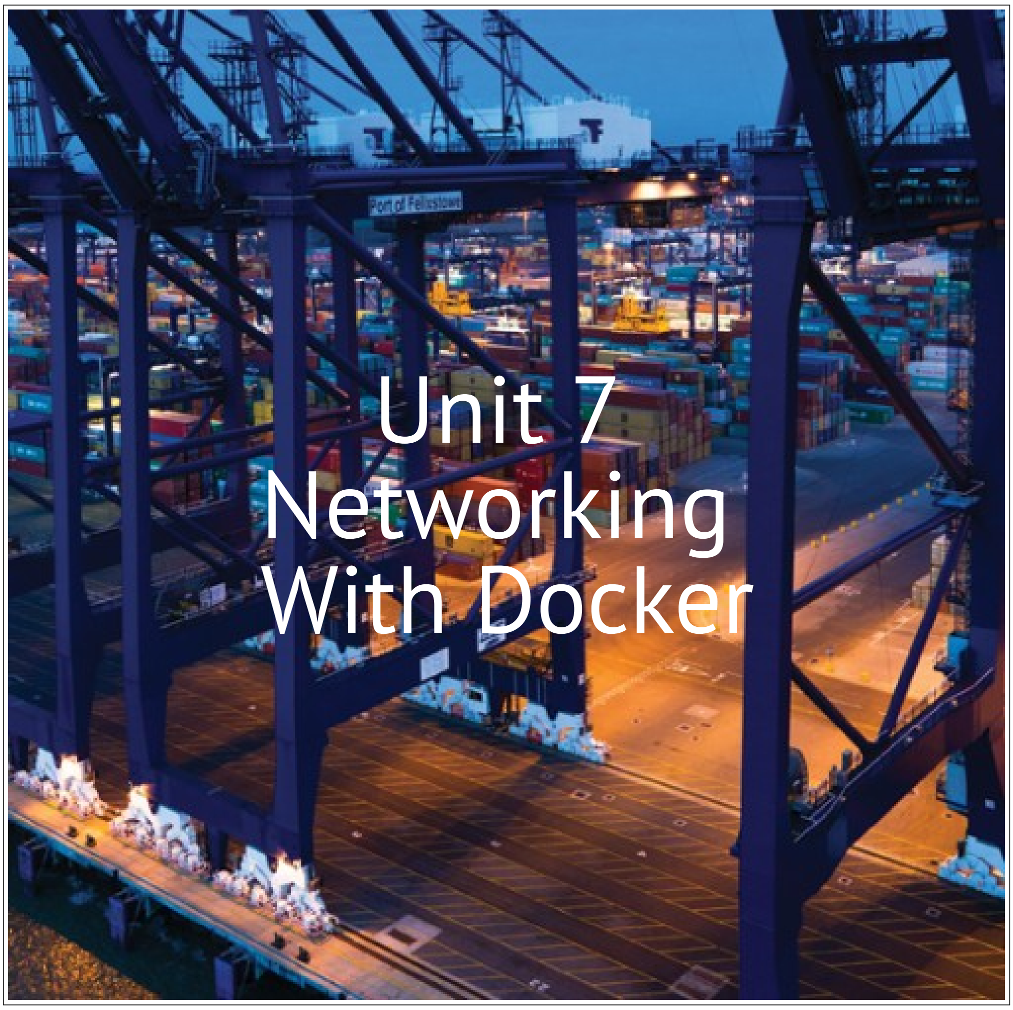


Unit 7 Networking With Docker

Objectives

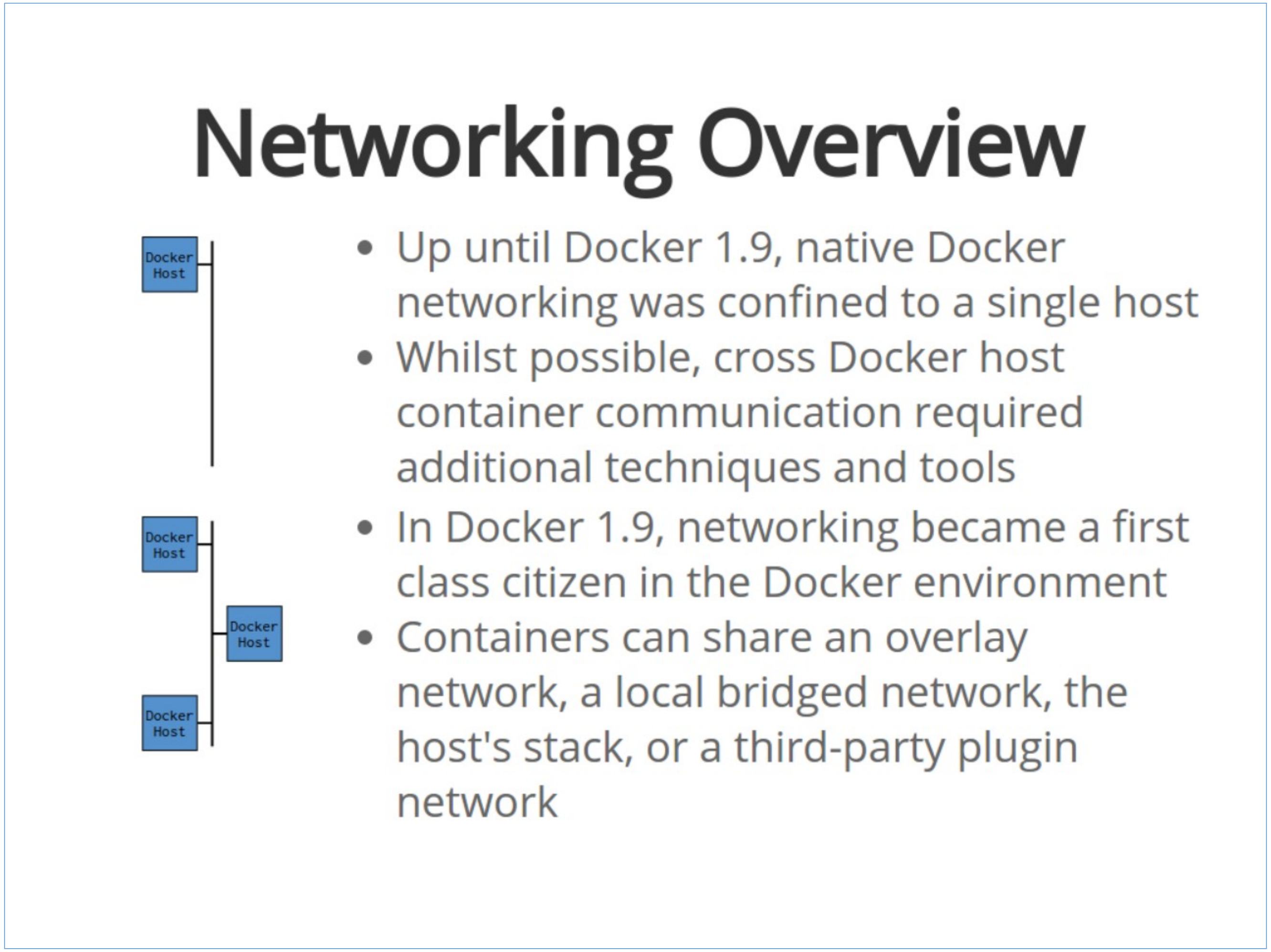
Upon completion of this unit, you will be able to:

- Describe Docker's approach to networking containers
- Distinguish between legacy and current networking modes
- Manipulate Docker networks with the docker network command
- Detail how to expose a container's service for consumption
- Configure name and lookup services for containers

Agenda

The following topics will be covered in this unit:

- Networking Overview
- Enabling Container Networking
- Legacy Networking
- Container Network Model
- The docker network Commands
- Network Types
- Exposing Container Ports
- Names and Name Resolution
- The docker-proxy
- Lab Exercise: Networking Containers



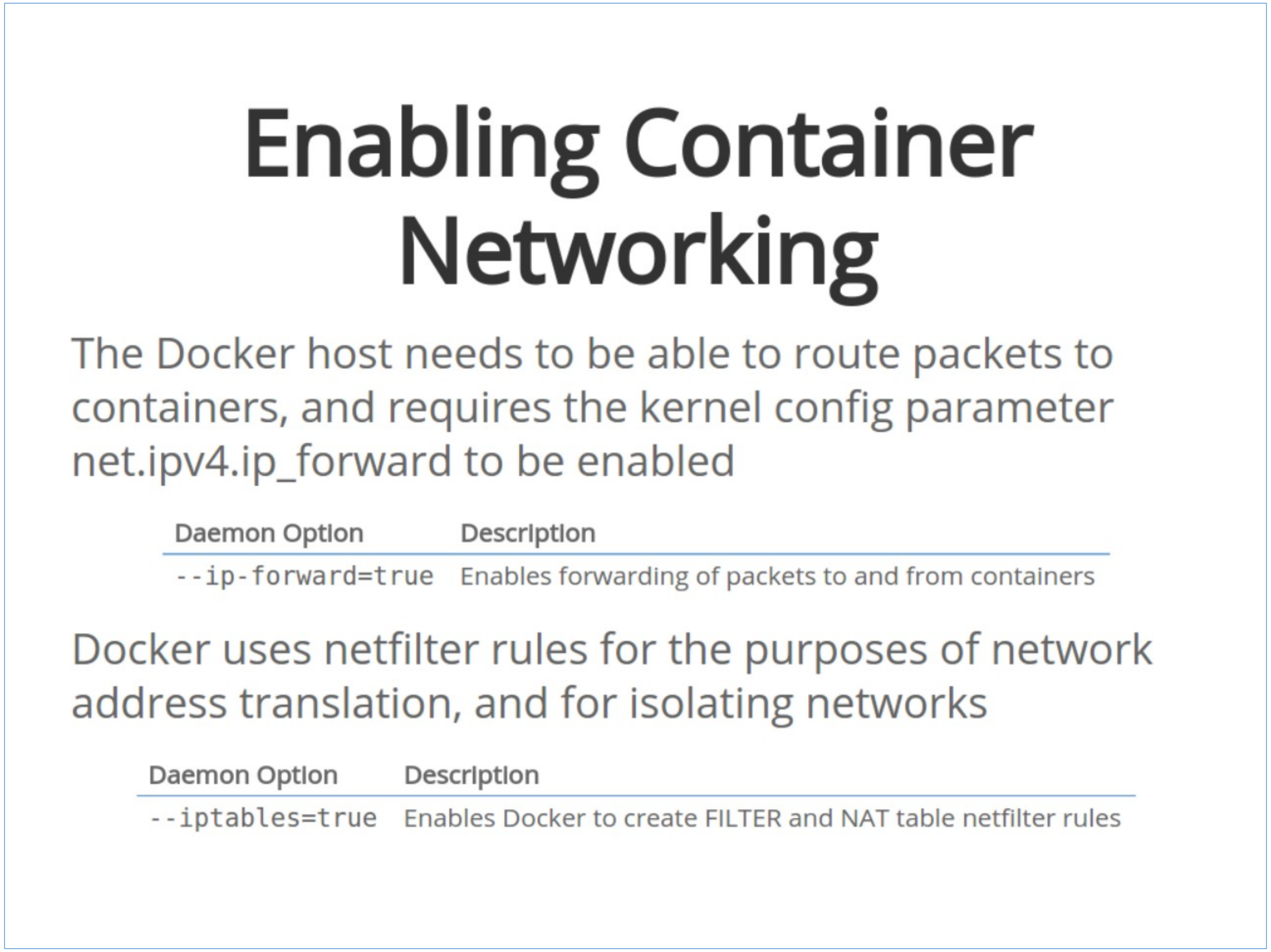
Networking Overview

- Up until Docker 1.9, native Docker networking was confined to a single host
- Whilst possible, cross Docker host container communication required additional techniques and tools
- In Docker 1.9, networking became a first class citizen in the Docker environment
- Containers can share an overlay network, a local bridged network, the host's stack, or a third-party plugin network

Networking Overview

- Docker provides a legacy networking capability (for backwards compatibility), and a more recent and comprehensive networking capability
- Prior to Docker version 1.9, there were several networking options available to containers, but they were confined to a single Docker host
- Inter-host networking for containers was possible, but required the use of proxy or ambassador containers, and service discovery techniques
- Single host networking served development environments reasonably well, but is not sufficient for enterprise grade, production scenarios
- Docker Inc. acquired SocketPlane in March 2015, whose expertise in networking subsequently yielded a complete overhaul of Docker's networking capabilities in version 1.9
- Docker's native networking now allows for the creation of overlay networks which span multiple Docker hosts, allowing containers running on different Docker hosts to communicate as if they were co-hosted
- Recognising the investment other third parties had made in developing Docker-related networking solutions, networking is delivered via a pluggable architecture
- Via plugins, it is possible to bypass Docker's 'batteries included' networking solution, and opt for a solution from one of the growing number of solutions from third parties, which may better fit a specific scenario
- From version 1.9, Docker comes with three networks pre-defined, which reflect its legacy networking, and which cannot be removed; they are named: `bridge`, `none`, and `host`
- By default, and unless otherwise specified, created containers belong to the legacy `bridge` network

Notes



Enabling Container Networking

The Docker host needs to be able to route packets to containers, and requires the kernel config parameter net.ipv4.ip_forward to be enabled

Daemon Option	Description
`--ip-forward=true`	Enables forwarding of packets to and from containers

Docker uses netfilter rules for the purposes of network address translation, and for isolating networks

Daemon Option	Description
`--iptables=true`	Enables Docker to create FILTER and NAT table netfilter rules

Enabling Container Networking

- In order for packets to be sent from and to containers, the host for the Docker daemon needs to be able to route packets
- Most Linux distributions don’t have IP forwarding enabled by default, as the assumption is that the host will be a ‘leaf node’, and unless configured otherwise, Docker will alter the host’s state in order to enable routing
- The Docker daemon command line option `--ip-forward` controls this behaviour, and must be set to `false` to stop Docker from turning the host into a router

`--ip-forward=true`: specifies whether the Docker host should route packets to and from containers

