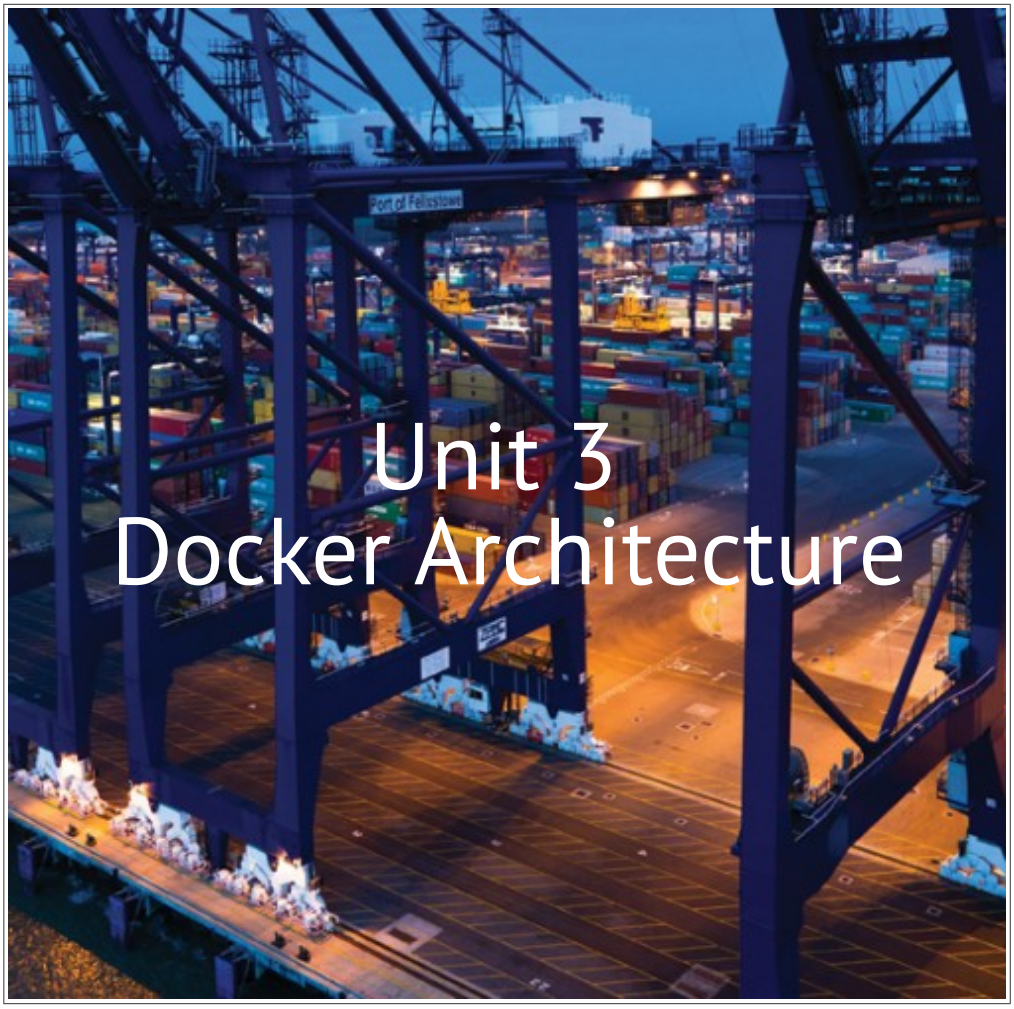


Unit 3
Docker Architecture

Objectives

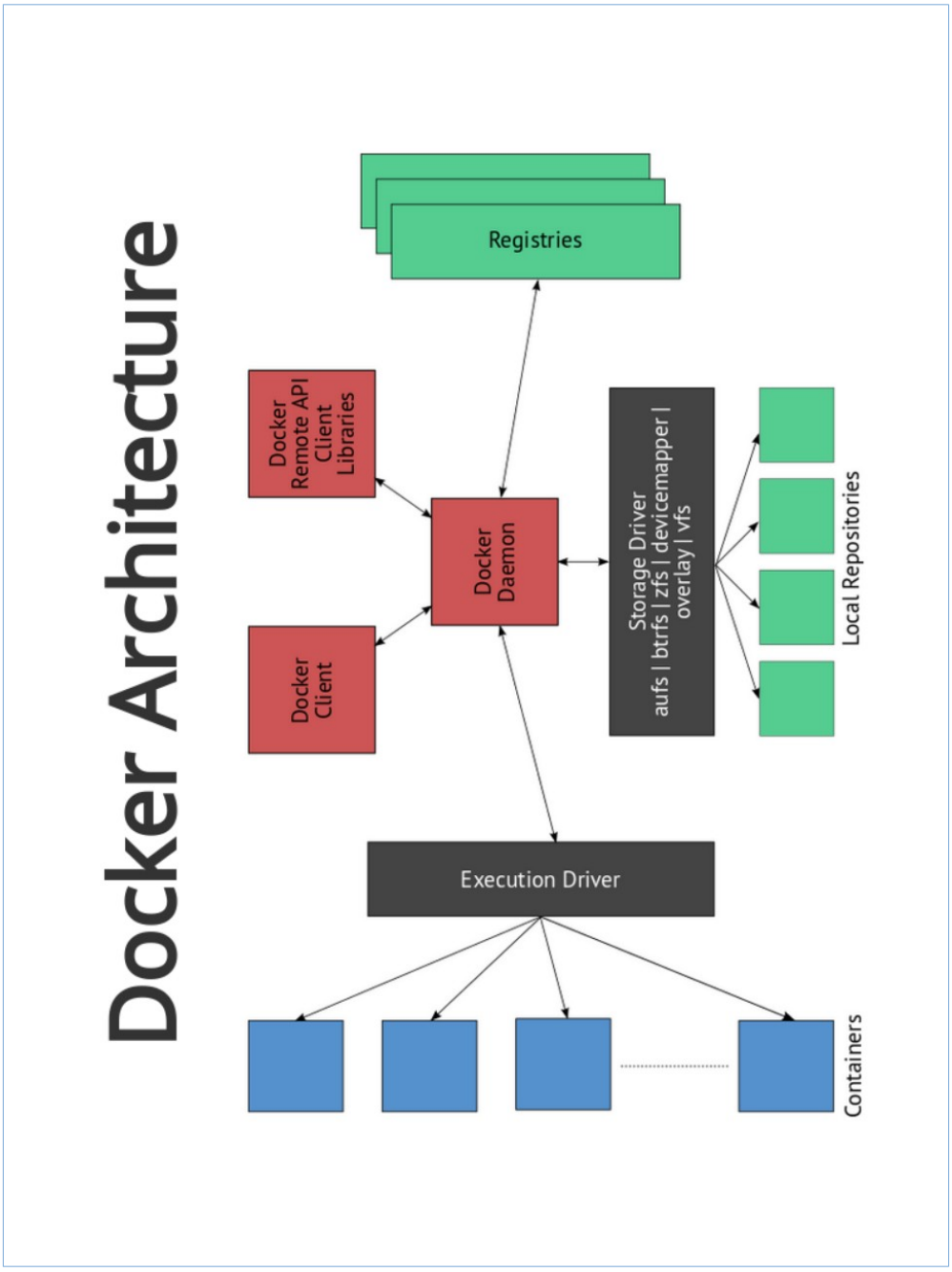
Upon completion of this unit, you will be able to:

- Outline the components of Docker's architecture
- Differentiate between Docker images and container filesystems
- Describe how to address Docker images from different locations
- Explain the purpose of the execution and storage drivers
- Summarise the purpose of the Docker client and daemon
- Detail how the client and daemon communicate

Agenda

The following topics will be covered in this unit:

- Architecture Overview
- Docker Images
- Execution Driver
- Storage Driver
- Docker Client
- Docker Daemon
- Lab exercise: Configuring the Docker Daemon


Docker Architecture
Registries
Docker Remote API Client Libraries
Docker Daemon
Storage Driver aufs | btrfs | zfs | devicemapper | overlay | vfs
Local Repositories
Docker Client
Execution Driver
Containers

Docker Architecture

- From an architectural perspective, Docker comprises of multiple sub-systems, which work in unison to manage the lifecycle of containers and the Docker images, from which they are derived:

  Docker Daemon: the Docker daemon is the server component of the Docker Engine, currently runs on Linux, and is responsible for dealing with client requests, and orchestrating sub-systems to comply with those requests. It communicates via a mainly RESTful API, using JSON objects to exchange data.

