensure that
# signals get correctly handled - without it the
# container may hang on exit.
echo '#!/bin/bash\npulseaudio &\nwebsockify 5800 localhost:59
00 --web /usr/local/bin/spice-html5 -D\nexec /usr/bin/Xspice --
auto $DISPLAY --password $(cat ${HOME}/.xserver-xspice-passwd)
--xsession /usr/local/bin/xsession --vdagent --video-codecs ${S
PICE_VIDEO_CODECS:-gstreamer:h264;gstreamer:vp8;gstreamer:mjpeg
;spice:mjpeg} --audio-fifo-dir $XDG_RUNTIME_DIR/pulse > /dev/n
ull' > /usr/local/bin/start-server && \
chmod +x /usr/local/bin/start-server && \
# Tidy up
apt-get clean && \
apt-get purge -y curl && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*
```

CMD ["/usr/local/bin/start-server"]

The image may be built **on the remote host** with:

```
docker build -t xserver-xspice-html5 .
```

The xserver-xspice base image has done the majority of the work, so the responsibility of this Dockerfile is mostly about installing Websockify and spice-html5 via curl and tar then modifying the configuration in index.html.

The index.html modification simply adds the following lines, which populate the hostname and port of the main UI window.

```
document.getElementById('host').value = window.location.hostname;  
document.getElementById('port').value = window.location.port;
```

This Dockerfile replaces the original xserver-xspice start-server script with the following script, which starts Websockify in addition to PulseAudio, Xspice and the SPICE VDAgent.

```
#!/bin/bash  
pulseaudio &  
websockify 5800 localhost:5900 --web /usr/local/bin/spice-html5  
-D  
exec /usr/bin/Xspice \  
--auto $DISPLAY \  
--password $(cat ${HOME}/.xserver-xspice-passwd) \  
--xsession /usr/local/bin/xsession \  
--vdagent --video-codecs ${SPICE_VIDEO_CODECS:-gstreamer:h264  
;gstreamer:vp8;gstreamer:mjpeg;spice:mjpeg} \  
--audio-fifo-dir $XDG_RUNTIME_DIR/pulse > /dev/null
```

The [xserver-xspice-html5.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -f "$(id -un)/.xserver-xspice-passwd"; then
    echo "creating password"
    read -s -p "Enter password: " password
    echo
    read -s -p "Confirm password: " confirmation
    echo
    if [ $confirmation != $password ]; then
        echo "Confirmation doesn't match password, exiting!"
        exit 1
    fi
    echo $password > $(id -un)/.xserver-xspice-passwd
fi

$DOCKER_COMMAND run --rm -it -d \
    --init \
    --shm-size 2g \
    --name=xserver-xspice-html5 \
    -p 5900:5900 \
    -p 5800:5800 \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR \
    -e DISPLAY=:1 \
    xserver-xspice-html5
```

```

# Use --volumes-from to mount /tmp/.X11-unix
# from xserver-xspice-html5 and also use its IPC
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:xserver-xspice-html5 \
    --volumes-from xserver-xspice-html5 \
    firefox

$DOCKER_COMMAND stop xserver-xspice-html5

```

This script is identical to the original xserver-xspice.sh apart from the image name and that port 5800 is being exposed in addition to port 5900. It may be launched remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/spice/xserver-xspice-html5; ./xserver-xspice-html5.sh"
```

To use xserver-xspice-html5, simply navigate a browser to <http://remote-host:5800>.

This SPICE HTML5 client generally works very well despite being referred to as a “proof of concept”. Audio works out of the box for both Firefox and Chrome, however some things, like YouTube videos, work less well with some glitching and loss of lip sync.

8-5-2 xserver-xspice-eyeos

An alternative SPICE HTML5 client has been developed by eyeos and that project has itself been forked by flexVDI, which has some modifications that make it a little easier to integrate than the original eyeos one.

The main [Dockerfile](#) is in the [xserver-xspice-eyeos](#) directory of [docker-gui](#) under [8-web-applications/spice](#). This Dockerfile again uses [xserver-xspice](#) as a base image, as described in the [xserver-xspice](#) section, however a [Dockerfile-standalone](#) is also provided in the GitHub repository.

```
FROM xserver-xspice

# Install eyeos spice-web-client.
RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    curl python-numpy python-setuptools && \
    # Download websockify and flexVDI fork of
    # eyeos spice-web-client.
    WS_VERSION=0.9.0 && \
    SPICE=3.1.0 && \
    curl -SSL https://github.com/novnc/websockify/archive/v${WS_V
ERSION}.tar.gz | tar -xzf -C /usr/local/bin && \
    curl -SSL https://github.com/flexVDI/spice-web-client/archive
/${SPICE}.tar.gz | tar -xzf -C /usr/local/bin && \
    cd /usr/local/bin/websockify-${WS_VERSION} && \
    python setup.py install && \
    mv /usr/local/bin/spice-web-client-${SPICE} \
    /usr/local/bin/spice-web-client && \
```

```
# Tweak spice-web-client to add a basic
# host/port/password entry UI.
sed -i 's/document.location.port/port/g' \
    /usr/local/bin/spice-web-client/lib/utils.js && \
    sed -i 's/<div class="float-right">/<div class="float-right">
\n\n                <label for="host">Host: </label><input typ
e="text" id="host" value=""\/><label for="port"> Port: </label
><input type="text" id="port" value=""\/><label for="password">
Password: </label><input type="password" id="password" value="
" onkeyup="checkIfEnterPressed(event)"\>\n/g' /usr/local/bin/s
pice-web-client/index.html && \
    sed -i 's/$(document).ready(start);$(document).ready(init);/
g' /usr/local/bin/spice-web-client/run.js && \
    sed -i 's/translate();//g' \
        /usr/local/bin/spice-web-client/run.js && \
    sed -i 's/function start ()/\nfunction init () {\n    transla
te();\n    document.getElementById("showclientid").style.displa
y = "none";\n    document.getElementById("uploadfile").style.di
splay = "none";\n    document.getElementById("host").value = do
cument.location.hostname;\n    document.getElementById("port").
value = document.location.port;\n}\n\nfunction checkIfEnterPres
sed (event) {\n    if (event.keyCode === 13) {\n        start()
;\n    }\n}\n\nfunction start ()/g' /usr/local/bin/spice-web-clie
nt/run.js && \
    sed -i "s/data\[['spice_address']\] || ''/document.getElementById(
'host').value/g" /usr/local/bin/spice-web-client/run.js && \
    sed -i "s/data\[['spice_port']\] || 0/document.getElementById(
'port').value/g" /usr/local/bin/spice-web-client/run.js && \
    sed -i "s/data\[['spice_password']\] || ''/document.getElementById(
'password').value/g" /usr/local/bin/spice-web-client/run.js
&& \
```

```

# Create simple launch script to start pulseaudio,
# xserver-xspice and the spice vdagent used to support
# server resizing and copy/paste. Note that the exec
# before /usr/bin/Xspice is important to ensure that
# signals get correctly handled - without it the
# container may hang on exit.
echo '#!/bin/bash\npulseaudio &\nwebssockify 5800 localhost:59
00 --web /usr/local/bin/spice-web-client -D\nexec /usr/bin/Xspice
--auto $DISPLAY --password $(cat ${HOME}/.xserver-xspice-pas
swd) --xsession /usr/local/bin/xsession --vdagent --video-codec
s ${SPICE_VIDEO_CODECS:-gstreamer:h264;gstreamer:vp8;gstreamer:
mjpeg;spice:mjpeg} --audio-fifo-dir $XDG_RUNTIME_DIR/pulse > /d
ev/null' > /usr/local/bin/start-server && \
chmod +x /usr/local/bin/start-server && \
# Tidy up
apt-get clean && \
apt-get purge -y curl && \
apt-get autoremove -y && \
rm -rf /var/lib/apt/lists/*

CMD ["/usr/local/bin/start-server"]

```

The image may be built **on the remote host** with:

```
docker build -t xserver-xspice-eyeos .
```

As with xserver-xspice-html5 the xserver-xspice base image has done the majority of the work, so the responsibility of this Dockerfile is mostly about installing Webssockify and [spice-web-client](#). The modifications to the eyeos/flexVDI spice-web-client are, however, a little more involved than those we had to make to xserver-xspice-html5.

We first modify [lib/utils.js](#) to replace the URL returned by generateWebSocketUrl in

```
wdi.Utils = {
  generateWebSocketUrl: function(protocol, host, port, destHost
, destPort, type, destInfoToken) {
    if (document.location.protocol == "https:") {
      scheme = "wss://";
    } else {
      scheme = "ws://";
    }
    url = scheme + host + ":" + document.location.port + "/" +
"?ver=2&token=" + port;
    return url;
  }
};
```

with

```
url = scheme + host + ":" + port + "/" + "?ver=2&token=" + por
t;
```

We next modify [index.html](#) to add the following HTML to the “float-right” div. This adds a *very* simple UI to allow users to enter the required host, port and password.

```
<label for="host">Host: </label>
<input type="text" id="host" value="" />

<label for="port"> Port: </label>
<input type="text" id="port" value="" />

<label for="password"> Password: </label>
<input type="password" id="password" value="" onkeyup="checkIfE
nterPressed(event)"/>
```

Next, we patch [run.js](#) to replace

```
$(document).ready(start);
```

with

```
$(document).ready(init);
```

so that we call the init() function rather than the start() function when the document loads. We also remove the

```
translate();
```

call from the body of the start() function, as we shall be calling it from our new init() function as follows.

```
function init () {
    translate();
    document.getElementById("showclientid").style.display = "none";
    document.getElementById("uploadfile").style.display = "none";
    document.getElementById("host").value = document.location.hostname;
    document.getElementById("port").value = document.location.port;
}

function checkIfEnterPressed (event) {
    if (event.keyCode === 13) {
        start();
    }
}
```

The init() function calls translate() and populates the UI with the current hostname and port when the document loads. The checkIfEnterPressed() function is called when typing occurs in the password entry box and will call start() when the password has been entered.

Finally, we update the [run.js](#) app.run call to replace the lines:

```
data['spice_address'] || ''  
data['spice_port'] || 0  
data['spice_password'] || ''
```

with

```
document.getElementById('host').value  
document.getElementById('port').value  
document.getElementById('password').value
```

which allows the UI to retrieve the supplied hostname, port and password.

The Dockerfile also replaces the start-server script with:

```
#!/bin/bash  
pulseaudio &  
websockify 5800 localhost:5900 --web /usr/local/bin/spice-web-client -D  
exec /usr/bin/Xspice \  
--auto $DISPLAY \  
--password $(cat ${HOME}/.xserver-xspice-passwd) \  
--xsession /usr/local/bin/xsession \  
--vdagent --video-codecs ${SPICE_VIDEO_CODECS:-gstreamer:h264  
;gstreamer:vp8;gstreamer:mjpeg;spice:mjpeg} \  
--audio-fifo-dir $XDG_RUNTIME_DIR/pulse > /dev/null
```

The [xserver-xspice-eyeos.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-dbus-session-launch.sh # Run before dbus-all
. $BIN/docker-command.sh
. $BIN/docker-dbus-all.sh

mkdir -p $(id -un)/.config/pulse
mkdir -p $(id -un)/.config/dconf

# Create password if required.
if ! test -f "$(id -un)/.xserver-xspice-passwd"; then
    echo "creating password"
    read -s -p "Enter password: " password
    echo
    read -s -p "Confirm password: " confirmation
    echo
    if [ $confirmation != $password ]; then
        echo "Confirmation doesn't match password, exiting!"
        exit 1
    fi
    echo $password > $(id -un)/.xserver-xspice-passwd
fi

$DOCKER_COMMAND run --rm -it -d \
    --init \
    --shm-size 2g \
    --name=xserver-xspice-eyeos \
    -p 5900:5900 \
    -p 5800:5800 \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    -e XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR \
    -e DISPLAY=:1 \
    xserver-xspice-eyeos
```

```

# Use --volumes-from to mount /tmp/.X11-unix
# from xserver-xspice-eyeos and also use its IPC
$DOCKER_COMMAND run --rm \
    --shm-size 2g \
    -u $(id -u):$(id -g) \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v /etc/passwd:/etc/passwd:ro \
    $APPARMOR_FLAGS \
    $DCONF_FLAGS \
    -e PULSE_SERVER=unix:$XDG_RUNTIME_DIR/pulse/native \
    -e DISPLAY=:1 \
    --ipc=container:xserver-xspice-eyeos \
    --volumes-from xserver-xspice-eyeos \
    firefox

$DOCKER_COMMAND stop xserver-xspice-eyeos

```

This script is identical to the original xserver-xspice.sh apart from the image name and that port 5800 is being exposed in addition to port 5900. It may be launched remotely with:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/8-web-applications/spice/xserver-xspice-eyeos; ./xserver-xspice-eyeos.sh"
```

To use xserver-xspice-eyeos, simply navigate a browser to <http://remote-host:5800>.

The eyeos/flexVDI HTML5 client works exceptionally well and, empirically, appears to give the best overall user experience of all of the HTML5 clients covered in this book. Audio and automatic display resizing work out of the box and audio lip sync is good, though the author did experience some audio glitching on YouTube videos.

9 Virtual Desktops

When first learning about Docker, it is very common to get confused about the differences between containers and virtual machines. This chapter *deliberately* blurs some of those differences, so readers are strongly encouraged to read and understand the preceding chapters before returning here to explore the art of the possible with containers.

At the most fundamental level virtual machines provide a virtual representation, or emulation, of a physical computer. Containers, on the other hand, provide a form of Operating System virtualisation where virtualised applications share the same host kernel and physical devices.

Because containerisation is often, quite reasonably, thought of as an application abstraction it is common to read a lot of hyperbole and dogma about what containers should or should not do. One common example of this dogma is “one process per container”, though that incorrectly paraphrases [Docker best practice](#) guidance which *actually* refers to *one concern*.

To be clear, the author of this book supports and endorses much of the *reasoning* behind this, which relates to horizontal scaling, reuse, decoupling, immutability, isolation, [PID 1 handling](#) and a myriad of other good reasons. Ultimately however Docker is simply a tool and provided one understands the “safety instructions” it is perfectly reasonable to use it however one chooses to solve a problem. So, whilst the rest of this book largely follows one concern per container, this chapter illustrates the opposite end of the spectrum.

9-1 xephyr

Thus far we have focussed on running individual applications inside containers, either targeting the host's X Server or a virtual framebuffer. Because we have been focussing on *applications*, one of the things we have largely glossed-over is the importance of the [window manager](#) in determining the look and feel of a system. For individual applications the main window is managed by a window manager, but for a virtual desktop the main window **is** a window manager.

The importance of the window manager was illustrated most clearly in the [Remote Applications](#) chapter where, although our intention was to remote individual applications, we noted that many remoting systems are designed for *desktop* sharing. For those examples the lightweight window manager [jwm](#) was bundled, as running many applications without a window manager can make them behave in unexpected ways.

Although our Remote Applications chapter shows that an X11 window manager is just another X client, one subtlety is that it is only possible to directly run one window manager at a time on an X Server. For our remote applications running a window manager was not an issue because we were running a headless X Server in the container, however if we try to run a window manager directly on the host's display it will fail.

One solution to the problem of running a nested window manager is [Xephyr](#), which is a kdrive based X Server that targets a window on a host X Server as its framebuffer.

The [Dockerfile](#) is in the [Xephyr](#) directory of [docker-gui](#) under [9-virtual-desktops](#).

```
FROM ubuntu:18.04

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    xserver-xephyr && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["Xephyr"]
```

and may be built with:

```
docker build -t xephyr .
```

The Dockerfile is trivial, simply installing xserver-xephyr.

The [xephyr.sh](#) launch script is:

```
# The X11 DISPLAY number of the nested Xephyr X server.
# Accepts user input value or defaults to :1
NESTED_DISPLAY=${@:-:1}

BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-command.sh

$DOCKER_COMMAND run --rm \
    --ipc=host \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    $X11_FLAGS_RW \
    xephyr $NESTED_DISPLAY -resizeable -ac -reset -terminate 2>
/dev/null
```

Running `xephyr.sh` will launch a nested, resizeable, X Server that defaults to listening for inbound X11 connections on display :1. To change the display number of the nested X Server use the `NESTED_DISPLAY` environment variable, e.g.:

```
NESTED_DISPLAY=:10 ./xephyr.sh
```

which will launch Xephyr listening on display :10.

There are two important and quite subtle points of note in the `xephyr.sh` launch script.

The first point is the use of `--ipc=host` to allow the container to share the host's IPC, which is important for Xephyr as it uses SysV shared memory for efficiency.

The second point is the use of `X11_FLAGS_RW`, which mounts the X11 socket directory `/tmp/.X11-unix` read-write. Mounting this directory read-write is important, as for Xephyr to work correctly it must create an X11 Unix domain socket on the host for each display that it opens.

Although it is possible to launch applications that directly target the nested X Server, this example mainly serves to illustrate exactly why window managers are so important. Any application launched directly onto the nested display will appear without any window decorators and other important usability features.

The virtual desktop examples in this chapter launch Xephyr as their X Server, then open their window managers onto the nested X Server.

9-2 debian-buster-lxde

There are many possible ways to approach the problem of running a complete virtual desktop. On the more minimal end of the spectrum we have already used jwm in our remote applications. Extending that approach to launch multiple applications and bundling [supervisord](#) for PID 1 handling and process control is reasonable for simple custom desktops.

One disadvantage of such a minimal approach is that the desktop doesn't really "feel" like the desktop a user would be greeted with had they installed the OS directly or in a VM. A bigger disadvantage however is that minimal approaches often do not *behave* correctly in a way that users might expect. The issue is that modern Linux desktops usually depend on a wide range of system services like systemd, logind, D-bus, Polkit, etc. being available in order to behave correctly.

For the virtual desktops presented in this chapter we take a more complete approach, where we shall install a full distribution rather closer to one that we might install natively or in a VM. In these examples we shall install systemd then run init to start the systemd graphical.target and launch the container directly into the [display manager](#) and greeter.

For our first example we shall install Debian with the [LXDE](#) desktop environment. This will give us a very lightweight system, but still running all of the services required for a complete and fully functioning Linux desktop experience.

The [Dockerfile](#) is in the [debian-buster-lxde](#) directory of [docker-gui](#) under [9-virtual-desktops](#).

```
FROM debian:buster

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN \
    # Update base packages.
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get -fy -o Dpkg::Options:::="--force-confnew" \
        -o APT::Immediate-Configure=false \
        dist-upgrade && \
    # Add the main packages
    DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    base-files vim ntp locales apt-transport-https curl \
    lxde lxde-settings-daemon lxlock lxmusic desktop-base \
    synaptic libnss-mdns thunderbird inkscape pidgin \
    gimp gimp-help-en transmission-gtk tracker-miner-fs \
    avahi-utils avahi-discover firefox-esr-110n-en-gb \
    vlc mesa-utils thunderbird-l110n-en-gb libpam-kwallet4 \
    remmina remmina-common remmina-plugin-rdp \
    remmina-plugin-vnc remmina-plugin-nx libpam-kwallet5 \
    remmina-plugin-spice remmina-plugin-xdmcp sudo geany \
    libcanberra-pulse pulseaudio-module-bluetooth paprefs \
    pavucontrol gstreamer1.0-pulseaudio cups-pk-helper \
    pulseaudio-module-zeroconf cups system-config-printer \
    # Install Display Manager and dependencies
    lightdm slick-greeter dbus-x11 && \
    # Stop synaptic package manager being painfully slow
    rm /etc/apt/apt.conf.d/docker-gzip-indexes && \
    rm -rf /var/lib/apt/lists/* && apt-get update && \
    # Generate locales
    sed -i "s/^# *\\($LANG\\)/\\1/" /etc/locale.gen && \
    locale-gen && \
```

```

# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Configure LightDM Display Manager to use Xephyr
# instead of X. Reorganise /usr/share/xsessions to set
# LXDE as default session as slick-greeter uses
# hardcoded names to select the default session.
rm /usr/share/xsessions/lightdm-xsession.desktop && \
rm /usr/share/xsessions/openbox.desktop && \
# Need to set dpi here for LXDE or fonts are HUGE
echo '#!/bin/bash\nexport XAUTHORITY=/root/.Xauthority.docker
\nexport DISPLAY=:0\nexec Xephyr $1 -ac -dpi 48 >> /var/log/lig
htdm/x-1.log' > /usr/bin/Xephyr-lightdm-wrapper && \
chmod +x /usr/bin/Xephyr-lightdm-wrapper && \
# Debian LightDM config is in /etc/lightdm/lightdm.conf
echo '[LightDM]\nminimum-display-number=1\n[Seat:0]\nsession-
setup-script=sh -c "xdpyinfo | grep -q RANDR && exec xrandr --o
utput default --mode 1600x1200 || true"\ngreeter-hide-users=fal
se\nxserver-command=/usr/bin/Xephyr-lightdm-wrapper' > /etc/lig
htdm/lightdm.conf && \
# Ensure LightDM is set as Display Manager
rm -f /etc/systemd/system/display-manager.service && \
ln -s /lib/systemd/system/lightdm.service \
/etc/systemd/system/display-manager.service && \
echo '/usr/sbin/lightdm' > \
/etc/X11/default-display-manager && \
# Set greeter background
echo '[Greeter]\nbackground=/usr/share/desktop-base/futurepro
tototype-theme/plymouth/plymouth_background_future.png\n' > /etc/
lightdm/slick-greeter.conf && \
# Set up Keyboard mapping
echo 'XKBMODEL="pc105"\nXKBLAYOUT="gb"\nXKBVARIANT=""\nXKBOPT
IONS=""' > /etc/default/keyboard && \

```

```
# Configure console
echo "console-setup console-setup/charmap select UTF-8" | deb
conf-set-selections && \
# Fix synaptic Empty Dir::Cache::pkgcache setting not
# handled correctly which causes synaptic to barf with:
# E: Could not open file - open (2: No such file or directory
)
# E: _cache->open() failed, please report.
sed -i 's/Dir::Cache::pkgcache "";//' \
/etc/apt/apt.conf.d/docker-clean && \
# Disable getty@tty1.service to speed up desktop loading.
rm -f /etc/systemd/system/getty.target.wants/getty@tty1.servi
ce && \
# Fix issues with slow shutdown
sed -i 's/#DefaultTimeoutStopSec=90s/DefaultTimeoutStopSec=5s
/' /etc/systemd/system.conf && \
# Fix Polkit issues caused by container login being
# considered to be an "inactive" session.
chmod 755 /etc/polkit-1/localauthority && \
# Shutdown & Restart
# Note that auth_admin_keep may be better than yes
# here, but there seems to be an issue with the
# authentication dialog appearing.
echo "[Shutdown & Restart]\nIdentity=unix-user:*\nAction=org.
freedesktop.login1.power-off;org.freedesktop.login1.power-off-m
ultiple-sessions;org.freedesktop.login1.reboot;org.freedesktop.
login1.reboot-multiple-sessions\nResultAny=yes\nResultInactive=
yes\nResultActive=yes\n" > /etc/polkit-1/localauthority/50-loc
al.d/10-shutdown.pkla
```

The image may be built with:

```
docker build -t debian-lxde:buster .
```

The rather large set of packages installed by this Dockerfile is intended to illustrate a reasonably feature-rich desktop. The packages whose names start with lx are required for the LXDE desktop and, unusually, we also install the lightdm display manager package and the related packages slick-greeter and dbus-x11. The remainder of the packages are, however, somewhat arbitrary and readers are encouraged to modify the Dockerfile to suit their own preferences.

After installing the packages we now have to configure the system to behave more like a regular desktop distribution.

This desktop uses the synaptic graphical package manager, however with the default Docker installation synaptic indexing was *painfully* slow. The problem is caused by Docker based distributions compressing the indexes and the following line in the Dockerfile resolves the issue:

```
rm /etc/apt/apt.conf.d/docker-gzip-indexes
```

After fixing the synaptic indexing issue we next set the locale and timezone, which follows exactly the same process employed in many other Debian images throughout this book.

The next section of the Dockerfile is arguably one of the most interesting of the whole book. Here we shall configure the LightDM display manager to use Xephyr instead of the default X Server, so that when init is run the container will launch a graphical login screen in exactly the same way as we'd expect to see with any regular Linux desktop distribution.

Although LightDM is very configurable and the default session really *should* be configurable via “user-session=” in the

lightdm configuration, that doesn’t seem to be the case. The issue appears to be that slick-greeter hard codes the defaults from a fixed list, which may be seen in the get_default_session method in <https://github.com/linuxmint/slick-greeter/blob/master/src/slick-greeter.vala>.

The greeter appears to search /usr/share/xsessions/, but then chooses the first match **from the hardcoded list**. We workaround this by reorganising /usr/share/xsessions/.