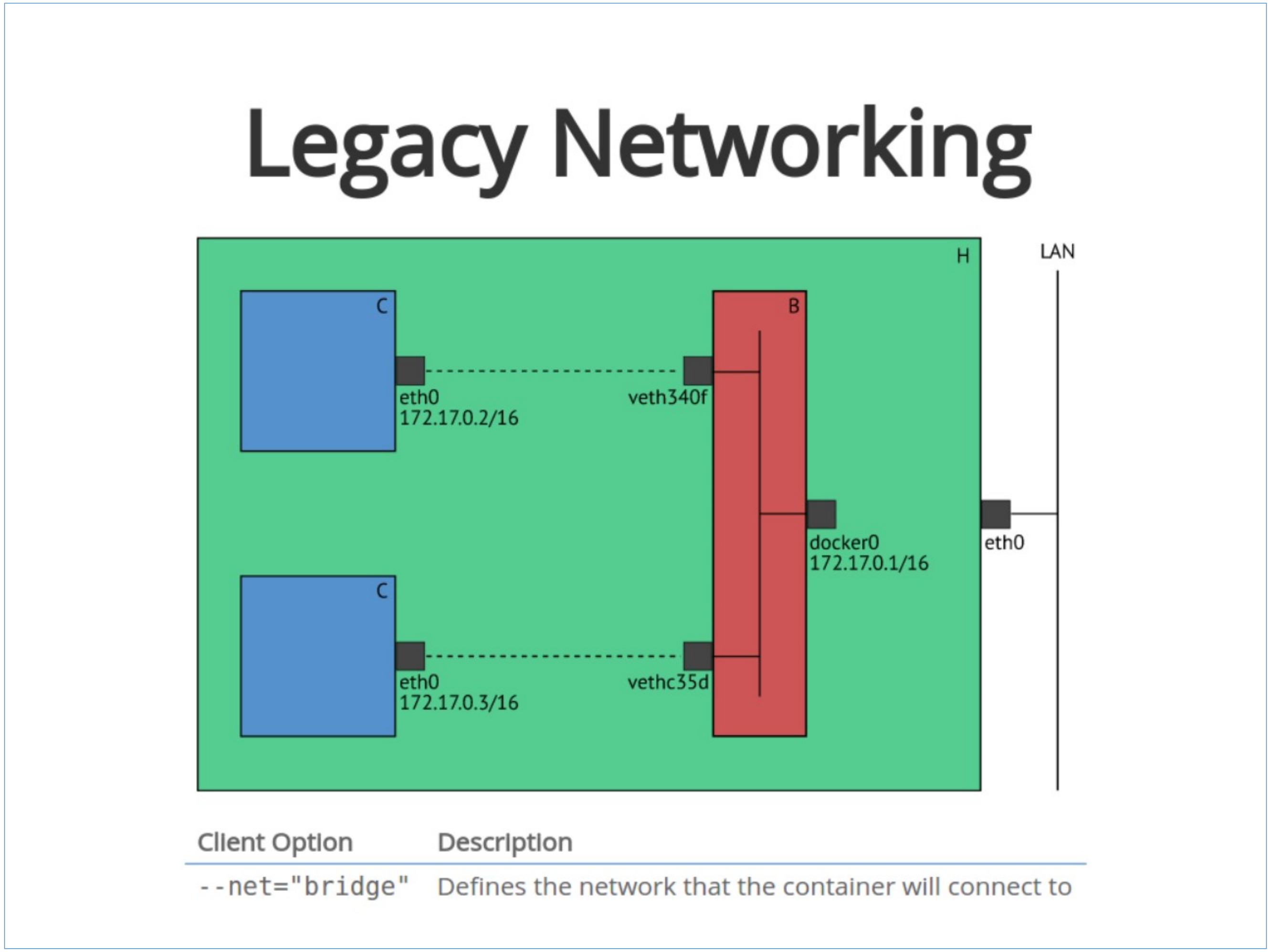
- The kernel parameter for IP forwarding can be queried and set using the `sysctl` command:

```
# sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
# sysctl -w net.ipv4.ip_forward=1
# sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

- As well as requiring routing to be enabled, Docker also makes use of the kernel’s netfilter framework for managing and controlling the data that is sent and received by containers
- Docker will create netfilter rules in both the FILTER and NAT tables by default, but this behaviour can be curtailed with the use of the `--iptables` daemon configuration option

`--iptables=true`: specifies whether the Docker host can write netfilter rules to control the flow of data to and from containers

Notes



Legacy Networking

Client Option	Description
`--net="bridge"`	Defines the network that the container will connect to

Legacy Networking Modes

- Docker uses the `--net` client configuration option to specify which network a container connects to, irrespective of whether it uses legacy networking or Docker's newer, more advanced networking features

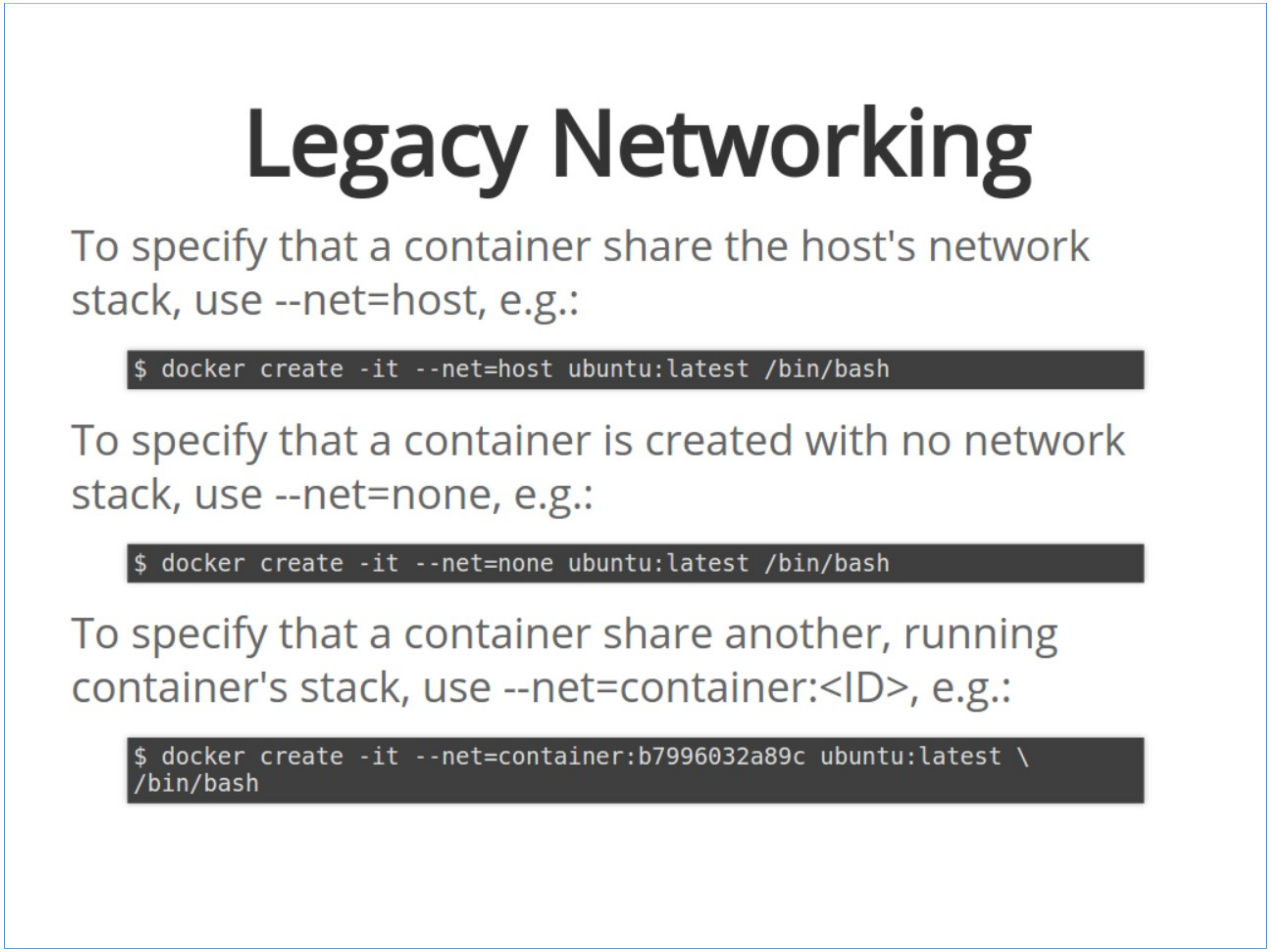
`--net=”bridge”`: sets the mode of networking for a container; can be `bridge` (default), `host`, `container`, `none`, or a user-defined network

- At start-up, the Docker daemon creates three default networks, called '`bridge`', '`none`' and '`host`'

<u>Bridge</u>

- The default bridge network is provided for backwards compatibility; it is recommended that it is not used (it is expected to disappear in the future)
- This default ethernet bridge is virtual in nature, and works in conjunction with netfilter rules to aid inter-container communication, and for exposing services to and beyond the Docker host
- By default, any container invoked will be connected to this bridge, with its network stack contained within its own NET namespace, but with its primary network interface connected to the bridge
- When a local Docker environment is being used, it is suggested that a new bridge network is created in place of the legacy, default bridge network
- There are subtle differences between the default bridge network (called `bridge`), and any user defined bridge networks
- In the legacy, default bridge scenario, in order for a container to consume another container's service, the containers must be 'linked', a mechanism which provides the consumer with the IP address of the provider (amongst other things)
- With a user-defined bridge, containers connected to the bridge's network can discover the services of other containers with the use of Docker's embedded DNS server for service discovery

Notes



Legacy Networking

To specify that a container share the host's network stack, use --net=host, e.g.:

```
$ docker create -it --net=host ubuntu:latest /bin/bash
```

To specify that a container is created with no network stack, use --net=none, e.g.:

```
$ docker create -it --net=none ubuntu:latest /bin/bash
```

To specify that a container share another, running container's stack, use --net=container:<ID>, e.g.:

```
$ docker create -it --net=container:b7996032a89c ubuntu:latest \
/bin/bash
```

Legacy Networking Modes

Host

- If the Docker client command specifies host mode for networking, then the container shares the same NET namespace and network stack (interfaces, ports etc.) as the host
- Care should be taken when using host mode, as the default user in the container has a UID of `0` (albeit with diminished capabilities), and will have the ability to access network resources and services that unprivileged users will not have, although it will not be able to reconfigure services

```
$ docker create -it --net=host ubuntu:latest /bin/bash
```

None

- With `--net=none`, the container is created with just a loopback interface to allow container-specific services to communicate
- Although no primary network interface is created, the container is still created in its own NET namespace
- This allows for scenarios where a container's networking is established manually rather than using the default setup provided by the Docker daemon

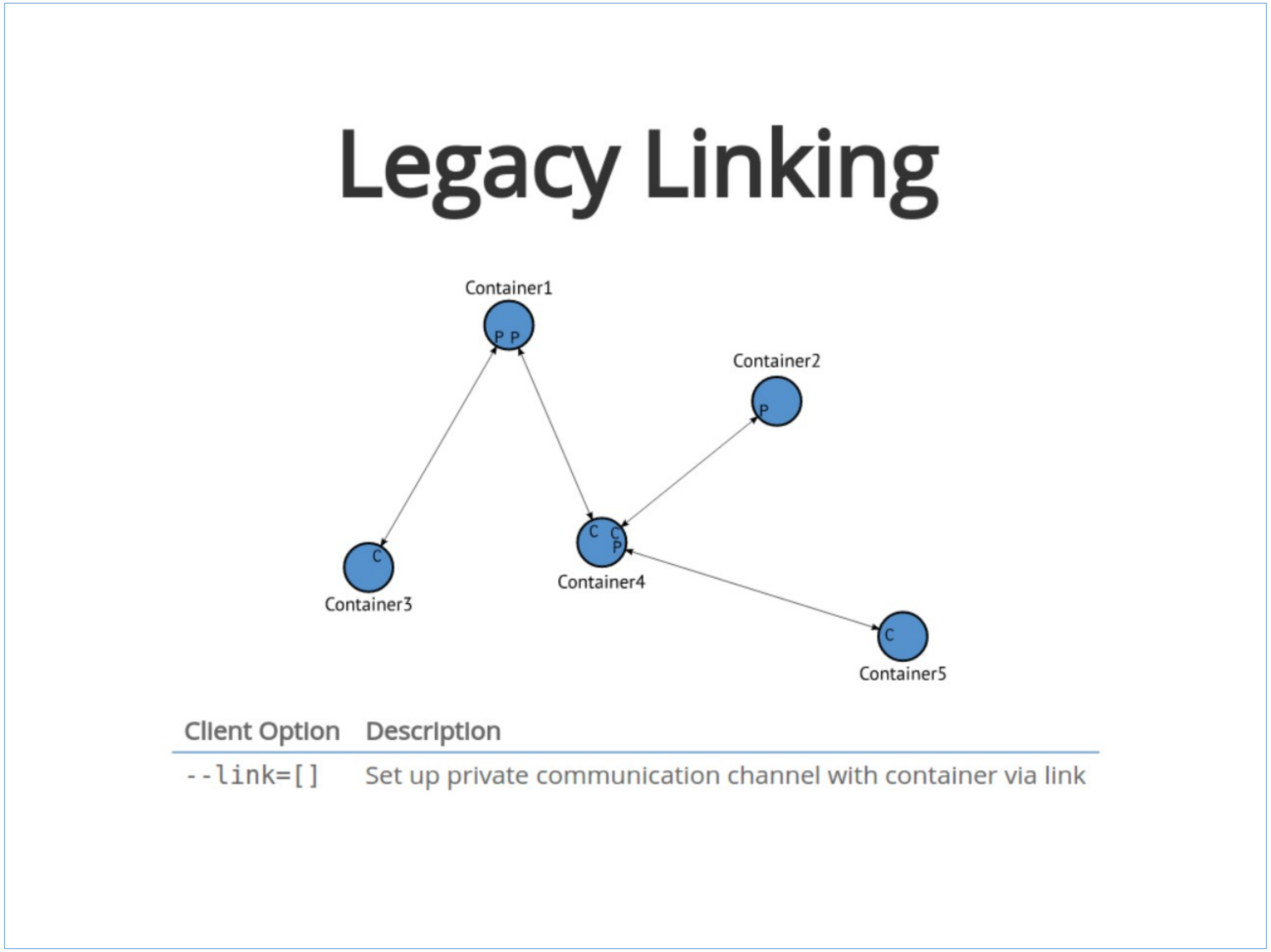
```
$ docker create -it --net=none ubuntu:latest /bin/bash
```

Container Mode

- When a container is invoked with networking set to container mode, the container shares the network stack of another specified container, which must already be running and attached to the same bridge network
- The containers share the same NET namespace and network resources