  Docker Client: the Docker client provides a command line interface (CLI) for providing user instructions to the Docker daemon. The client takes Docker commands issued at the terminal, translates them into Docker Remote API calls, which are sent to the Docker daemon for action.

  Execution Driver: the execution driver is a sub-component that takes care of creating, starting, stopping and destroying containers at the request of the daemon.

  Storage Driver: the storage driver sub-component, is responsible for managing the local cache of Docker images, as well as assembling and disassembling filesystems for containers.

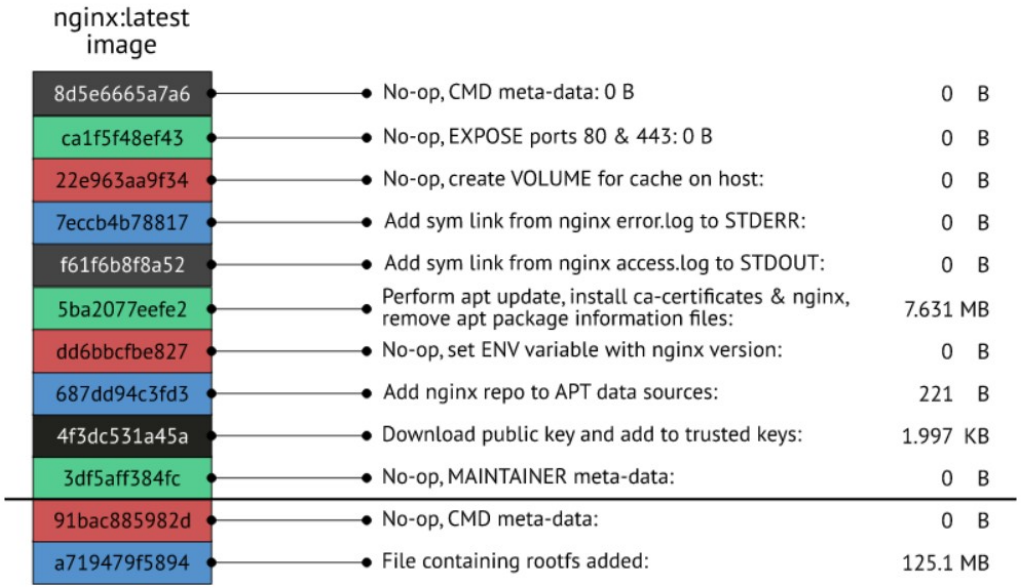
  Local Repositories: the local repositories are a collection of related, cached Docker images, that have been either retrieved from a Docker registry, or created on the local Docker host.

  Registries: registries provide a service for storing and distributing repositories of Docker images using an HTTP API. The default registry is the Docker Hub, hosted by Docker Inc., but there are third party registry implementations, and it's possible to host your own registry.

- There are numerous third party Remote API client libraries for a number of different programming languages

Notes

Docker Images



Docker Images

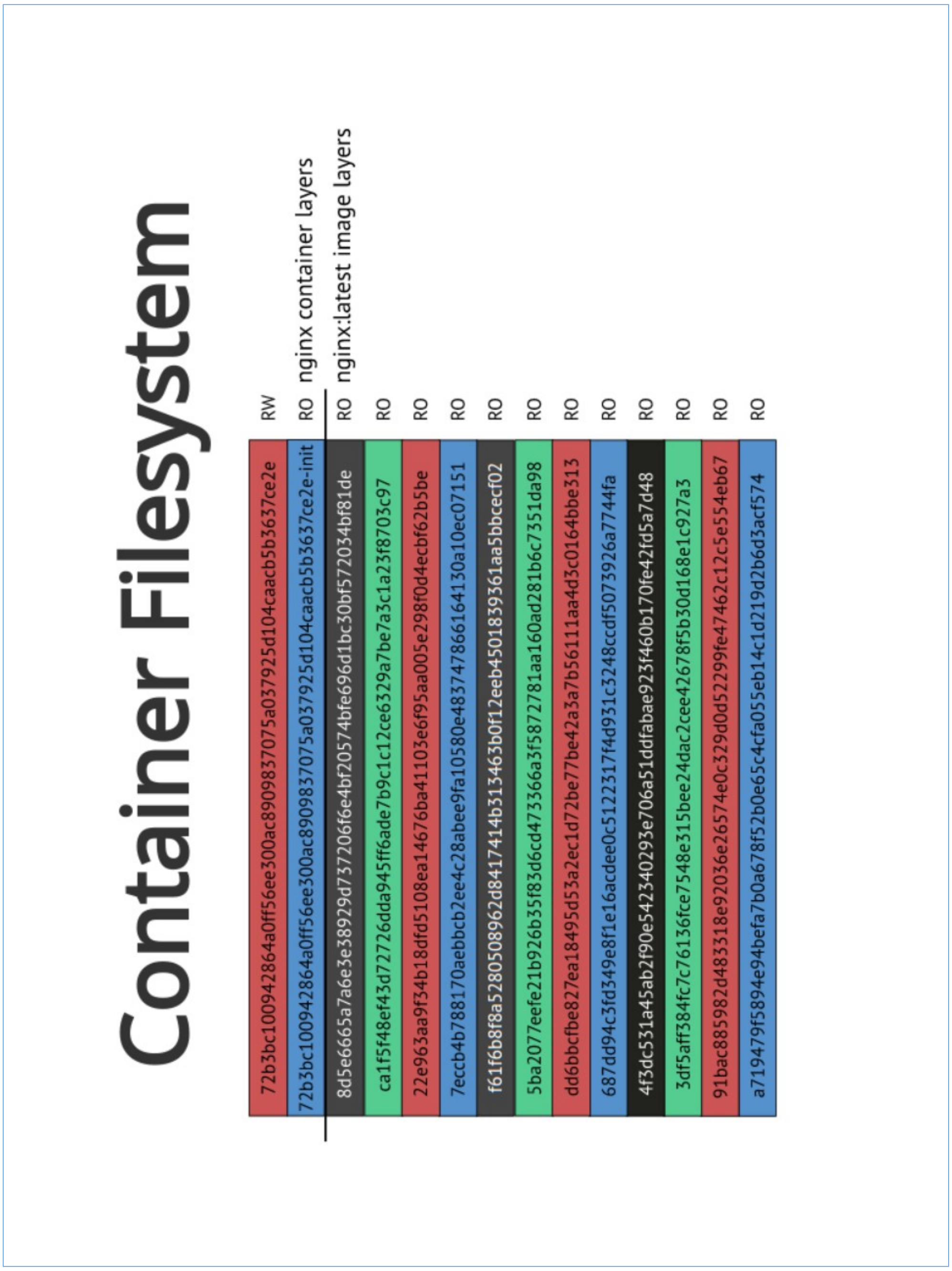
- A Docker image can be likened to a template, from which containers can be invoked, taking the form of the image characteristics
- Docker images consist of multiple layers of directories and files, which are combined together and then mounted at the point of use, in order to create a container filesystem
- Confusingly, Docker terminology refers to the individual layers as images, but a set of related layers is also an image, as is each sub-set of layers
- Each image has a parent image which it is layered on top of, unless the image is a base image
- In order to uniquely identify an image or layer, it is assigned and identified by a 256-bit ID*, which is presented to a Docker user as a 64 digit hex string, e.g.

```
e1cdf371fbdeaa08209e7c7af2fec6764f471402b193040e97fc224715250a11
```