```
rm /usr/share/xsessions/lightdm-xsession.desktop
rm /usr/share/xsessions/openbox.desktop
```

We next have to configure LightDM to use Xephyr.

The first part of this configuration is to create a wrapper script in /usr/bin/Xephyr-lightdm-wrapper to launch Xephyr.

The main purpose of this script is to ensure that Xephyr targets the host’s display :0 and uses the correct authentication for that display, however it also sets the dpi otherwise LXDE will use huge fonts. The exec ensures that the display manager receives the correct signal (SIGUSR1) from Xephyr, which is required for correct LightDM startup.

```
#!/bin/bash
export XAUTHORITY=/root/.Xauthority.docker
export DISPLAY=:0
exec Xephyr $1 -ac -dpi 48 >> /var/log/lightdm/x-1.log
```

The second part of the LightDM configuration is to configure /etc/lightdm/lightdm.conf to use our Xephyr-lightdm-wrapper as its X Server command.

```
[LightDM]
minimum-display-number=1
[Seat:*]
session-setup-script=sh -c "xdpyinfo | grep -q RANDR && exec xr
andr --output default --mode 1600x1200 || true"
greeter-hide-users=false
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
```

Note that this configuration sets the minimum-display-number to 1, because display :0 is the host's X Server. This configuration also uses xrandr to set the initial display geometry, checking first if the RANDR extension is supported .

After we have configured LightDM and the greeter, we set LightDM as the default systemd display manager service.

```
rm -f /etc/systemd/system/display-manager.service
ln -s /lib/systemd/system/lightdm.service \
/etc/systemd/system/display-manager.service
echo '/usr/sbin/lightdm' > \
/etc/X11/default-display-manager
```

We also set the greeter background in /etc/lightdm/slick-greeter.conf to give a Debian theme.

```
[Greeter]
background=/usr/share/desktop-base/futureprototype-theme/plymou
th/plymouth_background_future.png
```

After configuring LightDM, we next configure the console.

```
echo "console-setup console-setup/charmap select UTF-8" | debconf-set-selections
```

We then fix another synaptic issue described in <https://bugs.launchpad.net/ubuntu/+source/synaptic/+bug/1243615>, where we see an error:

```
E: Could not open file - open (2: No such file or directory)
E: _cache->open() failed, please report.
```

The fix is as follows, which simply removes the setting causing the error from /etc/apt/apt.conf.d/docker-clean.

```
sed -i 's/Dir::Cache::pkgcache "";//' \
/etc/apt/apt.conf.d/docker-clean
```

When this application was being developed the desktop didn't load as quickly as expected, which was resolved by disabling the systemd getty@tty1.service.

```
rm -f /etc/systemd/system/getty.target.wants/getty@tty1.service
```

Similarly, system shutdown was slower than expected due to systemd timing out on some non-existent services. This was resolved by reducing the default timeout period.

```
sed -i 's/#DefaultTimeoutStopSec=90s/DefaultTimeoutStopSec=5s/' \
/etc/systemd/system.conf
```

One final issue to be overcome was due to Polkit considering the container login to be an “inactive” session, which is one that doesn’t give ownership of the console that X runs on. By default “inactive” or remote sessions can’t perform certain actions that require elevated privileges, like shutdown.

As we actually **want** our container to behave like a real desktop session, we first change the permissions of /etc/polkit-1/localauthority.

```
chmod 755 /etc/polkit-1/localauthority
```

We then we create /etc/polkit-1/localauthority/50-local.d/10-shutdown.pkla to give the required privileges.

```
[Shutdown & Restart]
Identity=unix-user:*
Action=org.freedesktop.login1.power-off;org.freedesktop.login1.
power-off-multiple-sessions;org.freedesktop.login1.reboot;org.f
reedesktop.login1.reboot-multiple-sessions
ResultAny=yes
ResultInactive=yes
ResultActive=yes
```

The [debian.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh

IMAGE=debian-1xde:buster
CONTAINER=debian
```

```

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $IMAGE /sbin/init

# Wait for container to run before cp credentials
sleep 0.25

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

The launch script first sets environment variables for the image and container names, then creates a “credentials bundle” etc.tar.gz. This bundle is just an archive containing /etc/passwd, /etc/shadow, and /etc/group, created by launching a container that runs `adduser`, using a process very similar to our earlier [`xrdp-storepassword.sh`](#)

The `docker run` for this container is rather more involved than many we have developed and requires some additional privileges to work correctly.

Adding `--device=/dev/tty0` is odd, as the device isn’t *actually* used, but having it visible inside the container “tricks” systemd-logind into cleanly starting the user session. Without it we get “Failed to create session: Seat has no VTs but VT number not 0” and XDG_RUNTIME_DIR isn’t set.

Using `--ipc=host` allows the container to share the host’s IPC which, as mentioned previously, is important for Xephyr as it uses SysV shared memory.

As we wish to run systemd we require some additional privileges to be set as follows:

```
--security-opt apparmor=unconfined \
--cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
-v /sys/fs/cgroup:/sys/fs/cgroup \
```

Setting apparmor=unconfined is done for simplicity and is required because we want to be able to call mount from inside the container. A better approach would be to create a custom AppArmor profile explicitly enabling mount.

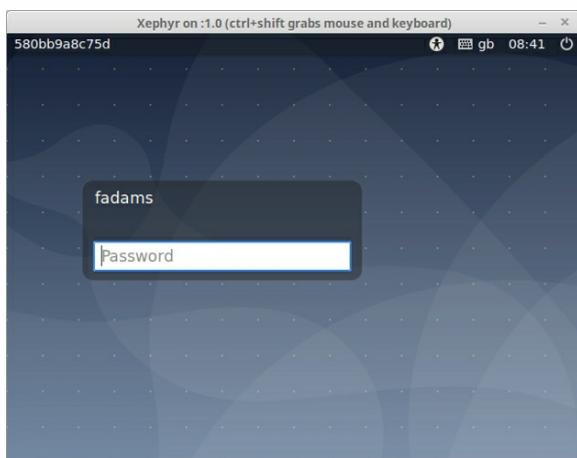
The SYS_ADMIN capability is added to let systemd run cleanly, though SYS_ADMIN is overloaded and it *may* be possible to use a combination of less privileged capabilities.

The SYS_BOOT capability lets the container cleanly shut down. As containers are immutable this isn't strictly necessary, but from a user experience perspective most users would expect the desktop shutdown menus to work correctly.

Access to cgroup controllers by bind-mounting /sys/fs/cgroup is also required for systemd to run cleanly.

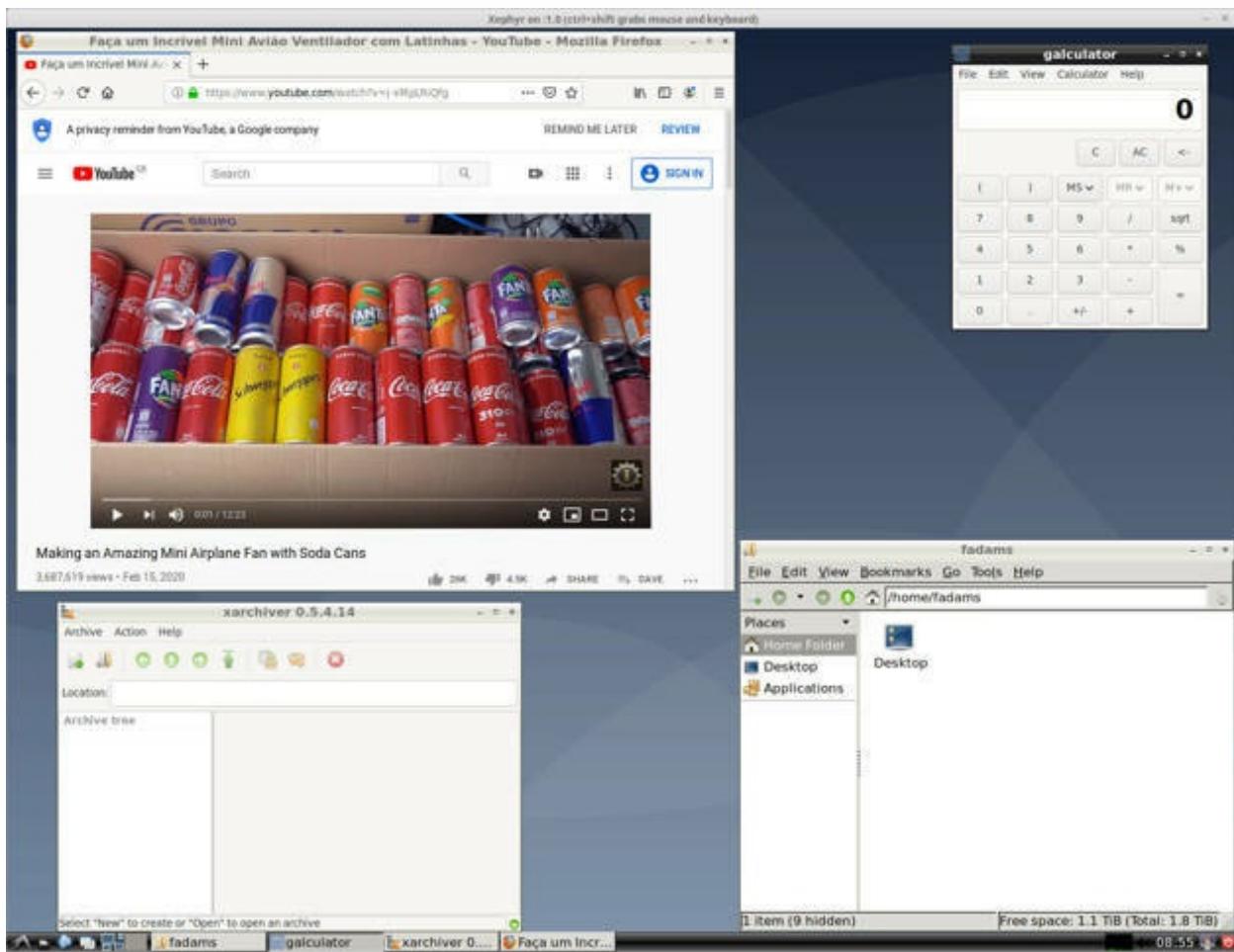
The last few steps provide access to the host's X Server, though in this case we bind-mount the .Xauthority in the container's root directory as the display manager is started by root before dropping privileges after login. Note too that we don't mount the X11 socket *directory* only the host's socket, because the nested X Server runs inside the container.

Finally we call /sbin/init, which will start all of the systemd services including the full graphical login.



This container is designed to look and feel very similar to running a desktop in a VM. It's important to remember though that it is not a VM and the container is immutable.

One difference relates to login credentials. After the container has started we copy the credentials bundle to the container. We do this in preference to bind-mounting the credentials so adduser and related graphical tools will work, however any users added will not survive container restart. Similarly apt and the graphical synaptic tool both work correctly, but packages added to the container won't survive a restart.



9-3 ubuntu-18-04-gnome

LXDE is an excellent choice of desktop where CPU and memory resources are limited, however if that is not the case it is possible to run a fully functioning Ubuntu GNOME desktop in a container.

The [Dockerfile](#) is in the [ubuntu-18.04-gnome](#) directory of [docker-gui](#) under [9-virtual-desktops](#).

```
FROM ubuntu:18.04

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN \
    # Enable partner repository (needed for codecs)
    sed -i 's/# deb http:\/\/archive.canonical.com\/ubuntu bionic
partner/deb http:\/\/archive.canonical.com\/ubuntu bionic partn
er/' /etc/apt/sources.list && \
    # Remove "This system has been minimized" warning.
    rm -f /etc/update-motd.d/60-unminimize && \
    rm -f /etc/update-motd.d/98-fsck-at-reboot && \
    # Update base packages.
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get -fy -o Dpkg::Options::="--force-confnew" \
        -o APT::Immediate-Configure=false \
        dist-upgrade && \
    # Add the main packages
    DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    base-files curl ntp add-apt-key aptdaemon \
    apt-transport-https language-pack-en gnome \
    ubuntu-minimal ubuntu-standard ubuntu-desktop \
    ubuntu-system-service ubuntu-restricted-extras \
```

```
gsettings-ubuntu-schemas gnome-system-log \
gnome-shell-extension-ubuntu-dock libpam-kwallet4 \
gnome-shell-extension-appindicator libpam-kwallet5 \
gnome-control-center-faces gnome-software-plugin-snap \
# Add packages to be similar to Ubuntu iso installation.
thunderbird-locale-en thunderbird-locale-en-us \
firefox-locale-en tracker-miner-fs avahi-utils vlc \
transmission-gtk synaptic inkscape gimp pidgin remmina \
remmina-common remmina-plugin-rdp remmina-plugin-vnc \
remmina-plugin-nx remmina-plugin-spice \
remmina-plugin-xdmcp mesa-utils libcanberra-pulse \
pulseaudio-module-bluetooth paprefs pavucontrol \
gstreamer1.0-pulseaudio pulseaudio-module-zeroconf \
cups system-config-printer-gnome \
# Install Display Manager and dependencies
lightdm slick-greeter dbus-x11 && \
# Default libgl1-mesa-dri causes "black window" issues
# when software rendering. Use ppa to upgrade version.
add-apt-repository -y ppa:oibaf/graphics-drivers && \
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y libgl1-mesa-dri && \
# Stop synaptic package manager being painfully slow
rm /etc/apt/apt.conf.d/docker-gzip-indexes && \
rm -rf /var/lib/apt/lists/* && apt-get update && \
# Generate locales
echo LANG=$LANG > /etc/default/locale && \
update-locale LANG=$LANG && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Configure LightDM Display Manager to use
# Xephyr instead of X
rm /usr/share/lightdm/lightdm.conf.d/50-ubuntu.conf && \
rm /usr/share/wayland-sessions/*.desktop && \
```

```
# Reorganise /usr/share/xsessions to set Ubuntu as
# default session as slick-greeter uses hardcoded names
# to select the default session.
rm /usr/share/xsessions/gnome.desktop && \
mv /usr/share/xsessions/ubuntu.desktop \
/usr/share/xsessions/ubuntu-xorg.desktop && \
mv /usr/share/xsessions/gnome-xorg.desktop \
/usr/share/xsessions/gnome-classic.desktop && \
echo '#!/bin/bash\nexport XAUTHORITY=/root/.Xauthority.docker
\nexport DISPLAY=:0\nexec Xephyr $1 -ac >> /var/log/lightdm/x-1
.log' > /usr/bin/Xephyr-lightdm-wrapper && \
chmod +x /usr/bin/Xephyr-lightdm-wrapper && \
echo '[LightDM]\nminimum-display-number=1\n[Seat: *]\nuser-ses
sion=ubuntu-xorg\nserver-command=/usr/bin/Xephyr-lightdm-wrapp
er' > /etc/lightdm/lightdm.conf.d/70-ubuntu.conf && \
echo '[Greeter]\nbackground=/usr/share/backgrounds/warty-fina
l-ubuntu.png\n' > /etc/lightdm/slick-greeter.conf && \
# Configure console
echo "console-setup console-setup/charmap select UTF-8" | deb
conf-set-selections && \
# Fix synaptic Empty Dir::Cache::pkgcache setting not
# handled correctly https://bugs.launchpad.net/ubuntu/+source
/synaptic/+bug/1243615
# which causes synaptic to barf with: E: Could not
# open file - open (2: No such file or directory)
# E: _cache->open() failed, please report.
sed -i 's/Dir::Cache::pkgcache ""; //'\ \
/etc/apt/apt.conf.d/docker-clean && \
# Disable getty@tty1.service to speed up desktop loading.
rm -f /etc/systemd/system/getty.target.wants/getty@tty1.servi
ce && \
```

```
# Fix issues with slow shutdown
sed -i 's/#DefaultTimeoutStopSec=90s/DefaultTimeoutStopSec=5s
/' /etc/systemd/system.conf && \
# Fix Polkit issues caused by container login being
# considered to be an "inactive" session.
chmod 755 /etc/polkit-1/localauthority && \
# Date & Time
echo "[Date & Time]\nIdentity=unix-user:*\nAction=org.gnome.c
ontrolcenter.datetime.configure\nResultAny=auth_admin_keep\nRes
ultInactive=auth_admin_keep\nResultActive=auth_admin_keep\n" >
/etc/polkit-1/localauthority/50-local.d/10-datetime.mechanism.pk
la && \
# User Accounts
echo "[Manage user accounts]\nIdentity=unix-user:*\nAction=or
g.gnome.controlcenter.user-accounts.administration\nResultAny=a
uth_admin_keep\nResultInactive=auth_admin_keep\nResultActive=au
th_admin_keep\n" > /etc/polkit-1/localauthority/50-local.d/10-u
ser-accounts.pkla && \
# Gnome System Log
echo "[Gnome System Log]\nIdentity=unix-user:*\nAction=org.de
bian.pkexec.gnome-system-log.run\nResultAny=auth_admin_keep\nRe
sultInactive=auth_admin_keep\nResultActive=auth_admin_keep\n" >
/etc/polkit-1/localauthority/50-local.d/10-system-log.pkla && \
# System Color Manager
echo "[System Color Manager]\nIdentity=unix-user:*\nAction=or
g.freedesktop.color-manager.create-device;org.freedesktop.color
-manager.create-profile;org.freedesktop.color-manager.delete-de
vice;org.freedesktop.color-manager.delete-profile;org.freedesk
top.color-manager.modify-device;org.freedesktop.color-manager.mo
dify-profile;org.freedesktop.color-manager.device-inhibit;org.f
reedesktop.color-manager.sensor-lock\nResultAny=yes\nResultInac
tive=yes\nResultActive=yes\n" > /etc/polkit-1/localauthority/50
-local.d/10-color.pkla && \
```

```
# Shutdown & Restart
# Note that auth_admin_keep may be better than yes
# here, but there seems to be an issue with the
# authentication dialog appearing.
echo "[Shutdown & Restart]\nIdentity=unix-user:*\nAction=org.
freedesktop.login1.power-off;org.freedesktop.login1.power-off-m
ultiple-sessions;org.freedesktop.login1.reboot;org.freedesktop.
login1.reboot-multiple-sessions\nResultAny=yes\nResultInactive=
yes\nResultActive=yes\n" > /etc/polkit-1/localauthority/50-loc
1.d/10-shutdown.pkla
```

The image may be built with:

```
docker build -t ubuntu-gnome:18.04 .
```

This Dockerfile has many similarities with the Debian LXDE one, so we shall only concentrate on the differences here.

The first thing we do is to enable the partner repository by uncommenting the line in /etc/apt/sources.list. This repository is required for codecs and other proprietary and closed-source software that is disabled by default.

We next remove the “This system has been minimized” and “fsck at reboot” warnings that are set in the MOTD of Ubuntu Docker images.

Again we install a rather large set of packages to give a close approximation of a standard desktop Ubuntu distribution, including a range of gnome packages and again installing the lightdm display manager package and the related packages.

With Ubuntu 18.04 the author experienced a number of odd “black window” rendering issues with the default mesa software renderer library swrast_dri.so. It is possible to downgrade the version by using libgl1-mesa-dri=18.0.0~rc5-1ubuntu1 in the apt install, but instead we upgrade using the oibaf PPA archive.

```
add-apt-repository -y ppa:oibaf/graphics-drivers  
apt-get update && DEBIAN_FRONTEND=noninteractive \  
apt-get install -y libgl1-mesa-dri
```

When we generate the locales we note that the procedure for this is subtly different to the one used for Debian, as first described in the [Office](#) section.

```
echo LANG=$LANG > /etc/default/locale  
update-locale LANG=$LANG
```

Although GDM is arguably more idiomatic for GNOME, we again use LightDM as the display manager. GDM has fewer options for configuring the X Server started by the display manager and we need the display manager to launch Xephyr instead of Xorg for this application.

Configuring LightDM on Ubuntu is similar to the procedure on Debian, though there are a number of differences.

Because slick-greeter searches /usr/share/xsessions/ and chooses the first match from a hard-coded list we reorganise to remove unwanted sessions and ensure that the session we want is chosen as the default.

```
rm /usr/share/lightdm/lightdm.conf.d/50-ubuntu.conf
rm /usr/share/wayland-sessions/*.desktop
rm /usr/share/xsessions/gnome.desktop
mv /usr/share/xsessions/ubuntu.desktop \
  /usr/share/xsessions/ubuntu-xorg.desktop
mv /usr/share/xsessions/gnome-xorg.desktop \
  /usr/share/xsessions/gnome-classic.desktop
```

To configure LightDM to use Xephyr we again create a wrapper script in /usr/bin/Xephyr-lightdm-wrapper. This is very similar to the one we created for Debian LXDE, but for Ubuntu GNOME we don't need to set the dpi.

```
#!/bin/bash
export XAUTHORITY=/root/.Xauthority.docker
export DISPLAY=:0
exec Xephyr $1 -ac >> /var/log/lightdm/x-1.log
```

The LightDM configuration is similar to our Debian one, though for Ubuntu the configuration file is in /etc/lightdm/lightdm.conf.d/70-ubuntu.conf and we don't need to call xrandr on session-setup.

```
[LightDM]
minimum-display-number=1
[Seat: *]
user-session=ubuntu-xorg
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
```

We also set the greeter background in /etc/lightdm/slick-greeter.conf to give an Ubuntu theme.