```
$ docker create -it --net=container:b7996032a89c ubuntu:latest \
/bin/bash
```

Notes



Legacy Linking

Client Option	Description
`--link=[]`	Set up private communication channel with container via link

Legacy Container Linking

- Containers connected to the legacy bridge can be linked together to provide a more private method of communication, possibly to the exclusion of all other containers
- The link between two containers is instigated by a 'consumer' container to a 'provider' container, with the provider providing the consumer with details concerning its configuration
- A provider container can be linked to numerous consumers, and a consumer container can have numerous providers
- Clearly, the provider container must already exist before the consumer container attempts to link to it
- If the link is established whilst the provider container is not running, then it needs to be started before the consumer can be started, otherwise the `docker run` command will fail

`--link=[]`: in the context of a legacy bridge network, specifies that a consumer container links to a provider container, which provides the consumer with its configuration details*

- The `--link` configuration option must be accompanied by the name of the provider container to link to, along with an (optional) alias for the link (name:alias), e.g.

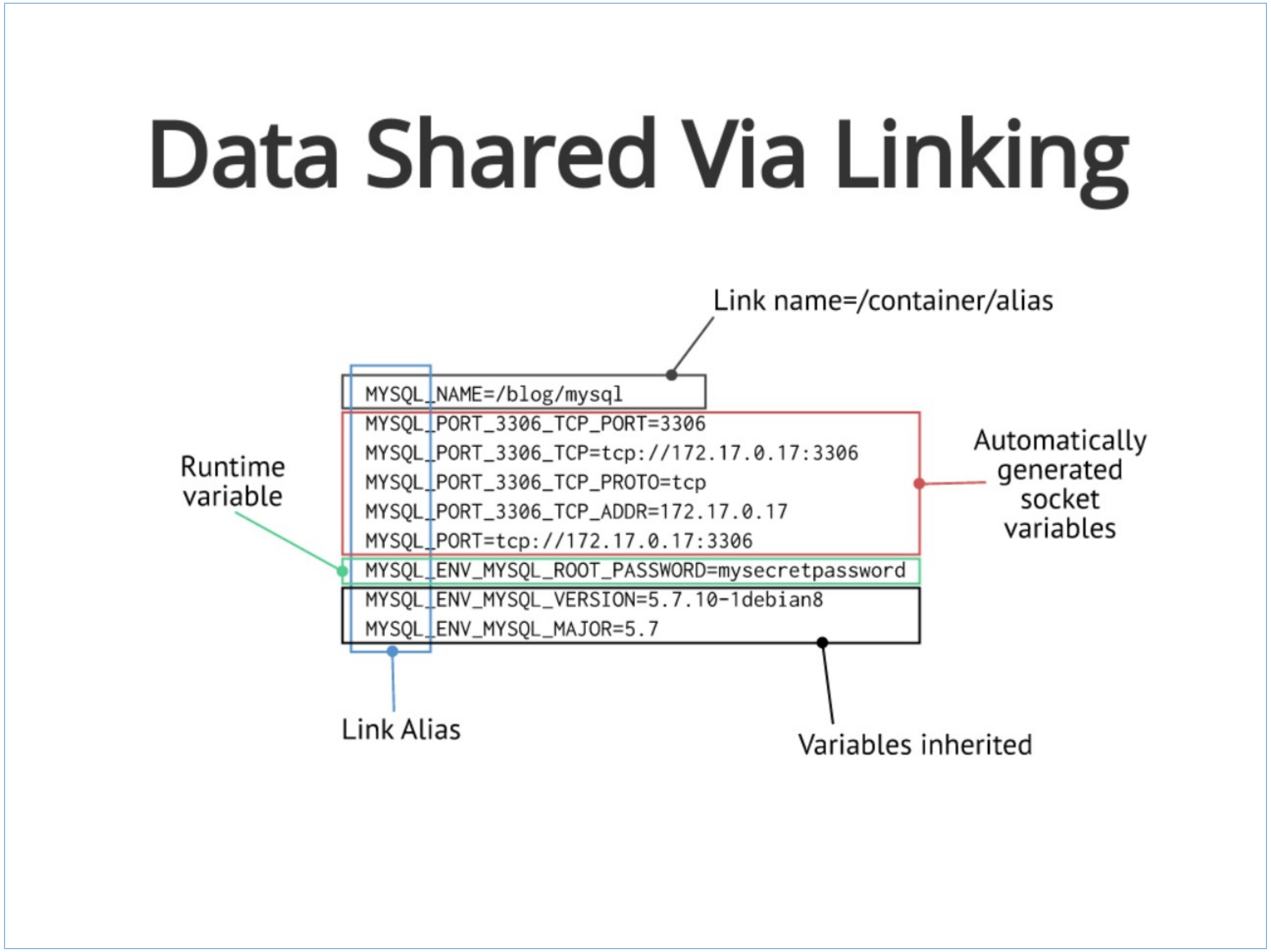
  ```
  $ docker run -d --name=blog --link=mysql_db:my_link wordpress
  ```

- Details of any links a container has established, can be retrieved with the `docker inspect` command:

  ```
  $ docker inspect -f '{{.HostConfig.Links}}' blog
  [/mysql_db:/blog/my_link]
  ```

- Docker stores the names of containers, and any links that exist between containers, in a sqlite database called `linkgraph.db`

* The '`--link`' configuration option has an entirely different behaviour in the context of non-legacy bridge networks



Data Shared Via Linking

```
MYSQL_NAME=/blog/mysql
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP=tcp://172.17.0.17:3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_PORT_3306_TCP_ADDR=172.17.0.17
MYSQL_PORT=tcp://172.17.0.17:3306
MYSQL_ENV_MYSQL_ROOT_PASSWORD=mysecretpassword
MYSQL_ENV_MYSQL_VERSION=5.7.10-1debian8
MYSQL_ENV_MYSQL_MAJOR=5.7
```

Link name=/container/alias
Runtime variable
Automatically generated socket variables
Link Alias
Variables inherited

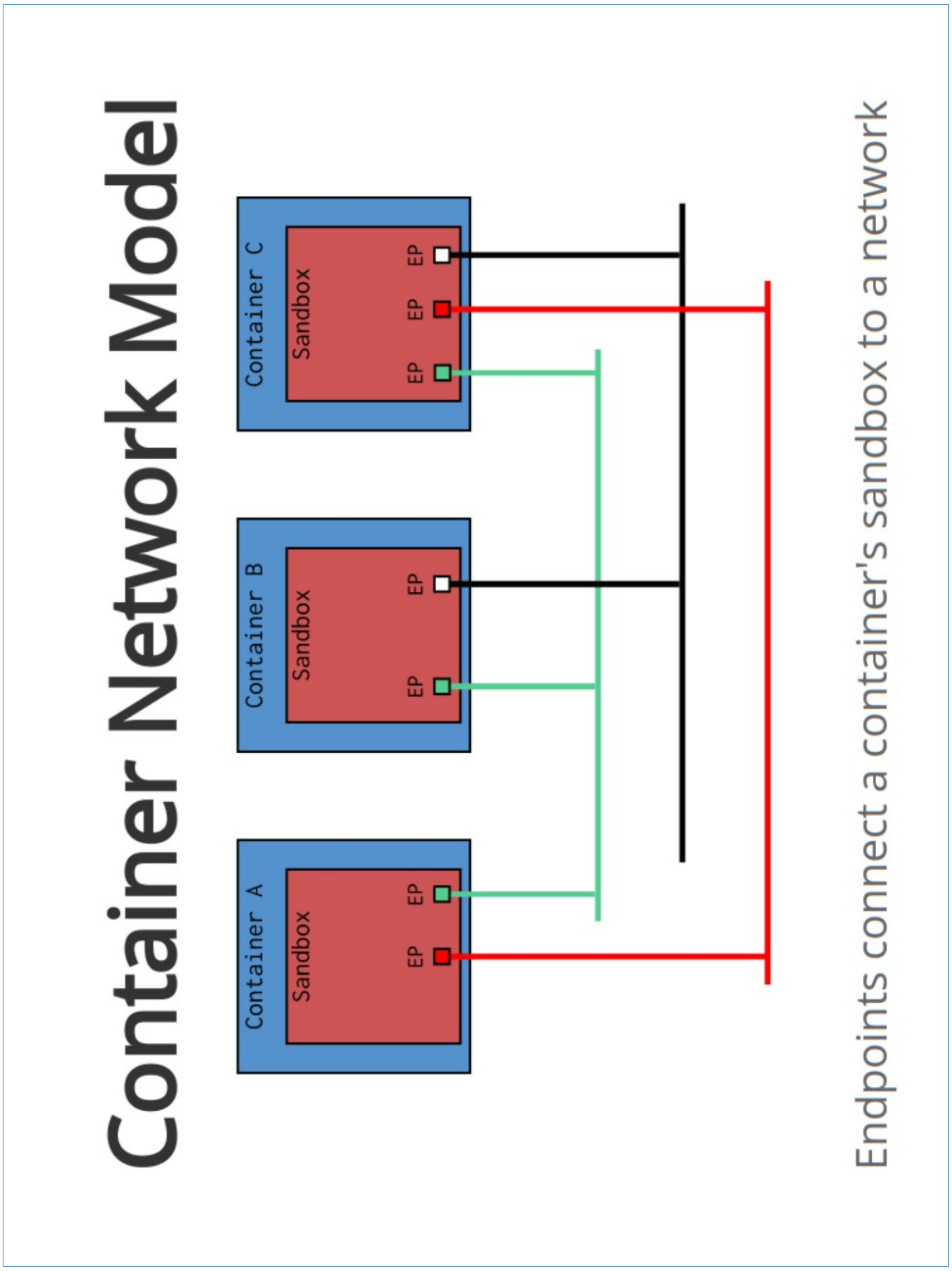
Data Shared Via Linking

- When two containers are linked, the provider container provides the consumer container with items of data (in the form of environment variables) that may be of use to the consuming container when consuming the child container's service
- An environment variable with the link name is provided, and if a port or ports are exposed in the provider container, then a number of environment variables are created that relate to the exposed socket(s):

```
MYSQL_NAME=/blog/mysql
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP=tcp://172.17.0.17:3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_PORT_3306_TCP_ADDR=172.17.0.17
MYSQL_PORT=tcp://172.17.0.17:3306
```

- As well as port related information, the consumer container will also receive any environment variables that are defined in the image from which the provider container is derived from, or that were specified from using the `-e`, `--env` or `--env-file` configuration options for `docker create` or `docker run`
- In addition to the environment variables created in the consumer container when it is linked with a provider container, an entry is also created in the consumer container's `/etc/hosts` file referencing the provider container; the names that accompany the provider container's IP address are the link alias, the container ID and the container's name
- It should be noted that in the event that the provider container is restarted, whilst the new IP address associated with the restarted container is reflected in the consumer container's `/etc/hosts` file, any port-related environment variables in the consumer container referencing the old IP address, are not updated without a restart of the consumer container

Notes



Container Network Model

Endpoints connect a container's sandbox to a network

Container Network Model

- Docker's new (as of v1.9) networking capability, revolves around the Container Network Model (CNM)
- The CNM provides the necessary abstraction, implemented as a set of APIs, to cater for numerous networking use cases
- The CNM is written in Golang, and is delivered as a library called libnetwork
- The CNM comprises of the following components:

  Sandbox

  A sandbox provides the network stack configuration for a container (e.g. interfaces, routes), and is currently implemented as a Linux NET namespace. A container has a single sandbox associated with it.

  Endpoint

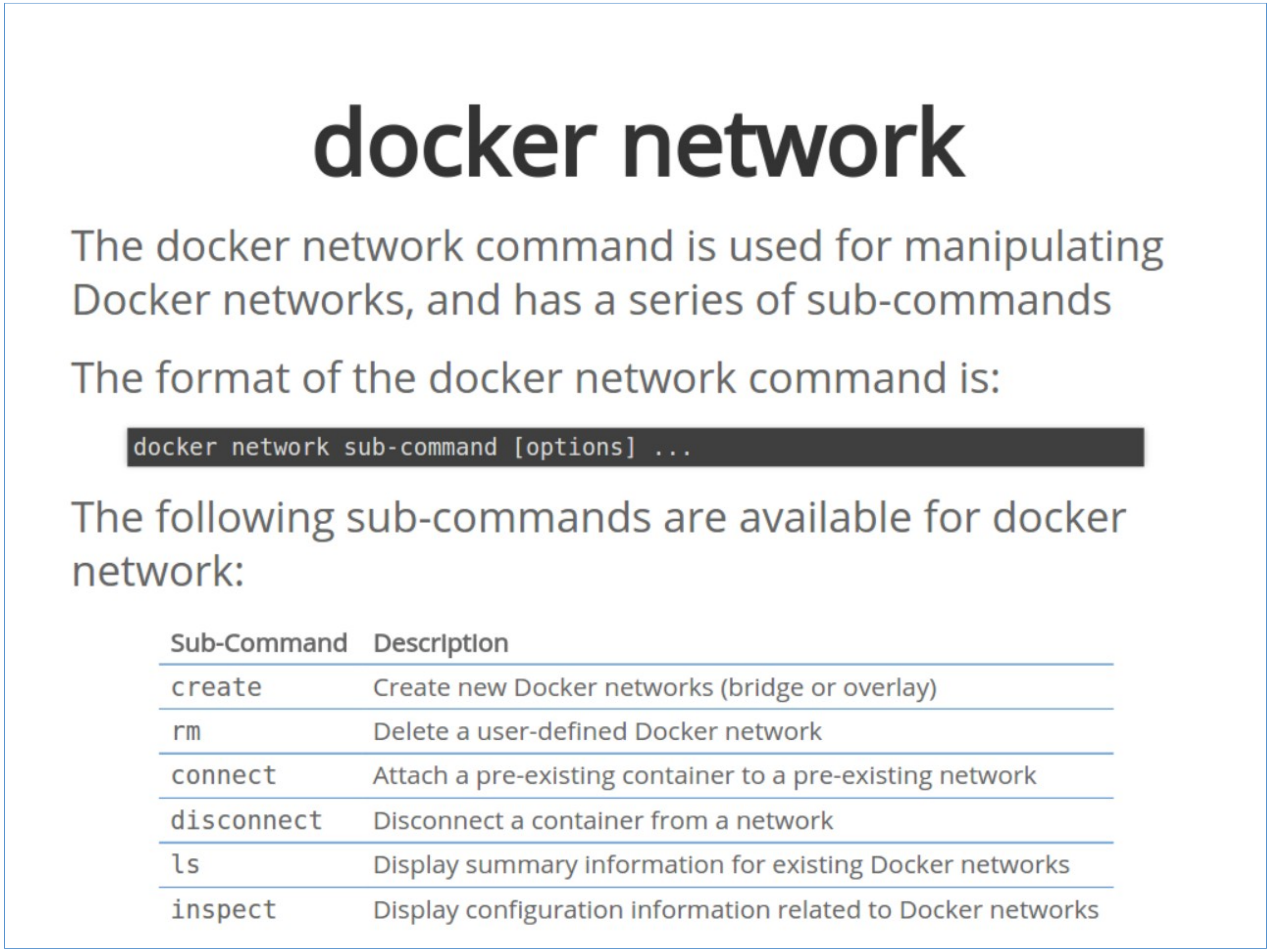
  An endpoint can be viewed as a 'service endpoint', and is used to join a container's sandbox to a specific network. Multiple endpoints, from different networks, can co-exist in a sandbox, allowing a container to be connected to multiple networks. An endpoint can only be connected to one network and one sandbox.

  Network

  A network is a collection of connected endpoints, and can be local or global in scope, depending on whether the network spans multiple Docker hosts. Each network has an associated driver, which takes care of implementing the network in terms of connectivity and isolation. Docker has several 'built-in' drivers, but also has the facility to make use of third-party drivers (or plugins) to support the 'batteries included, but swappable' concept

- An example of a third-party CNM-compliant driver is Weaveworks' Weave Net

Notes



docker network

The docker network command is used for manipulating Docker networks, and has a series of sub-commands

The format of the docker network command is:

```
docker network sub-command [options] ...
```

The following sub-commands are available for docker network:

Sub-Command	Description
`create`	Create new Docker networks (bridge or overlay)
`rm`	Delete a user-defined Docker network
`connect`	Attach a pre-existing container to a pre-existing network
`disconnect`	Disconnect a container from a network
`ls`	Display summary information for existing Docker networks
`inspect`	Display configuration information related to Docker networks

docker network

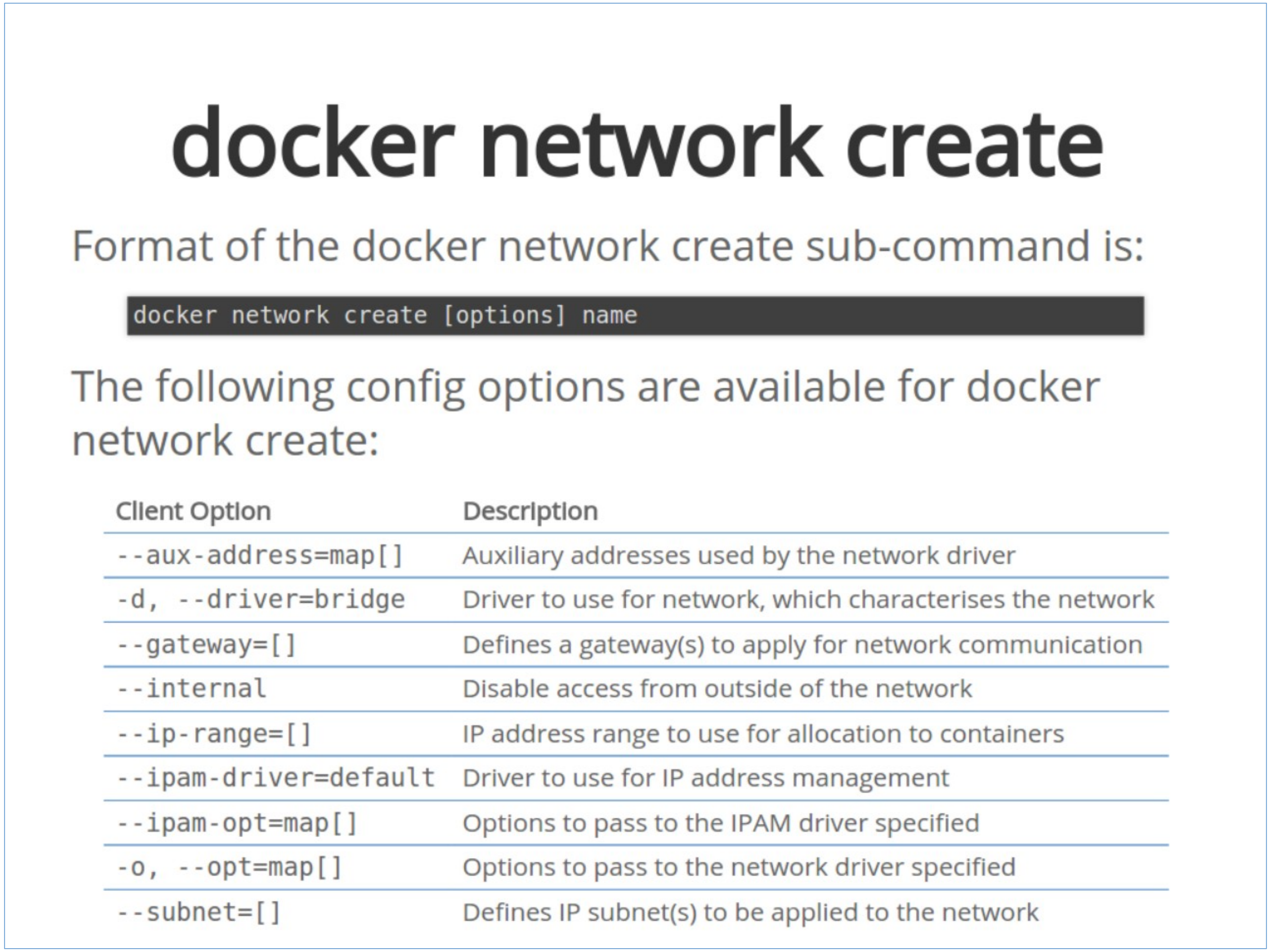
- Access to Docker's networking capabilities is provided via the Docker client CLI or the Docker Remote API
- The Docker client command is `docker network`, and has several sub-commands
- The format of the `docker network` command is:

```
      docker network sub-command [options] ...
```

- The available docker network sub-commands are:

  create: used for the creation of independent Docker networks
  rm: used to remove the specified network
  connect: used to attach a running container to an existing network
  disconnect: used to disconnect a running container from a network it is attached to
  ls: used to list the available networks registered with a Docker host (or cluster)
  inspect: used to inspect the details associated with a network, including the containers connected to it

Notes



docker network create

Format of the docker network create sub-command is:

```
docker network create [options] name
```

The following config options are available for docker network create:

Client Option	Description
`--aux-address=map[]`	Auxiliary addresses used by the network driver
`-d, --driver=bridge`	Driver to use for network, which characterises the network
`--gateway=[]`	Defines a gateway(s) to apply for network communication
`--internal`	Disable access from outside of the network
`--ip-range=[]`	IP address range to use for allocation to containers
`--ipam-driver=default`	Driver to use for IP address management
`--ipam-opt=map[]`	Options to pass to the IPAM driver specified
`-o, --opt=map[]`	Options to pass to the network driver specified
`--subnet=[]`	Defines IP subnet(s) to be applied to the network

docker network create

- The `docker network create` sub-command is used to create networks according to the driver provided by the `--driver` configuration option
- The format of the `docker network create` sub-command is:

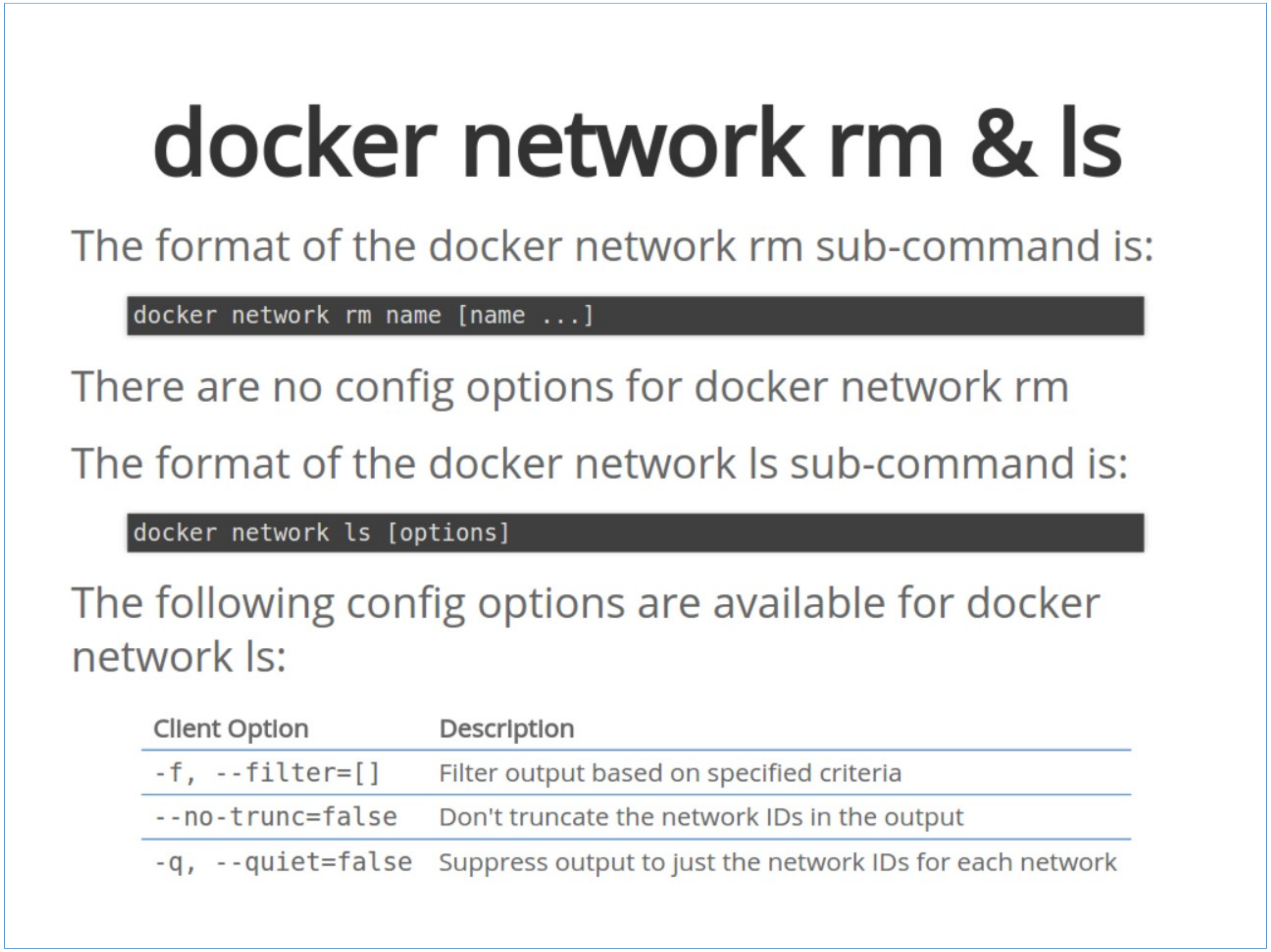
```
docker network create [options] name
```

- The configuration options available for the `docker network create` sub-command are:

`--aux-address=map[]`: specifies an auxiliary IPv4/IPv6 address for the device driver
`-d, --driver=bridge`: the driver to be used to manage the network, which may be `bridge`, `overlay` or that designated by a custom or third-party plugin
`--gateway=[]`: the gateway to be used for the network's subnet
`--internal`: disables external access to an overlay network
`--ip-range=[]`: specifies a range of IP addresses that will be used to allocate to containers
`--ipam-driver=default`: the driver to be used for managing IP addresses, the default being Docker's built-in IPAM
`--ipam-opt=map[]`: used to pass IPAM-specific configuration options for the IPAM driver in operation
`-o, --opt=map[]`: used to pass network-driver specific configuration options for the network driver
`--subnet=[]`: specifies an IP subnet (or non-overlapping IP subnets) in CIDR format, for the network to be created with (bridge networks are limited to a single subnet)

- Libnetwork provides a built-in IP Address Management driver for networks created with `docker network create`, which can be replaced by an alternative third party driver
- Whilst it's possible to specify a single IP subnet for a bridge network, it is possible to specify multiple subnets for overlay networks, usually with accompanying gateway specification

Notes



docker network rm & ls

The format of the docker network rm sub-command is:

```
docker network rm name [name ...]
```

There are no config options for docker network rm

The format of the docker network ls sub-command is:

```
docker network ls [options]
```

The following config options are available for docker network ls:

Client Option	Description
`-f, --filter=[]`	Filter output based on specified criteria
`--no-trunc=false`	Don't truncate the network IDs in the output
`-q, --quiet=false`	Suppress output to just the network IDs for each network

docker network rm & ls

- The `docker network rm` sub-command is used to remove Docker networks
- The format of the `docker network rm` sub-command is:

```
        docker network rm name [name ...]
```

- There are no configuration options for the `docker network rm` sub-command
- Before a network can be removed, there must be no containers attached to it (strictly speaking, no network endpoints)
- Multiple networks can be specified for removal, and attempts are made to remove all networks specified, even if the removal of one or more networks fail
- The `docker network ls` sub-command lists all of the networks registered with the Docker daemon
- The format of the `docker network ls` sub-command is:

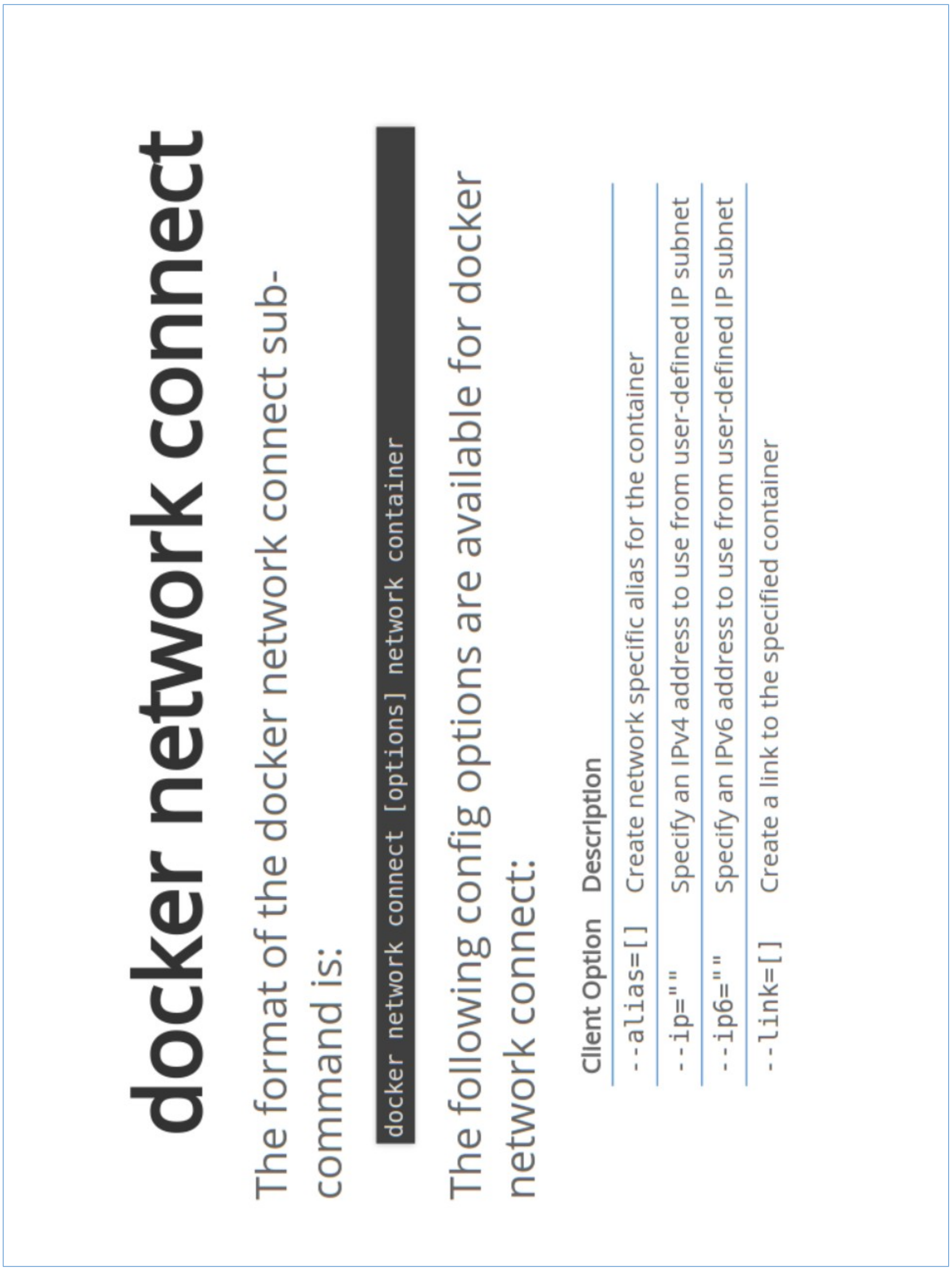
```
        docker network ls [options]
```

- The configuration options available for the `docker network ls` sub-command are:

`-f, --filter=[]`: filters the output based on the criteria provided
`--no-trunc=false`: specifies that the IDs of each network in the output be truncated to a 12-digit hex string
`-q, --quiet=false`: specifies that the output is limited to the IDs of the network only; useful when concatenating `docker network ls` and `docker network rm` commands together

- Filters are specified as key/value pairs, and the available filters are: id (`id=746ab1cf50b5`), name (`name=local_bridge`), or type (where type is `builtin|custom`, e.g. `type=builtin` or `type=custom`)
- Specification of multiple filter criteria represents and behaves like an OR construct

Notes



docker network connect

The format of the docker network connect sub-command is:

```
docker network connect [options] network container
```

The following config options are available for docker network connect:

Client Option	Description
`--alias=[]`	Create network specific alias for the container
`--ip=""`	Specify an IPv4 address to use from user-defined IP subnet
`--ip6=""`	Specify an IPv6 address to use from user-defined IP subnet
`--link=[]`	Create a link to the specified container

docker network connect

- The `docker network connect` sub-command is used to attach a running container to a pre-existing network
- The format of the `docker network connect` sub-command is:

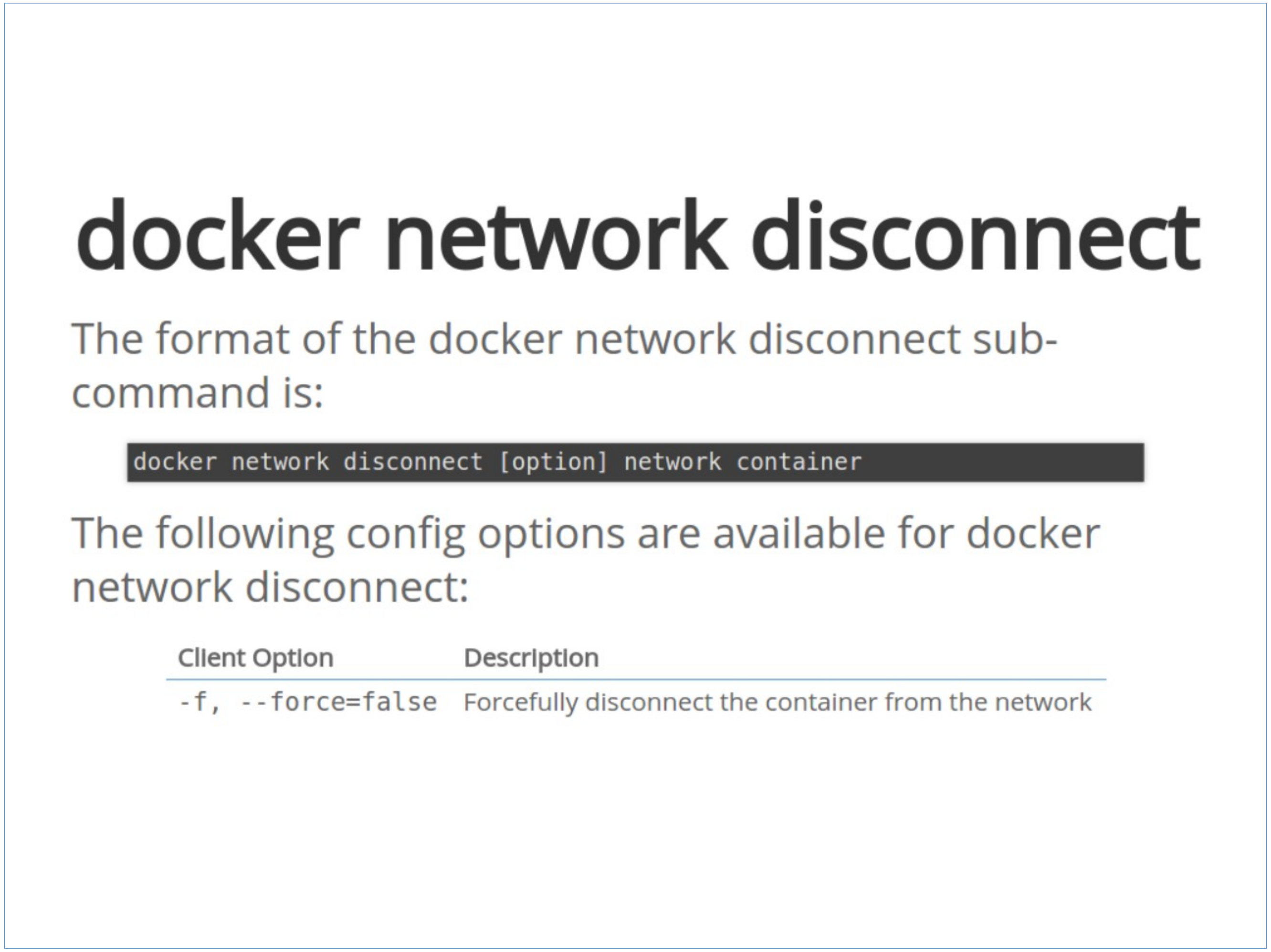
```
docker network connect [options] network container
```

- The configuration options available for the `docker network connect` sub-command are:

`--alias=[]`: adds an alias for the container, which is scoped to the specific network only
`--ip=””`: allocate a specific IPv4 IP address for the container, provided network was created with user-defined IP subnet
`--ip6=””`: allocate a specific IPv6 IP address for the container, provided network was created with user-defined IP subnet
`--link=[]`: add a link (for the purposes of an alias) to another container on the same network

- The `--ip` and `--ip6` configuration options can also be used to the same effect, in the context of creating or running a container, as options for the `docker create` or `docker run` commands

Notes



docker network disconnect

The format of the docker network disconnect sub-command is:

```
docker network disconnect [option] network container
```

The following config options are available for docker network disconnect:

Client Option	Description
`-f, --force=false`	Forcefully disconnect the container from the network

docker network disconnect

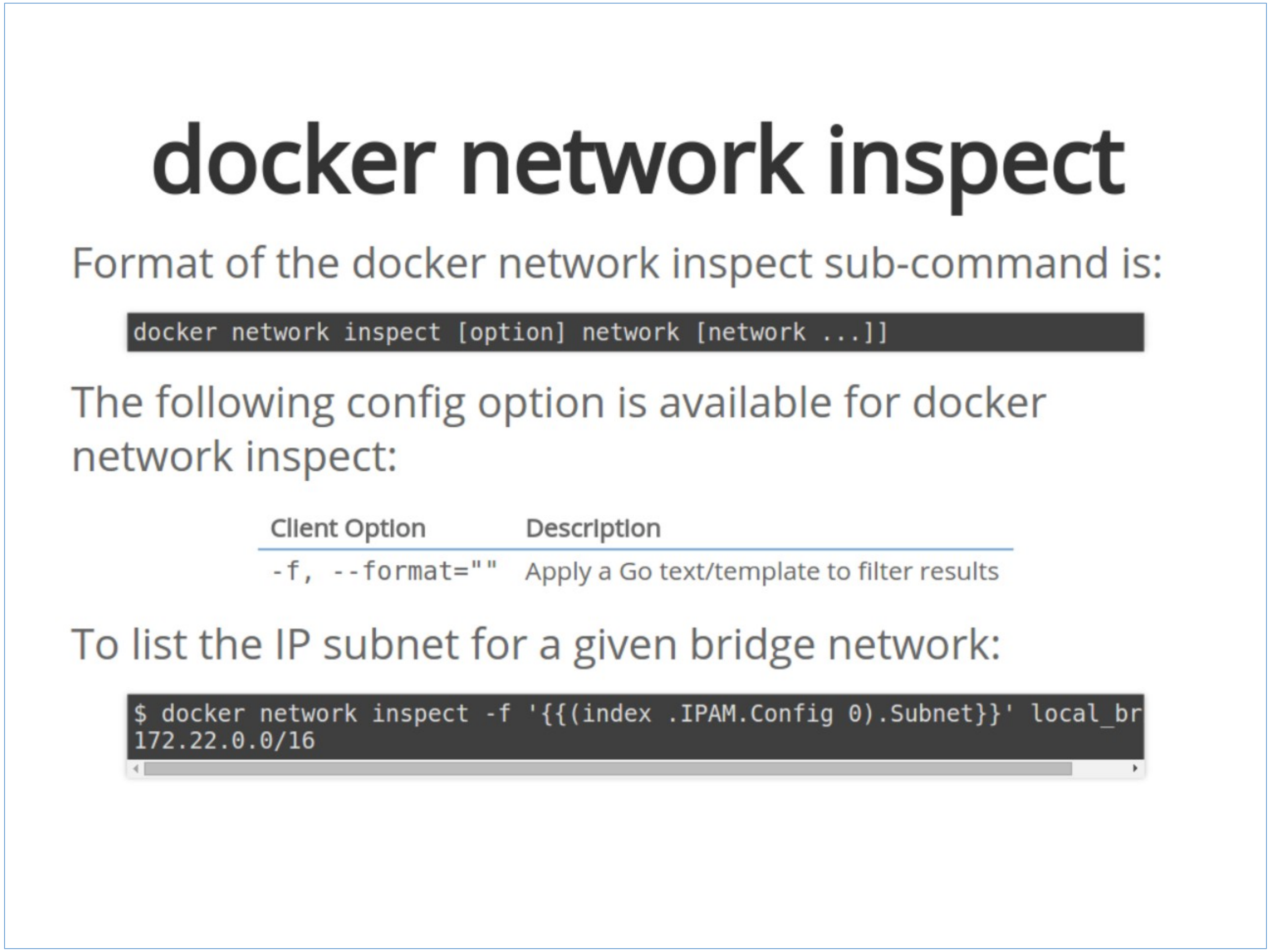
- The docker network disconnect sub-command is used to detach a running container from a network
- The format of the docker network disconnect sub-command is:

```
docker network disconnect [option] network container
```

- The configuration options available for the docker network disconnect sub-command are:

`--force=false`: forcefully disconnects the container from a network

Notes



docker network inspect

Format of the docker network inspect sub-command is:

```
docker network inspect [option] network [network ...]]
```

The following config option is available for docker network inspect:

Client Option	Description
`-f, --format=""`	Apply a Go text/template to filter results

To list the IP subnet for a given bridge network:

```
$ docker network inspect -f '{{(index .IPAM.Config 0).Subnet}}' local_br
172.22.0.0/16
```

docker network inspect

- The `docker network inspect` sub-command is used to glean information for each of the networks specified
- The format of the `docker network disconnect` sub-command is:

```
docker network inspect [option] network [network ...]
```

- The configuration options available for the `docker network inspect` sub-command are:

`-f, --format=””`: specifies a Go text/template to apply to the data objects in order to filter the results

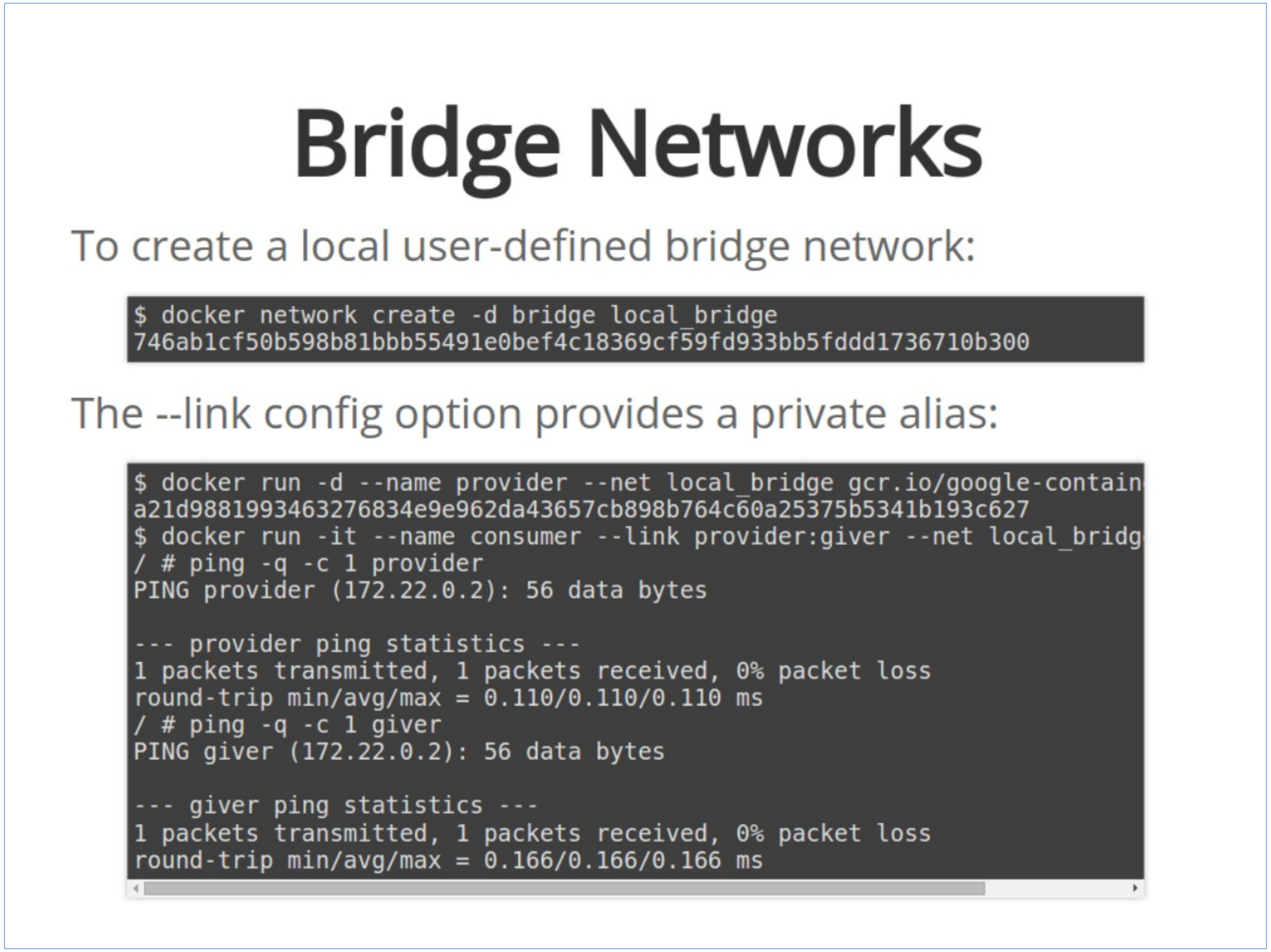
- If no format is specified, `docker network inspect` returns the complete configuration of the specified network(s) as a JSON object
- In order to list the containers connected to a specific network, using their names and IP addresses, the following Go text/template could be used:

```
$ docker network inspect -f '{{range $c := .Containers}}\
{{$c.Name}} -> {{$c.IPv4Address}}    {{end}}' local_bridge
elegant_roentgen -> 172.22.0.3/16    big_bhabha -> 172.22.0.2/1
```

- To list the IP subnet for a given bridge network:

```
$ docker network inspect -f '{{(index .IPAM.Config 0).Subnet}}' \
local_bridge
172.22.0.0/16
```

Notes



Bridge Networks

To create a local user-defined bridge network:

```
$ docker network create -d bridge local_bridge
746ab1cf50b598b81bbb55491e0bef4c18369cf59fd933bb5fddd1736710b300
```

The --link config option provides a private alias:

```
$ docker run -d --name provider --net local_bridge gcr.io/google-contain
a21d9881993463276834e9e962da43657cb898b764c60a25375b5341b193c627
$ docker run -it --name consumer --link provider:giver --net local_bridg
/ # ping -q -c 1 provider
PING provider (172.22.0.2): 56 data bytes

--- provider ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.110/0.110/0.110 ms
/ # ping -q -c 1 giver
PING giver (172.22.0.2): 56 data bytes