- In order to make working with image IDs easier, most of the Docker CLI commands will output, or accept as an argument, a truncated 12 digit hex string instead of the 64 digit alternative (internally, Docker uses the 256-bit ID)
- In fact, it's possible to use as few of the hex digits as will serve to uniquely identify the image or layer, so `e1c` (or even `e1` or `e`), providing no other image or layer also starts with `e1c`
- Every Docker image has a base image, which usually comes from the Docker Hub registry, but it is possible to create proprietary base images
- Base images can be a minimal set of files and directories that represent a major Linux distribution, such as Debian Jessie, but can also be even more minimalist (e.g. BusyBox, Tiny Core Linux, Alpine Linux)
- A special repository called 'scratch' is hosted by the Docker Hub, which is a completely empty base image, but it cannot be pulled from the registry!
- This an historical anomaly; since Docker 1.5 defining 'scratch' as a base image is a no-op instruction in a Dockerfile

* Other Docker components, such as containers and volumes, are also assigned 256-bit IDs

Notes


Container Filesystem
72b3bc100942864a0ff56ee300ac8909837075a037925d104caacb5b3637ce2e RW
72b3bc100942864a0ff56ee300ac8909837075a037925d104caacb5b3637ce2e-init RO
nginx container layers
8d5e6665a7a6e3e38929d737206f6e4bf20574bfe696d1bc30bf572034bf81de RO
nginx:latest image layers
ca1f5f48ef43d72726dda945ff6ade7b9c1c12ce6329a7be7a3c1a23f8703c97 RO
22e963aa9f34b18dfd5108ea14676ba41103e6f95aa005e298f0d4ecbf62b5be RO
7eccb4b788170aebbcb2ee4c28abee9fa10580e483747866164130a10ec07151 RO
f61f6b8f8a5280508962d8417414b313463b0f12eeb4501839361aa5bbcecf02 RO
5ba2077eefe21b926b35f83d6cd473366a3f5872781aa160ad281b6c7351da98 RO
dd6bbcfbe827ea18495d53a2ec1d72be77be42a3a7b56111aa4d3c0164bbe313 RO
687dd94c3fd349e8f1e16acdee0c5122317f4d931c3248ccdf5073926a7744fa RO
4f3dc531a45ab2f90e542340293e706a51ddfabae923f460b170fe42fd5a7d48 RO
3df5aff384fc7c76136fce7548e315bee24dac2cee42678f5b30d168e1c927a3 RO
91bac885982d483318e92036e26574e0c329d0d52299fe47462c12c5e554eb67 RO
a719479f5894e94befa7b0a678f52b0e65c4cfa055eb14c1d219d2b6d3acf574 RO

Container Filesystem

- A container's filesystem is based on the layers of the image that is specified at runtime (e.g. `nginx:latest`)
- Two additional, non-persistent layers are created by the storage driver, that are specific to any given container instance
- The first additional layer is a special 'init' layer, where Docker creates mount points in order to inject some necessary runtime files into the container's filesystem (e.g . `/dev/console`, `/etc/hosts`, `/etc/resolv.conf`, `/etc/hostname`)
- The init layer has the same name as the container ID, with '-init' appended , e.g.

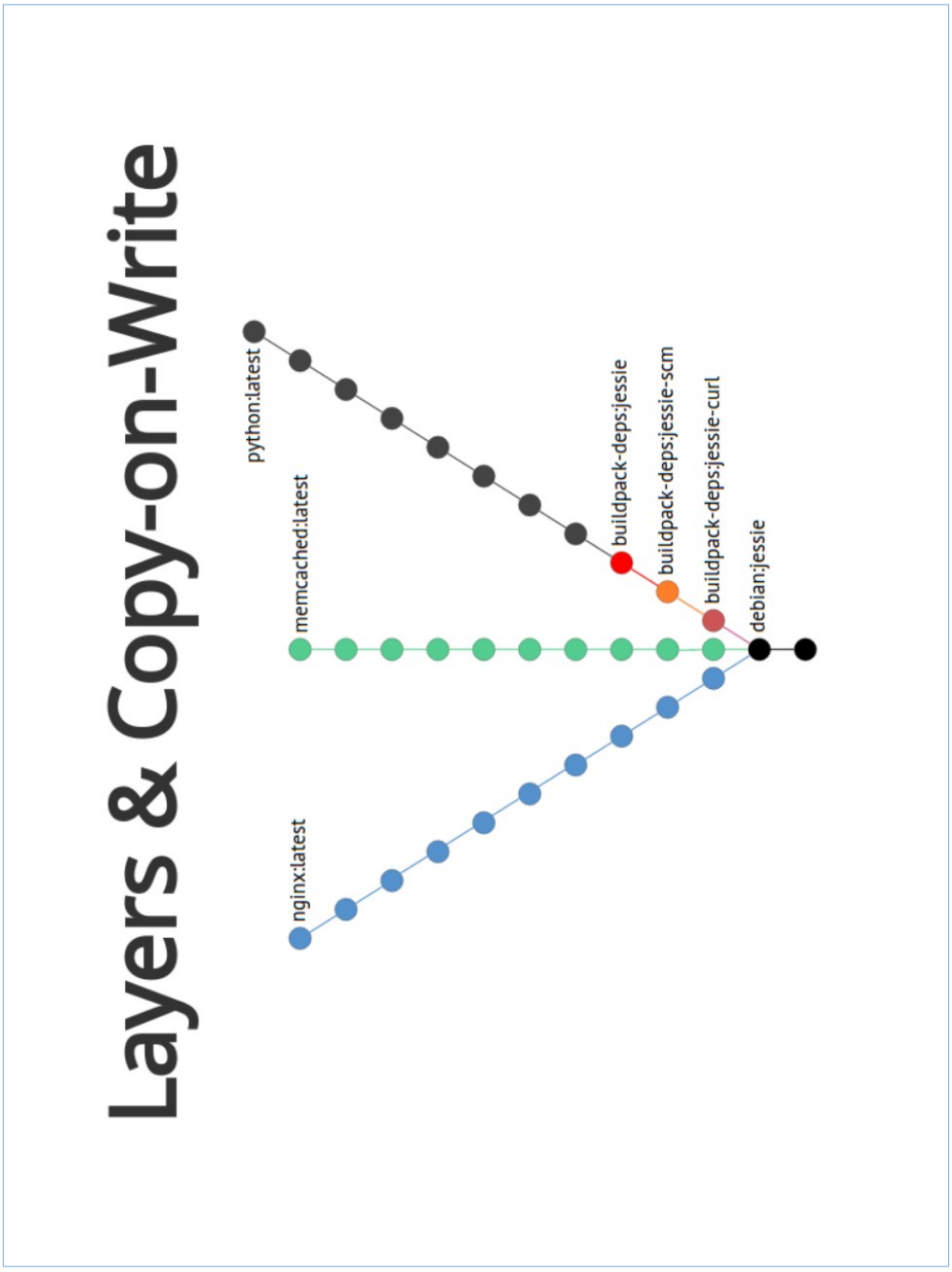
  ```
  72b3bc100942864a0ff56ee300ac8909837075a037925d104caacb5b3637ce2e-init
  ```

- The second additional layer is the container's read-write layer, which allows the container to modify existing files or directories, and create new ones
- The directory name for the read-write content is the same as the container's ID; e.g.

  ```
  72b3bc100942864a0ff56ee300ac8909837075a037925d104caacb5b3637ce2e
  ```