```
[Greeter]
background=/usr/share/backgrounds/warty-final-ubuntu.png
```

After configuring LightDM and slick-greeter we configure the console, fix the synaptic “_cache->open() failed” bug, disable the getty@tty1.service and resolve the slow system shutdown in exactly the same way as we did for the Debian desktop.

Finally, as we did for Debian, we create a number of entries in /etc/polkit-1/localauthority/50-local.d/ to give elevated privileges to “inactive” sessions.

10-datetime-mechanism.pkla

```
[Date & Time]
Identity=unix-user:*
Action=org.gnome.controlcenter.datetime.configure
ResultAny=auth_admin_keep
ResultInactive=auth_admin_keep
ResultActive=auth_admin_keep
```

10-user-accounts.pkla

```
[Manage user accounts]
Identity=unix-user:*
Action=org.gnome.controlcenter.user-accounts.administration
ResultAny=auth_admin_keep
ResultInactive=auth_admin_keep
ResultActive=auth_admin_keep
```

10-system-log.pkla

```
[Gnome System Log]
Identity=unix-user:*
Action=org.debian.pkexec.gnome-system-log.run
ResultAny=auth_admin_keep
ResultInactive=auth_admin_keep
ResultActive=auth_admin_keep
```

10-color.pkla

```
[System Color Manager]
Identity=unix-user:*
Action=org.freedesktop.color-manager.create-device;org.freedesktop.color-manager.create-profile;org.freedesktop.color-manager.delete-device;org.freedesktop.color-manager.delete-profile;org.freedesktop.color-manager.modify-device;org.freedesktop.color-manager.modify-profile;org.freedesktop.color-manager.device-inhibit;org.freedesktop.color-manager.sensor-lock
ResultAny=yes
ResultInactive=yes
ResultActive=yes
```

10-shutdown.pkla

```
[Shutdown & Restart]
Identity=unix-user:*
Action=org.freedesktop.login1.power-off;org.freedesktop.login1.power-off-multiple-sessions;org.freedesktop.login1.reboot;org.freedesktop.login1.reboot-multiple-sessions
ResultAny=yes
ResultInactive=yes
ResultActive=yes
```

The [ubuntu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh
IMAGE=ubuntu-gnome:18.04
CONTAINER=ubuntu
```

```

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

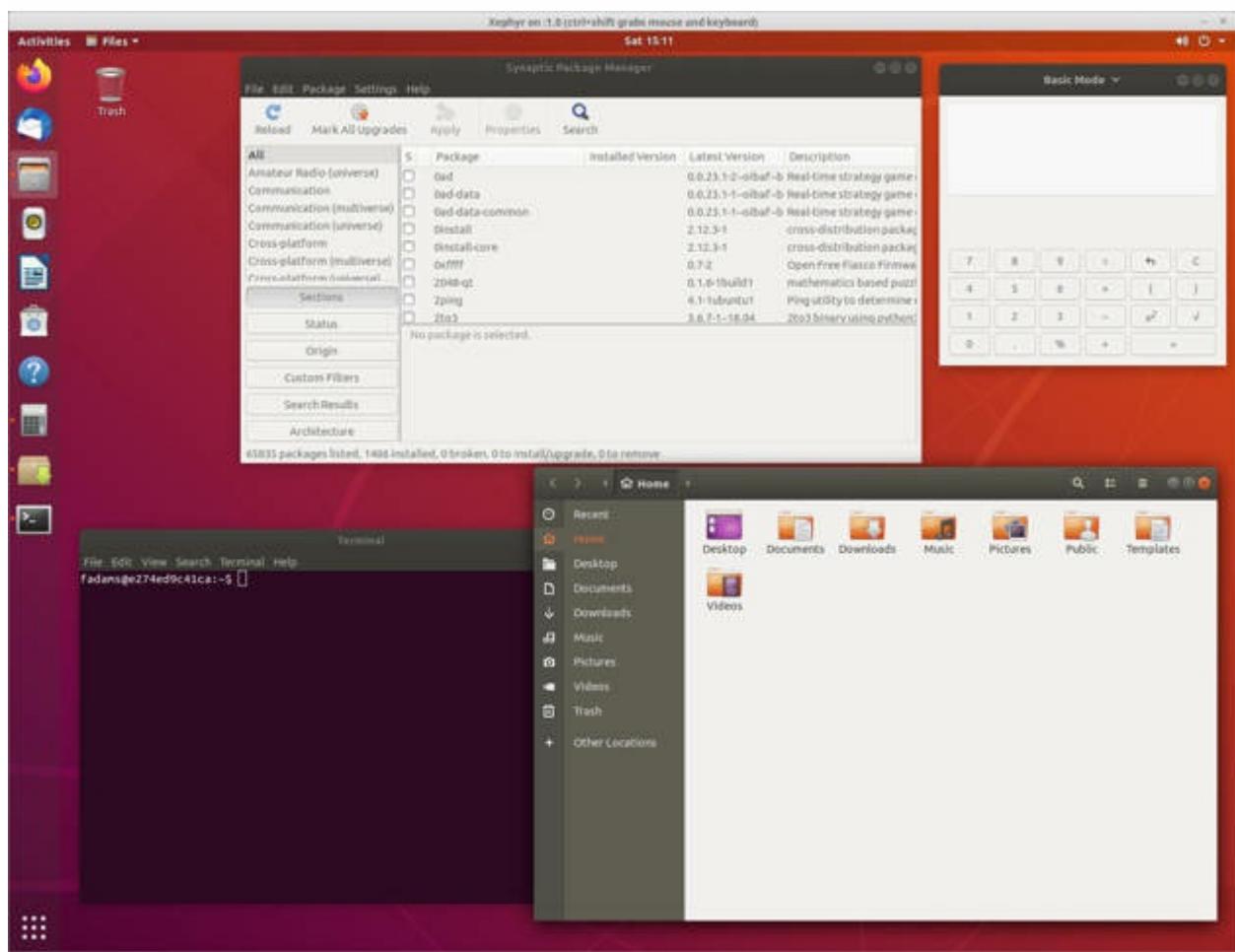
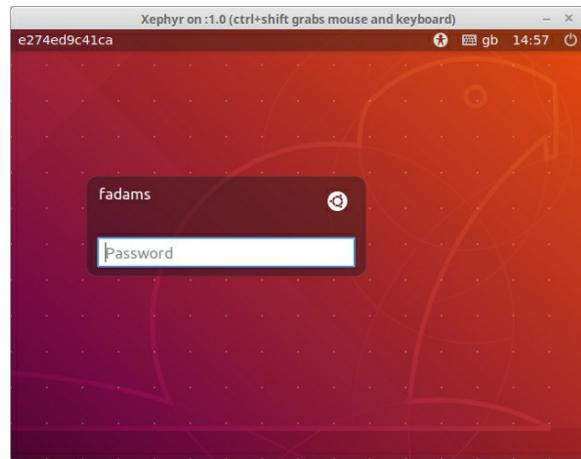
# Create home directory
mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

This script is identical to the `debian.sh` launch script except for the image and container names, so the explanation of the script given in the [debian-buster-lxde](#) section holds here too.



9-4 linuxmint-19-3-cinnamon

[Linux Mint](#), which is derived from Ubuntu, is one of the most popular desktop Linux distributions. The [Dockerfile](#) is in the [linuxmint-19.3-cinnamon](#) directory of [docker-gui](#) under [9-virtual-desktops](#).

```
FROM ubuntu:18.04

ENV LANG=en_GB.UTF-8
ENV TZ=Europe/London

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends gnupg && \
    # Enable partner repository (for mint-meta-codecs)
    sed -i 's/# deb http:\/\/archive.canonical.com\/ubuntu bionic \
partner/deb http:\/\/archive.canonical.com\/ubuntu bionic partn
er/' /etc/apt/sources.list && \
    # Remove "This system has been minimized" warning.
    rm -f /etc/update-motd.d/60-unminimize && \
    # Set up the repositories for Linux Mint 19.3 "Tricia"
    echo "deb http://packages.linuxmint.com/ tricia main upstream
import backport\n$(cat /etc/apt/sources.list)" > /etc/apt/sou
rces.list.d/official-package-repositories.list && \
    rm /etc/apt/sources.list && \
    LINUX_MINT_KEY=$(apt update 2>&1 | \
        grep -o '[0-9A-Z]\{16\}$' | xargs) && \
    apt-key adv --recv-keys --keyserver \
    hkp://keyserver.ubuntu.com:80 ${LINUX_MINT_KEY} && \
    apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get -fy -o Dpkg::Options::="--force-confnew" \
        -o APT::Immediate-Configure=false \
        dist-upgrade && \
```

```
DEBIAN_FRONTEND=noninteractive \
apt-get install -y --allow-unauthenticated \
    linuxmint-keyring && \
unset LINUX_MINT_KEY && \
# Add the main Mint flavoured packages
DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends \
base-files ntp apt-utils add-apt-key aptdaemon \
apt-transport-https locales tzdata language-pack-en \
language-pack-gnome-en mintlocale mint-meta-cinnamon \
mintupload mint-common mint-meta-core mint-themes \
mintdesktop mintmenu mintstick mintsystem mintwelcome \
xserver-xephyr wamerican cinnamon-core cinnamon-doc \
cinnamon-desktop-environment mint-info-cinnamon \
mintnanny mintreport mint-meta-codecs wbritish \
# Add packages to be similar to Mint iso installation.
gnome-terminal keyboard-configuration pidgin xed \
gimp gimp-help-en hexchat firefox firefox-locale-en \
libreoffice openoffice.org-hyphenation thunderbird \
thunderbird-locale-en thunderbird-gnome-support \
gucharmap remmina remmina-common remmina-plugin-rdp \
remmina-plugin-vnc remmina-plugin-nx gvfs-backends \
remmina-plugin-spice remmina-plugin-xdmcp drawing \
transmission-gtk pix rhythmbox gnome-calculator \
gnome-screenshot xreader xviewer xplayer gnote \
simple-scan inkscape vlc vlc-data baobab blueberry \
gnome-power-manager gufw dmz-cursor-theme vino \
gnome-system-log gnome-system-monitor libglu1-mesa \
gnome-disk-utility pulseaudio libpulsedsp paprefs \
pulseaudio-utils libcanberra-pulse pavucontrol \
pulseaudio-module-bluetooth gstreamer1.0-pulseaudio \
pulseaudio-module-zeroconf avahi-utils libnss-mdns \
mesa-utils cheese cups system-config-printer-gnome \
colord system-config-printer-udev gnome-keyring \
libpam-gnome-keyring libpam-kwallet4 libpam-kwallet5 \
# Install Display Manager and dependencies
lightdm slick-greeter dbus-x11 && \
```

```
# Default libgl1-mesa-dri causes "black window" issues
# when software rendering. Use ppa to upgrade version.
add-apt-repository -y ppa:oibaf/graphics-drivers && \
apt-get update && DEBIAN_FRONTEND=noninteractive \
apt-get install -y libgl1-mesa-dri && \
# Stop synaptic package manager being painfully slow
rm /etc/apt/apt.conf.d/docker-gzip-indexes && \
rm -rf /var/lib/apt/lists/* && apt-get update && \
# Generate locales
echo LANG=$LANG > /etc/default/locale && \
update-locale LANG=$LANG && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
DEBIAN_FRONTEND=noninteractive \
dpkg-reconfigure tzdata && \
# Configure LightDM Display Manager to use
# Xephyr instead of X
echo '#!/bin/bash\nexport XAUTHORITY=/root/.Xauthority.docker
\nexport DISPLAY=:0\nexec Xephyr $1 -ac >> /var/log/lightdm/x-1
.log' > /usr/bin/Xephyr-lightdm-wrapper && \
chmod +x /usr/bin/Xephyr-lightdm-wrapper && \
echo '[LightDM]\nminimum-display-number=1\n[Seat:]*]\nxserver-
command=/usr/bin/Xephyr-lightdm-wrapper' > /etc/lightdm/lightdm
.conf.d/70-linu xmint.conf && \
# Change nemo icons from standard to small
# https://unix.stackexchange.com/questions/250266/rhel-7-gnom
e-shell-decrease-desktop-icon-size
sed -i '/default-zoom-level/{n; s/standard/small/}' \
/usr/share/glib-2.0/schemas/org.nemo.gschema.xml && \
glib-compile-schemas /usr/share/glib-2.0/schemas/ > \
/dev/null 2>&1 && \
# Configure console
echo "console-setup console-setup/charmap select UTF-8" | deb
conf-set-selections && \
```

```
# Fix mintupdate "APT-cache damaged" error
cp /usr/share/linuxmint/mintsystem/apt/official-package-repositories.pref \
    /etc/apt/preferences.d/official-package-repositories.pref
&& \
# Fix synaptic Empty Dir::Cache::pkgcache setting not
# handled correctly https://bugs.launchpad.net/ubuntu/+source
/synaptic/+bug/1243615
# which causes synaptic to barf with: E: Could not
# open file - open (2: No such file or directory)
# E: _cache->open() failed, please report.
sed -i 's/Dir::Cache::pkgcache ""; //' \
    /etc/apt/apt.conf.d/docker-clean && \
# Disable getty@tty1.service to speed up desktop loading.
rm -f /etc/systemd/system/getty.target.wants/getty@tty1.servi
ce && \
# Fix issues with slow shutdown
sed -i 's/#DefaultTimeoutStopSec=90s/DefaultTimeoutStopSec=5s
/' /etc/systemd/system.conf && \
# Fix Polkit issues caused by container login being
# considered to be an "inactive" session.
chmod 755 /etc/polkit-1/localauthority && \
# Mint Software Sources
echo "[Mint Software Sources]\nIdentity=unix-user:*\nAction=c
om.linuxmint.mintsources\nResultAny=auth_self_keep\nResultInact
ive=auth_self_keep\nResultActive=auth_self_keep\n" > /etc/polki
t-1/localauthority/50-local.d/10-mintsources.pkla && \
# Date & Time
echo "[Date & Time]\nIdentity=unix-user:*\nAction=org.cinnamo
n.settingsdaemon.datetimemechanism.configure\nResultAny=auth_ad
min_keep\nResultInactive=auth_admin_keep\nResultActive=auth_adm
in_keep\n" > /etc/polkit-1/localauthority/50-local.d/10-datetim
emechanism.pkla && \
```

```
# Gnome System Log
echo "[Gnome System Log]\nIdentity=unix-user:*\nAction=org.debian.pkexec.gnome-system-log.run\nResultAny=auth_admin_keep\nResultInactive=auth_admin_keep\nResultActive=auth_admin_keep\n" > /etc/polkit-1/localauthority/50-local.d/10-system-log.pkla && \
# Shutdown & Restart
# Note that auth_admin_keep may be better than yes
# here, but there seems to be an issue with the
# authentication dialog appearing.
echo "[Shutdown & Restart]\nIdentity=unix-user:*\nAction=org.freedesktop.login1.power-off;org.freedesktop.login1.power-off-multiple-sessions;org.freedesktop.login1.reboot;org.freedesktop.login1.reboot-multiple-sessions\nResultAny=yes\nResultInactive=yes\nResultActive=yes\n" > /etc/polkit-1/localauthority/50-local.d/10-shutdown.pkla
```

The image may be built with:

```
docker build -t linuxmint-cinnamon:19.3 .
```

The first part of this Dockerfile enables the partner repository by uncommenting the line in /etc/apt/sources.list in the same way as we did for Ubuntu. This repository is required for mint-meta-codecs and other proprietary and closed-source software that is disabled by default.

We next remove a “This system has been minimized” warning that is set in the MOTD of Ubuntu Docker images.

The next part of the Dockerfile adds the repositories for Linux Mint 19.3 “Tricia” from <http://packages.linuxmint.com> and removes the original /etc/apt/sources.list.

```
echo "deb http://packages.linuxmint.com/ tricia main upstream import backport\n$(cat /etc/apt/sources.list)" > /etc/apt/sources.list.d/official-package-repositories.list  
rm /etc/apt/sources.list
```

After adding the Linux Mint repositories we next add the key, run apt update, then install linuxmint-keyring.

```
LINUX_MINT_KEY=$(apt update 2>&1 | \  
    grep -o '[0-9A-Z]\{16\}$' | xargs)  
apt-key adv --recv-keys --keyserver \  
    hkp://keyserver.ubuntu.com:80 ${LINUX_MINT_KEY}  
apt-get update && DEBIAN_FRONTEND=noninteractive \  
apt-get -fy -o Dpkg::Options::="--force-confnew" \  
        -o APT::Immediate-Configure=false \  
        dist-upgrade  
DEBIAN_FRONTEND=noninteractive \  
apt-get install -y --allow-unauthenticated \  
    linuxmint-keyring  
unset LINUX_MINT_KEY
```

From now on apt should favour pulling packages from the Mint repository in preference to the vanilla Ubuntu repositories.

The next step in the Dockerfile is to install the large range of packages that aims to make the container behave in a similar way to a Linux Mint iso installation.

For this example we install the [Cinnamon](#) desktop environment, which is the default for Linux Mint, and the lightdm display manager package and related packages.

After installing the main packages we follow the same procedure as we did with Ubuntu to upgrade libgl1-mesa-dri from the oibaf PPA, as again the author experienced a number of odd “black window” rendering issues with the default mesa software renderer library.

Next, our Linux Mint Dockerfile uses the same procedures to fix synaptic performance, generate locales and set up timezone as we did for Ubuntu.

Because LightDM is the default Linux Mint display manager the configuration procedure is simpler than for Debian or Ubuntu. We simply need to create the wrapper script in /usr/bin/Xephyr-lightdm-wrapper, which is the same as we used for Ubuntu.

```
#!/bin/bash
export XAUTHORITY=/root/.Xauthority.docker
export DISPLAY=:0
exec Xephyr $1 -ac >> /var/log/lightdm/x-1.log
```

then write the configuration that sets the X Server to Xephyr to /etc/lightdm/lightdm.conf.d/70-linuxmint.conf.

```
[LightDM]
minimum-display-number=1
[Seat:*]
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
```

Icons for the nemo file manager default to “standard” size, so we change those to small following the procedure described in <https://unix.stackexchange.com/questions/250266/rhel-7-gnome-shell-decrease-desktop-icon-size>.

```
sed -i '/default-zoom-level/{n; s/standard/small/}' \
/usr/share/glib-2.0/schemas/org.nemo.gschema.xml
glib-compile-schemas /usr/share/glib-2.0/schemas/ > \
/dev/null 2>&1
```

As with Debian and Ubuntu we configure the console, fix the synaptic “_cache->open() failed” bug, disable the getty@tty1.service and resolve the slow system shutdown.

For Mint, we also fix an “APT-cache damaged” error that we observe with the mintupdate application, described in <https://bugs.launchpad.net/linuxmint/+bug/1530809>.

```
cp /usr/share/linuxmint/mintsystem/apt/official-package-repositories.pref \
/etc/apt/preferences.d/official-package-repositories.pref
```

Finally, we again create entries in /etc/polkit-1/localauthority/50-local.d/ to elevate “inactive” session privileges.

10-mintsources.pkla

```
[Mint Software Sources]
Identity=unix-user:*
Action=com.linuxmint.mintsources
ResultAny=auth_self_keep
ResultInactive=auth_self_keep
ResultActive=auth_self_keep
```

10-datetime-mechanism.pkla

```
[Date & Time]
Identity=unix-user:*
Action=org.cinnamon.settingsdaemon.datetime-mechanism.configure
ResultAny=auth_admin_keep
ResultInactive=auth_admin_keep
ResultActive=auth_admin_keep
```

The 10-system-log.pkla and 10-shutdown.pkla entries are same as the ones we created for Ubuntu.

The [mint.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh

IMAGE=linuxmint-cinnamon:19.3
CONTAINER=mint

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)
```

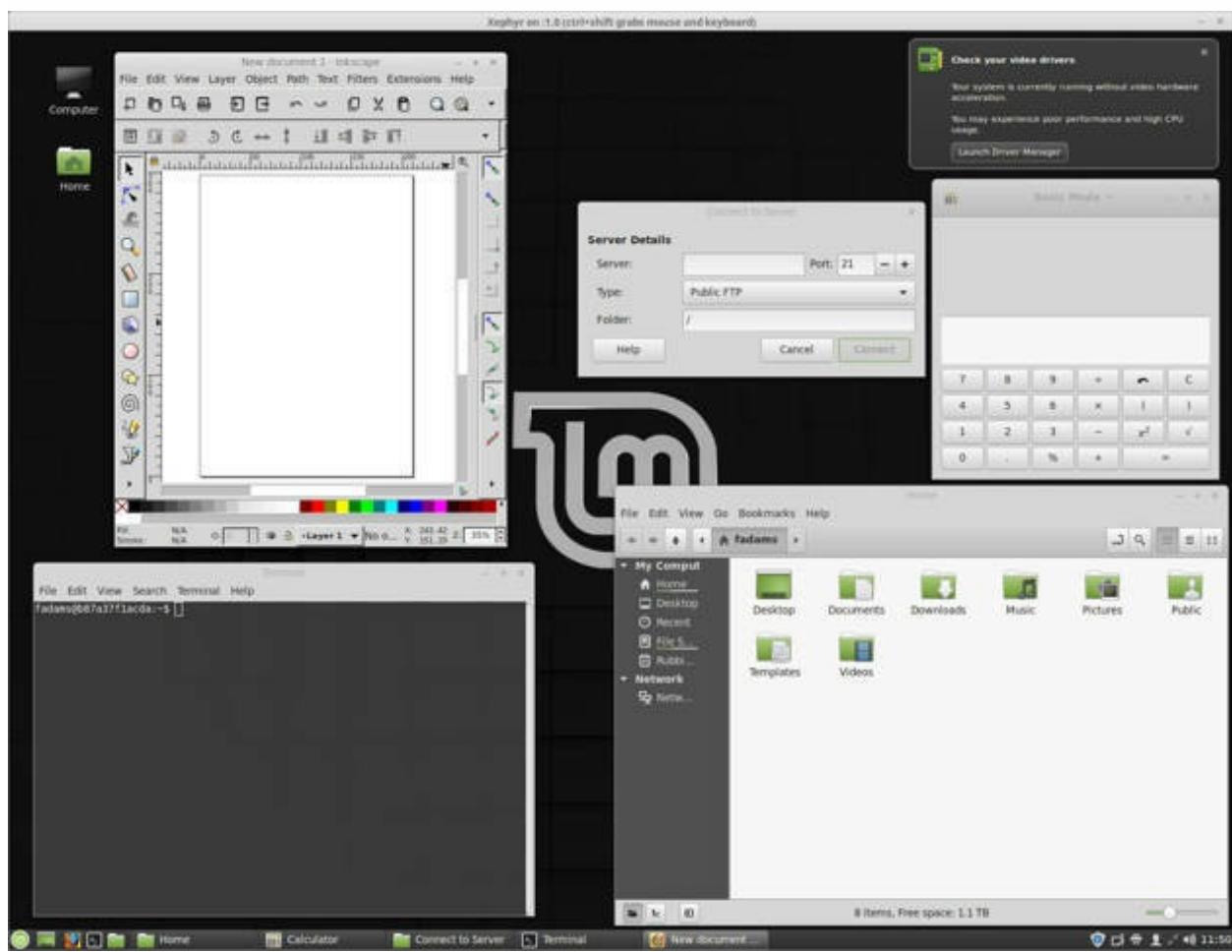
```

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

This script is identical to the debian.sh and ubuntu.sh launch script except for the image and container names, so the explanation of the script previously given in the debian-buster-lxde section holds here too.



9-5 centos-7-gnome

[CentOS](#) is a community-supported Linux distribution functionally compatible with Red Hat Enterprise Linux. The [Dockerfile](#) is in the [centos-7-gnome](#) directory of [docker-gui](#) under [9-virtual-desktops](#).