--- giver ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.166/0.166/0.166 ms
```

Bridge Networks

- For backwards compatibility, the Docker daemon creates a legacy bridge network (`docker0`) on startup, to which containers will be attached to by default*
- In order to create a user-defined bridge network called 'local_bridge', use the following command (the `-d` configuration option is optional as the bridge driver is the default driver):

```
$ docker network create -d bridge local_bridge
746ab1cf50b598b81bbb55491e0bef4c18369cf59fd933bb5fddd1736710b300
```

- Docker creates a Linux bridge device, with a name of the form `br-746ab1cf50b5`, where the numbers represent the shortened ID of the network representation in Docker
- Unless directed otherwise by the user on the command line, Docker assigns an IP subnet for the network, along with a gateway
- A number of entries are added to the Docker host's netfilter rules (`FILTER` and `NAT` tables), which are used to route traffic to and from containers connected to the network, isolate different bridge networks, and to provide IP masquerading when communicating with external services

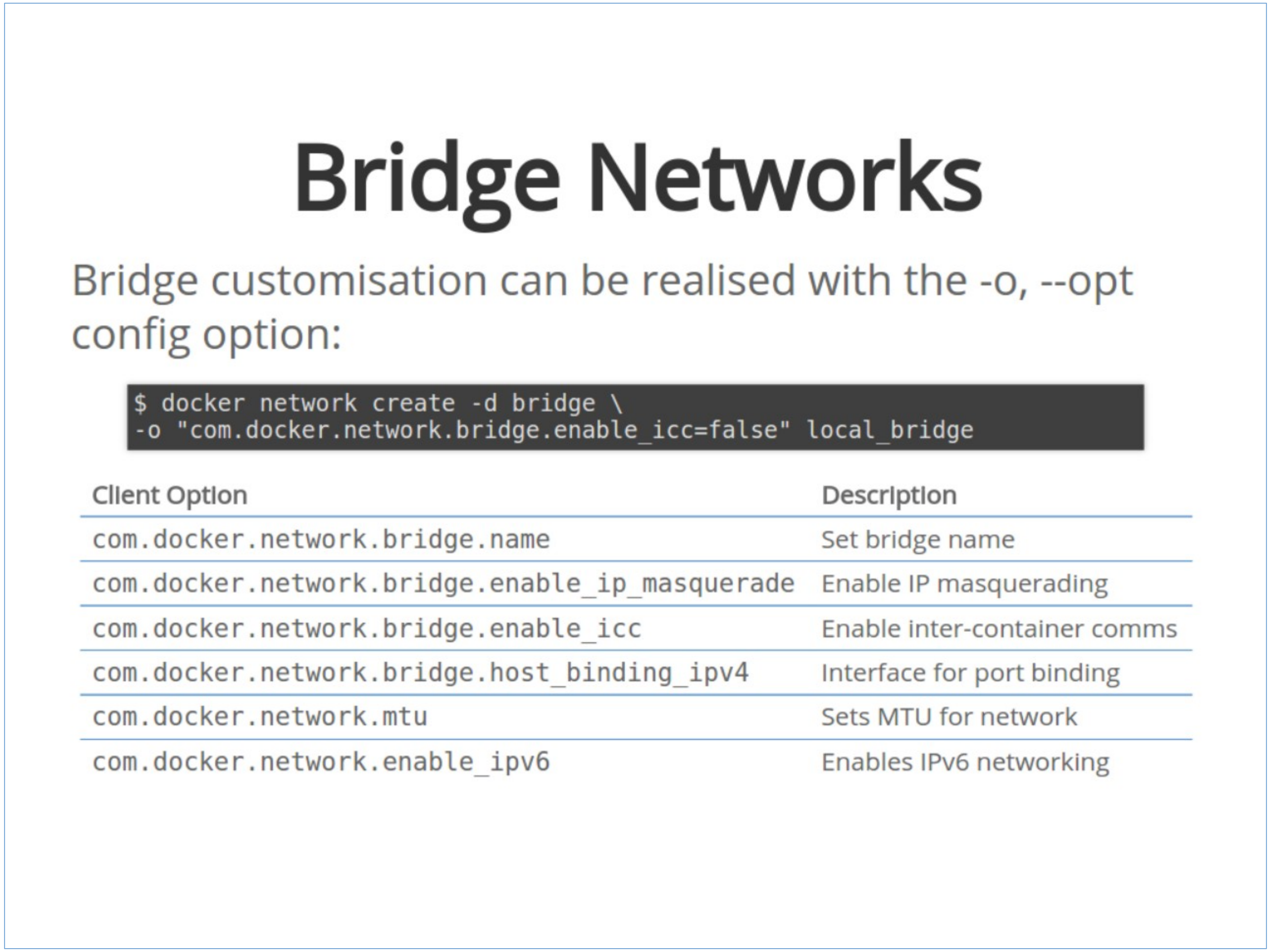
All containers on a bridge network are able to communicate with each other, and can make use of Docker's internal DNS capability to lookup an IP address by container hostname, name or alias

`--link=[]`: in the context of non-legacy bridge networks, specifies an alias associated with the provider container for the benefit of the consumer container

- When using the `--link` configuration option for non-legacy bridge networks, the consumer container acquires a private alias for the provider container, which is scoped exclusively to the network in question

* The default bridge can be disabled by using the daemon configuration option '`--bridge none`'

Notes



Bridge Networks

Bridge customisation can be realised with the -o, --opt config option:

```
$ docker network create -d bridge \
-o "com.docker.network.bridge.enable_icc=false" local_bridge
```

Client Option	Description
com.docker.network.bridge.name	Set bridge name
com.docker.network.bridge.enable_ip_masquerade	Enable IP masquerading
com.docker.network.bridge.enable_icc	Enable inter-container comms
com.docker.network.bridge.host_binding_ipv4	Interface for port binding
com.docker.network.mtu	Sets MTU for network
com.docker.network.enable_ipv6	Enables IPv6 networking

Bridge Networks

- Several options are available for customising user-defined bridge networks, effected using the `-o`, `--opt` configuration option for the `docker network create` sub-command
- By and large, each of these options has a corresponding Docker daemon configuration option that applies to the default, legacy bridge:

  com.docker.network.bridge.name – this allows for setting a custom bridge device name, instead of the auto-generated name of the form `br-XXXXXXXXXXXX`. There is no equivalent legacy configuration option.

  com.docker.network.bridge.enable_ip_masquerade – set true by default, this option (when set false) enables inhibiting IP masquerading of container IP addresses in the bridge. The equivalent legacy configuration option is `--ip-masq`.

  com.docker.network.bridge.enable_icc – set true by default, this option (when set false) enables inhibiting inter-container communication using netfilter rules. The equivalent legacy configuration option is `--icc`.

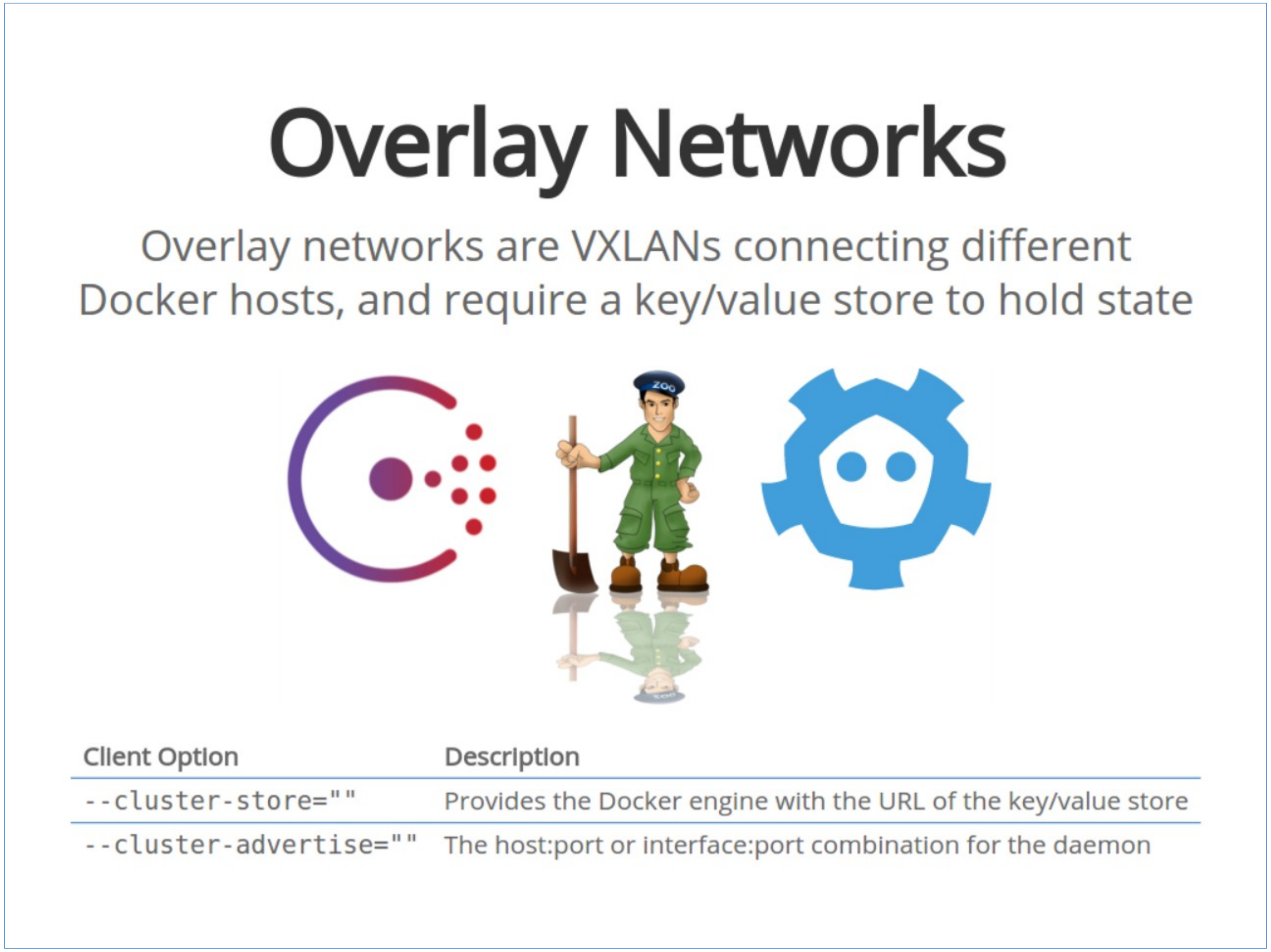
  com.docker.network.bridge.host_binding_ipv4 – set to `0.0.0.0` (all interfaces) by default, this option enables specification of which host interfaces, container ports are forwarded to. The equivalent legacy configuration option is `--ip`.

  com.docker.network.mtu – set to `0` by default, this option enables the setting of the maximum transmission unit (MTU) for the network. The equivalent legacy configuration option is `--mtu`.

  com.docker.network.enable_ipv6 – set false by default, this option (when set true) enables support for IPv6. The equivalent legacy configuration option is `--ipv6`.
- To disable inter-container communication on the network, use the following:

```
        -o “com.docker.network.bridge.enable_icc=false”
```

Notes



Overlay Networks

Overlay networks are VXLANs connecting different Docker hosts, and require a key/value store to hold state

Client Option	Description
`--cluster-store=""`	Provides the Docker engine with the URL of the key/value store
`--cluster-advertise=""`	The host:port or interface:port combination for the daemon

Overlay Networks

- A Docker overlay network is a VXLAN, which allows containers running on different Docker hosts to belong to, and communicate on, the same network
- Implementation of overlay networks in Docker, rely on a key/value store, which needs to be accessible from each Docker host that participates in the overlay network
- The key/value store retains information regarding the network state
- Docker uses a Golang library called `libkv`, which provides an abstraction of common key/value operations, and it provides support for the following key/value stores:

    - Hashicorp Consul – 'tool for service discovery, configuration and orchestration'
    - Apache Zookeeper – 'centralised service for maintaining configuration information'
    - CoreOS etcd – 'a distributed key value store'

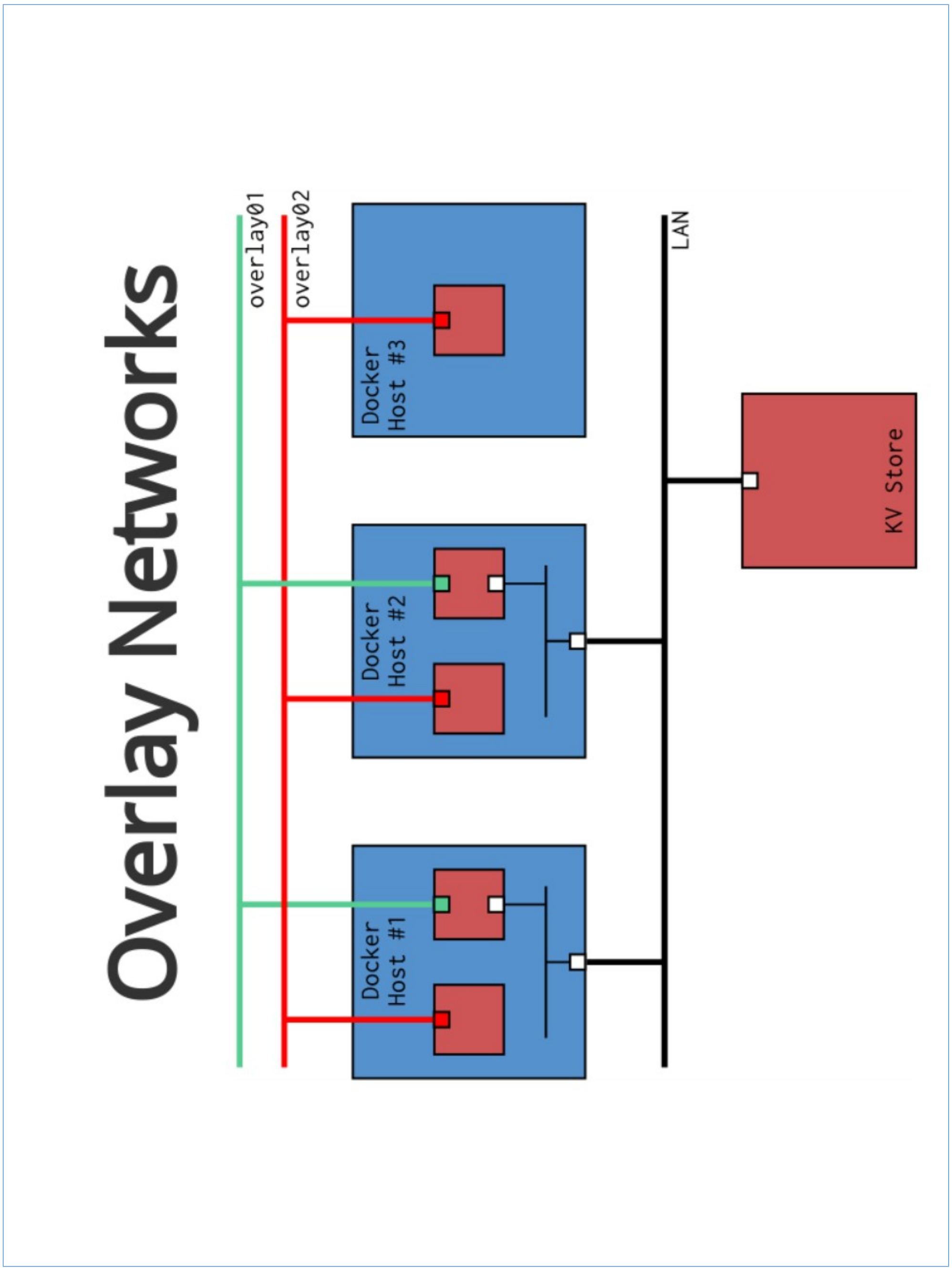
- The daemon on each Docker host must be started with the key/value store configuration options, `--cluster-store` and `--cluster-advertise`
- The configuration option uses the term 'cluster', because multiple Docker hosts are often configured in a cluster for workload purposes, and use key/value stores for maintaining cluster membership
- Strictly speaking, Docker overlay networking does not require the hosts to be members of a formal cluster, but does require registration with a distributed key/value store

`--cluster-store=””`: provides the Docker engine with the URL of the key/value store
`--cluster-advertise=””`: specifies the `host:port` combination, or `interface:port` the daemon can be found on

- To register a Docker host with a key/value store located at `172.17.0.2`, the daemon needs to be started with the following configuration option for each :