- Every layer of the image is implemented read-only, as is the container's 'init' layer, but the final layer is read-write, allowing the container to amend its filesystem whilst its running
- How the various layers are combined into an homogeneous filesystem is entirely dependent on the underlying storage driver used

Notes


Layers & Copy-on-Write
python:latest
memcached:latest
nginx:latest
buildpack-deps:jessie
buildpack-deps:jessie-scm
buildpack-deps:jessie-curl
debian:jessie

Layers & Copy-on-Write

- Image layers are built on top of each other before they are assembled into a coherent filesystem for a container
- Images can be derived from other images; for example the `nginx:latest` image is derived from the `debian:jessie` image, and the `debian:jessie` image is contained within and is a part of the `nginx:latest` image
- Similarly, the `memcached:latest` image is derived from the `debian:jessie` image, as is the `buildpack-deps:jessie-curl` image, and so on
- The layers for the `debian:jessie` image, and the files and directories that constitute the layers, are stored only once on the Docker host
- Because they are read-only, these common layers can be shared by images that reference those common layers
- This makes Docker very efficient in terms of storage, and significantly aids container start times
- When a container wants to alter a file that resides in its filesystem, but is actually located in one of its ancestor image layers, the file is 'copied' up to the read-write layer in the container, before it is altered and saved in that uppermost layer; this is called copy-on-write
- From the container's perspective, the original file in the ancestor image layer is masked by the new version in the read-write layer
- The mechanics of the copy up and subsequent masking of the original file, is entirely dependent on the type of storage driver employed

Notes

Docker Registries

- A Docker registry is a service to manage information about docker images, store their content, and to enable their distribution
- Registries can be self-hosted, or be provided by a third party, but the default registry is the Docker Hub, which is hosted by Docker, Inc.
- The Docker Hub provides both private and public repositories
- Subscribers to the Docker Hub are provided with one private repository and an unlimited number of public repositories
- Additional private repositories on the Docker Hub registry can be purchased from Docker, Inc.
- Official, certified repositories of images are provided by software vendors (e.g. Nginx, Inc.) and others in the Docker community, and are stored on the Docker Hub registry
- The Docker Hub official repositories provide optimised images (to run in containers), and are regularly updated
- The content of the official repositories are open to inspection, giving credence to their content, and automatically built and pushed to the Docker Hub registry
- The code that underpins the Docker Hub registry is open source, and therefore it is possible to self-host a registry within the confines of a firewall or in a cloud environment
- Docker Inc. also provide an enhanced commercially supported registry called the Docker Trusted Registry (also available from Amazon, IBM and Microsoft)
- Docker Trusted Registry can be deployed on premise or in the cloud

Notes

Anatomy of Image Names

Syntax

```
[hostname[:port]]/[username]/reponame[:tag]
```

Docker Hub official image example

```
centos:latest
```

Docker Hub user namespace example

```
phusion/baseimage:0.9.9
```

Third party registry, user namespace example

```
quay.io/signalfuse/zookeeper:3.4.6-1
```

Self-hosted registry example

```
localhost:5000/revealjs:latest
```

Anatomy of Image Names

- In order to use a Docker image, it needs to be addressed by its name in order to be located
- A fully qualified image name (FQIN) consists of four main components; a registry location, a namespace, a repository name, and an image tag:

```
[hostname[:port]]/[username]/reponame[:tag]
```

- The registry location is specified in terms of a hostname and optional port number (e.g. `quay.io:80`), and when omitted, implicitly refers to the Docker Hub registry (whose location is `registry-1.docker.io`)
- Registries can, however, be hosted by third parties supplying services to customers, or by individuals or organisations needing to keep their Docker images proprietary behind firewalls
- The username namespace component corresponds to the user or entity account for the repository where the image resides, and when left blank implies a special namespace called `library` (which is reserved for Docker's official images on the Docker Hub registry)
- The repository name refers to a collection of similar images (e.g. `ubuntu`)
- Any particular image can optionally be labelled with a tag in order to distinguish it from other, similar images held in the same repository (e.g. `ubuntu:utopic` or `ubuntu:trusty`)
- If the tag element of the image name is omitted, Docker assumes the special tag called 'latest'
- So, whilst it is normal practice to 'address' the latest official CentOS image on the Docker Hub registry simply as `centos`, it could equally well be addressed with a FQIN:

```
registry-1.docker.io/library/centos:latest
```

- Currently, RHEL-based RPM packages for Docker provided by the distribution in question, allow for blocking access to registries, including the Docker Hub registry

Notes

Execution Driver

The execution driver creates, starts, stops and destroys containers

Docker uses a native, purpose built execution driver called **libcontainer**, a Golang library

Open Container Initiative established to define vendor agnostic standard for container runtime environments

Docker contributed libcontainer as basis of a new reference implementation of the OCI specs, called **runC**

Execution Driver

- The execution driver (sometimes referred to as the 'execdriver'), is the component responsible for creating, starting, stopping and destroying containers
- It bootstraps the container's specified command as a process (PID 1) inside the container
- It takes care of creating the relevant namespaces, establishing control groups for the process, applying security controls, generating the container's networking environment, mounting the container's filesystem and executing the process
- Historically, Docker used the LXC userland tools for this purpose, but subsequently defaulted to a purpose built, native execdriver embodied in a library called `libcontainer`
- Upon the creation of the Open Container Initiative, whose aim is to develop a standard set of specifications for container runtime environments, the Docker project donated the `libcontainer` code as a basis for a reference implementation
- The code is being developed in conjunction with the definition of the specifications, and has been renamed `runC`
- Whilst Docker still relies on `libcontainer` for its container's runtime environments, it is slowly migrating toward the adoption of `runC` as it matures, which will be a separate binary altogether
- In theory, in the future it will be possible to choose an alternative container runtime to `libcontainer/runC`, and organisations are already developing OCI-compliant container runtime alternatives

Notes

Storage Driver

The storage driver manages the Docker host's cache of **image layers** and their contents

Several different storage drivers are supported

Choice comes down to: **user vs. daemon**, **kernel support**, **backing filesystem**, **workload type vs. performance**

Supported storage drivers:
aufs, **btrfs**, **devicemapper**, **zfs** & **overlay**