```
FROM centos:7.7.1908

ENV LANG=en_GB.UTF-8
ENV LC_ALL=en_GB.UTF-8
ENV TZ=Europe/London

RUN \
    # When installing from yum repo it is common to see:
    # Error : Public key for *.rpm is not installed
    # This can be resolved by importing the public key files
    # for RPM, which may be found by running the following
    # docker run --rm centos:7.7.1908 find / -name *GPG*
    rpm --import \
        /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7 && \
    rpm --import \
        /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-Testing-7 && \
    rpm --import \
        /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-Debug-7 && \
    # Install required packages. Add deltarpm to get rid
    # of some warnings & epel-release is needed for lightdm.
    yum -y install deltarpm epel-release && \
    yum -y update && \
    rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7 && \
    yum -y --exclude=abrt* groups install \
    "GNOME Desktop" || true && \
    yum -y groups install \
    "Office Suite and Productivity" && \
    yum -y groups install \
    "Graphics Creation Tools" && \
```

```
yum -y install \
xorg-x11-server-Xephyr thunderbird pavucontrol lightdm \
pulseaudio-module-zeroconf slick-greeter dbus-x11 \
nss-mdns mesa-libGLU && \
# Generate locales
echo LANG="$LANG" > /etc/locale.conf && \
localeddef -i "${LANG%.*}" -f "${LANG#*.}" $LANG && \
# Set up the timezone
echo $TZ > /etc/timezone && \
ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
# Set up Keyboard mapping
echo -e 'XKBMODEL="pc105"\nXKBLAYOUT="gb"\nXKBVARIANT=""\nXKB
OPTIONS=""' > /etc/default/keyboard && \
sed -i 's/KEYMAP="us"/KEYMAP="uk"/' \
/etc/vconsole.conf && \
sed -i 's/Option "XkbLayout" "us"/Option "XkbLayout" "gb"\n
    Option "XkbModel" "pc105"\n        Option "XkbOptions" "t
erminate:ctrl_alt_bksp"/' /etc/X11/xorg.conf.d/00-keyboard.conf
&& \
# Configure LightDM Display Manager to use Xephyr
# instead of X. Reorganise /usr/share/xsessions to set
# gnome-classic as default session as slick-greeter
# uses hardcoded names to select the default session.
mv /usr/share/xsessions/gnome.desktop \
/usr/share/xsessions/gnome-xorg.desktop && \
mv /usr/share/xsessions/gnome-classic.desktop \
/usr/share/xsessions/gnome.desktop && \
echo -e '#!/bin/bash\nexport XAUTHORITY=/root/.Xauthority.doc
ker\nexport DISPLAY=:0\nexec Xephyr $1 -ac >> /var/log/lightdm/
x-1.log' > /usr/bin/Xephyr-lightdm-wrapper && \
chmod +x /usr/bin/Xephyr-lightdm-wrapper && \
echo -e '[LightDM]\nminimum-display-number=1\n[Seat: *]\nsessi
on-setup-script=xrandr --output default --mode 1600x1200\nuser-
session=gnome\nxserver-command=/usr/bin/Xephyr-lightdm-wrapper'
> /etc/lightdm/lightdm.conf.d/70-centos.conf && \
```

```

# CentOS defaults to GDM, change to LightDM
rm -f /etc/systemd/system/display-manager.service && \
ln -s /usr/lib/systemd/system/lightdm.service \
/etc/systemd/system/display-manager.service && \
# CentOS defaults init to multi-user.target,
# change to graphical.target
rm -f /etc/systemd/system/default.target && \
ln -s /usr/lib/systemd/system/graphical.target \
/etc/systemd/system/default.target && \
# Change nautilus and desktop icons from large to small
# https://unix.stackexchange.com/questions/250266/rhel-7-gnom
e-shell-decrease-desktop-icon-size
sed -i '/default-zoom-level/{n; s/large/small/}' \
/usr/share/glib-2.0/schemas/org.gnome.nautilus.gschem
a.xml && \
glib-compile-schemas \
/usr/share/glib-2.0/schemas/ >/dev/null 2>&1 && \
# Prevent PulseAudio trying to launch rtkit-daemon
# because the container doesn't have the required caps
# and the failures slow desktop launch.
sed -i 's/load-module module-udev-detect/load-module module-u
dev-detect tsched=0/' /etc/pulse/default.pa && \
# Disable org.freedesktop.RealtimeKit1.service DBus svc
rm -f /usr/share/dbus-1/system-services/org.freedesktop.Realt
imeKit1.service && \
# Fix issues with slow shutdown
sed -i 's/#DefaultTimeoutStopSec=90s/DefaultTimeoutStopSec=5s
/' /etc/systemd/system.conf && \
# Fix Polkit issues caused by container login being
# considered to be an "inactive" session.
chmod 755 /etc/polkit-1/localauthority && \
# Date & Time
echo -e "[Date & Time]\nIdentity=unix-user:*\nAction=org.gnom
e.controlcenter.datetime.configure\nResultAny=auth_admin_keep\n
ResultInactive=auth_admin_keep\nResultActive=auth_admin_keep\n"
> /etc/polkit-1/localauthority/50-local.d/10-datetime.mechanism.
pkla && \

```

```
# User Accounts
echo -e "[Manage user accounts]\nIdentity=unix-user:*\nAction
=org.gnome.controlcenter.user-accounts.administration\nResultAn
y=auth_admin_keep\nResultInactive=auth_admin_keep\nResultActive
=auth_admin_keep\n" > /etc/polkit-1/localauthority/50-local.d/1
0-user-accounts.pkla && \
# Gnome System Log
echo -e "[Gnome System Log]\nIdentity=unix-user:*\nAction=org
.gnome.logview.pkexec.run\nResultAny=auth_admin_keep\nResultIna
ctive=auth_admin_keep\nResultActive=auth_admin_keep\n" > /etc/p
olkit-1/localauthority/50-local.d/10-system-log.pkla && \
# System Color Manager
echo -e "[System Color Manager]\nIdentity=unix-user:*\nAction
=org.freedesktop.color-manager.create-device;org.freedesktop.co
lor-manager.create-profile;org.freedesktop.color-manager.delete
-device;org.freedesktop.color-manager.delete-profile;org.freede
sktop.color-manager.modify-device;org.freedesktop.color-manager
.modify-profile;org.freedesktop.color-manager.device-inhibit;or
g.freedesktop.color-manager.sensor-lock\nResultAny=yes\nResultI
nactive=yes\nResultActive=yes\n" > /etc/polkit-1/localauthority
/50-local.d/10-color.pkla && \
# Shutdown & Restart
# Note that auth_admin_keep may be better than yes
# here, but there seems to be an issue with the
# authentication dialog appearing.
echo -e "[Shutdown & Restart]\nIdentity=unix-user:*\nAction=o
rg.freedesktop.login1.power-off;org.freedesktop.login1.power-of
f-multiple-sessions;org.freedesktop.login1.reboot;org.freedeskt
op.login1.reboot-multiple-sessions\nResultAny=yes\nResultInacti
ve=yes\nResultActive=yes\n" > /etc/polkit-1/localauthority/50-1
ocal.d/10-shutdown.pkla
```

The image may be built with:

```
docker build -t centos-gnome:7.7 .
```

Most of the Dockerfiles in this book use Debian base images and the apt package manager. This CentOS image is a little different as it uses RPM packages and yum, though the procedure for installing packages is very similar to apt.

The first thing we do is import the public key files for the main RPM repositories to prevent installation errors.

```
rpm --import \
/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7
rpm --import \
/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-Testing-7
rpm --import \
/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-Debug-7
```

We next install deltarpm to fix some deltarpm warnings that would otherwise occur and install the epel-release repository and public key file, which is needed for lightdm.

```
yum -y install deltarpm epel-release
yum -y update
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7
```

With yum we can install the key packages as groups.

```
yum -y --exclude=abrt* groups install \
"GNOME Desktop" || true
yum -y groups install "Office Suite and Productivity"
yum -y groups install "Graphics Creation Tools"
```

There is, however, still a need to install some packages individually.

```
yum -y install \
xorg-x11-server-Xephyr thunderbird pavucontrol lightdm \
pulseaudio-module-zeroconf slick-greeter dbus-x11 \
nss-mdns mesa-libGLU && \
```

Once the packages have been installed we next generate locales, set up the timezone and set the keyboard mapping. Setting up the timezone for CentOS is the same as for Debian and Ubuntu, but generating locales is as follows.

```
echo LANG=\"$LANG\" > /etc/locale.conf
localeddef -i "${LANG%.*}" -f "${LANG#*.}" $LANG
```

Setting the keyboard mapping for CentOS is slightly involved and described in more detail in this article:

<https://serverfault.com/questions/669762/cant-change-the-keyboard-layout-on-centos-7>.

```
echo -e 'XKBMODEL="pc105"\nXKBLAYOUT="gb"\nXKBVARIANT=""\nXKBOPTIONS=""' > /etc/default/keyboard
sed -i 's/KEYMAP="us"/KEYMAP="uk"/' \
/etc/vconsole.conf
sed -i 's/Option "XkbLayout" "us"/Option "XkbLayout" "gb"\n    Option "XkbModel" "pc105"\n            Option "XkbOptions" "terminate:ctrl_alt_bksp"/' /etc/X11/xorg.conf.d/00-keyboard.conf
```

Configuring LightDM for CentOS is similar to the procedure that we used for Debian, where we first reorganise /usr/share/xsessions/ because slick-greeter chooses the first match from a hard-coded list as its default.

```
mv /usr/share/xsessions/gnome.desktop \
    /usr/share/xsessions/gnome-xorg.desktop
mv /usr/share/xsessions/gnome-classic.desktop \
    /usr/share/xsessions/gnome.desktop
```

To configure LightDM to use Xephyr we again create a wrapper script in /usr/bin/Xephyr-lightdm-wrapper.

```
#!/bin/bash
export XAUTHORITY=/root/.Xauthority.docker
export DISPLAY=:0
exec Xephyr $1 -ac >> /var/log/lightdm/x-1.log
```

The LightDM configuration is similar to our Debian one, though for CentOS the configuration file is in /etc/lightdm/lightdm.conf.d/70-centos.conf.

```
[LightDM]
minimum-display-number=1
[Seat: *]
session-setup-script=xrandr --output default --mode 1600x1200
user-session=gnome
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
```

CentOS GNOME uses GDM as the default display manager, so we must change the default display manager to LightDM.

```
rm -f /etc/systemd/system/display-manager.service  
ln -s /usr/lib/systemd/system/lightdm.service \  
/etc/systemd/system/display-manager.service
```

The CentOS Docker image defaults systemd to use multi-user.target, so we must change that to graphical.target.

```
rm -f /etc/systemd/system/default.target  
ln -s /usr/lib/systemd/system/graphical.target \  
/etc/systemd/system/default.target
```

Like our Linux Mint example we change the size of the nautilus and desktop icons using the procedure described in <https://unix.stackexchange.com/questions/250266/rhel-7-gnome-shell-decrease-desktop-icon-size>.

```
sed -i '/default-zoom-level/{n; s/large/small/}' \  
/usr/share/glib-2.0/schemas/org.gnome.nautilus.gschem.xml  
glib-compile-schemas \  
/usr/share/glib-2.0/schemas/ >/dev/null 2>&1
```

When developing this application the author initially experienced very slow desktop launch times. This turned out to be due to PulseAudio trying to launch rtkit-daemon, which fails because the container hasn't been given the required capabilities and then eventually times out after 30 seconds.

To resolve the problem we disable this in the PulseAudio configuration and also disable the org.freedesktop.RealtimeKit1.service D-bus service.

```
sed -i 's/load-module module-udev-detect/load-module module-ude  
v-detect tsched=0/' /etc/pulse/default.pa  
rm -f /usr/share/dbus-1/system-services/org.freedesktop.RealtimeKit1.service
```

As with our other virtual desktop containers system shutdown was slower than expected due to systemd timing out on some non-existent services. This was again resolved by reducing the default timeout period.

```
sed -i 's/#DefaultTimeoutStopSec=90s/DefaultTimeoutStopSec=5s/'  
/etc/systemd/system.conf
```

Finally, we again create entries in /etc/polkit-1/localauthority/50-local.d/ to elevate “inactive” session privileges in exactly the same way as we did for our Ubuntu GNOME example.

The [centos.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin  
. $BIN/docker-command.sh  
. $BIN/docker-xauth.sh  
  
IMAGE=centos-gnome:7.7  
CONTAINER=centos
```

```

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; passwd '$(id -un)'; usermod -aG wheel '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

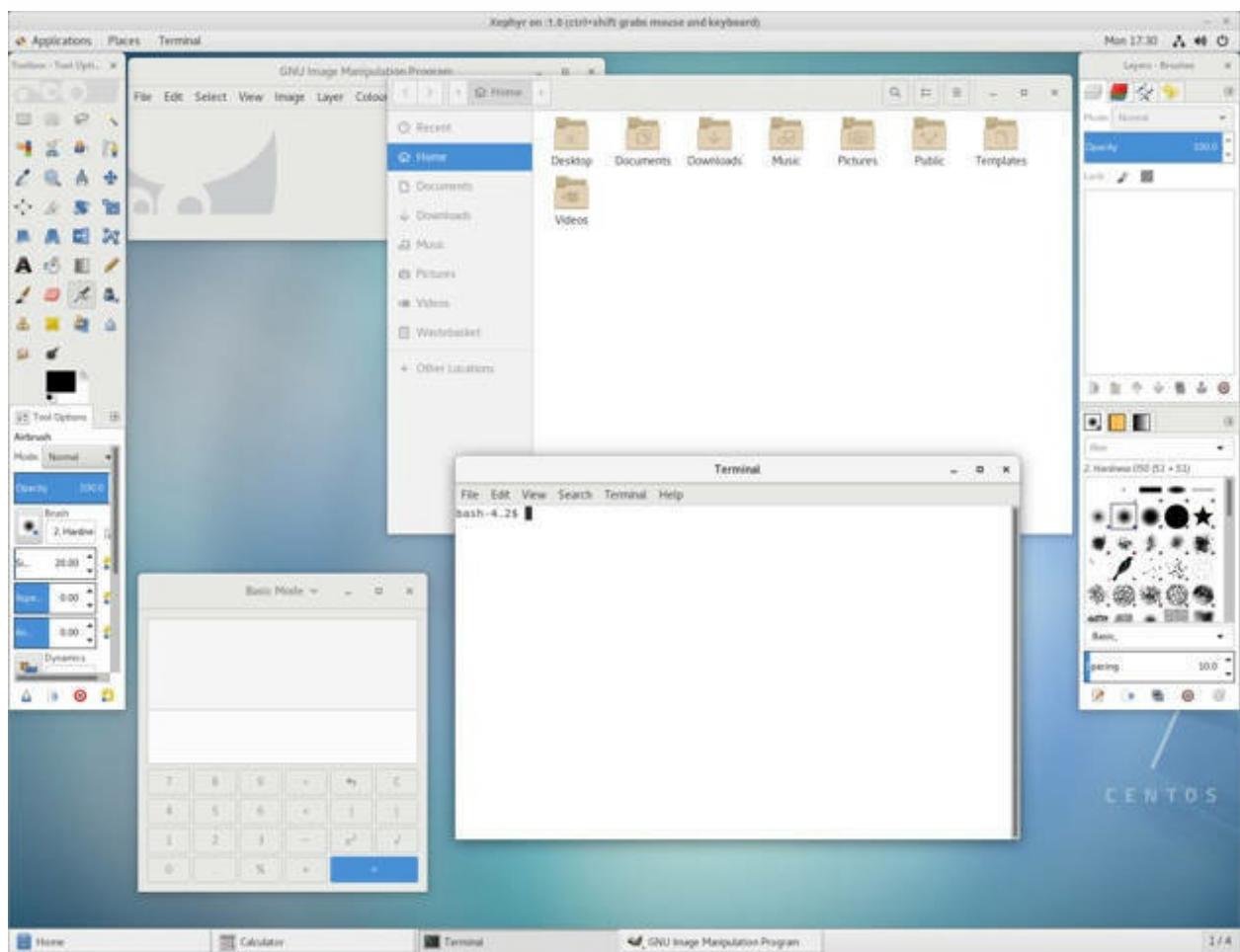
mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

Like our previous launch scripts the debian-buster-lxde explanation holds here too, with a slight difference in the adduser command that creates the credentials bundle.



9-6 fedora-32-kde

[Fedora](#) is a community-supported Linux distribution sponsored primarily by Red Hat. Fedora works closely with [upstream](#) communities and many applications and services are quite “bleeding edge”, making it interesting and challenging to containerise but a good learning experience. The [Dockerfile](#) is in the [fedora-32-kde](#) directory of [docker-gui](#) under [9-virtual-desktops](#).

```
FROM fedora:32

ENV LANG=en_GB.UTF-8
ENV LC_ALL=en_GB.UTF-8
ENV TZ=Europe/London

RUN dnf -y update && \
    dnf -y install \
        xorg-x11-server-utils xorg-x11-utils xorg-x11-xinit && \
        dnf -y groups install "KDE Plasma Workspaces" || true && \
        dnf -y groups install "KDE Applications" && \
        dnf -y groups install "KDE Multimedia support" && \
        dnf -y groups install "LibreOffice" && \
    dnf -y install \
        xorg-x11-server-Xephyr passwd procps cracklib-dicts \
        ntp glibc-locale-source glibc-langpack-en thunderbird \
        open-sans-fonts firefox pulseaudio-module-zeroconf \
        pavucontrol gimp inkscape nss-mdns mesa-libGLU \
    # Install Display Manager and dependencies
    lightdm slick-greeter dbus-x11 || true && \
    # Generate locales
    echo LANG=\"$LANG\" > /etc/locale.conf && \
    localeddef -i "${LANG%.*}" -f "${LANG##*.}.$LANG" && \
    # Set up the timezone
    echo $TZ > /etc/timezone && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
```

```

# Set up Keyboard mapping
echo -e 'XKBMODEL="pc105"\nXKBLAYOUT="gb"\nXKBVARIANT=""\nXKB
OPTIONS=""' > /etc/default/keyboard && \
sed -i 's/KEYMAP="us"/KEYMAP="uk"/' \
/etc/vconsole.conf && \
sed -i 's/Option "XkbLayout" "us"/Option "XkbLayout" "gb"\n
      Option "XkbModel" "pc105"\n      Option "XkbOptions" "t
erminate:ctrl_alt_bksp"/' /etc/X11/xorg.conf.d/00-keyboard.conf
&& \
# Configure LightDM Display Manager to use Xephyr
# instead of X.
echo -e '#!/bin/bash\nexport XAUTHORITY=/root/.Xauthority.doc
ker\nexport DISPLAY=:0\nexec Xephyr $1 -ac >> /var/log/lightdm/
x-1.log' > /usr/bin/Xephyr-lightdm-wrapper && \
chmod +x /usr/bin/Xephyr-lightdm-wrapper && \
echo -e '[LightDM]\nminimum-display-number=1\n[Seat: *]\nsessi
on-setup-script=sh -c "xdpyinfo | grep -q RANDR && exec xrandr
--output default --mode 1600x1200 || true"\nxserver-command=/us
r/bin/Xephyr-lightdm-wrapper' > /etc/lightdm/lightdm.conf.d/70-
fedora.conf && \
# Change default Display Manager to LightDM
rm -f /etc/systemd/system/display-manager.service && \
ln -s /usr/lib/systemd/system/lightdm.service \
/etc/systemd/system/display-manager.service && \
# Set greeter background
echo -e '[Greeter]\nbackground=/usr/share/wallpapers/F32/cont
ents/images/1600x1200.png\n' > /etc/lightdm/slick-greeter.conf
&& \
# Fedora defaults init to multi-user.target,
# change to graphical.target
rm -f /etc/systemd/system/default.target && \
ln -s /usr/lib/systemd/system/graphical.target \
/etc/systemd/system/default.target && \

```

```

# Stop fonts being HUGE. Is there a better way?
echo -e 'mkdir -p ~/.config\nif [ ! -f ~/.config/kcmfonts ];\nthen\n  echo -e "[General]\ndontChangeAASettings=true\nforceFontDPI=96\n" > ~/.config/kcmfonts\nfi' > /etc/profile.d/fix-fonts.sh && \
# Fedora 32 defaults to using dbus-broker rather than
# dbus-daemon which seems to cause some significant
# systemd issues, so revert back to dbus-daemon.
# First disable dbus-broker
rm /etc/systemd/system/dbus.service && \
# Needed for Fedora 31 breaks Fedora 32!! so use -f
rm -f /etc/systemd/system/messagebus.service && \
# Then enable dbus-daemon
ln -s /usr/lib/systemd/system/dbus-daemon.service \
      /etc/systemd/system/dbus.service && \
ln -s /usr/lib/systemd/system/dbus-daemon.service \
      /etc/systemd/system/messagebus.service && \
ln -s /usr/lib/systemd/system/dbus-daemon.service \
      /etc/systemd/system/multi-user.target.wants/dbus-daemon \
.service && \
# Unmask systemd-logind.service
rm /etc/systemd/system/systemd-logind.service && \
# Fix systemd-logind.service config that borked user
# session creation. With DevicesAllow enabled a call to
# access("/dev/tty0", F_OK) fails with EPERM and
# session creation fails with the error:
# Seat has no VTs but VT number not 0 error
# Passing in /dev/tty0 usually suppresses that failure
# but post systemd v241 the config borked that albeit
# hacky fix to systemd demanding hardware for seats.
sed -i 's/DeviceAllow/#DeviceAllow/g' \
      /usr/lib/systemd/system/systemd-logind.service && \
# Mask tmp.mount which was overwriting the X11
# socket we were bind-mounting for Xephyr to use.
ln -s /dev/null /etc/systemd/system/tmp.mount && \
# Prevent PulseAudio trying to launch rtkit-daemon
# because the container doesn't have the required caps
# and the failures slow desktop launch.

```

```
sed -i 's/load-module module-udev-detect/load-module module-u
dev-detect tsched=0/' /etc/pulse/default.pa && \
# Disable org.freedesktop.RealtimeKit1.service DBus svc
rm -f /usr/share/dbus-1/system-services/org.freedesktop.Realt
imeKit1.service && \
# Fix Polkit issues caused by container login being
# considered to be an "inactive" session.
chmod 755 /etc/polkit-1/localauthority && \
# dnf
echo -e "[dnf]\nIdentity=unix-user:*\nAction=org.baseurl.DnfS
ystem.write;org.baseurl.DnfSystem.read\nResultAny=auth_admin_ke
ep\nResultInactive=auth_admin_keep\nResultActive=auth_admin_ke
p\n" > /etc/polkit-1/localauthority/50-local.d/10-dnf.pkla && \
# PCSC
echo -e "[PCSC]\nIdentity=unix-user:*\nAction=org.debian.pcsc
-lite.access_pcsc;org.org.debian.pcsc-lite.access_card\nResultA
ny=yes\nResultInactive=yes\nResultActive=yes\n" > /etc/polkit-1
/localauthority/50-local.d/10-pcsc.pkla && \
# NetworkManager
echo -e "[NetworkManager]\nIdentity=unix-user:*\nAction=org.f
reedesktop.NetworkManager.network-control\nResultAny=yes\nResul
tInactive=yes\nResultActive=yes\n" > /etc/polkit-1/localauthori
ty/50-local.d/10-NetworkManager.pkla && \
# packagekit refresh
echo -e "[packagekit]\nIdentity=unix-user:*\nAction=org.freed
esktop.packagekit.system-sources-refresh\nResultAny=yes\nResult
Inactive=yes\nResultActive=yes\n" > /etc/polkit-1/localauthorit
y/50-local.d/10-packagekit.pkla && \
```

```
# System Color Manager
echo -e "[System Color Manager]\nIdentity=unix-user:*\nAction
=org.freedesktop.color-manager.create-device;org.freedesktop.co
lor-manager.create-profile;org.freedesktop.color-manager.delete
-device;org.freedesktop.color-manager.delete-profile;org.freedes
ktop.color-manager.modify-device;org.freedesktop.color-manager
.modify-profile;org.freedesktop.color-manager.device-inhibit;or
g.freedesktop.color-manager.sensor-lock\nResultAny=yes\nResultI
nactive=yes\nResultActive=yes\n" > /etc/polkit-1/localauthority
/50-local.d/10-color.pkla && \
# Shutdown & Restart
# Note that auth_admin_keep may be better than yes
# here, but there seems to be an issue with the
# authentication dialog appearing.
echo -e "[Shutdown & Restart]\nIdentity=unix-user:*\nAction=o
rg.freedesktop.login1.power-off;org.freedesktop.login1.power-of
f-multiple-sessions;org.freedesktop.login1.reboot;org.freedesk
op.login1.reboot-multiple-sessions\nResultAny=yes\nResultInacti
ve=yes\nResultActive=yes\n" > /etc/polkit-1/localauthority/50-l
ocal.d/10-shutdown.pkla
```

The image may be built with:

```
docker build -t fedora-kde:32 .
```

Although Fedora uses dnf as its package manager the Dockerfile begins in a similar way to our CentOS 7 example, this time using KDE Plasma as the desktop environment for variety.

After installing the required packages we generate locales, set up the timezone and set the keyboard mapping using exactly the same process we used in our CentOS 7 example.

Configuring LightDM for Fedora is similar to the procedure that we used for CentOS 7, where we again create a wrapper script in /usr/bin/Xephyr-lightdm-wrapper.

```
#!/bin/bash
export XAUTHORITY=/root/.Xauthority.docker
export DISPLAY=:0
exec Xephyr $1 -ac >> /var/log/lightdm/x-1.log
```

The LightDM configuration is similar to our CentOS 7 one, though for Fedora the configuration file is in /etc/lightdm/lightdm.conf.d/70-fedora.conf.

```
[LightDM]
minimum-display-number=1
[Seat:*]
session-setup-script=sh -c "xdpyinfo | grep -q RANDR && exec xr
andr --output default --mode 1600x1200 || true"
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
```

Fedora KDE uses SDDM as the default display manager, so we must change the systemd default display manager to LightDM.

```
rm -f /etc/systemd/system/display-manager.service
ln -s /usr/lib/systemd/system/lightdm.service \
/etc/systemd/system/display-manager.service
```

We also set the greeter background in /etc/lightdm/slick-greeter.conf to give a Fedora theme.

```
[Greeter]
background=/usr/share/wallpapers/F32/contents/images/1600x1200.
png
```

The Fedora Docker image defaults systemd to use multi-user.target, so we must change that to graphical.target.

```
rm -f /etc/systemd/system/default.target
ln -s /usr/lib/systemd/system/graphical.target \
      /etc/systemd/system/default.target
```

Because we are using Xephyr the KDE automatic DPI calculation is broken, resulting in huge fonts by default. To resolve this we create the following script in /etc/profile.d/fix-fonts.sh. This script is run automatically at session startup to force the DPI we require.