```
Consul:      --cluster-store “consul://172.17.0.2:8500”
Zookeeper:   --cluster-store “zk://172.17.0.2:2181”
etcd:        --cluster-store “etcd://172.17.0.2:2379”
```

Notes



Overlay Networks

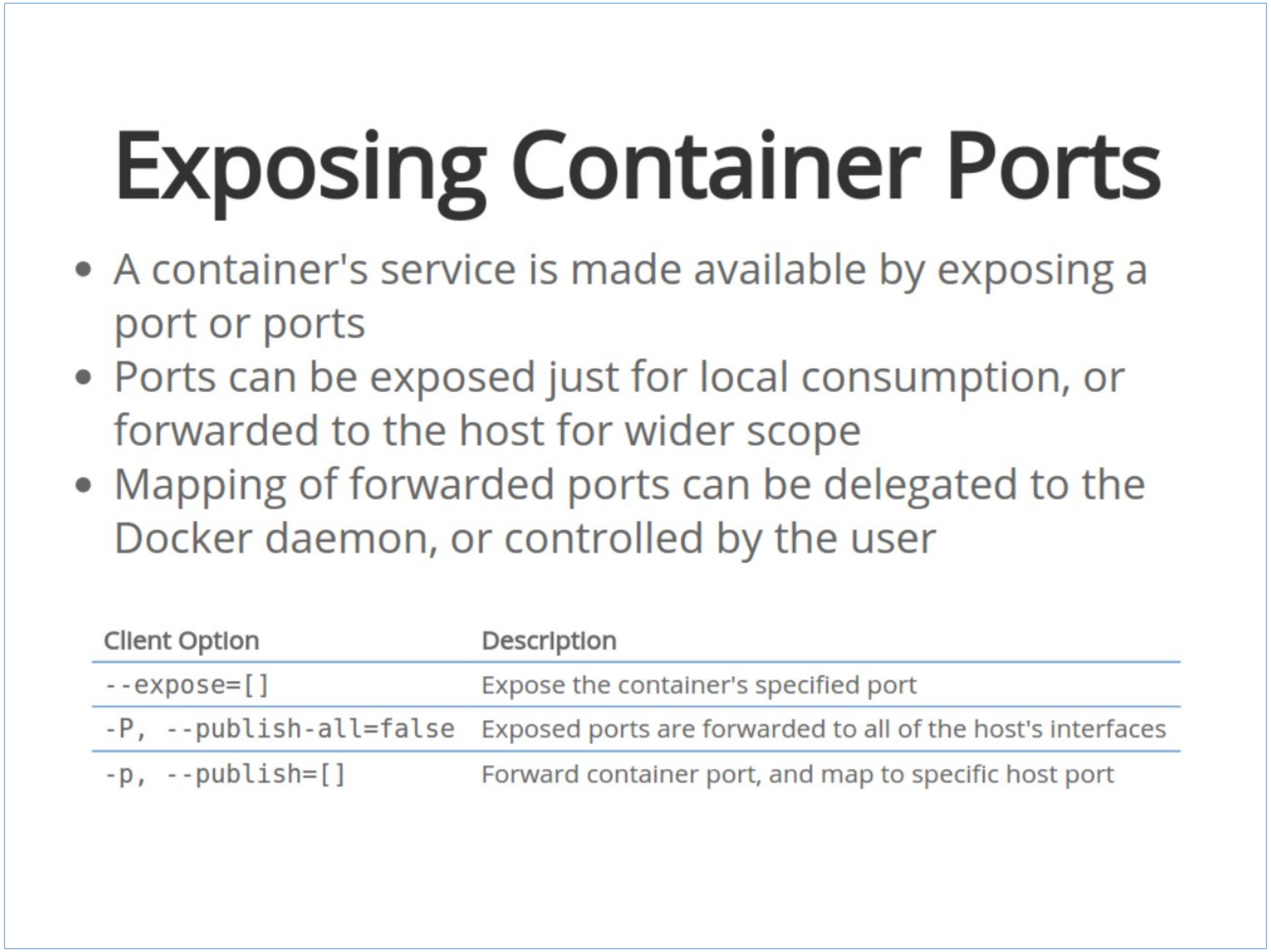
Overlay Networks

- Once a Docker host has registered with a key/value store, it is possible to use the `docker network create` sub-command to create an overlay network
- To create an overlay network called 'overaly_nw':

```
$ docker network create -d overlay overlay_nw
9544a4ce433dcebf69207bef47bc83f8c3303439dfb3adf6c1efc0d6132982aa
```

- After the creation of the overlay network, any Docker host that has registered with the key/value store will be able to 'see' the new network, and attach containers to it
- Containers running on a Docker host that are connected to the overlay network, can communicate freely with containers running on a different Docker host that are also connected to the same network
- Containers connected to an overlay network, may need to publish their services to consumers beyond the Docker host on which they are running
- In order to achieve this, containers need to forward the service's port to the host, which Docker facilitates by creating a specific bridge network called 'docker_gwbridge' on each Docker host
- The 'docker_gwbridge' network is created on a given Docker host, the very first time one of its containers connects to an overlay network
- The container in question has its primary network interface (`eth0`) connected to the overlay network, and a secondary network interface (`eth1`) connected to the 'docker_gwbridge' network
- Containers use the 'docker_gwbridge' bridge as a NAT router to publish their services beyond their Docker host
- In situations where containers running on an overlay network provide and consume each others services, and don't require external access beyond the network, use of the `--internal` configuration option when creating the overlay network, will not result in containers being added to the 'docker_gwbridge' network
- Containers are unable to communicate with each other via the 'docker_gwbridge' network

Notes



Exposing Container Ports

- A container's service is made available by exposing a port or ports
- Ports can be exposed just for local consumption, or forwarded to the host for wider scope
- Mapping of forwarded ports can be delegated to the Docker daemon, or controlled by the user

Client Option	Description
`--expose=[]`	Expose the container's specified port
`-P, --publish-all=false`	Exposed ports are forwarded to all of the host's interfaces
`-p, --publish=[]`	Forward container port, and map to specific host port

Exposing Container Ports

- In order to make practical use of a container's purpose, it will often be necessary to expose its services on a predetermined port(s)
- The port can be made exclusively available to the host and other containers, or it can be mapped or forwarded to the host, where it will be exposed to the rest of the network
- To achieve the former, the Docker client commands `docker create` and `docker run`, have a command line option called `--expose`, which takes a port number or range, which the container exposes to the host and other containers

`--expose=[]`: specifies a port or a port range, which the container exposes to the host and other containers (e.g. `--expose=80` or `--expose=5000-5005`)

- To achieve the latter, that is to expose and forward a port to the host, the Docker client commands `docker create` and `docker run` have two command line options, which provide for different behaviour

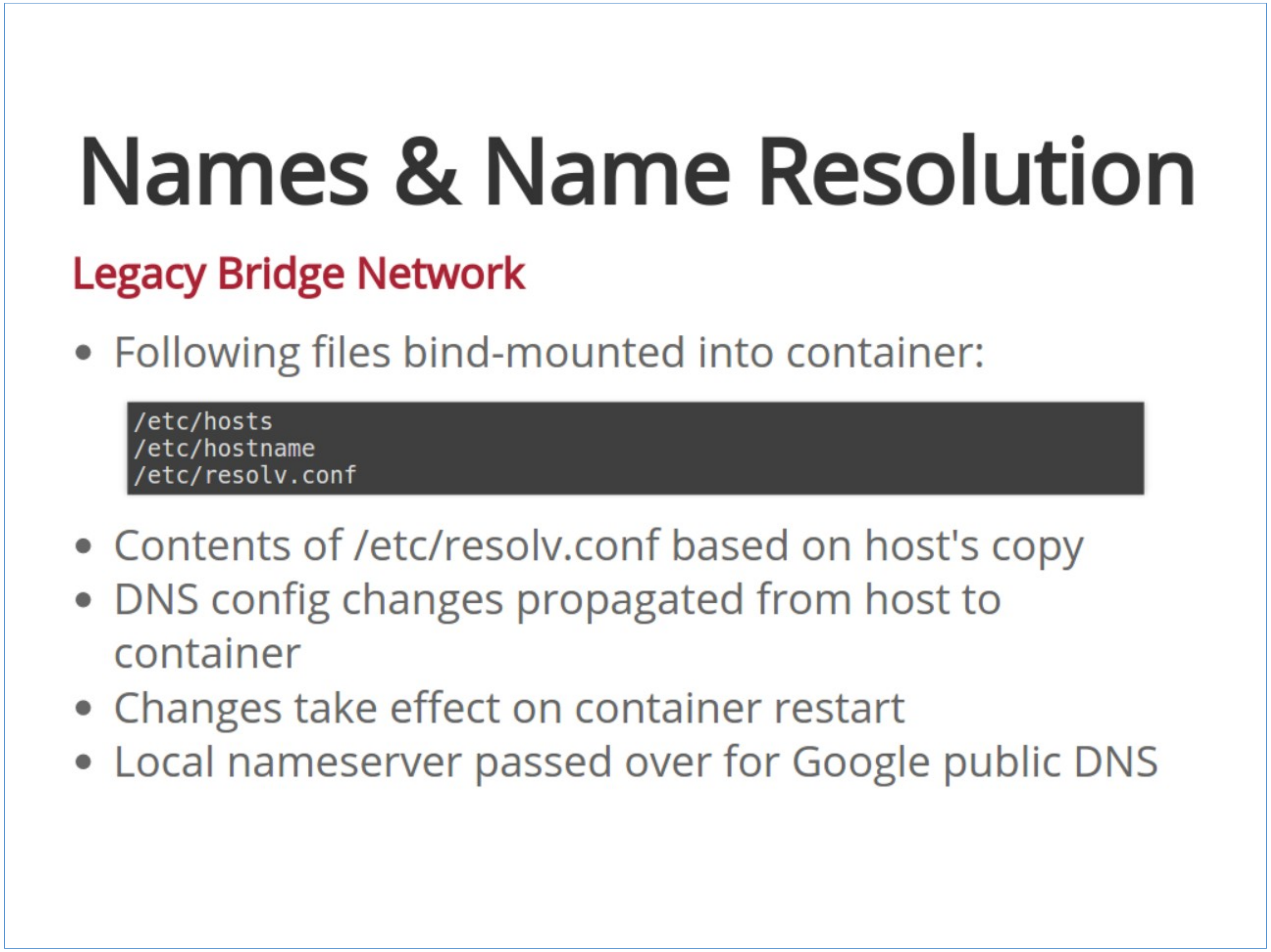
`-P, --publish-all=false`: specifies that any ports exposed by a container, are forwarded to all of the host's interfaces

- In order to make use of the `-P`, `--publish-all` command line option, it is necessary to define the ports to expose as part of an image build (we'll see how to do this later), or use in conjunction with `--expose`
- Any ports exposed and forwarded to the host in this way, are forwarded to host ports allocated by the daemon, which will use the range specified in the kernel parameter `/proc/sys/net/ipv4/ip_local_port_range`

`-p, --publish=[]`: specifies that a specific container port is published to a specific host port

- The `-p`, `--publish` command line option allows for more fine grained control of port forwarding, and takes the form `[ip:][hostPort:]containerPort`
- The default behaviour is to forward ports to ALL of the host's interfaces

Notes



Names & Name Resolution

Legacy Bridge Network

- Following files bind-mounted into container:

```
/etc/hosts
/etc/hostname
/etc/resolv.conf
```

- Contents of /etc/resolv.conf based on host's copy
- DNS config changes propagated from host to container
- Changes take effect on container restart
- Local nameserver passed over for Google public DNS

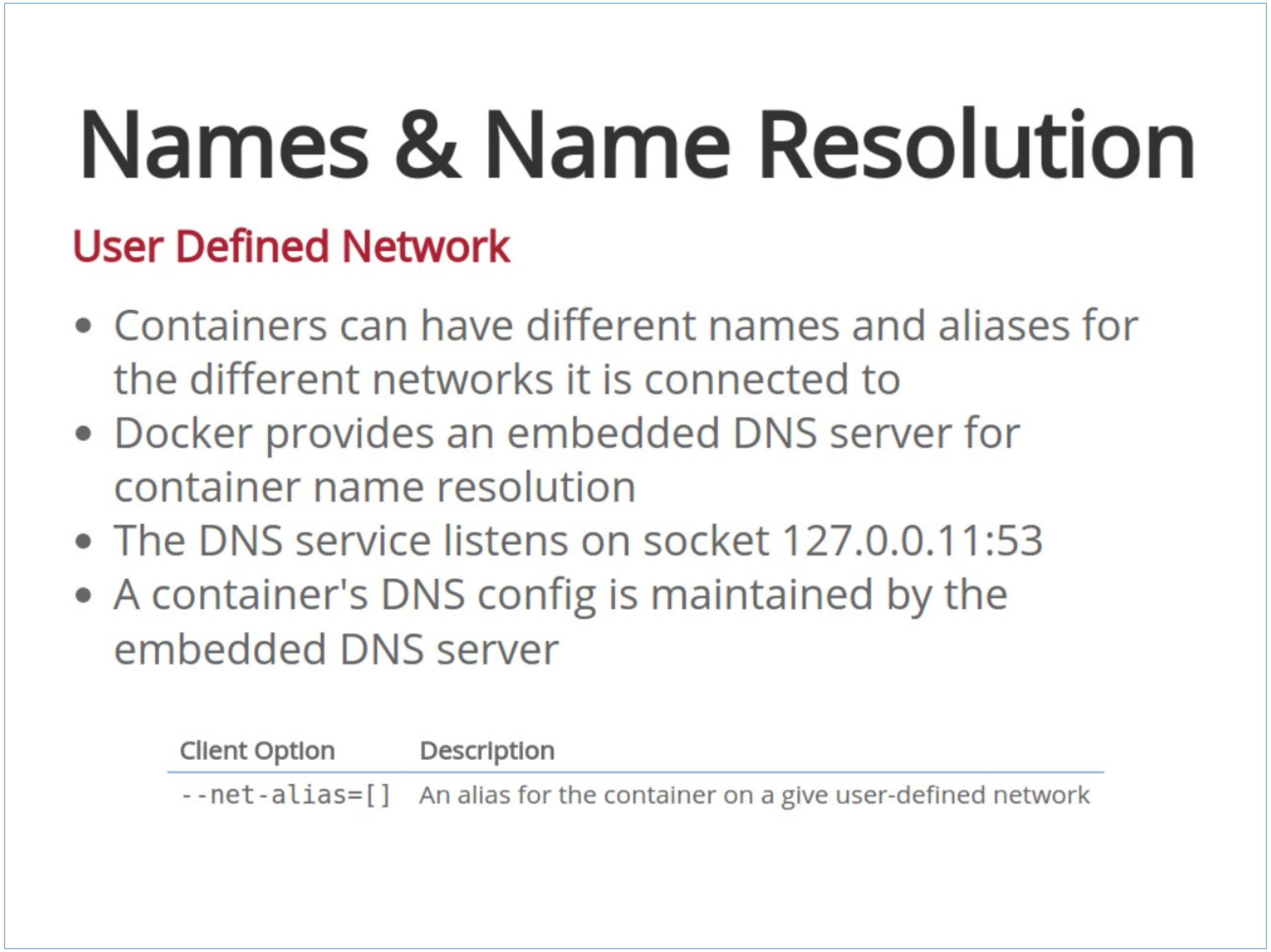
Names & Name Resolution

- How Docker handles hostnames, aliases and name resolution, depends on the container's context; is the context in relation to the default, legacy bridge network, or a user-defined network
- The difference in treatment is required in order to provide backwards compatibility for the legacy network, whilst taking advantage of the improvements in networking
- Irrespective of network type, Docker provides each container with a hostname, a shortened version of the 64-digit ID of the container, unless this is overridden with the Docker client `-h`, `--hostname` configuration option

  Default, legacy bridge network

- Each container gets its own private versions of `/etc/hosts`, `/etc/hostname` and `/etc/resolv.conf`
- These files are provided from outside of the container, and bind mounted into the container's filesystem, otherwise every container invoked from a particular image would have the same identity
- It's possible to place additional information in these files once the container has started, but this information will be transient, will be lost between container restarts, and is ignored when an image is built from an existing container
- Docker creates a container's `/etc/resolv.conf` based on the host's version of `/etc/resolv.conf`
- If, however, the host uses local name resolution (in the form of `dnsmasq`), then Docker needs to create a custom `/etc/resolv.conf` for each container, because the containers cannot access the host's local nameserver
- In such circumstances, a custom `/etc/resolv.conf` is used, and it is configured to use Google's public DNS servers (`8.8.8.8` and `8.8.4.4`)
- Provided a container's `/etc/resolv.conf` file remains identical to the host's copy, any changes to the host's copy will be reflected in a changed container copy
- When a container is eligible for an updated version of `/etc/resolv.conf`, it will receive the update when it is in a stopped state, and not whilst it is running

Notes



Names & Name Resolution

User Defined Network

- Containers can have different names and aliases for the different networks it is connected to
- Docker provides an embedded DNS server for container name resolution
- The DNS service listens on socket 127.0.0.11:53
- A container's DNS config is maintained by the embedded DNS server

Client Option	Description
`--net-alias=[]`	An alias for the container on a give user-defined network

Names & Name Resolution

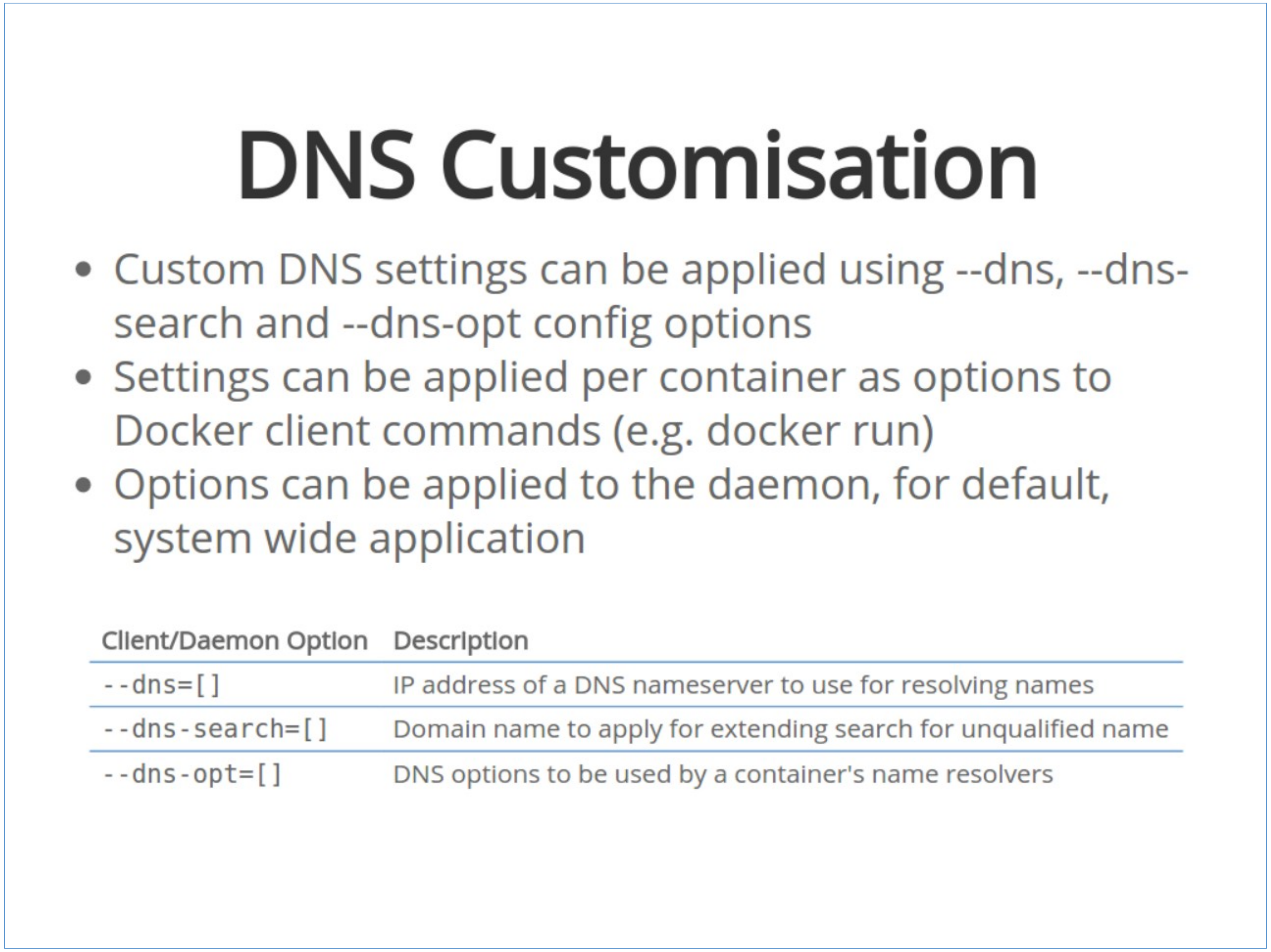
User-defined networks

- In user-defined networks, container names (for lookup purposes), can be set using a number of options
- We've already seen that a container is allocated a hostname, either implicitly or explicitly, and if it is also allocated a name by the user (with the `--name` configuration option), this name can also be used to lookup the container
- Additionally, further name aliases for the container, which are scoped to the network to which it is joining, can be defined at runtime, with the use of the `--net-alias` configuration option

`--net-alias=[]`: specifies a network specific alias for the container

- The aliasing capability provided by the `--net-alias` configuration option, is analogous to, and has the same effect as, the `--alias` configuration option for the `docker network connect` sub-command
- A container can have different name aliases in different networks
- For user-defined networks, Docker provides an embedded DNS server for the purpose of resolving container names
- On container creation, Docker creates an `/etc/resolv.conf` file which only references the embedded DNS server, which is found at `127.0.0.11:53`
- If the embedded DNS server is unable to resolve a name lookup, it will use nameservers specified in the host's `/etc/resolv.conf` file
- If the hosts uses localhost name resolution, then in the absence of any other references to external names, the request will be forwarded to Google's public DNS servers (`8.8.8.8` and `8.8.4.4`)
- The additional nameserver references are NOT reflected in the container's `/etc/resolv.conf` file, but are maintained as configuration items of the embedded DNS server

Notes



DNS Customisation

- Custom DNS settings can be applied using --dns, --dns-search and --dns-opt config options
- Settings can be applied per container as options to Docker client commands (e.g. docker run)
- Options can be applied to the daemon, for default, system wide application

Client/Daemon Option	Description
`--dns=[]`	IP address of a DNS nameserver to use for resolving names
`--dns-search=[]`	Domain name to apply for extending search for unqualified name
`--dns-opt=[]`	DNS options to be used by a container's name resolvers

DNS Customisation

- Customisation of the operation of DNS can be achieved with the configuration options `--dns`, `--dns-search` and `--dns-opt`
- The options can be applied to the daemon, giving a host-wide, default customisation for all containers, or to the client, which gives a container-specific customisation
- Client customisation takes precedence over daemon customisation
- The effect of applying the configuration has different effects, depending on whether it is applied to the default, legacy bridge network, or a user-defined network

`--dns=[]`: specifies an external IP address that references a nameserver to use for name lookup

- For the default, legacy bridge network, the 'nameserver' contents of the container's `/etc/resolv.conf` file will be altered to that supplied, whereas with user-defined networks, the embedded DNS server will use the nameserver supplied when it cannot resolve a lookup

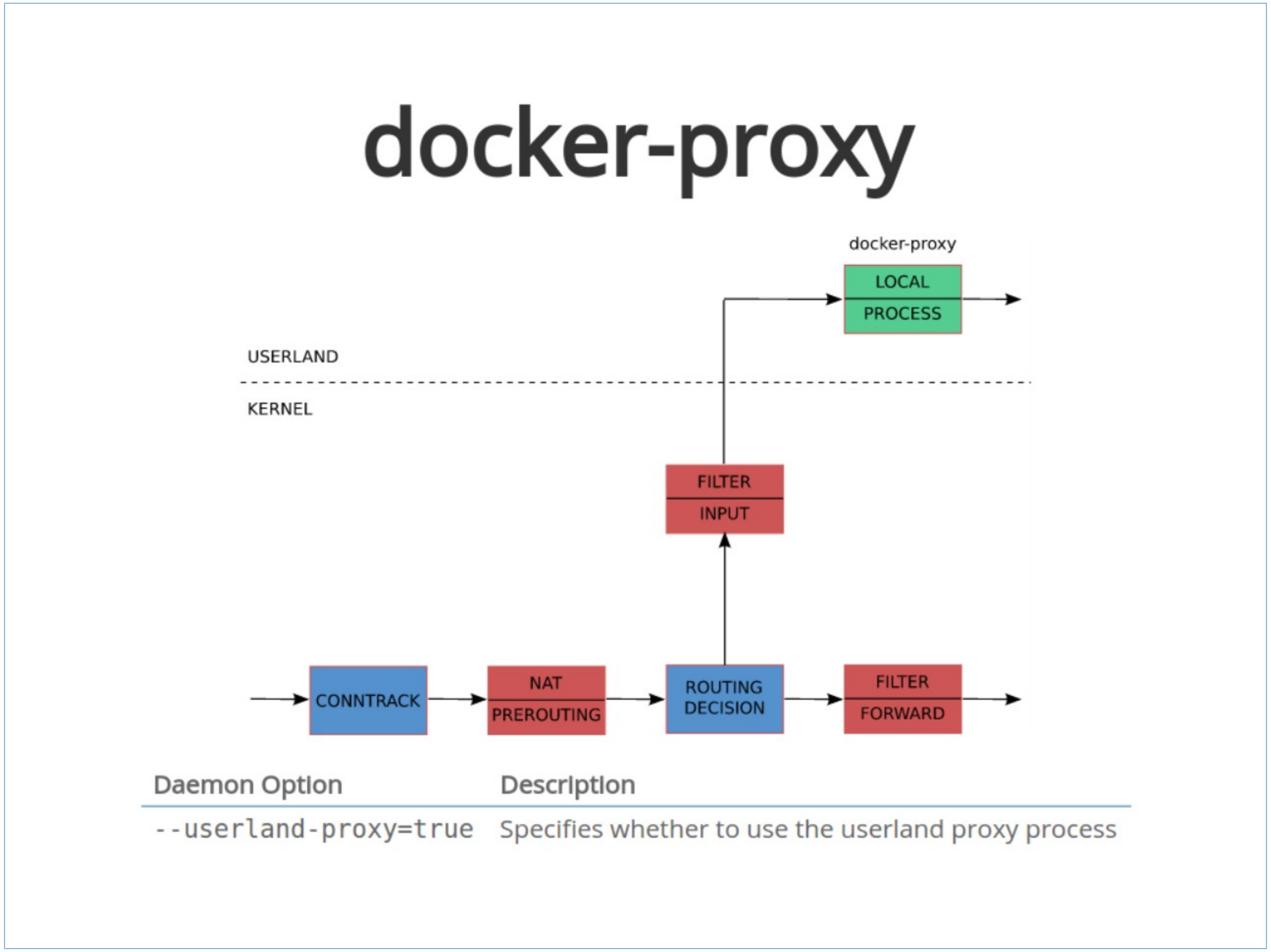
`--dns-search=[]`: specifies a domain name in which to search for an unqualified name (e.g. a lookup of name `skut`, will be extended to `skut.example.com`)

- For the default, legacy bridge network, the 'search' contents of the container's `/etc/resolv.conf` file will be altered to that supplied, whereas with user-defined networks, the embedded DNS server will extend the search to include the supplied domainname

`--dns-opt=[]`: specifies options to be used by name resolvers, and can be any valid option that can be used in an `/etc/resolv.conf` file

- For the default, legacy bridge network, the 'options' contents of the container's `/etc/resolv.conf` file will be altered to that supplied, whereas with user-defined networks, the embedded DNS server will apply the options when resolving

Notes



docker-proxy

Daemon Option	Description
`--userland-proxy=true`	Specifies whether to use the userland proxy process

docker-proxy

- Whilst Docker relies on netfilter rules to route traffic to containers on bridge networks, it also provides an alternative, legacy mechanism in the form of a 'userland' process called `docker-proxy`
- In the early stages of Docker's development, some kernels did not support routing of traffic from the host's loopback network, and the `docker-proxy` process facilitates this communication outside of the Linux kernel
- Whenever a container's port is forwarded to the host, a `docker-proxy` process is started and can be viewed using a ps process listing, e.g.

```
docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 80 \
-container-ip 172.17.0.8 -container-port 80
```

- It is considered to be inefficient, a significant compromise, and whilst it currently remains the default behaviour, it can be disabled with the `--userland-proxy=false` configuration option setting
- With appropriate use of the `--userland-proxy` configuration option, Docker sets the kernel parameter `net.ipv4.route_localnet` to `1`, thereby allowing the routing of loopback traffic

`--userland-proxy=true`: specifies whether the 'userland' `docker-proxy` process is used when forwarding container ports to the host

Notes