Storage Driver

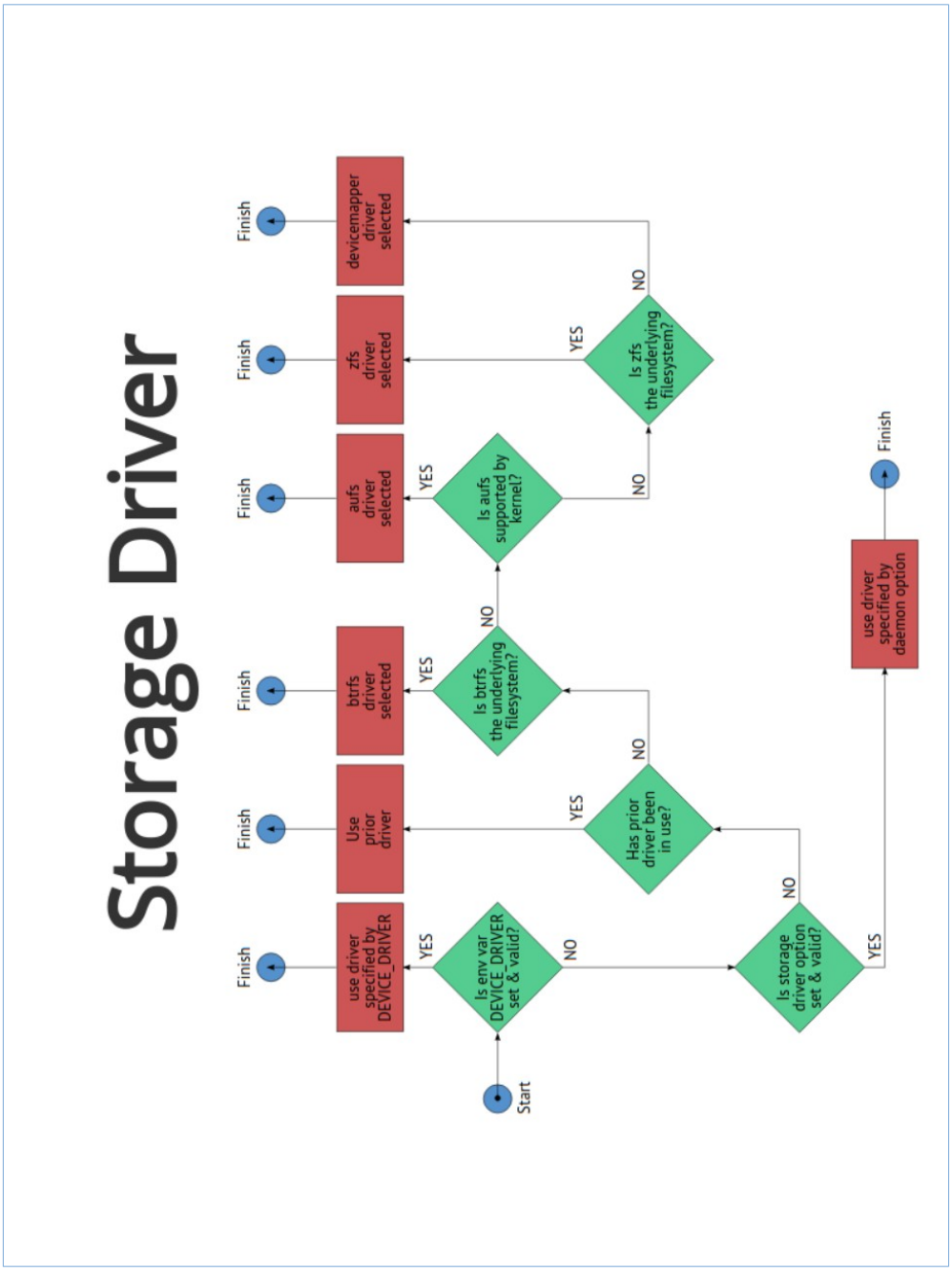
- The storage driver (sometimes referred to as the 'graphdriver') takes care of managing the local storage of image layers and their contents, and assembling and dis-assembling the filesystems for containers
- The following different storage drivers are supported by Docker: aufs, btrfs, devicemapper, zfs and overlay
- Which storage driver is employed by the Docker daemon, is dependent on several factors:

    1. User specified or daemon specified
    2. Kernel support
    3. Backing filesystem support
    4. Performance characteristics

- Storage driver characteristics:

aufs:	Advanced Multi-Layered Unification Filesystem takes directories and mounts them together in a union over the top of a regular filesystem (e.g. ext4). Not in mainline kernel.
btrfs:	B-tree Filesystem is an advanced copy-on-write filesystem (subvolumes & snapshots), and made it into mainline kernel in August 2014).
devicemapper:	The devicemapper storage driver uses the device mapper framework with the thin provisioning target (dm- thinp) to provide COW snapshots. Contributed by Red Hat in lieu of a supported alternative on RHEL.
zfs:	ZFS is a copy-on-write filesystem provided as a loadable kernel module. It uses snapshots and clones to implement image layers.
overlay:	Overlay (formally overlayfs) is a union mount filesystem, and was merged into the mainline Linux kernel as of 3.18 (December 2014), and operates over a regular filesystem.

Notes


Storage Driver
Start
Is env var DEVICE_DRIVER set & valid?
YES
use driver specified by DEVICE_DRIVER
Finish
NO
Has prior driver been in use?
YES
Use prior driver
Finish
NO
Is storage driver option set & valid?
YES
use driver specified by daemon option
Finish
NO
Is btrfs the underlying filesystem?
YES
btrfs driver selected
Finish
NO
Is aufs supported by kernel?
YES
aufs driver selected
Finish
NO
Is zfs the underlying filesystem?
YES
zfs driver selected
Finish
NO
devicemapper driver selected
Finish

Storage Driver

- The storage driver is defined when the Docker daemon is started
- If the DOCKER_DRIVER environment variable is set to one of `aufs`, `btrfs`, `zfs`, `devicemapper` or `overlay`, the daemon will try to use that specified as its first option
- If it is unset, and the `-s <storage driver>` option has been specified, the daemon will try to use that which has been specified (e.g. `-s overlay`)
- If neither of these configuration options have been used, Docker will determine the storage driver itself
- First, it will check to see whether there is prior historical use of particular storage drivers (based on stored data), and will match the first found in its hard-coded priority list (see below)
- If there is no prior storage driver use, Docker uses its priority list in conjunction with which storage drivers are supported by the kernel, and backing filesystem constraints
- The priority order that Docker uses is: aufs → btrfs → zfs → devicemapper
- At present, it is necessary to explicitly request the `overlay` driver as the storage driver, and its kernel module must be loaded

Notes

Docker Client

```
docker attach    Attach to a running container
docker build     Build an image from a Dockerfile
docker commit    Create a new image from a container's changes
docker cp        Copy files/folders between a container and the local filesystem
docker create    Create a new container
docker diff      Inspect changes on a container's filesystem
docker events    Get real time events from the server
docker exec      Run a command in a running container
docker export    Export a container's filesystem as a tar archive
docker history   Show the history of an image
docker images    List images
docker import    Import the contents from a tarball to create a filesystem image
docker info      Display system-wide information
docker inspect   Return low-level information on a container or image
docker kill      Kill a running container
docker load      Load an image from a tar archive or STDIN
docker login     Register or log in to a Docker registry
docker logout    Log out from a Docker registry
docker logs      Fetch the logs of a container
docker network   Manage Docker networks
```

Docker Client

- The Docker client provides a CLI for requesting services of the Docker daemon
- The commands available number 41, and provide the means to manipulate and interact with images and containers, as well as retrieve information concerning the Docker host system, images and containers
- The following client commands are provided:

```
docker attach    Attach to a running container
docker build     Build an image from a Dockerfile
docker commit    Create a new image from a container's changes
docker cp        Copy files/folders between a container and the local filesystem
docker create    Create a new container
docker diff      Inspect changes on a container's filesystem
docker events    Get real time events from the server
docker exec      Run a command in a running container
docker export    Export a container's filesystem as a tar archive
docker history   Show the history of an image
docker images    List images
docker import    Import the contents from a tarball to create a filesystem image
docker info      Display system-wide information
docker inspect   Return low-level information on a container or image
docker kill      Kill a running container
docker load      Load an image from a tar archive or STDIN
docker login     Register or log in to a Docker registry
docker logout    Log out from a Docker registry
docker logs      Fetch the logs of a container
docker network   Manage Docker networks
docker pause     Pause all processes within a container
docker port      List port mappings or a specific mapping for the CONTAINER
docker ps        List containers
docker pull      Pull an image or a repository from a registry
docker push      Push an image or a repository to a registry
docker rename    Rename a container
docker restart   Restart a container
docker rm        Remove one or more containers
docker rmi       Remove one or more images
docker run       Run a command in a new container
docker save      Save an image(s) to a tar archive
docker search    Search the Docker Hub for images
```