```
mkdir -p ~/.config
if [ ! -f ~/.config/kcmfonts ]; then
    echo -e "[General]\ndontChangeAASettings=true\nforceFontDPI=9
6\n" > ~/.config/kcmfonts
fi
```

As the Fedora Dockerfile was being developed the author rather naively assumed that it would follow the same general “pattern” as the CentOS 7 Dockerfile. Unfortunately however due to the “bleeding edge” nature of Fedora a number of additional complications were discovered that took considerable time and experimentation to work through.

The first issue manifest itself by the systemctl systemd control commands taking an extremely long time to complete. The reason for this turned out to be due to Fedora 32 using [dbus-broker](#) rather than dbus-daemon as the default D-bus service.

Unfortunately the author couldn't work out the underlying issue with dbus-broker (though it may be related to kernel version), so as a work-around we revert back to dbus-daemon.

```
# First disable dbus-broker
rm /etc/systemd/system/dbus.service
rm -f /etc/systemd/system/messagebus.service
# Then enable dbus-daemon
ln -s /usr/lib/systemd/system/dbus-daemon.service \
      /etc/systemd/system/dbus.service
ln -s /usr/lib/systemd/system/dbus-daemon.service \
      /etc/systemd/system/messagebus.service
ln -s /usr/lib/systemd/system/dbus-daemon.service \
      /etc/systemd/system/multi-user.target.wants/dbus-daemon.s
ervice
```

The next issue was something of a nightmare to get to the bottom of. Recall that in our launch scripts we add `--device=/dev/tty0`. This is really a work-around to systemd being very “opinionated” on the topic of [seats](#) and user sessions, whereby seats are generally associated with physical hardware like monitors, keyboards etc..

This often makes sense, but for the case of headless and nested X Servers and virtual sessions it can be really annoying as logind doesn't have a clean way to force session creation. Adding tt0 “tricks” logind to correctly create the user session, which is required in order to set up `XDG_RUNTIME_DIR`, which in turn is relied on by PulseAudio and a number of other user services.

Unfortunately with Fedora 32 the tt0 trick didn't work and journalctl logs revealed a “Seat has no VTs but VT number not 0” error, and the desktop had no `XDG_RUNTIME_DIR`.

Working out what had changed was awkward, as the relevant section of code in systemd v245 was largely the same as in systemd v241 used in Fedora 30, which worked as expected. It took compiling systemd from source and running through a debugger to find a call to `access("/dev/tty0", F_OK)` that failed with EPERM in v245, but worked as expected in v241.

EPERM suggested a change to permitted capabilities in the `systemd-logind.service` configuration, and comparing `/etc/systemd/system/systemd-logind.service` between systemd v241 and v245 yielded the following solution.

```
# Unmask systemd-logind.service
rm /etc/systemd/system/systemd-logind.service
sed -i 's/DeviceAllow/#DeviceAllow/g' \
    /usr/lib/systemd/system/systemd-logind.service
```

Another surprise issue was that the X11 socket bind-mounted from the host was being overwritten in the container, causing Xephyr to fail. This turned out to be due to the `tmp.mount` service and was resolved by masking that service.

```
ln -s /dev/null /etc/systemd/system/tmp.mount
```

We also had a number of rtkit-daemon errors that were resolved with the same solution we used in CentOS 7.

```
sed -i 's/load-module module-udev-detect/load-module module-ude
v-detect tsched=0/' /etc/pulse/default.pa
rm -f /usr/share/dbus-1/system-services/org.freedesktop.Realtime
eKit1.service
```

To round off the Dockerfile, we again create entries in /etc/polkit-1/localauthority/50-local.d/ to elevate “inactive” session privileges in exactly the same way as we did for our previous virtual desktop examples.

The [fedora.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh

IMAGE=fedora-kde:32
CONTAINER=fedora

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; passwd '$(id -un)'; usermod -aG wheel '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
```

```

$DOCKER_COMMAND run --rm -d \
--device=/dev/tty0 \
--name $CONTAINER \
--ipc=host \
--shm-size 2g \
--security-opt apparmor=unconfined \
--cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
-v /sys/fs/cgroup:/sys/fs/cgroup \
-v $PWD/$(id -un):/home/$(id -un) \
-v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$IMAGE /sbin/init

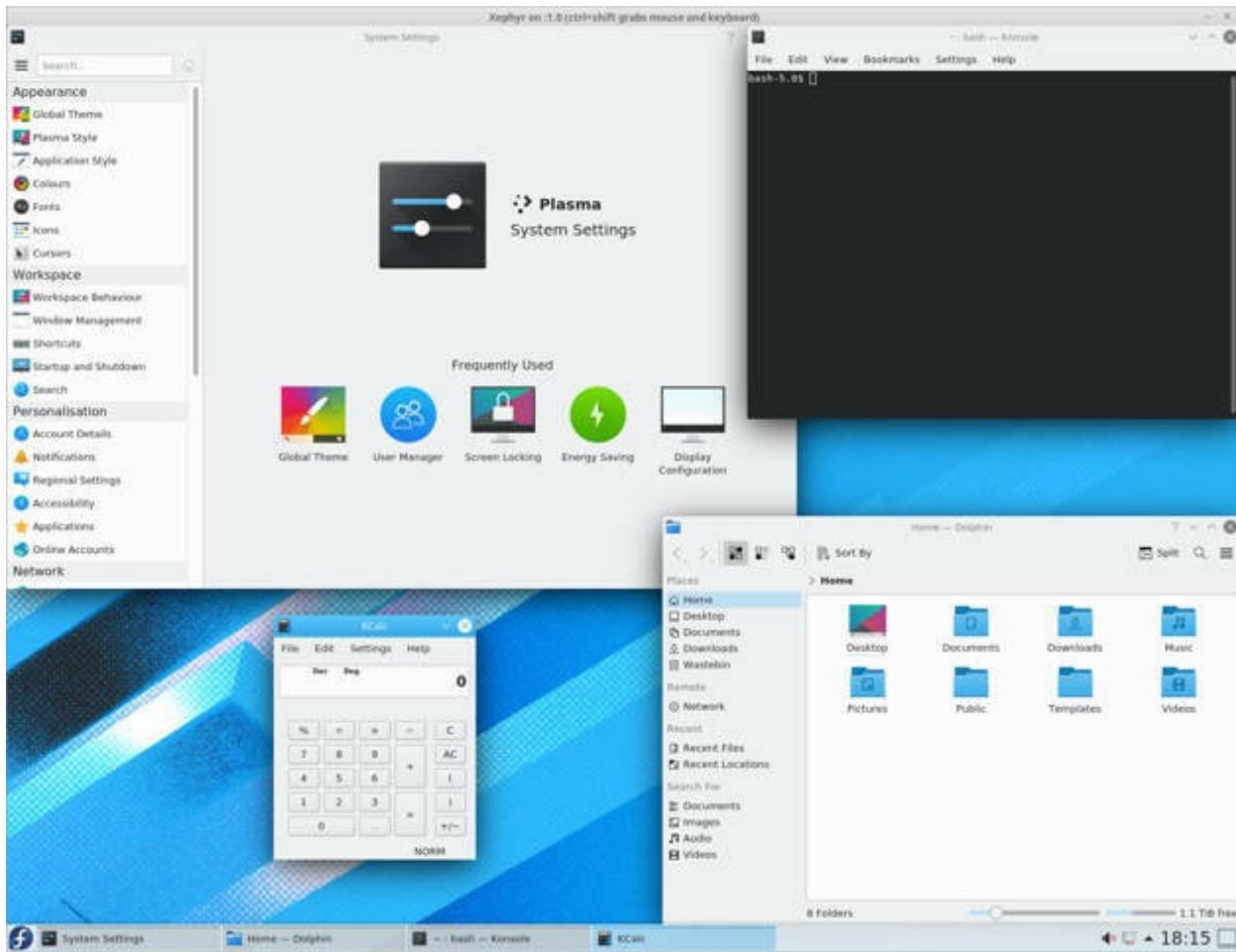
# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

This script is identical to our previous virtual desktop launch scripts apart from the image and container names and the debian-buster-lxde explanation holds here too.

As with our previous virtual desktop examples running this script will give us a full graphical login.





10 3D Accelerated Virtual Desktops

One place where container based virtual desktops can significantly outperform virtual machines is with respect to resource utilisation. With VMs the storage and CPU resources are pre-allocated and relatively fixed. In practice however the *actual* resource requirements are often extremely variable, especially for a user desktop. This situation can result in significant over-allocation and under-utilisation of resources.

Virtual Machines often provide especially poor 3D performance and most Virtual GPUs offer only a fraction of the performance of physical GPUs. Although multi-GPU hosts can provide direct GPU access to VMs via PCI pass through this model generally means one GPU per VM, which again can result in significant under-utilisation of resources.

The [3D Accelerated Applications](#) section earlier in the book illustrates that it is possible to run 3D accelerated applications in containers at near native performance, however for desktops there are added complications. The main issue is that it is not possible to run a window manager *directly* in a window on the host, so we must use a nested X Server like Xephyr and unfortunately Xephyr does not directly support 3D acceleration. Fortunately, our work in the [Remote 3D Accelerated Applications](#) section provides us with an effective workaround in the form of [VirtualGL](#).

This chapter extends our earlier virtual desktops with VirtualGL to provide good 3D desktop performance albeit with more overhead than running individual applications.

10-1 ubuntu-18-04-gnome

The [Dockerfile](#) is in the [ubuntu-18.04-gnome](#) directory of [docker-gui](#) under [10-3D-accelerated-virtual-desktops](#).

```
FROM ubuntu-gnome:18.04

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    libvdpau1 mesa-vdpau-drivers && \
    # Attempt to work out the latest VirtualGL version from
    # https://sourceforge.net/projects/virtualgl/files/
    VGL_VERSION=$(curl -sSL https://sourceforge.net/projects/virt
    ualgl/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 |
    cut -d \> -f2 | cut -d \< -f1) && \
    echo "VirtualGL version: ${VGL_VERSION}" && \
    # Given the version download and install VirtualGL
    curl -sSL https://sourceforge.net/projects/virtualgl/files/${
    VGL_VERSION}/virtualgl_${VGL_VERSION}_amd64.deb -o virtualgl_${{
    VGL_VERSION}}_amd64.deb && \
    dpkg -i virtualgl_*_amd64.deb && \
    rm virtualgl_*_amd64.deb && \
    # Give VGL access to host X Server for 3D rendering
    echo 'XAUTHORITY=$HOME/.Xauthority.docker' > \
        /etc/profile.d/Xauthority-fix.sh && \
    echo 'export LD_PRELOAD=/usr/lib/libdlfaker.so:/usr/lib/libvg
    lfaker.so:$LD_PRELOAD' > /etc/profile.d/virtualgl.sh && \
    # Stop openssh/xorg bug clobbering LD_PRELOAD
    sed -i 's/use-ssh-agent/no-use-ssh-agent/' \
        /etc/X11/Xsession.options
```

The image may be built with:

```
docker build -t ubuntu-gnome-vgl:18.04 .
```

As we can see from the Dockerfile, extending our virtual desktops from the previous chapter to support 3D acceleration is surprisingly straightforward.

The Dockerfile begins by installing the libvdpau1 and mesa-vdpau-drivers packages. This step is unnecessary if we only wish to support OpenGL acceleration, but including it means that our new virtual desktop will also support hardware accelerated video decoding where available.

The next steps revisit what we learned in [Remote 3D Accelerated Applications](#), where we attempt to “scrape” the most recent VirtualGL version from SourceForge then download and install the VirtualGL deb package.

When we used VirtualGL for individual applications we simply used the vglrun script to launch the applications. This approach isn’t suitable for running virtual desktops as we have a range of applications and, moreover, we also want the desktop rendering to be accelerated where possible.

The approach we have taken to enable VirtualGL more generally is to use the fact that scripts placed in /etc/profile.d will be run automatically as users login. We can set the environment variables needed by VirtualGL there and know that they will be available in the environment for each user.

In /etc/profile.d/Xauthority-fix.sh we ensure that XAUTHORITY points to the Xauthority that will be bind-mounted from the host to allow VirtualGL access to the host's X Server for 3D rendering via GLX Forking.

```
XAUTHORITY=$HOME/.Xauthority.docker
```

In /etc/profile.d/virtualgl.sh we set LD_PRELOAD to point to the VirtualGL interposing libraries, which is actually what vglrun does but setting it in /etc/profile.d causes the preloading to be done more generally.

```
export LD_PRELOAD=/usr/lib/libdlfaker.so:/usr/lib/libvglfaker.so:$LD_PRELOAD
```

To work around an openssh/Xorg bug that causes LD_LIBRARY_PATH to be unset, the final part of the Dockerfile modifies /etc/X11/Xsession.options to disable ssh-agent, as described in <https://bugs.launchpad.net/ubuntu/+source/openssh/+bug/47958>

```
sed -i 's/use-ssh-agent/no-use-ssh-agent/'
```

The [ubuntu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh
```

```

IMAGE=ubuntu-gnome-vgl:18.04
CONTAINER=ubuntu-vgl

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro \
    \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    $IMAGE /sbin/init
# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

This launch script is somewhat similar to our previous non-accelerated virtual desktop launch scripts, however there are a number of subtle differences that might easily be overlooked.

Firstly, we source \$BIN/docker-gpu.sh rather than \$ BIN/docker-command.sh to include the required GPU switches for our `docker run` call.

Then, in the `docker run` call, we bind-mount the Xauthority from the host to the *user's* home directory in the container. This is in addition to bind-mounting to /root as we did for the non-accelerated case. This is because when we run 3D applications in the user session they will need access to the host's X Server to perform GLX Forking.

```
-v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro
```

Finally, we add \$GPU_FLAGS to the `docker run` call to enable the required environment variables and bind-mounts.

10-2 linuxmint-19-3-cinnamon

The [Dockerfile](#) is in the [linuxmint-19.3-cinnamon](#) directory of [docker-gui](#) under [10-3D-accelerated-virtual-desktops](#).

The Dockerfile for linuxmint-19-3-cinnamon and the [mint.sh](#) launch script are identical to those for our ubuntu-18-04-gnome virtual desktop aside from some trivial differences like the image and container names and so won't be explored in depth here, as there is not much more to add.

10-3 centos-7-gnome

The [Dockerfile](#) is in the [centos-7-gnome](#) directory of [docker-gui](#) under [10-3D-accelerated-virtual-desktops](#).

```
FROM centos-gnome:7.7

RUN yum -y install libvdpau mesa-vdpau-drivers && \
    # Attempt to work out the latest VirtualGL version from
    # https://sourceforge.net/projects/virtualgl/files/
    VGL_VERSION=$(curl -sSL https://sourceforge.net/projects/virt
ualgl/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 | 
cut -d \> -f2 | cut -d \<; -f1) && \
    echo "VirtualGL version: ${VGL_VERSION}" && \
    # Given the version download and install VirtualGL
    curl -sSL https://sourceforge.net/projects/virtualgl/files/${
VGL_VERSION}/VirtualGL-${VGL_VERSION}.x86_64.rpm -o VirtualGL-$
{VGL_VERSION}.x86_64.rpm && \
    rpm -i VirtualGL-${VGL_VERSION}.x86_64.rpm && \
    rm VirtualGL-${VGL_VERSION}.x86_64.rpm && \
    # Give VGL access to host X Server for 3D rendering
    echo 'XAUTHORITY=$HOME/.Xauthority.docker' > \
        /etc/profile.d/Xauthority-fix.sh && \
    echo 'export LD_PRELOAD=/usr/lib64/libdlfaker.so:/usr/lib64/l
ibvglfaker.so:$LD_PRELOAD' > /etc/profile.d/virtualgl.sh
```

The image may be built with:

```
docker build -t centos-gnome-vgl:7.7 .
```

The Dockerfile is similar to our Ubuntu and Linux Mint examples, though for CentOS we use yum and rpm rather than apt for package management and there are some differences in the package names and library paths.

The [centos.sh](#) launch script is also very nearly identical to those we developed for Ubuntu and Linux Mint, aside from the container and image names and the subtle difference to the adduser command that creates the credentials bundle:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

IMAGE=centos-gnome-vgl:7.7
CONTAINER=centos-vgl

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for c
ontainer."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home
'$id -un'; passwd '$(id -un)'; usermod -aG wheel '$(id -un)';
tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/gr
oup'
fi

# Create home directory
mkdir -p $(id -un)
```

```

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro
\
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
$IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

10-4 fedora-32-kde

The [Dockerfile](#) is in the [fedora-32-kde](#) directory of [docker-gui](#) under [10-3D-accelerated-virtual-desktops](#).

The Dockerfile for fedora-32-kde and the [fedora.sh](#) launch script are identical to those for our centos-7-gnome virtual desktop aside from some trivial differences like the image and container names and using dnf instead of yum, so won't be explored in depth here as there is not much more to add.

11 Remote Virtual Desktops

After developing containerised virtual desktops for a range of popular Linux distributions and then subsequently enhancing those with GPU accelerated 3D graphics and video decoding, the natural conclusion of our journey is to enable remote access to our desktops.

This chapter revisits many of the services that we first introduced in the [Remote Applications](#) chapter, though as it turns out installing and configuring them in the context of full virtual desktops comes with some unforeseen challenges.

11-1 VNC

Because most of the VNC servers that we covered previously also expose a built-in X Server, there are many different ways to enable VNC access to a remote desktop. One issue that is often glossed over however is that modern desktops often make use a wide range of underlying system services and many of those services have dependencies on concepts like seats and sessions in order to behave correctly.

To ensure that our remote virtual desktops have behaviour that mirrors local desktops as closely as possible, our approach will be to make use of a feature of LightDM that actually allows the display manager to be configured to listen for VNC connections and launch VNC servers. This approach *should* keep our configuration set up simple, whilst also enabling the correct behaviour.

11-1-1 debian-buster-lxde-tightvnc

The [Dockerfile](#) is in the [debian-buster-lxde-tightvnc](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/vnc](#).

```
FROM debian-lxde:buster

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    tightvncserver xffonts-base xffonts-75dpi \
    xffonts-100dpi xffonts-scalable && \
    # Debian LightDM config is in /etc/lightdm/lightdm.conf
    echo '[LightDM]\nminimum-display-number=1\n[Seat:*\nsession-
setup-script=sh -c "xdpyinfo | grep -q RANDR && exec xrandr --o
utput default --mode 1600x1200 || true"\ngreeter-hide-users=fal
se\nxserver-command=/usr/bin/Xephyr-lightdm-wrapper\n[VNCServer
]\nenabled=true\ndepth=24\ncommand=Xvnc -ac -rfbauth /tmp/light
dm/.vnc/passwd' > /etc/lightdm/lightdm.conf
```

The image may be built **on the remote host** with:

```
docker build -t debian-lxde-tightvnc:buster .
```

This Dockerfile illustrates how straightforward it can be to enable remote access to our virtual desktops with VNC. By simply installing tightvncserver and configuring the display manager we can export a fully functioning remote desktop.

The most interesting part of the Dockerfile is the LightDM configuration in /etc/lightdm/lightdm.conf.

```
[LightDM]
minimum-display-number=1
[Seat:*]
session-setup-script=sh -c "xdpyinfo | grep -q RANDR && exec xr
andr --output default --mode 1600x1200 || true"
greeter-hide-users=false
xserver-command=/usr/bin/Xephyr -lightdm-wrapper
[VNCServer]
enabled=true
depth=24
command=Xvnc -ac -rfbauth /tmp/lightdm/.vnc/passwd
```

The first part of this configuration starts out exactly the same as the lightdm.conf in our original [debian-buster-lxde](#) example. Running the session-setup-script using `sh -c` and ending it with `|| true` is particularly important for this example as tightvncserver does not support the X Resize and Rotate (RandR) extension, so calling xrandr without first checking if the extension is supported will cause the X Server launch to fail.

The second part of this configuration configures LightDM to listen for VNC connections, on port 5900 by default, then launch Xvnc with basic password authentication when a connection occurs.

The [debian.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%*/}docker-gui/bin)
. $BIN/docker-command.sh
. $BIN/docker-xauth.sh

IMAGE=debian-lxde-tightvnc:buster
CONTAINER=debian-tightvnc
```

```

if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

mkdir -p $(id -un)/.vnc

# Create VNC password if required.
if ! test -f "$(id -un)/.vnc/passwd"; then
    echo "creating VNC password"
    $DOCKER_COMMAND run --rm -it \
        -u $(id -u):$(id -g) \
        -v /etc/passwd:/etc/passwd:ro \
        -v $PWD/$(id -un):/home/$(id -un) \
        $IMAGE vncpasswd
fi

# Launch container as root to init core Linux services
$DOCKER_COMMAND run --rm -d \
    -p 5900:5900 \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $PWD/$(id -un)/.vnc:/tmp/lightdm/.vnc \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $IMAGE /sbin/init

```

```
# Trivial wait for container to be running before cp credential
#
sleep 0.25

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/
```

This script mostly follows the same pattern as the launch script for our original, local, debian-buster-lxde. The first difference is the addition of a section to create the VNC password, which performs the same role as the storepasswd.sh scripts in our earlier VNC examples. The remaining changes are in the `docker run`, where we expose port 5900 and bind-mount the .vnc directory containing the VNC password to a path accessible to the lightdm user.

The container may be launched remotely, supplying a username, host and path and using any VNC client to connect:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-
virtual-desktops/vnc/debian-buster-lxde-tightvnc; ./debian.sh"
```

As this example extends our earlier LightDM configuration it will launch Xephyr on the remote server. If this behaviour is not desired, an alternative [Dockerfile-no-xephyr](#) is provided in the same directory of the repository, that replaces Xephyr with a dummy Xvfb instance that is launched by the display manager, but never used.

11-1-2 ubuntu-18-04-gnome-tigervnc

Whilst tightvncserver works well for a simple desktop like LXDE, if we were to simply replace the base image in the previous example with one of our more sophisticated desktops it will fail. With Linux Mint the desktop will launch, but will only run in “fallback mode” which uses MATE. With Ubuntu, GNOME refuses to start at all.

The reason for this is because the built-in tightvncserver X Server doesn’t support X RandR or OpenGL, both of which are required by most of the more resource hungry desktops. Fortunately, the built-in X Server provided by tigervnc *does* support them.

The [Dockerfile](#) is in the [ubuntu-18.04-gnome-tigervnc](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/vnc](#).

```
FROM ubuntu-gnome-vgl:18.04

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    tigervnc-common tigervnc-standalone-server \
    xfonts-base xfonts-75dpi xfonts-100dpi \
    xfonts-scalable && \
    echo '[LightDM]\nminimum-display-number=1\n[Seat: *]\nuser-session=ubuntu-xorg\nxserver-command=/usr/bin/Xephyr-lightdm-wrapper\n[VNCServer]\nenabled=true\ndepth=24\ncommand=Xvnc -ac -rfbauth /tmp/lightdm/.vnc/passwd' > /etc/lightdm/lightdm.conf.d/70-ubuntu.conf
```

The image may be built **on the remote host** with:

```
docker build -t ubuntu-gnome-tigervnc:18.04 .
```

This Dockerfile is very similar to our tightvnc example, though obviously installs tigervnc-common and tigervnc-standalone-server instead of tightvncserver.

The LightDM configuration is similar, though for Ubuntu it is in /etc/lightdm/lightdm.conf.d/70-ubuntu.conf.

```
[LightDM]
minimum-display-number=1
[Seat:*]
user-session=ubuntu-xorg
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
[VNCServer]
enabled=true
depth=24
command=Xvnc -ac -rfbauth /tmp/lightdm/.vnc/passwd
```

Again, this is just an extension of the configuration used in our earlier ubuntu-18.04-gnome example.

The [ubuntu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

IMAGE=ubuntu-gnome-tigervnc:18.04
CONTAINER=ubuntu-tigervnc
```

```

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)/.vnc

# Create VNC password if required.
if ! test -f "$(id -un)/.vnc/passwd"; then
    echo "creating VNC password"
    $DOCKER_COMMAND run --rm -it \
        -u $(id -u):$(id -g) \
        -v /etc/passwd:/etc/passwd:ro \
        -v $PWD/$(id -un):/home/$(id -un) \
        $IMAGE vncpasswd
fi

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    -p 5900:5900 \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \

```

```

-v /sys/fs/cgroup:/sys/fs/cgroup \
-v $PWD/$(id -un):/home/$(id -un) \
-v $PWD/$(id -un)/.vnc:/tmp/lightdm/.vnc \
-v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
-v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro
\
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
$IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

This script is rather similar to the original `ubuntu-18.04-gnome` launch script, but as with the Debian VNC example the launch script is extended to create the VNC password, expose port 5900 and bind-mount the `.vnc` directory.

The container may be launched remotely, supplying a username, host and path and using any VNC client to connect:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-virtual-desktops/vnc/ubuntu-18.04-gnome-tigervnc; ./ubuntu.sh"
```

Of particular note with this example is that it provides a remote **3D accelerated** desktop.

As with the Debian VNC example, this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by replacing Xephyr with a dummy, unused, Xvfb instance.

11-1-3 linuxmint-19-3-cinnamon-turbovnc

Although the TigerVNC example provides a fairly good remote 3D accelerated desktop experience, TurboVNC+VirtualGL claims to be optimised for 3D workloads. In this example, for added interest, we shall also use noVNC and Websockify to expose an HTML5 interface to our desktop.

The [Dockerfile](#) is in the [linuxmint-19.3-cinnamon-turbovnc](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/vnc](#).

```
FROM linuxmint-cinnamon-vgl:19.3

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    xfonts-base xfonts-75dpi xfonts-100dpi \
    xfonts-scalable python-numpy python-setuptools && \
    # Attempt to work out the latest turbovnc version from
    # https://sourceforge.net/projects/turbovnc/files/
    TVNC_VERSION=$(curl -sSL https://sourceforge.net/projects/turbovnc/files/ | grep "<span class=\"name\">[0-9]" | head -n 1 | \
    cut -d \> -f2 | cut -d \<; -f1) && \
    echo "turbovnc version: ${TVNC_VERSION}" && \
    curl -sSL https://sourceforge.net/projects/turbovnc/files/${TVNC_VERSION}/turbovnc_${TVNC_VERSION}_amd64.deb -o turbovnc_${TVNC_VERSION}_amd64.deb && \
    dpkg -i turbovnc_*_amd64.deb && \
    rm turbovnc_*_amd64.deb && \
    ln -snf /opt/TurboVNC/bin/Xvnc /usr/bin/Xvnc && \
```

```
# Download websockify
WS_VERSION=0.9.0 && \
curl -sSL https://github.com/novnc/websockify/archive/v${WS_V
ERSION}.tar.gz | tar -xzv -C /usr/local/bin && \
cd /usr/local/bin/websockify-${WS_VERSION} && \
python setup.py install && \
# Download noVNC
NOVNC_VERSION=1.1.0 && \
NOVNC=/usr/local/bin/noVNC-${NOVNC_VERSION} && \
APP=/usr/local/bin/noVNC/app && \
curl -sSL https://github.com/novnc/noVNC/archive/v${NOVNC_VER
SION}.tar.gz | tar -xzv -C /usr/local/bin && \
# Temporarily install Node.js and npm to transpile
# ECMAScript 6 modules. As well as speeding up load times
# on browsers that don't support modules this circumvents
# an issue with pre 0.9.12 LibVNCServer where the MIME
# type for Javascript was set incorrectly causing Chrome
# to reject them due to strict MIME type checking being
# enabled for modules.
curl -sSL https://nodejs.org/dist/v10.16.3/node-v10.16.3-linu
x-x64.tar.xz | tar -xJv -C /usr/local/lib && \
NODE=/usr/local/lib/node-v10.16.3-linux-x64/bin && \
ln -s ${NODE}/node /usr/local/bin/node && \
ln -s ${NODE}/npm /usr/local/bin/npm && \
ln -s ${NODE}/npx /usr/local/bin/npx && \
npm install -g es6-module-transpiler && \
npm install -g @babel/core @babel/cli && \
npm install -g @babel/preset-env && \
ln -s ${NODE}/babel /usr/local/bin/babel && \
ln -s ${NODE}/compile-modules \
/usr/local/bin/compile-modules && \
cd ${NOVNC} && \
```

```
# Tweak the vnc.html to use the transpiled app.js
sed -i 's/type="module" crossorigin="anonymous" src="app\/ui.js"/src="app.js"/g' vnc.html && \
sed -i 's/<script src="vendor\/promise.js"><\\/script>//g' vnc.html && \
sed -i 's/if (window._noVNC_has_module_support) //g' vnc.html && \
# Transpile the Javascript to speed up loading
echo '{"presets": ["@babel/preset-env"]}' > .babelrc && \
npm install --save-dev @babel/core @babel/preset-env && \
compile-modules convert app/ui.js > app.js && \
babel app.js --out-file app.js && \
mkdir -p ${APP} && \
mv ${NOVNC}/app/images ${APP}/images && \
mv ${NOVNC}/app/locale ${APP}/locale && \
mv ${NOVNC}/app/sounds ${APP}/sounds && \
mv ${NOVNC}/app/styles ${APP}/styles && \
mv ${NOVNC}/app/error-handler.js \
${APP}/error-handler.js && \
mv ${NOVNC}/app.js ${APP}.js && \
mv ${NOVNC}/vnc.html \
/usr/local/bin/noVNC/index.html && \
rm -rf ${NOVNC} && \
rm -rf /usr/local/lib/node-v10.16.3-linux-x64 && \
rm -rf /root/.npm && \
rm /usr/local/bin/node && \
rm /usr/local/bin/npm && \
rm /usr/local/bin/npx && \
# Create systemd service to launch noVNC
echo '[Unit]\nDescription=HTML5 VNC WebSocket proxy\nAfter=sy
slog.target network.target\n\n[Service]\nUser=lightdm\nType=sim
ple\nExecStart=/usr/local/bin/websockify 5800 localhost:5900 --
web /usr/local/bin/noVNC\nTimeoutStopSec=20\nKillMode=process\n
Restart=always\nRestartSec=2\n\n[Install]\nWantedBy=multi-user.
target\nAlias=websocket.service\n' > /lib/systemd/system/websoc
ket.service && \
```

```
ln -snf /lib/systemd/system/websocket.service \
        /etc/systemd/system/multi-user.target.wants/websocket.serv
ice && \
    # Create Turbovnc-lightdm-wrapper
    echo '#!/bin/bash\nRFBPORT=$((5900+${1#::}))\nXvnc $1 -ac -noI
isten tcp -localhost -rfbport $RFBPORT -rfbauth /tmp/lightdm/.v
nc/passwd&\nXVNC_PID=$!\ncleanup() {\n    kill -TERM $XVNC_PID\n}
\ntrap cleanup SIGINT SIGTERM EXIT\nsleep 0.25\nkill -USR1 $PPI
D\nsocat - TCP:localhost:$RFBPORT' > /usr/bin/Turbovnc-lightdm-
wrapper && \
    chmod +x /usr/bin/Turbovnc-lightdm-wrapper && \
    echo '[LightDM]\nminimum-display-number=1\n[Seat:0]\nxserver-
command=/usr/bin/Xephyr-lightdm-wrapper\n[VNCServer]\nenabled=t
rue\ndepth=24\ncommand=Turbovnc-lightdm-wrapper' > /etc/lightdm
/lightdm.conf.d/70-linuxmint.conf
```

The image may be built **on the remote host** with:

```
docker build -t linuxmint-cinnamon-turbovnc:19.3 .
```

Although fairly long and complex, the majority of this Dockerfile actually revisits work from previous examples.

First TurboVNC is installed following the process described in the [turbovnc](#) section, where we attempt to “scrape” the most recent TurboVNC version from SourceForge then download and install the TurboVNC deb package. Because TurboVNC installs to /opt/TurboVNC/bin/Xvnc we also symlink to that from /usr/bin/Xvnc otherwise LightDM complains.

We next download and install Websockify, then we install noVNC using the process described in the [noVNC](#) section to install Node.js and use es6-module-transpiler and babel to transpile noVNC into a single JavaScript bundle app.js.

To expose our noVNC HTML5 endpoint, we create a systemd service in /lib/systemd/system/websocket.service.

```
[Unit]
Description=HTML5 VNC WebSocket proxy
After=syslog.target network.target

[Service]
User=lightdm
Type=simple
ExecStart=/usr/local/bin/websockify 5800 localhost:5900 --web /
usr/local/bin/noVNC
TimeoutStopSec=20
KillMode=process
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
Alias=websocket.service
```

The service is then enabled, so it will be launched at system startup as a dependency of multi-user.target.

```
ln -snf /lib/systemd/system/websocket.service \
/etc/systemd/system/multi-user.target.wants/websocket.servic
e
```

In our earlier VNC examples, we could simply use Xvnc as the command in the VNCServer stanza of the LightDM configuration. LightDM actually transparently adds the `-inetd` option, which configures Xvnc to use stdin/stdout for its I/O rather than listening on the RFB socket itself. As TurboVNC also has this option, the author naively assumed that the same approach could be used here.

Unfortunately the `-inetd` option for TurboVNC doesn't seem to work, so we create a `Turbovnc-lightdm-wrapper` script to work around the problem.

```
#!/bin/bash
RFBPORT=$((5900+${1#:}))
Xvnc $1 -ac -nolisten tcp -localhost -rfbport $RFBPORT -rfbauth
/tmp/lightdm/.vnc/passwd&
XVNC_PID=$!
cleanup() {
    kill -TERM $XVNC_PID
}
trap cleanup SIGINT SIGTERM EXIT
sleep 0.25
kill -USR1 $PPID
socat - TCP:localhost:$RFBPORT
```

This script is triggered by connections to port 5900 and gets passed the X11 DISPLAY number by LightDM, so we first use that to choose a suitable RFB port then launch TurboVNC with the specified DISPLAY and chosen RFB port.

We next get the PID of the TurboVNC server that we've just launched and add a trap so that it will be cleanly killed when the parent script is terminated. After a brief sleep, to ensure that TurboVNC is indeed running, we call `kill -USR1 $PPID` which is the signal sent by the X Server to LightDM to signal that it is running.

The final part of the script attempts to emulate the Xvnc `-inetd` option. It uses socat as a proxy, taking its input from stdin and redirecting that to our TurboVNC server. listening on our chosen RFB port.

The LightDM configuration for Linux Mint is in /etc/lightdm/lightdm.conf.d/70-linuxmint.conf and simply uses Turbovnc-lightdm-wrapper as the VNCServer command.

```
[LightDM]
minimum-display-number=1
[Seat:*]
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
[VNCServer]
enabled=true
depth=24
command=Turbovnc-lightdm-wrapper
```

The [mint.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

IMAGE=linuxmint-cinnamon-turbovnc:19.3
CONTAINER=mint-turbovnc

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)/.vnc
```

```

# Create VNC password if required.
if ! test -f "$(id -un) /.vnc/passwd"; then
    echo "creating VNC password"
    $DOCKER_COMMAND run --rm -it \
    -u $(id -u):$(id -g) \
    -v /etc/passwd:/etc/passwd:ro \
    -v $PWD/$(id -un):/home/$(id -un) \
    $IMAGE /opt/TurboVNC/bin/vncpasswd
fi

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    -p 5900:5900 \
    -p 5800:5800 \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $PWD/$(id -un)/.vnc:/tmp/lightdm/.vnc \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro
\
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
$IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

This script is nearly identical to our `ubuntu-18-04-gnome-tigervnc` launch script, aside from obvious things like the container and image names and small details like the path of the `vncpasswd` call and exposing port 5800 in addition to port 5900 to expose the HTML5 Web Application.

The container may be launched remotely, supplying a username, host and path and using any VNC client to connect, or simply navigate a browser to <http://remote-host:5800>:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-virtual-desktops/vnc/linuxmint-19.3-cinnamon-turbovnc; ./mint.sh"
```

As with the Debian and Ubuntu VNC examples, this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by replacing Xephyr with a dummy, unused, Xvfb instance.

11-1-4 fedora-32-kde-turbovnc

The [Dockerfile](#) is in the [fedora-32-kde-turbovnc](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/vnc](#).

The Dockerfile for `fedora-32-kde-turbovnc` and the `fedora.sh` launch script are very similar to those for our `linuxmint-19-3-cinnamon-turbovnc` virtual desktop, aside from some trivial differences like the image and container names and using `dnf` instead of `apt`, so won't be explored in depth here as there is not much more to add.

11-2 RDP

When we covered [RDP](#) in our [Remote Applications](#) chapter we noted that, unlike VNC, RDP logs in a remote user by effectively creating a real desktop session on the server. Because of this, we had to go to some lengths in that section to configure RDP to behave more like an *application* server.

Because RDP launches real desktop sessions, the expectation was for this chapter to be quite straightforward, but by default xrdp doesn't work well with some 3D window managers and requires patching and configuration to work at all.

11-2-1 ubuntu-18-04-gnome-xrdp

The [Dockerfile](#) is in the [ubuntu-18.04-gnome-xrdp](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/rdp](#).

```
FROM ubuntu-gnome-vgl:18.04

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    git autoconf libtool software-properties-common \
    libssl-dev libpam0g-dev nasm xsldproc flex bison \
    pkg-config libcap-dev libpulse-dev libudev-dev \
    intltool libltdl-dev libsndfile-dev bash-completion \
    libsystemd-dev libdbus-1-dev libspeexdsp-dev autopoint \
    libx11-dev xserver-xorg-dev xserver-xorg-core \
    libxfixes-dev libxrandr-dev libxml2-dev dpkg-dev \
    libmp3lame-dev libopus-dev libfdk-aac-dev libjpeg-dev \
    libturbojpeg0-dev libpixman-1-dev libfuse-dev \
    xfonts-base xfonts-75dpi xfonts-100dpi \
    xfonts-scalable xauth && \
```

```

# Clone and build xrdp and xorgxrdp from GitHub.
cd /usr/src && \
git clone --recursive \
    https://github.com/neutrinoLabs/xrdp.git && \
git clone \
    https://github.com/neutrinoLabs/xorgxrdp.git && \
git clone \
    https://github.com/pulseaudio/pulseaudio.git && \
git clone \
    https://github.com/neutrinoLabs/pulseaudio-module-xrdp.gi
t && \
cd xrdp && git checkout v0.9.13 -b build && \
./bootstrap && \
./configure --enable-opus --enable-fuse --enable-jpeg \
    --enable-tjpeg --enable-ipv6 --enable-vssock \
    --enable-pixman --enable-mp3lame \
    --enable-fdkaac --enable-rdpssndaudin && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
cd ../xorgxrdp && git checkout v0.2.13 -b build && \
# Need to patch xorgxrdp as GNOME is picky about RandR
# and an xorgxrdp update #153 causes RandR to fail, see:
# After update to v0.2.12 GNOME shell does not start
# https://github.com/neutrinoLabs/xorgxrdp/issues/156
sed -i 's/LLOGLN(0, ("rdpRRScreenSetSize: not allowing resize
"));if ((width == pScreen->width) && (height == pScreen->hei
ght) && (mmWidth == pScreen->mmWidth) && (mmHe
ight == pScreen->mmHeight))\n            {\n                LLOGLN(0, (
"rdpRRScreenSetSize: already this size"));}\n            return
TRUE;\n        }' \
    LLOGLN(0, ("rdpRRScreenSetSize: not a
llowing resize"));' /usr/src/xorgxrdp/module/rdpRandR.c && \
./bootstrap && ./configure && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
sed -i 's/ssl_protocols=TLSv1.2, TLSv1.3/ssl_protocols=TLSv1.
2/' \
    /etc/xrdp/xrdp.ini && \

```

```

# The following lines are needed to allow 3D WM to start.
sed -i 's/readenv=1/readenv=1\nexport XAUTHORITY=$HOME\/.Xauthority.docker/' \
    /etc/xrdp/startwm.sh && \
    sed -i 's/\.\ \\\etc\\\X11\\\Xsession/export GNOME_SHELL_SESSION_MODE=ubuntu\n      export XDG_CURRENT_DESKTOP=ubuntu:GNOME\n      export VGL_WM=1\n      dbus-launch --sh-syntax --exit-with-session x-session-manager --disable-acceleration-check/' /etc/xrdp/star
twm.sh && \
    sed -i 's/param=-config/param=-ac\nparam=-config/' \
        /etc/xrdp/sesman.ini && \
# Build xrdp source / sink modules
cd ../pulseaudio && git checkout v11.1 -b build && \
    ./autogen.sh && \
    cd ../pulseaudio-module-xrdp && \
    git checkout v0.4 -b build && \
    ./bootstrap && \
    ./configure PULSE_DIR=/usr/src/pulseaudio && \
    make -j$(getconf _NPROCESSORS_ONLN) && make install && \
# Modify PulseAudio daemon config to support xrdp.
echo "load-module module-xrdp-sink" >> \
    /etc/pulse/default.pa && \
echo "load-module module-xrdp-source" >> \
    /etc/pulse/default.pa && \
# From systemctl enable xrdp
ln -snf /lib/systemd/system/xrdp.service \
    /etc/systemd/system/multi-user.target.wants/xrdp.service

```

The image may be built **on the remote host** with:

```
docker build -t ubuntu-gnome-xrdp:18.04 .
```

The Dockerfile begins in a similar way to our earlier [xrdp-native-audio](#) by installing a range of development packages and cloning the xrdp and PulseAudio repositories.

After configuring and building xrdp we must patch xorgxrdp prior to building it. GNOME requires X11 RandR support, but an xorgxrdp update #153 causes RandR to fail, see: “After update to v0.2.12 GNOME shell does not start”

<https://github.com/neutrinolabs/xorgxrdp/issues/156>.

```
sed -i 's/LLOGLN(0, ("rdpRRScreenSetSize: not allowing resize")\n);/if ((width == pScreen->width) && (height == pScreen->height) &&\n        (mmWidth == pScreen->mmWidth) && (mmHeight == pScreen->mmHeight))\n    {\n        LLOGLN(0, ("rdpRRScreenSetSize: already this size"));}\n    return TRUE;\n}\nLLOGLN(0, ("rdpRRScreenSetSize: not allowing resize"));' /usr/src/xorgxrdp/module/rdpRandR.c
```

After building and installing our patched xorgxrdp, we must modify the configuration in /etc/xrdp/startwm.sh so xrdp will allow 3D window managers to start.

```
sed -i 's/readenv=1/readenv=1\nexport XAUTHORITY=$HOME/.Xauthority.docker/' \n/etc/xrdp/startwm.sh\n\nsed -i 's/\. \\\'/etc\\/X11\\/Xsession/export GNOME_SHELL_SESSION_MODE=ubuntu\n        export XDG_CURRENT_DESKTOP=ubuntu:GNOME\n        export VGL_WM=1\n        dbus-launch --sh-syntax --exit-with-session x\n        -session-manager --disable-acceleration-check/' /etc/xrdp/startwm.sh
```

The first of these lines sets the XAUTHORITY environment variable to point to the 3D X Server on the host.

The second line replaces the call to /etc/X11/Xsession in /etc/xrdp/startwm.sh with:

```
export GNOME_SHELL_SESSION_MODE=ubuntu
export XDG_CURRENT_DESKTOP=ubuntu:GNOME
export VGL_WM=1
dbus-launch --sh-syntax --exit-with-session x-session-manager -
--disable-acceleration-check
```

This configuration sets the environment variables required to run the Ubuntu GNOME desktop and uses `dbus-launch` to run `x-session-manager`. Both the `dbus-launch` and `--disable-acceleration-check` are necessary for xrdp to correctly launch GNOME.

It is also necessary to modify /etc/xrdp/sesman.ini to add `-ac` to the X Server launch. This disables authentication to the 2D X Server launched by xrdp, which isn't ideal but a relatively minor concern as it is bound to a Unix domain socket only visible to the container.

```
sed -i 's/param=-config/param=-ac\nparam=-config/' \
/etc/xrdp/sesman.ini
```

After configuring xrdp the Dockerfile adds support for audio redirection by building the xrdp-source and xrdp-sink PulseAudio modules based on the instructions found in the xrdp Wiki:
<https://github.com/neutrinolabs/pulseaudio-module-xrdp/wiki/README>.

Finally, the Dockerfile enables the systemd xrdp service.

The [ubuntu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

IMAGE=ubuntu-gnome-xrdp:18.04
CONTAINER=ubuntu-xrdp

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for c
ontainer."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home
'$$(id -un)'; usermod -aG sudo '$$(id -un)'; tar zcf /mnt/etc.tar
.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    -p 3389:3389 \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SADMIN --cap-add=SBOOT \
```

```

-v /sys/fs/cgroup:/sys/fs/cgroup \
-v $PWD/$(id -un):/home/$(id -un) \
-v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
-v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro
\
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
$IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

The launch script is identical to the basic 3D accelerated ubuntu-18.04-gnome launch script except for the container and image names and exporting the RDP port 3389.

The container may be launched remotely, supplying a username, host and path and using any RDP client to connect:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-virtual-desktops/rdp/ubuntu-18.04-gnome-xrdp; ./ubuntu.sh"
```

Although this RDP example is a little complex. it has the benefit of providing native audio forwarding in addition to a remote 3D accelerated desktop.

As with the VNC examples, this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by removing 70-ubuntu.conf, which configured LightDM with Xephyr.

11-2-2 linuxmint-19-3-cinnamon-xrdp

Because Linux Mint is derived from Ubuntu, simply changing the base image should have been enough to create a remote Mint desktop. That is certainly true of the VNC examples.

Unfortunately, with RDP that's not the case, though RDP on Linux Mint is slightly simpler than with Ubuntu. The [Dockerfile](#) is in the [linuxmint-19.3-cinnamon-xrdp](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/rdp](#).

```
FROM linuxmint-cinnamon-vgl:19.3

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    git autoconf libtool software-properties-common \
    libssl-dev libpam0g-dev nasm xsllibproc flex bison \
    pkg-config libcap-dev libpulse-dev libudev-dev \
    intltool libltdl-dev libsndfile-dev bash-completion \
    libsystemd-dev libdbus-1-dev libspeexdsp-dev autopoint \
    libx11-dev xserver-xorg-dev xserver-xorg-core \
    libxfixes-dev libxrandr-dev libxml2-dev dpkg-dev \
    libmp3lame-dev libopus-dev libfdk-aac-dev libjpeg-dev \
    libturbojpeg0-dev libpixman-1-dev libfuse-dev \
    xfonts-base xfonts-75dpi xfonts-100dpi \
    xfonts-scalable xauth && \
# Clone and build xrdp and xorgxrdp from GitHub.
cd /usr/src && \
git clone --recursive \
    https://github.com/neutrino-labs/xrdp.git && \
git clone \
    https://github.com/neutrino-labs/xorgxrdp.git && \
```

```

git clone \
    https://github.com/pulseaudio/pulseaudio.git && \
git clone \
    https://github.com/neutrino-labs/pulseaudio-module-xrdp.gi
t && \
cd xrdp && git checkout v0.9.13 -b build && \
./bootstrap && \
./configure --enable-opus --enable-fuse --enable-jpeg \
    --enable-tjpeg --enable-ipv6 --enable-vsock \
    --enable-pixman --enable-mp3lame \
    --enable-fdkaac --enable-rdpsndaudin && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
cd ../../xorgxrdp && git checkout v0.2.13 -b build && \
# Hack to make gb layout the default if client fails
# to send correct keylayout (Vinagre seems to do that).
# https://github.com/neutrino-labs/xrdp/issues/337
# Replace pc105 and gb with required model and layout.
sed -i 's/set.model = g_pc104_str;/set.model = "pc105";/' \
    /usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c && \
sed -i 's/set.layout = g_us_str;/set.layout = "gb";/' \
    /usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c && \
sed -i 's/strlen(client_info->layout)/0/' \
    /usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c && \
./bootstrap && ./configure && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
sed -i 's/ssl_protocols=TLSv1.2, TLSv1.3/ssl_protocols=TLSv1.
2/' \
    /etc/xrdp/xrdp.ini && \
# The follows lines are needed to allow 3D WM to start
sed -i 's/readenv=1/readenv=1\nexport XAUTHORITY=$HOME\/.Xaut
hority.docker/' \
    /etc/xrdp/startwm.sh && \
sed -i 's/param=-config/param=-ac\nparam=-config/' \
    /etc/xrdp/sesman.ini && \
# Build xrdp source / sink modules
cd ../../pulseaudio && git checkout v11.1 -b build && \
./autogen.sh && \

```

```
cd ../pulseaudio-module-xrdp && \
git checkout v0.4 -b build && \
./bootstrap && \
./configure PULSE_DIR=/usr/src/pulseaudio && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
# Modify PulseAudio daemon config to support xrdp.
echo "load-module module-xrdp-sink" >> \
    /etc/pulse/default.pa && \
echo "load-module module-xrdp-source" >> \
    /etc/pulse/default.pa && \
# From systemctl enable xrdp
ln -snf /lib/systemd/system/xrdp.service \
    /etc/systemd/system/multi-user.target.wants/xrdp.service
```

The image may be built **on the remote host** with:

```
docker build -t linuxmint-cinnamon-xrdp:19.3 .
```

The Dockerfile starts the same as our ubuntu-18-04-gnome-xrdp one, however as Cinnamon does not fail without X11 RandR support we don't need the patch that was required for GNOME.