Notes

Docker Client

docker pause	Pause all processes within a container
docker port	List port mappings or a specific mapping for the CONTAINER
docker ps	List containers
docker pull	Pull an image or a repository from a registry
docker push	Push an image or a repository to a registry
docker rename	Rename a container
docker restart	Restart a container
docker rm	Remove one or more containers
docker rmi	Remove one or more images
docker run	Run a command in a new container
docker save	Save an image(s) to a tar archive
docker search	Search the Docker Hub for images
docker start	Start one or more stopped containers
docker stats	Display a live stream of container(s) resource usage statistics
docker stop	Stop a running container
docker tag	Tag an image into a repository
docker top	Display the running processes of a container
docker unpause	Unpause all processes within a container
docker version	Show the Docker version information
docker volume	Manage Docker volumes
docker wait	Block until a container stops, then print its exit code

Docker Client

- Client commands continued:

```
docker start     Start one or more stopped containers
docker stats     Display a live stream of container(s) resource usage statistics
docker stop      Stop a running container
docker tag       Tag an image into a repository
docker top       Display the running processes of a container
docker unpause   Unpause all processes within a container
docker version   Show the Docker version information
docker volume    Manage Docker volumes
docker wait      Block until a container stops, then print its exit code
```

- The CLI commands are translated into Docker Remote API calls by the client, before transmission to the daemon
- The client utilises the same binary as the daemon (default location: `/usr/bin/docker`), with the relevant branch of code executed based on context
- By default, the client communicates with the daemon via a local UNIX domain socket, but this can be overridden to allow the client to be remote from the daemon
- Remote connections between the client and daemon will be unencrypted and unauthenticated, unless Transport Layer Security (TLS) is implemented via command line options in the client and daemon
- The TLS-related command line options for the client can be omitted for remote communication, provided the environment variables `DOCKER_HOST` and `DOCKER_TLS_VERIFY` are set accordingly (more later)

Notes

Stop/Start Docker Daemon

Docker Machine:

```
# docker-machine ls
NAME   ACTIVE   DRIVER       STATE     URL                         SWARM
box    *        virtualbox   Running   tcp://192.168.99.100:2376
# docker-machine stop box
# docker-machine ls
NAME   ACTIVE   DRIVER       STATE     URL   SWARM
box             virtualbox   Stopped
# docker-machine start
Starting VM...
Started machines may have new IP addresses. You may need to re-run the
`docker-machine env` command.
```

Upstart Init System:

```
# service docker stop
docker stop/waiting
# service docker start
docker start/running, process 7685
```

Stop/Start the Docker Daemon

Docker Machine

- If the Docker host is remote, and under the supervision of Docker Machine, its daemon can be stopped and started by simply using the `stop`, `start` and `restart` sub-commands:

```
# docker-machine ls
NAME   ACTIVE   DRIVER       STATE     URL                         SWARM
box    *        virtualbox   Running   tcp://192.168.99.100:2376
# docker-machine stop box
# docker-machine ls
NAME   ACTIVE   DRIVER       STATE     URL   SWARM
box             virtualbox   Stopped
# docker-machine start
Starting VM...
Started machines may have new IP addresses. You may need to re-run the
`docker-machine env` command.
```

Upstart

- For Linux distributions using the upstart init system (Ubuntu variants prior to 15.04), the upstart job configuration file for Docker is `/etc/init/docker.conf`
- Amongst other things, the upstart pre-script sources the file `/etc/default/docker`, which can contain definitions for various Docker-related environment variables, enabling configuration parameters to be set for the daemon on startup
- The daemon can be stopped and started using the `service` utility:

```
# service docker stop
docker stop/waiting
# service docker start
docker start/running, process 7685
```

Notes

Stop/Start Docker Daemon

Systemd Init System:

```
# systemctl is-enabled docker.service
disabled
# systemctl enable docker.service
Created symlink From /etc/systemd/system/multi-user.target.wants/docker.
to /usr/lib/systemd/system/docker.service
# systemctl start docker.service
# systemctl stop docker.service
```

Unit file changes require reload & restart

```
# systemctl daemon-reload
# systemctl restart docker.service
```

Stop/Start the Docker Daemon

Systemd

- For Linux distributions using the systemd init system, the systemd unit file for the Docker daemon service is located at `/lib/systemd/system/docker.service`
- To ensure that Docker starts at boot time, be sure to check that it has been enabled, and if not, enable it:

```
# systemctl is-enabled docker.service
disabled
# systemctl enable docker.service
```

- The unit file for the Docker service specifies that it requires the creation of the Docker socket which Docker uses to provide communication between client and daemon, and that the service will only be started after the socket has been created
- To stop the Docker daemon use the following command:

```
# systemctl stop docker.service
```

- To start the Docker daemon when it is stopped, use the following command:

```
# systemctl start docker.service
```

- After configuration changes have been made, register the changes and then restart the daemon:

```
# systemctl daemon-reload
# systemctl restart docker.service
```

Notes

Applying Config Options

Upstart Init System:

- Environment variables and config options are defined in `/etc/default/docker`
- Use `DOCKER_OPTS` variable to define config options; e.g `DOCKER_OPTS="--storage-driver=overlay"`

Systemd Init System:

- Default unit file can be overridden with config files in `/etc/systemd/system/docker.service.d`

Applying Configuration Options for Docker Daemon

Upstart

- Environment variables and configuration options are specified in the configuration file `/etc/default/docker`; the daemon needs to be restarted after the file is changed
- Configuration options are defined in the `DOCKER_OPTS` variable; e.g.

```
DOCKER_OPTS="--storage-driver=overlay"
```

Systemd*

- If the default parameters for the Docker daemon need to be overridden at startup, this can be safely achieved in one of two ways
- If a complete replacement unit file is required, a drop-in file can be placed in the `/etc/systemd/system` directory, which will override the package provided equivalent
- If, however, specific directives need altering rather than the whole unit definition, best practice is to add files (with a `.conf` suffix) in a directory located at: `/etc/systemd/system/docker.service.d` (daemon requires re-load and restart)
- For example, to specify an alternative location for the directory that the daemon uses to store temporary files during some of it's operations, place something similar in a file called `tmp.conf` in `/etc/systemd/system/docker.service.d`:

```
[Service]
Environment="DOCKER_TMP=/tmp"
```

- To apply changes to the configuration options for the Docker daemon (e.g. change storage driver), use the following snippet in a configuration file in `docker.service.d`:

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// --storage-driver=overlay
```

* RPM packages for Docker provided by RHEL use an EnvironmentFile `/etc/sysconfig/docker` for supplying config options

Notes

General Config Options

Daemon Option	Description
`--default-ulimit=[]`	Default ulimit parameters for containers
`-e, --exec-driver="native"`	Execution driver the Docker Engine uses
`--exec-opt=[]`	Options for the execution driver
`--exec-root="/var/run/docker"`	Root for execution driver
`-g, --graph="/var/lib/docker"`	Location of Docker runtime environment
`-p, --pidfile="/var/run/docker.pid"`	File containing the PID of the daemon
`-s, --storage-driver=""`	Storage driver used by Docker Engine
`--selinux-enabled=false`	Apply SELinux security to Docker daemon
`--storage-opt=[]`	Options for the storage driver

General Configuration Options for Docker Daemon

- The following general system-related configuration options are available for the Docker daemon:

`--default-ulimit=[]`: specifies a list of default ulimit parameters to apply to all containers (e.g. `nofile`, `noproc`), which can be overridden by the Docker client at container run time
`-e, --exec-driver="native"`: specifies the execution driver to use for managing containers
`--exec-opt=[]`: specifies configuration options for the execution driver
`--exec-root="/var/run/docker"`: specifies the root directory for the execution driver's data
`-g, --graph="/var/lib/docker"`: specifies Docker's runtime directory
`-p, --pidfile="/var/run/docker.pid"`: specifies the file which contains the PID of the running Docker daemon
`-s, --storage-driver=""`: specifies the storage driver to use
`--selinux-enabled=false`: specifies that Docker daemon will run with SELinux enabled
`--storage-opt=[]`: specifies options for the storage driver

- The ulimit parameters are set with the following format, `type=soft:[hard]` (e.g. `msgqueue=204800:409600`. If the `hard` value is omitted, it takes the same as the `soft` value
- The sole option available to the `--exec-opt` configuration option, which is passed on to the execution driver, is `native.cgroupdriver`, which can be given either of the values `cgroupfs` or `systemd` (e.g. `--exec-opt native.cgroupdriver=cgroupfs`). This is used to define whether all container's control groups are managed by `cgroupfs` or by `systemd`. The default behaviour is for the execution driver to attempt to use `systemd`, but when unavailable to fall back on `cgroupfs`

Notes

docker version

Provides basic information for both the daemon and the client:

```
$ docker version
Client:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:12:04 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:12:04 UTC 2015
 OS/Arch:      linux/amd64
```

Client Option	Description
`-f, --format[=FORMAT]`	Golang text/template to apply to the output

docker version

- Whilst running Docker on a single host ensures that the same binary is used for the client and the daemon, it is not necessarily the case that the client and daemon are compatible when the Docker host is remote
- The `docker version` command provides basic information for both the daemon and the client:

```
$ docker version
Client:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:12:04 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:12:04 UTC 2015
 OS/Arch:      linux/amd64
```

- The `docker version` command provides basic diagnostic information, which should always be provided when raising an issue with the Docker project on GitHub
- The `docker version` command can take a single command line option:

  `-f, --format[=FORMAT]`: specifies a Golang text/template to apply to the output

- For example, to retrieve just the Git commit for the client:

```
$ docker version -f '{{.Client.GitCommit}}'
a34a1d5
```

Notes

docker info

Provides detailed information for the configuration of the Docker host:

```
$ docker info
Containers: 2
Images: 82
Server Version: 1.9.1
Storage Driver: overlay
 Backing Filesystem: extfs
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-26-generic
Operating System: Ubuntu 14.04.3 LTS
CPUs: 2
Total Memory: 3.751 GiB
Name: castafiore
ID: GGZ5:AH7A:YGT3:CCUL:LUWZ:MMNN:32EN:I22P:G2DU:PKYK:TT5Y:UBMA
WARNING: No swap limit support
```

Additional data is provided when the daemon is run with log level set to DEBUG

docker info

- The `docker info` command provides important configuration data related to the Docker host, and should be provided when raising an issue with the Docker project on GitHub:

```
$ docker info
Containers: 2
Images: 82
Server Version: 1.9.1
Storage Driver: overlay
 Backing Filesystem: extfs
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-26-generic
Operating System: Ubuntu 14.04.3 LTS
CPUs: 2
Total Memory: 3.751 GiB
Name: castafiore
ID: GGZ5:AH7A:YGT3:CCUL:LUWZ:MMNN:32EN:I22P:G2DU:PKYK:TT5Y:UBMA
WARNING: No swap limit support
```

- When the Docker daemon is run in debug mode, the information is further augmented:

```
Debug mode (server): true
 File Descriptors: 12
 Goroutines: 20
 System Time: 2015-11-25T17:16:59.02258723Z
 EventsListeners: 0
 Init SHA1: 4fc7c03a06c675f115b39958c13486d53df10a14
 Init Path: /usr/lib/docker/dockerinit
 Docker Root Dir: /var/lib/docker
```

Notes

Docker Daemon Logging

Docker daemon's logging can be set to one of debug, **info**, warn, error or fatal

Host Type	Log Location
Upstart	Logs located at `/var/log/upstart/docker.log` and are ascii
Systemd	Logs are binary, can be viewed with systemd `journalctl` utility
boot2docker	Use `docker-machine ssh <host> 'tail -f /var/log/docker.log'`

The logging config options for the daemon are:

Daemon Option	Description
`-D=false`	Sets logging level of Docker daemon to debug
`-l, --log-level="info"`	Sets logging level of Docker dameon

Docker Daemon Logging

- The Docker daemon can provide five different levels of logging: `debug`, `info`, `warn`, `error`, `fatal`, with the default set to `info`
- The `debug` level is the most verbose, with reducing verbosity for each subsequent level: `debug` → `info` → `warn` → `error` → `fatal`

Upstart

- Daemons managed by the upstart init system have output sent to `stdout` and `stderr`, written to a logfile located in `/var/log/upstart`
- Hence, the Docker daemon's logfile is `docker.log` within `/var/log/upstart`

Systemd

- For systemd, the output produced by the daemon writing to `stdout` and `stderr` can be accessed via systemd's journal system, using the command (add the `-f` option to follow):

```
journaltctl -u docker.service
```

boot2docker

- Although a symlink to another location, the daemon's logs for a remote boot2docker host can be found in `/var/log/docker.log`, and could be accessed with the following:

```
$ docker-machine ssh box "tail -f /var/log/docker.log"
```

- The following logging related configuration options are available for the daemon:

`-D=false`: original option for specifying debug level, same as `--log-level=debug`
`-l, --log-level="info"`: specifies logging level for Docker daemon

Notes

Local Daemon Comms

- By default, Docker daemon listens on a local UNIX socket at `/var/run/docker.sock`
- The `-H` config option can be used to customise the default socket configuration
- Systemd socket activation requires option `-H fd://`

The local client/server config options for the daemon are:

Daemon Option	Description
`-G, --group="docker"`	Sets group ownership of daemon socket
`-H, --host=[]`	List of sockets the daemon listens on

Local Client and Server Communication

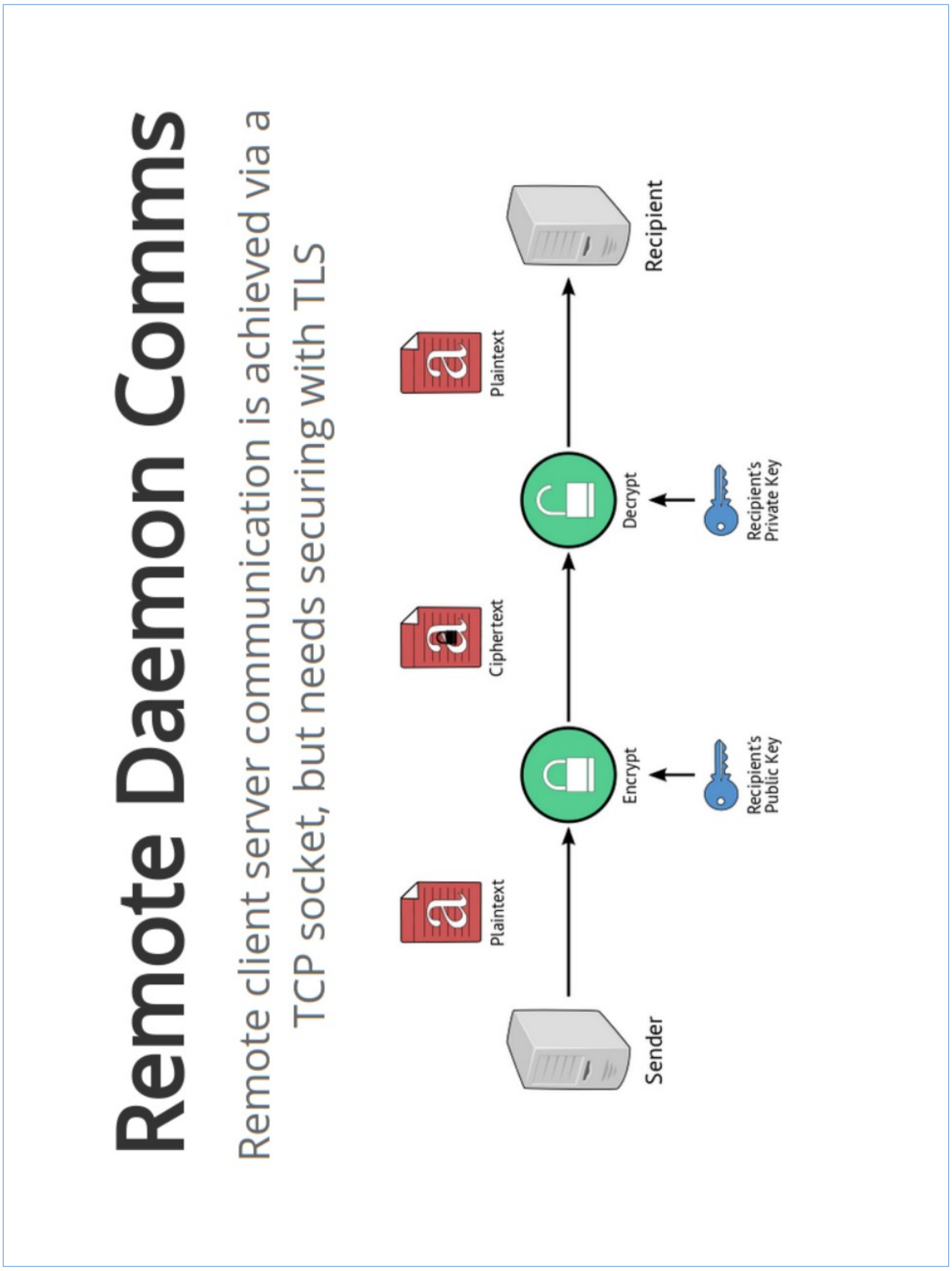
- By default, unless provisioned with Docker Machine, the Docker daemon will be configured to communicate locally via a UNIX domain socket
- The socket is established at the default location `/var/run/docker.sock`, where the daemon listens for service requests
- Clearly, this restricts the server to receiving service requests from processes running on the same host, and not from the network
- This default setup can be replaced or augmented, with the use of the `-H` command line
- option; additional sockets can be added and the location of the single default socket can be changed:

```
docker daemon -H unix:///var/run/docker_server.sk
```

- Remember, for distributions running Docker under systemd supervision, the socket's activation will be managed by systemd, and the daemon will require one of the following forms, `-H fd:// | fd://* | fd://socketfd`
- The Docker socket(s) are owned by `root`, by default belong to a group called `docker`, and have their file mode bits set to `0660`
- This means the Docker client needs to be run as `root` in order to read and write to the socket, unless non-privileged users are made members of the `docker` group
- Group ownership of the socket can be changed with the `-G` command line option
- The Docker daemon configuration options that apply to the client/server communication are:

`-G, --group="docker"`: specifies the group ownership for UNIX domain sockets used by the daemon to listen for requests
`-H, --host=[]`: specifies a list of sockets (UNIX domain or TCP) the Docker daemon listens on; takes the form of `tcp://[host][:port][/path]`, `unix:///path/to/socket`, `fd://*` or `fd://socketfd`

Notes


Remote Daemon Comms
Remote client server communication is achieved via a TCP socket, but needs securing with TLS
Plaintext
Ciphertext
Plaintext
Sender
Encrypt
Decrypt
Recipient
Recipient's Public Key
Recipient's Private Key

Remote Client Server Communication

- Providing a means to access the Docker server on the same host that it runs on is all well and good, but what if we want to access the server from a different host?
- Docker Machine provides this capability for the Docker host running in a remote virtual machine, and largely masks the complexity behind remote client server communication
- What if we want a local client to communicate with a remote 'bare metal' Linux host running the Docker daemon?
- Remote client server communication is achieved by specifying a TCP socket for the daemon to listen to, instead of, or in addition to, the local UNIX domain socket, using the `-H` command line option again; e.g.:

```
docker daemon -H tcp://192.168.1.40:2375
```

- The `-H` option is also a client option, and can be used to address a remote Docker host; e.g.:

```
docker -H tcp://192.168.1.40:2375 info
```

- However, this is insecure, and allows unauthorised access to the daemon by anyone who knows the socket address
- Docker provides the means for secure communication between the client and server using Transport Layer Security (TLS)
- A number of command line arguments, which are common to the server and the client, can be used to configure secure communication between the server and the client
- The Internet Assigned Numbers Authority (IANA) designated ports for the Docker Rest API are `2375` (plain), and `2376` (secure)

Notes

Securing Remote Comms

- Docker provides several config options to enable TLS
- The options apply to both the client and the daemon

The TLS config options for the client and daemon are:

Daemon Option	Description
`--tls`	Use TLS - implied by `--tlsverify`
`--tlscacert="~/.docker/ca.pem"`	Only trust certificates signed by this CA
`--tlscert="~/.docker/cert.pem"`	Path to certificate containing public key
`--tlskey="~/.docker/key.pem"`	Path to private key
`--tlsverify=false`	Use TLS to verify the client or daemon

Securing Remote Communication

- Docker provides five configuration options for the client and the daemon, which facilitate TLS communication
- The daemon and the client are invoked with settings which are independent of each other, but the client must be invoked with a knowledge of the daemon's configuration, otherwise the connection may fail
- Use of the `--tlsverify` command line option implies `--tls`, which is then not required on the side in question

  Daemon

  Mode 1: using all four options with appropriate values, the daemon authenticates a client's certificate against the certificate authority provided (`--tlsverify` and `--tlscacert`), and expects to be authenticated by the client (`--tlscert` and `–tlskey`)

  Mode 2: using `--tls`, `--tlscert` and `--tlskey`, the daemon responds to TLS (`--tls`) communication requests, expects to be authenticated (`--tlscert` and `--tlskey`) by the client, but does not authenticate the client itself

  Client

  Mode 1: using all four options with appropriate values, as above for the daemon, but vice versa; if the client omits `--tlscert` and `--tlskey`, however, the daemon needs to be operating in mode 2 for the connection to be successful

  Mode 2: using `--tls` alone, or in conjunction with `--tlscert` and `--tlskey`, the client will authenticate server based on default or public CA pool

- It goes without saying that mode 1 should be employed at both ends in production

Notes

Securing Remote Comms

- Client commands can be simplified with use of environment variables
- Client's certificates and key can be placed in user's directory `~/.docker`

The environment variables used to simplify client command execution are:

Environment Variable	Description
`DOCKER_CERT_PATH`	Alternate location to `.docker` for certificates
`DOCKER_HOST`	Socket to connect to, e.g. `tcp://miarka:2376`
`DOCKER_TLS_VERIFY`	When set to '1', client authenticates daemon against CA

Securing Remote Communication

- Docker's client can retrieve a certificate, key and CA certificate automatically, if they are placed in a directory called `.docker` within the user's home directory, and the files are named `cert.pem`, `key.pem` and `ca.pem`, respectively
- An alternative location for the these files can be set using the environment variable `DOCKER_CERT_PATH`, e.g.

```
DOCKER_CERT_PATH=/home/jbloggs/.certs
```

- It's also possible to omit the host configuration option (`-H`) if the `DOCKER_HOST` environment variable is set appropriately for the user, e.g.

```
DOCKER_HOST=tcp://miarka:2376
```

- Additionally, for convenience, setting the `DOCKER_TLS_VERIFY` environment variable will force the Docker client to authenticate the daemon against the CA certificate located in `.docker` without the need to specify `--tlsverify`, e.g.

```
DOCKER_TLS_VERIFY=1
```

- Care should be taken to protect the keys used for remote communication, as anyone with access to the keys can legitimately interact with the Docker daemon, effectively giving them `root` access to the Docker host
- Change the permissions on the keys to 0400, and 0444 for the certificates
- If you have the Docker engine running locally on Linux, in your own time you can attempt the extra lab in Appendix 2; Secure Communication Between Client and Daemon

Notes