A different issue was encountered however, where some clients fail to send the correct keylayout. This example illustrates a “brute force” way to work around this issue by patching the xorgxrdp source code prior to building.

```
sed -i 's/set.model = g_pc104_str;/set.model = "pc105";/' \
    /usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c
sed -i 's/set.layout = g_us_str;/set.layout = "gb";/' \
    /usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c
sed -i 's/strlen(client_info->layout)/0/' \
    /usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c
```

Patching the source code is a very inelegant workaround and ideally it should be possible to modify configuration in /etc/xrdp/xrdp_keyboard.ini, however none of the configuration changes worked for the author and it seems to be a known issue: <https://github.com/neutrinoLabs/xrdp/issues/337>.

Like the Ubuntu GNOME example, after building and installing our patched xorgxrdp we must modify the configuration in /etc/xrdp/startwm.sh so that xrdp will allow Cinnamon to start. For Cinnamon we only need to set the XAUTHORITY environment variable to point to the 3D X Server on the host.

```
sed -i 's/readenv=1/readenv=1\nexport XAUTHORITY=$HOME/.Xauthority.docker/' \
/etc/xrdp/startwm.sh
```

We again need to modify /etc/xrdp/sesman.ini to include `-ac` so that we disable authentication to the 2D X Server launched by xrdp.

```
sed -i 's/param=-config/param=-ac\nparam=-config/' \
/etc/xrdp/sesman.ini
```

The rest of the Dockerfile is the same as the Ubuntu one.

The [mint.sh](#) launch script is identical to the Ubuntu RDP launch script except for the image and container names.

The container may be launched remotely, supplying a username, host and path and using any RDP client to connect:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-virtual-desktops/rdp/linuxmint-19.3-cinnamon-xrdp; ./mint.sh"
```

As with the VNC examples this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by removing 70-linuxmint.conf, which configured LightDM with Xephyr.

11-2-3 centos-7-gnome-xrdp

The [Dockerfile](#) is in the [centos-7-gnome-xrdp](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/rdp](#).

```
FROM centos-gnome-vgl:7.7

RUN yum -y install \
    cmake patch gcc make autoconf libtool automake \
    pkgconfig openssl-devel gettext file pam-devel \
    git libX11-devel libXfixes-devel libjpeg-devel \
    libXrandr-devel nasm flex bison gcc-c++ libxslt \
    perl-libxml-perl xorg-x11-font-utils xmlto-tex \
    fuse-devel opus-devel lame-devel pixman-devel \
    turbojpeg-devel xorg-x11-server-devel intltool \
    libtool-ltdl-devel libcap-devel libsndfile-devel \
    speex-devel libudev-devel dbus-devel rpmdevtools \
    pulseaudio-libs-devel libXfont2-devel \
    xorg-x11-fonts-base xorg-x11-fonts-75dpi \
    xorg-x11-fonts-100dpi && \
```

```

# Clone and build xrdp and xorgxrdp from GitHub.
cd /usr/src && \
# Need to get fdk-aac from different repo.
curl -SSL https://li.nux.ro/download/nux/dextop/el7/x86_64/fd
k-aac-0.1.4-1.x86_64.rpm -o fdk-aac-0.1.4-1.x86_64.rpm && \
curl -SSL https://li.nux.ro/download/nux/dextop/el7/x86_64/fd
k-aac-devel-0.1.4-1.x86_64.rpm -o fdk-aac-devel-0.1.4-1.x86_64.
rpm && \
rpm -i fdk-aac-0.1.4-1.x86_64.rpm fdk-aac-devel-0.1.4-1.x86_6
4.rpm && \
git clone --recursive \
https://github.com/neutrinoLabs/xrdp.git && \
git clone \
https://github.com/neutrinoLabs/xorgxrdp.git && \
git clone \
https://github.com/neutrinoLabs/pulseaudio-module-xrdp.gi
t && \
cd xrdp && git checkout v0.9.13 -b build && \
./bootstrap && \
./configure --enable-opus --enable-fuse --enable-jpeg \
--enable-tjpeg --enable-ipv6 --enable-vsock \
--enable-pixman --enable-mp3lame \
--enable-fdkaac --enable-rdpsndaudin && \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
cd ../xorgxrdp && git checkout v0.2.13 -b build && \
# Hack to make gb layout the default if client fails
# to send correct keylayout (Vinagre seems to do that).
# https://github.com/neutrinoLabs/xrdp/issues/337
# Replace pc105 and gb with required model and layout.
sed -i 's/set.model = g_pc104_str;/set.model = "pc105";/' \
/usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c && \
sed -i 's/set.layout = g_us_str;/set.layout = "gb";/' \
/usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c && \
sed -i 's/strlen(client_info->layout)/0/' \
/usr/src/xorgxrdp/xrdpkeyb/rdpKeyboard.c && \

```

```

# Need to patch xorgxrdp as GNOME is picky about RandR
# and an xorgxrdp update #153 causes RandR to fail, see:
# After update to v0.2.12 GNOME shell does not start
# https://github.com/neutrinolabs/xorgxrdp/issues/156
sed -i 's/LLOGLN(0, ("rdpRRScreenSetSize: not allowing resize
"));if ((width == pScreen->width) && (height == pScreen->hei
ght) &&\n                (mmWidth == pScreen->mmWidth) &&\n                (mmHe
ight == pScreen->mmHeight))\n                {\n                    LLOGLN(0, (
"rdpRRScreenSetSize: already this size"));\n                return
TRUE;\n            }\n            LLOGLN(0, ("rdpRRScreenSetSize: not a
llowing resize"));' /usr/src/xorgxrdp/module/rdpRandR.c && \
./bootstrap && ./configure XRDP_CFLAGS=-I/usr/local/include &
& \
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
sed -i 's/ssl_protocols=TLSv1.2, TLSv1.3/ssl_protocols=TLSv1.
2/' \
/etc/xrdp/xrdp.ini && \
# The follows lines are needed to allow 3D WM to start
# GNOME is particularly picky needing lots of environment
# variables, dbus-launch and --disable-acceleration-check
sed -i 's/readenv=1/readenv=1\nexport XAUTHORITY=$HOME/.Xaut
hority.docker/' \
/etc/xrdp/startwm.sh && \
sed -i 's/\. \\\etc\\X11\\xinit\\Xsession/export GNOME_SHELL_S
ESSION_MODE=classic\n    export XDG_CURRENT_DESKTOP=GNOME-Class
ic:GNOME\n    export VGL_WM=1\n    dbus-launch --sh-syntax --ex
it-with-session gnome-session --session gnome-classic --disable
-acceleration-check' /etc/xrdp/startwm.sh && \
sed -i 's/param=-config/param=-ac\nparam=-config/' \
/etc/xrdp/sesman.ini && \

```

```
# For CentOS pulseaudio modules must be built with
# CentOS's pulseaudio source and the module builds,
# but won't load, when using the pulseaudio git repo.
# https://github.com/neutrinoLabs/xrdp/issues/1178
# https://github.com/neutrinoLabs/pulseaudio-module-xrdp/wiki
/README
cd /usr/src && \
yumdownloader --source pulseaudio && \
rpm --install pulseaudio*.src.rpm && \
yum-builddep -y pulseaudio && \
rpmbuild -bb --noclean /root/rpmbuild/SPECS/pulseaudio.spec &
& \
# Build xrdp source / sink modules
cd pulseaudio-module-xrdp && \
git checkout v0.4 -b build && \
./bootstrap && \
./configure PULSE_DIR=/root/rpmbuild/BUILD/pulseaudio-10.0 &&
\
make -j$(getconf _NPROCESSORS_ONLN) && make install && \
# Modify PulseAudio daemon config to support xrdp.
echo "load-module module-xrdp-sink" >> \
/etc/pulse/default.pa && \
echo "load-module module-xrdp-source" >> \
/etc/pulse/default.pa && \
# systemctl enable xrdp
ln -snf /usr/lib/systemd/system/xrdp.service \
/etc/systemd/system/multi-user.target.wants/xrdp.service
```

The image may be built **on the remote host** with:

```
docker build -t centos-gnome-xrdp:7.7 .
```

The centos-7-gnome-xrdp Dockerfile follows the pattern of our
ubuntu-18-04-gnome-xrdp and linuxmint-19-3-cinnamon-xrdp
Dockerfiles quite closely.

We again install a range of development packages that are required to build xrdp from source, though of course for CentOS we use yum rather than apt and the package names are slightly different.

A more significant package difference is that the fdk-aac and fdk-aac-devel packages are not available in the main CentOS repositories, so we have to download those from a separate repository and install via

```
rpm -i .
```

With CentOS we experienced the same keylayout issue as Linux Mint, so we apply the same “brute force” workaround to the xorgxrdp source code. As we are using GNOME, we also apply the X11 RandR patch described in [ubuntu-18.04-gnome-xrdp](#) prior to building xorgxrdp.

Like the Ubuntu GNOME example, after building and installing our patched xorgxrdp we must modify the configuration in /etc/xrdp/startwm.sh so that xrdp will allow GNOME to start correctly. We again need to set the XAUTHORITY environment variable to point to the 3D X Server on the host, then replace the call to /etc/X11/Xsession in /etc/xrdp/startwm.sh with:

```
export GNOME_SHELL_SESSION_MODE=classic
export XDG_CURRENT_DESKTOP=GNOME-Classic:GNOME
export VGL_WM=1
dbus-launch --sh-syntax --exit-with-session gnome-session --session gnome-classic --disable-acceleration-check
```

This is similar to our Ubuntu example, though this time we are launching a gnome-classic session.

We complete our xrdp configuration by again modifying /etc/xrdp/sesman.ini to include `-ac` so that we disable authentication to the 2D X Server launched by xrdp.

A notable difference with our CentOS Dockerfile is the process for building the PulseAudio modules, which must be built with CentOS's PulseAudio source. The modules build, but won't load using the code from the PulseAudio GitHub repository:

<https://github.com/neutrino-labs/xrdp/issues/1178>.

```
yumdownloader --source pulseaudio  
rpm --install pulseaudio*.src.rpm  
yum-builddep -y pulseaudio  
rpmbuild -bb --noclean /root/rpmbuild/SPECS/pulseaudio.spec
```

The `centos.sh` launch script is nearly identical to the Ubuntu and Linux Mint RDP launch scripts except for the image and container names and the subtle difference to the adduser command that creates the credentials bundle.

The container may be launched remotely, supplying a username, host and path and using any RDP client to connect:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-  
virtual-desktops/rdp/centos-7-gnome-xrdp; ./centos.sh"
```

As with the other remote virtual desktop examples, this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by removing `70-centos.conf`, which configured LightDM with Xephyr.

11-3 NX

As with the [NX](#) section in the [Remote Applications](#) chapter, in this section we shall cover examples for both [X2Go](#) and [NoMachine](#). Fortunately, all of the examples in this section are significantly more straightforward than the xrdp examples of our previous section.

11-3-1 ubuntu-18-04-gnome-x2go

The [Dockerfile](#) is in the [ubuntu-18.04-gnome-x2go](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

```
FROM ubuntu-gnome-vgl:18.04

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    x2goserver x2goserver-xsession && \
    sed -i 's/use-ssh-agent/no-use-ssh-agent/' \
    /etc/X11/Xsession.options
```

The image may be built **on the remote host** with:

```
docker build -t ubuntu-gnome-x2go:18.04 .
```

This Dockerfile simply installs x2goserver and x2goserver-xsession from the Ubuntu repositories. The only thing noteworthy is the patch to modify /etc/X11/Xsession.options to disable ssh-agent, as described in <https://bugs.launchpad.net/ubuntu/+source.openssh/+bug/47958>.

The [ubuntu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

IMAGE=ubuntu-gnome-x2go:18.04
CONTAINER=ubuntu-x2go

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    -p 2222:22 \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
```

```

-v /sys/fs/cgroup:/sys/fs/cgroup \
-v $PWD/$(id -un):/home/$(id -un) \
-v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
-v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro
\
-v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
$IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

The launch script is identical to the basic 3D accelerated ubuntu-18.04-gnome launch script except for the container and image names and exporting the SSH port as port 2222.

The container may be launched remotely, supplying a username, host and path and using an X2Go client to connect:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-virtual-desktops/nx/ubuntu-18.04-gnome-x2go; ./ubuntu.sh"
```

Client instructions are in the [x2goclient](#) section of the [NX](#) chapter. The most important detail is to select the **Session type** labelled *Connect to local desktop* or *X2go/X11 Desktop Sharing* depending on version. Also, note that X2Go doesn't resize on session launch, so disconnecting and reconnecting may be required to get a full-sized session window.

As with the other remote virtual desktop examples, this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by replacing Xephyr with [xorg-video-dummy](#).

11-3-2 linuxmint-19-3-cinnamon-x2go

The [Dockerfile](#) is in the [linuxmint-19.3-cinnamon-x2go](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

The Dockerfile for linuxmint-19.3-cinnamon-x2go and the [mint.sh](#) launch script are identical to those for our ubuntu-18-04-gnome-x2go desktop aside from some trivial differences like the image and container names and so won't be explored in depth here, as there is not much more to add.

11-3-3 centos-7-gnome-x2go

The [Dockerfile](#) is in the [centos-7-gnome-x2go](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

```
FROM centos-gnome-vgl:7.7

RUN yum -y install \
    x2goserver x2goserver-xsession
```

The image may be built **on the remote host** with:

```
docker build -t centos-gnome-x2go:7.7 .
```

Clearly this Dockerfile is very straightforward, simply installing the x2goserver and x2goserver-xsession packages from the CentOS repositories.

The [centos.sh](#) launch script is nearly identical to the Ubuntu and Linux Mint X2Go launch scripts except for the image and container names and the subtle difference to the adduser command that creates the credentials bundle.

11-3-4 fedora-32-kde-x2go

The [Dockerfile](#) is in the [fedora-32-kde-x2go](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

The Dockerfile for fedora-32-kde-x2go and the [fedora.sh](#) launch script are identical to those for our centos-7-gnome-x2go desktop aside from some trivial differences like the image and container names and using dnf instead of yum, so won't be explored in depth here as there is not much more to add.

11-3-5 ubuntu-18-04-gnome-nomachine

This section illustrates how to install the free version of the proprietary and closed source [NoMachine](#) to provide a high quality remote desktop with audio forwarding.

Important note: unlike the majority of applications illustrated in this book NoMachine is a commercial product. NoMachine is free to download and install for personal or individual use, including access to office computers, provided it is not to carry out activities that would be considered "commercial usage" as per the NoMachine [FAQ](#).

The [Dockerfile](#) is in the [ubuntu-18.04-gnome-nomachine](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

```
FROM ubuntu-gnome-vgl:18.04

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    xfonts-base xfonts-75dpi xfonts-100dpi \
    xfonts-scalable net-tools xauth && \
    # Install NoMachine binary and dependencies.
    # Ensure the nx user ID is not in the normal user range.
    groupadd -r -g 2000 nx && \
    useradd -u 2000 -r -g nx nx && \
    # Try to work out the latest version from the NoMachine
    # Linux download page. If this fails set the following
    # variables manually instead.
    NOMACHINE_VERSION=$(curl -sSL \
        "https://www.nomachine.com/download/download&id=3" | \
        grep "Linux/nomachine" | \
        cut -d \' -f2 | cut -d \_ -f2-3) && \
    NOMACHINE_MAJOR_VERSION=$(echo $NOMACHINE_VERSION | \
        cut -d \. -f1-2) && \
    echo "VERSION: ${NOMACHINE_VERSION}" && \
    echo "MAJOR_VERSION: ${NOMACHINE_MAJOR_VERSION}" && \
    curl -sSL https://download.nomachine.com/download/${NOMACHINE \
    _MAJOR_VERSION}/Linux/nomachine_${NOMACHINE_VERSION}_amd64.deb \
    -o nomachine_${NOMACHINE_VERSION}_amd64.deb && \
    dpkg -i nomachine_${NOMACHINE_VERSION}_amd64.deb && \
    rm nomachine_${NOMACHINE_VERSION}_amd64.deb && \
```

```
# Workaround for NoMachine configuring audio for
# the lightdm user but not for a normal user.
echo 'mkdir -p ~/.config/pulse\nif [ ! -f ~/.config/pulse/client.conf ]; then\ncp /etc/pulse/client.conf ~/.config/pulse/client.conf\nsed -i "s/; default-server =/default-server = unix:/\tmp/pulse-socket/g" ~/.config/pulse/client.conf\nfi' > /etc/profile.d/create-pulse-clientconf.sh && \
echo 'load-module module-native-protocol-UNIX auth-anonymous=1 socket=/tmp/pulse-socket' >> /etc/pulse/default.pa
```

The image may be built **on the remote host** with:

```
docker build -t ubuntu-gnome-nomachine:18.04 .
```

The Dockerfile begins in a similar way to the [nomachine](#) example described in the [Remote Applications](#) chapter.

As before, the Dockerfile installs net-tools to provide the required netstat command. It then adds the required nx user and group, then “scrapes” the most recent NoMachine version from the download page before downloading with curl and installing via `dpkg -i`.

The author was expecting it to be enough to largely repeat the instructions from the earlier [nomachine](#) section and indeed the remote desktop largely “just worked”. Unfortunately however, audio forwarding did not work as expected and took several days of effort to identify the following workaround.

The audio problem appears to be related to our use of the display manager, as NoMachine creates a user session for both the lightdm user and the regular user. There is a PulseAudio server started for the lightdm user session and NoMachine actually sets up audio forwarding for that server, but unfortunately it *does not* then go on to set up audio forwarding for the regular user.

Due to the closed source nature of NoMachine, the author unfortunately couldn't readily identify why audio forwarding was only established for the lightdm user session. Resolving that would obviously have been the most elegant solution.

In lieu of solving the *actual* problem, a workaround was identified that makes use of an example from
<https://wiki.archlinux.org/index.php/PulseAudio/Examples> describing how to allow multiple users to use PulseAudio at the same time.

It is sometimes desirable to run some programs as another user on the same desktop as the primary user in order to isolate the software. However, PulseAudio will not accept connections from the secondary users by default, since a PulseAudio daemon is already running for the primary user.

A new PulseAudio Unix domain socket can however be created in order to accept connections from other users to the main PulseAudio daemon run by the primary user.

To configure each user session to point to the shared Unix domain socket that will be created in /tmp/pulse-socket, we add the following script to /etc/profile.d/create-pulse-clientconf.sh to be run at user session startup.

```
mkdir -p ~/.config/pulse
if [ ! -f ~/.config/pulse/client.conf ]; then
    cp /etc/pulse/client.conf ~/.config/pulse/client.conf
    sed -i "s;/default-server =/default-server = unix:/tmp/pulse-socket/g" ~/.config/pulse/client.conf
fi
```

This script creates a client.conf for the user if one doesn't already exist and sets default-server = unix:/tmp/pulse-socket.

To create the shared Unix domain socket we modify /etc/pulse/default.pa as follows:

```
load-module module-native-protocol-unix auth-anonymous=1 socket=/tmp/pulse-socket
```

Now when the main user session starts it will actually use the PulseAudio daemon created for the lightdm user, which NoMachine has correctly set up for audio forwarding.

The [ubuntu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

IMAGE=ubuntu-gnome-nomachine:18.04
CONTAINER=ubuntu-nomachine
```

```

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for container."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home '$(id -un)'; usermod -aG sudo '$(id -un)'; tar zcf /mnt/etc.tar.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    -p 4000:4000 \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro \
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
    $GPU_FLAGS \
    $IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

The launch script is identical to the basic 3D accelerated ubuntu-18.04-gnome launch script except for the container and image names and exporting the NoMachine port 4000.

The container may be launched remotely, supplying a username, host and path and using the NoMachine client to connect:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-virtual-desktops/nx/ubuntu-18.04-gnome-nomachine; ./ubuntu.sh"
```

As with the other remote virtual desktop examples, this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by replacing Xephyr with [xorg-video-dummy](#).

11-3-6 linuxmint-19-3-cinnamon-nomachine

The [Dockerfile](#) is in the [linuxmint-19.3-cinnamon-nomachine](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

The Dockerfile for linuxmint-19.3-cinnamon-nomachine and the [mint.sh](#) launch script are identical to those for our ubuntu-18-04-gnome-nomachine desktop aside from some trivial differences like the image and container names and so won't be explored in depth here, as there is not much more to add.

11-3-7 centos-7-gnome-nomachine

The [Dockerfile](#) is in the [centos-7-gnome-nomachine](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

```
FROM centos-gnome-vgl:7.7

RUN yum -y install \
    xorg-x11-fonts-base xorg-x11-fonts-75dpi \
    xorg-x11-fonts-100dpi net-tools && \
    # Install NoMachine binary and dependencies.
    # Ensure the nx user ID is not in the normal user range.
    groupadd -r -g 2000 nx && \
    useradd -u 2000 -r -g nx nx && \
    # Try to work out the latest version from the NoMachine
    # Linux download page. If this fails set the following
    # variables manually instead.
NOMACHINE_VERSION=$(curl -sSL \
    "https://www.nomachine.com/download/download&id=3" | \
    grep "Linux/nomachine" | \
    cut -d \' -f2 | cut -d \_ -f2-3) && \
NOMACHINE_MAJOR_VERSION=$(echo $NOMACHINE_VERSION | \
    cut -d \_. -f1-2) && \
echo "VERSION: ${NOMACHINE_VERSION}" && \
echo "MAJOR_VERSION: ${NOMACHINE_MAJOR_VERSION}" && \
curl -sSL https://download.nomachine.com/download/${NOMACHINE \
_MAJOR_VERSION}/Linux/nomachine_${NOMACHINE_VERSION}_x86_64.rpm \
-o nomachine_${NOMACHINE_VERSION}_x86_64.rpm && \
rpm -i nomachine_${NOMACHINE_VERSION}_x86_64.rpm && \
rm nomachine_${NOMACHINE_VERSION}_x86_64.rpm && \
# Workaround for NoMachine configuring audio for
# the lightdm user but not for a normal user.
```

```
echo -e 'mkdir -p ~/.config/pulse\nif [ ! -f ~/.config/pulse/client.conf ]; then\n  cp /etc/pulse/client.conf ~/.config/pulse/client.conf\n  sed -i "s/; default-server =/default-server = unix :\tmp\pulse-socket/g" ~/.config/pulse/client.conf\nfi' > /etc/profile.d/create-pulse-clientconf.sh && \
echo 'load-module module-native-protocol-unix auth-anonymous=1 socket=/tmp/pulse-socket' >> /etc/pulse/default.pa
```

The image may be built **on the remote host** with:

```
docker build -t centos-gnome-nomachine:7.7 .
```

Aside from using yum this is identical to our Ubuntu example.

The [centos.sh](#) launch script is nearly identical to the Ubuntu and Linux Mint NoMachine launch scripts except for the image and container names and the subtle difference to the adduser command that creates the credentials bundle.

11-3-8 fedora-32-kde-nomachine

The [Dockerfile](#) is in the [fedora-32-kde-nomachine](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/nx](#).

The Dockerfile for fedora-32-kde-nomachine and the [fedora.sh](#) launch script are identical to those for our centos-7-gnome-nomachine desktop aside from some trivial differences like the image and container names and using dnf instead of yum, so won't be explored in depth here as there is not much more to add.

11-4 SPICE

Like NoMachine, SPICE provides a high quality remote desktop solution with audio forwarding, but unlike NoMachine it is free and open source. In this section we shall also incorporate the SPICE HTML5 client developed by eyeos and first encountered in the [xserver-xspice-eyeos](#) section, to illustrate a complete 3D accelerated remote desktop with audio forwarding available through a web browser.

11-4-1 ubuntu-18-04-gnome-spice

The [Dockerfile](#) is in the [ubuntu-18.04-gnome-spice](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/spice](#).

```
FROM ubuntu-gnome-vgl:18.04

RUN apt-get update && DEBIAN_FRONTEND=noninteractive \
    apt-get install -y --no-install-recommends \
    python-numpy python-setuptools \
    xserver-xspice-hwe-18.04 spice-vdagent && \
    # LightDM complains if Xvnc isn't present, even if
    # [VNCServer] config. specifies a command= option.
    ln -snf /usr/bin/Xspice /usr/bin/Xvnc && \
    mkdir /tmp/audio-fifo && \
    chmod 1777 /tmp/audio-fifo && \
    # Modify PulseAudio daemon config to support SPICE.
    echo "load-module module-pipe-sink file=/tmp/audio-fifo/playb
ack fifo format=s16 rate=48000 channels=2" >> /etc/pulse/default
.pa && \
```

```

# The Xspice --auto option is convenient, but if
# we have large display resolutions Xspice will
# crash with OOM. We need to modify the config in
# /etc/X11/spiceqxl.xorg.conf, which in turn means
# we can't use --auto and that exposes a bug in
# /usr/bin/Xspice where temp_dir isn't initialised.
sed -i 's/if args.auto:/temp_dir = ""\nif args.auto:/' /usr/b
in/Xspice && \
# Set buffers to handle larger resolutions.
sed -i 's/#Option "NumHeads" "4"/Option "NumHeads" "1"/' /etc
/X11/spiceqxl.xorg.conf && \
sed -i 's/#Option "SurfaceBufferSize" "128"/Option "SurfaceBu
fferSize" "512"/' /etc/X11/spiceqxl.xorg.conf && \
sed -i 's/#Option "CommandBufferSize" "128"/Option "CommandBu
fferSize" "512"/' /etc/X11/spiceqxl.xorg.conf && \
sed -i 's/#Option "FrameBufferSize" "16"/Option "FrameBufferS
ize" "32"/' /etc/X11/spiceqxl.xorg.conf && \
# If not using --auto the --audio-fifo-dir flag
# seems to be ignored, so we need to use the config.
sed -i 's/#Option "SpicePlaybackFIFODir" "\/tmp\//"/Option "Sp
icePlaybackFIFODir" "\/tmp\audio-fifo"/' /etc/X11/spiceqxl.xor
g.conf && \
# Download websockify and flexVDI fork of
# eyeos spice-web-client.
WS_VERSION=0.9.0 && \
SPICE=3.1.0 && \
curl -sSL https://github.com/novnc/websockify/archive/v${WS_V
ERSION}.tar.gz | tar -xv -C /usr/local/bin && \
curl -sSL https://github.com/flexVDI/spice-web-client/archive
/${SPICE}.tar.gz | tar -xv -C /usr/local/bin && \
cd /usr/local/bin/websockify-${WS_VERSION} && \
python setup.py install && \
mv /usr/local/bin/spice-web-client-${SPICE} \
/usr/local/bin/spice-web-client && \

```

```
# Tweak spice-web-client to add a basic
# host/port/password entry UI.
sed -i 's/document.location.port/port/g' \
    /usr/local/bin/spice-web-client/lib/utils.js && \
    sed -i 's/<div class="float-right">/<div class="float-right">
\n\n                <label for="host">Host: </label><input typ
e="text" id="host" value=""\/><label for="port"> Port: </label
><input type="text" id="port" value=""\/><label for="password">
Password: </label><input type="password" id="password" value="
" onkeyup="checkIfEnterPressed(event)"\>\n/g' /usr/local/bin/s
pice-web-client/index.html && \
    sed -i 's/$(document).ready(start);$(document).ready(init);/
g' /usr/local/bin/spice-web-client/run.js && \
    sed -i 's/translate();//g' \
        /usr/local/bin/spice-web-client/run.js && \
    sed -i 's/function start ()/\nfunction init () {\n    transla
te();\n    document.getElementById("showclientid").style.displa
y = "none";\n    document.getElementById("uploadfile").style.di
splay = "none";\n    document.getElementById("host").value = do
cument.location.hostname;\n    document.getElementById("port").
value = document.location.port;\n}\n\nfunction checkIfEnterPres
sed (event) {\n    if (event.keyCode === 13) {\n        start()
;\n    }\n}\n\nfunction start ()/g' /usr/local/bin/spice-web-clie
nt/run.js && \
    sed -i "s/data\[\'spice_address\'\] || ''/document.getElementById(
'host').value/g" /usr/local/bin/spice-web-client/run.js && \
    sed -i "s/data\[\'spice_port\'\] || 0/document.getElementById(
'port').value/g" /usr/local/bin/spice-web-client/run.js && \
    sed -i "s/data\[\'spice_password\'\] || ''/document.getElementById(
'password').value/g" /usr/local/bin/spice-web-client/run.js
&& \
```

```

# Create systemd service to launch html5 SPICE
echo '[Unit]\nDescription=HTML5 SPICE WebSocket proxy\nAfter=
syslog.target network.target\n[Service]\nUser=lightdm\nType=s
imple\nExecStart=/usr/local/bin/websockify 5800 localhost:5900
--web /usr/local/bin/spice-web-client\nTimeoutStopSec=20\nKillM
ode=process\nRestart=always\nRestartSec=2\n[Install]\nWantedB
y=multi-user.target\nAlias=websocket.service\n' > /lib/systemd/
system/websocket.service && \
ln -snf /lib/systemd/system/websocket.service \
/etc/systemd/system/multi-user.target.wants/websocket.serv
ice && \
# Script to launch Xspice "pretending" to be Xvnc
echo '#!/bin/bash\nif [ "$1" == ":2" ]; then\n  Xspice $1 -ac
-noreset -nolisten tcp --port 5902 --password $(cat /tmp/lightd
m/.xserver-xspice-passwd) --vdagent --video-codecs ${SPICE_VIDE
O_CODECS:-gstreamer:h264;gstreamer:vp8;gstreamer:mjpeg;spice:mj
peg} --deferred-fps 60 > /dev/null &\n  XSPICE_PID=$!\n  cleanu
p() {\n    kill -TERM $XSPICE_PID\n  }\n  trap cleanup SIGINT S
IGTERM EXIT\n  sleep 0.25\n  kill -USR1 $PPID\nfi\nsocat - TCP:
localhost:5902' > /usr/bin/Xspice-lightdm-wrapper && \
chmod +x /usr/bin/Xspice-lightdm-wrapper && \
echo '[LightDM]\nminimum-display-number=1\n[Seat: *]\nuser-ses
sion=ubuntu-xorg\nxserver-command=/usr/bin/Xephyr-lightdm-wrapp
er\n[VNCServer]\nenabled=true\ncommand=Xspice-lightdm-wrapper'
> /etc/lightdm/lightdm.conf.d/70-ubuntu.conf

VOLUME /tmp/audio-fifo

```

The image may be built **on the remote host** with:

```
docker build -t ubuntu-gnome-spice:18.04 .
```

With many things to configure, this Dockerfile is one of the more complex examples in the book, however by considering each section in turn it should be easier to digest, especially as many of the concepts have already been covered.

Very few additional packages are required by this Dockerfile. The SPICE specific packages are xserver-xspice-hwe-18.04 and spice-vdagent, where we use xserver-xspice-hwe-18.04 rather than xserver-xspice as it is a much newer version of SPICE that supports better codecs. The Python packages python-numpy and python-setuptools are used later by Websockify.

After installing the required packages we, slightly oddly, create a symbolic link from /usr/bin/Xvnc to /usr/bin/Xspice, which is used later when we “pretend” that the SPICE server is actually a VNC server.

We next set up SPICE audio forwarding, first by creating the audio FIFO directory:

```
mkdir /tmp/audio-fifo  
chmod 1777 /tmp/audio-fifo
```

Then modifying the PulseAudio daemon configuration in /etc/pulse/default.pa to support SPICE, by adding:

```
load-module module-pipe-sink file=/tmp/audio-fifo/playback.fifo  
format=s16 rate=48000 channels=2
```

to load the PulseAudio pipe-sink module, listening to the FIFO /tmp/audio-fifo/playback.fifo.

When we previously covered [xserver-xspice](#) we used the Xspice `--auto` option for convenience. However, for desktops with large display resolutions Xspice will crash with an out of memory error. To resolve this we need to modify the configuration in `/etc/X11/spiceqxl.xorg.conf`, which in turn means that we can't use `--auto` and that also exposes a bug in `/usr/bin/Xspice` where `temp_dir` isn't initialised.

First we patch `/usr/bin/Xspice` to fix the `temp_dir` bug.

```
sed -i 's/if args.auto:/temp_dir = ""\nif args.auto:/' /usr/bin/Xspice
```

Then we modify `/etc/X11/spiceqxl.xorg.conf` to set buffers to handle larger display resolutions.

```
sed -i 's/#Option "NumHeads" "4"/Option "NumHeads" "1"/' /etc/X11/spiceqxl.xorg.conf
sed -i 's/#Option "SurfaceBufferSize" "128"/Option "SurfaceBufferSize" "512"/' /etc/X11/spiceqxl.xorg.conf
sed -i 's/#Option "CommandBufferSize" "128"/Option "CommandBufferSize" "512"/' /etc/X11/spiceqxl.xorg.conf
sed -i 's/#Option "FrameBufferSize" "16"/Option "FrameBufferSize" "32"/' /etc/X11/spiceqxl.xorg.conf
```

Another Xspice quirk is that if the `--auto` option is not being used then the Xspice `--audio-fifo-dir` flag that is used to configure the audio FIFO path seems to be ignored, so we instead need to add it to `/etc/X11/spiceqxl.xorg.conf`.

```
sed -i 's/#Option "SpicePlaybackFIFODir" "\/tmp\/"/Option "SpicePlaybackFIFODir" "\/tmp\/audio-fifo"/' /etc/X11/spiceqxl.xorg.conf
```

After configuring Xspice, we download Websockify and [spice-web-client](#) via curl and tar then modify the eyeos/flexVDI spice-web-client exactly as described in [xserver-xspice-eyeos](#) to add a basic host/port/password entry UI.

After patching the spice-web-client, to expose our SPICE HTML5 endpoint we create a systemd service in /lib/systemd/system/websocket.service.

```
[Unit]
Description=HTML5 SPICE WebSocket proxy
After=syslog.target network.target

[Service]
User=lightdm
Type=simple
ExecStart=/usr/local/bin/websockify 5800 localhost:5900 --web /
usr/local/bin/spice-web-client
TimeoutStopSec=20
KillMode=process
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
Alias=websocket.service
```

The service is then enabled, so it will be launched at system startup as a dependency of multi-user.target.

```
ln -snf /lib/systemd/system/websocket.service \
/etc/systemd/system/multi-user.target.wants/websocket.servic
e
```

For most of our remote desktops it is trivial to launch both Xephyr and the remote server together and for many of the servers that is the default behaviour, but launching Xspice and Xephyr together is much more difficult and requires us to be quite “creative” with our solution.

Because we know that LightDM allows us to listen on a configurable port and launch a VNC server upon connection, our approach is to use a script that will launch Xspice masquerading as Xvnc.

The inspiration for this approach was the work we did in [linuxmint-19.3-cinnamon-turbovnc](#), where we had to work around the apparently broken TurboVNC `-inetd` option by using socat as a proxy. In this case we create an Xspice-lightdm-wrapper.

```
#!/bin/bash
if [ "$1" == ":2" ]; then
    Xspice $1 -ac -noreset -nolisten tcp --port 5902 --password $(cat /tmp/lightdm/.xserver-xspice-passwd) --vdagent --video-codescs ${SPICE_VIDEO_CODECS:-gstreamer:h264;gstreamer:vp8;gstreamer:mjpeg;spice:mjpeg} --deferred-fps 60 > /dev/null &
    XSPICE_PID=$!\n    cleanup() {
        kill -TERM $XSPICE_PID
    }
    trap cleanup SIGINT SIGTERM EXIT
    sleep 0.25
    kill -USR1 $PPID
fi
socat - TCP:localhost:5902
```

This Xspice-lightdm-wrapper script is triggered by connections to port 5900 and gets passed the X11 DISPLAY number by LightDM, so we first check if DISPLAY == “:2” and if so we launch Xspice listening on port 5902.

We next get the PID of the Xspice server that we’ve just launched and add a trap so that it will be cleanly killed when the parent script is terminated. After a brief sleep, to ensure that Xspice is indeed running, we call kill -USR1 \$PPID which is the signal sent by the X Server to LightDM to signal that it is running.

It is important that the socat call is outside the if statement as SPICE appears to open four separate connections and we need all of those to be redirected to port 5902. When a SPICE connection is made we will actually see four separate socat instances running.

Setting Xspice-lightdm-wrapper as the VNCServer command in /etc/lightdm/lightdm.conf.d/70-ubuntu.conf enables our workaround.

```
[LightDM]
minimum-display-number=1
[Seat:*]
user-session=ubuntu-xorg
xserver-command=/usr/bin/Xephyr-lightdm-wrapper
[VNCServer]
enabled=true
command=Xspice-lightdm-wrapper
```

The [ubuntu.sh](#) launch script is:

```
BIN=$(cd $(dirname $0); echo ${PWD%docker-gui*})docker-gui/bin
. $BIN/docker-xauth.sh
. $BIN/docker-gpu.sh

IMAGE=ubuntu-gnome-spice:18.04
CONTAINER=ubuntu-spice

# Create initial /etc/passwd /etc/shadow /etc/group
if ! test -f "etc.tar.gz"; then
    echo "Creating /etc/passwd /etc/shadow and /etc/group for c
ontainer."
    $DOCKER_COMMAND run --rm -it \
        -v $PWD:/mnt \
        $IMAGE sh -c 'adduser --uid '$(id -u)' --no-create-home
'$$(id -un)'; usermod -aG sudo '$$(id -un)'; tar zcf /mnt/etc.tar
.gz -C / ./etc/passwd ./etc/shadow ./etc/group'
fi

# Create home directory
mkdir -p $(id -un)

# Create SPICE password if required.
if ! test -f "$$(id -un)/.xserver-xspice-passwd"; then
    echo "Creating SPICE password"
    read -s -p "Enter password: " password
    echo
    read -s -p "Confirm password: " confirmation
    echo
    if [ $confirmation != $password ]; then
        echo "Confirmation doesn't match password, exiting!"
        exit 1
    fi
    echo $password > $$(id -un)/.xserver-xspice-passwd
fi
```

```

# Launch container as root to init core Linux services and
# launch the Display Manager and greeter. Switches to
# unprivileged user after login.
# --device=/dev/tty0 makes session creation cleaner.
# --ipc=host is set to allow Xephyr to use SHM XImages
$DOCKER_COMMAND run --rm -d \
    -p 5900:5900 \
    -p 5800:5800 \
    --device=/dev/tty0 \
    --name $CONTAINER \
    --ipc=host \
    --shm-size 2g \
    --security-opt apparmor=unconfined \
    --cap-add=SYS_ADMIN --cap-add=SYS_BOOT \
    -v /sys/fs/cgroup:/sys/fs/cgroup \
    -v $PWD/$(id -un):/home/$(id -un) \
    -v $PWD/$(id -un)/.xserver-xspice-passwd:/tmp/lightdm/.xser
ver-xspice-passwd \
    -v $DOCKER_XAUTHORITY:/root/.Xauthority.docker:ro \
    -v $DOCKER_XAUTHORITY:/home/$(id -un)/.Xauthority.docker:ro
\
    -v /tmp/.X11-unix/X0:/tmp/.X11-unix/X0:ro \
$GPU_FLAGS \
$IMAGE /sbin/init

# cp credentials bundle to container
cat etc.tar.gz | $DOCKER_COMMAND cp - $CONTAINER:/

```

This script is nearly identical to our basic 3D accelerated ubuntu-18.04-gnome launch script, aside from obvious things like the container and image names and small details like a section to create the SPICE password and exposing port 5800 in addition to port 5900 to expose the HTML5 Web Application.

The container may be launched remotely, supplying a username, host and path and using any SPICE client to connect, or navigate a browser to <http://remote-host:5800>:

```
ssh -t <user>@<remote-host> "cd <path-to>/docker-gui/11-remote-virtual-desktops/spice/ubuntu-18.04-gnome-spice; ./ubuntu.sh"
```

As with the other remote virtual desktop examples, this example will launch Xephyr on the remote server. The alternative [Dockerfile-no-xephyr](#) avoids this by replacing Xephyr with an Xspice-lightdm-wrapper that will launch Xspice as the primary X Server instead of Xephyr.

11-4-2 linuxmint-19-3-cinnamon-spice

The [Dockerfile](#) is in the [linuxmint-19.3-cinnamon-spice](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/spice](#).

The Dockerfile for linuxmint-19.3-cinnamon-spice and the [mint.sh](#) launch script are identical to those for our ubuntu-18-04-gnome-spice desktop aside from some trivial differences like the image and container names and a minor difference in the LightDM configuration, so they won't be explored in depth here, as there is not much more to add.

11-4-3 fedora-32-kde-spice

The [Dockerfile](#) is in the [fedora-32-kde-spice](#) directory of [docker-gui](#) under [11-remote-virtual-desktops/spice](#).

```
FROM fedora-kde-vgl:32

RUN dnf -y install \
    numpy python-setuptools socat \
    xorg-x11-server-Xspice spice-vdagent && \
    # LightDM complains if Xvnc isn't present, even if
    # [VNCServer] config. specifies a command= option.
    ln -snf /usr/bin/Xspice /usr/bin/Xvnc && \
    mkdir /tmp/audio-fifo && \
    chmod 1777 /tmp/audio-fifo && \
    # Modify PulseAudio daemon config to support SPICE.
    echo "load-module module-pipe-sink file=/tmp/audio-fifo/playb
ack fifo format=s16 rate=48000 channels=2" >> /etc/pulse/default.pa && \
    # The Xspice --auto option is convenient, but if
    # we have large display resolutions Xspice will
    # crash with OOM. We need to modify the config in
    # /etc/X11/spiceqxl.xorg.conf, which in turn means
    # we can't use --auto and that exposes a bug in
    # /usr/bin/Xspice where temp_dir isn't initialised.
    sed -i 's/if args.auto:/temp_dir = ""\nif args.auto:/' /usr/b
in/Xspice && \
    # Set buffers to handle larger resolutions.
    sed -i 's/#Option "NumHeads" "4"/Option "NumHeads" "1"/' /etc
/X11/spiceqxl.xorg.conf && \
    sed -i 's/#Option "SurfaceBufferSize" "128"/Option "SurfaceBu
fferSize" "512"/' /etc/X11/spiceqxl.xorg.conf && \
    sed -i 's/#Option "CommandBufferSize" "128"/Option "CommandBu
fferSize" "512"/' /etc/X11/spiceqxl.xorg.conf && \
    sed -i 's/#Option "FrameBufferSize" "16"/Option "FrameBuffers
ize" "32"/' /etc/X11/spiceqxl.xorg.conf && \
```

```

# If not using --auto the --audio-fifo-dir flag
# seems to be ignored, so we need to use the config.
sed -i 's/#Option "SpicePlaybackFIFODir" "\tmp\//Option "Sp
icePlaybackFIFODir" "\tmp\audio-fifo"/' /etc/X11/spiceqxl.xor
g.conf && \
# Download websockify and flexVDI fork of
# eyeos spice-web-client.
WS_VERSION=0.9.0 && \
SPICE=3.1.0 && \
curl -SSL https://github.com/novnc/websockify/archive/v${WS_V
ERSION}.tar.gz | tar -xvz -C /usr/local/bin && \
curl -SSL https://github.com/flexVDI/spice-web-client/archive
/${SPICE}.tar.gz | tar -xvz -C /usr/local/bin && \
cd /usr/local/bin/websockify-${WS_VERSION} && \
python setup.py install && \
mv /usr/local/bin/spice-web-client-${SPICE} \
/usr/local/bin/spice-web-client && \
# Tweak spice-web-client to add a basic
# host/port/password entry UI.
sed -i 's/document.location.port/port/g' \
/usr/local/bin/spice-web-client/lib/utils.js && \
sed -i 's/<div class="float-right">/<div class="float-right">
\n\n          <label for="host">Host: </label><input typ
e="text" id="host" value=""\><label for="port"> Port: </label
><input type="text" id="port" value=""\><label for="password">
Password: </label><input type="password" id="password" value="
" onkeyup="checkIfEnterPressed(event)\"\>\n/g' /usr/local/bin/s
pice-web-client/index.html && \
sed -i '$(document).ready(start);$(document).ready(init);/
g' /usr/local/bin/spice-web-client/run.js && \
sed -i 's/translate();//g' \
/usr/local/bin/spice-web-client/run.js && \

```

```
sed -i 's/function start ()/\nfunction init () {\n    translate();\n    document.getElementById("showclientid").style.display = "none";\n    document.getElementById("uploadfile").style.display = "none";\n    document.getElementById("host").value = document.location.hostname;\n    document.getElementById("port").value = document.location.port;\n}\n\nfunction checkIfEnterPressed (event) {\n    if (event.keyCode === 13) {\n        start()\n    }\n}\n\nfunction start ()/g' /usr/local/bin/spice-web-client/run.js && \
sed -i "s/data['spice_address'] || ''/document.getElementById('host').value/g" /usr/local/bin/spice-web-client/run.js && \
sed -i "s/data['spice_port'] || 0/document.getElementById('port').value/g" /usr/local/bin/spice-web-client/run.js && \
sed -i "s/data['spice_password'] || ''/document.getElementById('password').value/g" /usr/local/bin/spice-web-client/run.js && \
# Create systemd service to launch html5 SPICE
echo -e '[Unit]\nDescription=HTML5 SPICE WebSocket proxy\nAfter=syslog.target network.target\n\n[Service]\nUser=lightdm\nType=simple\nExecStart=/usr/local/bin/websockify 5800 localhost:5900 --web /usr/local/bin/spice-web-client\nTimeoutStopSec=20\nKillMode=process\nRestart=always\nRestartSec=2\n\n[Install]\nWantedBy=multi-user.target\nAlias=websocket.service\n' > /lib/systemd/system/websocket.service && \
ln -snf /lib/systemd/system/websocket.service \
/etc/systemd/system/multi-user.target.wants/websocket.service && \
```

```
# Script to launch Xspice "pretending" to be Xvnc
echo -e '#!/bin/bash\nif [ "$1" == ":2" ]; then\n  Xspice $1
-ac -noreset -nolisten tcp --port 5902 --password $(cat /tmp/li
ghtdm/.xserver-xspice-passwd) --vdagent --video-codecs ${SPICE_
VIDEO_CODECS:-gstreamer:h264;gstreamer:vp8;gstreamer:mjpeg;spic
e:mjpeg} --deferred-fps 60 > /dev/null &\n  XSPICE_PID=$!\n  cl
eanup() {\n    kill -TERM $XSPICE_PID\n  }\n  trap cleanup SIGI
NT SIGTERM EXIT\n  sleep 0.25\n  kill -USR1 $PPID\nfi\nsocat -
TCP:localhost:5902' > /usr/bin/Xspice-lightdm-wrapper && \
chmod +x /usr/bin/Xspice-lightdm-wrapper && \
echo -e '[LightDM]\nminimum-display-number=1\n[Seat:*\nuser-
session=gnome\nxserver-command=/usr/bin/Xephyr-lightdm-wrapper\
\n[VNCServer]\nenabled=true\ncommand=Xspice-lightdm-wrapper' > /
etc/lightdm/lightdm.conf.d/70-fedora.conf

VOLUME /tmp/audio-fifo
```

The image may be built **on the remote host** with:

```
docker build -t fedora-kde-spice:32 .
```

This Dockerfile is very similar to our Ubuntu and Linux Mint examples aside from using dnf instead of apt, echo calls requiring the `-e` flag, and a minor difference in the LightDM configuration.

The [fedora.sh](#) launch script is nearly identical to the Ubuntu and Linux Mint SPICE launch scripts except for the image and container names and the subtle difference to the adduser command that creates the credentials bundle.

About the Author

Fraser Adams is a software engineer with over thirty years of experience. He has a wide range of interests and experience from Digital Signal Processing and GPU Compute, through Middleware and Messaging, to Virtualisation, Containers and Cloud Native Architectures. He lives with his wife and son in rural England and when not coding and writing enjoys walking, cooking and playing guitar.