Notas sobre a infraestrutura e utilização de contentores *Docker em ambiente Linux*

draft 0.1

ISEL

dezembro de 2022

Conteúdo

Contentores versus máquinas virtuais

Contentores são uma forma de virtualização *lightweight* (de sistema operativo) que permite criar, sobre o mesmo *kernel*, contextos de execução virtualizados em vários aspetos, nomeadamente ao nível do *file system* e das interfaces de rede disponíveis. A figura 1 apresenta o modelo de contentores em comparação com o modelo de máquinas virtuais de sistema.

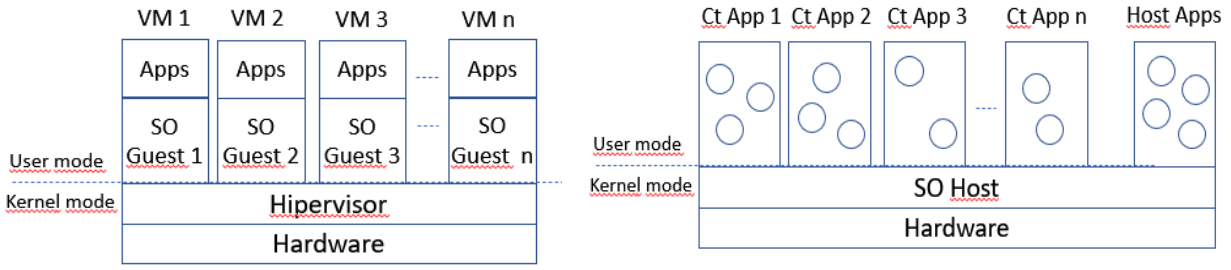

Figura 1 – Máquinas virtuais de sistema versus contentores

No Linux o suporte à construção de contentores é realizado através de um conjunto de serviços de *kernel* que permitem, nomeadamente, a criação de diferentes espaços de nomes (*namespaces*) para vários recursos, delimitando assim os recursos acessíveis aos processos, e a definição de quotas de acesso (*cgroups*) a determinados recursos físicos.

Tipo de espaço de nomes a virtualizar	**Vista alterada**	**Exemplos de utilização**
PID	Identificadores de processos	`unshare --fork --pid --mount-proc /bin/bash`
Mount	Sistema de ficheiros raiz	
UTS	hostname	`unshare -u /bin/bash`
Network	Tabelas de routing e interfaces de rede	
IPC	Nomes de message queues	
User	Credenciais de utilizadores	

Tabela 1 – tipos de *namespaces* disponíveis no Linux

Imagens e contentores

Introdução

A grande motivação para a virtualização via contentores está na facilidade de distribuição e instalação de *software* em contexto de execução isolada que estes permitem, de uma forma muito mais eficiente e escalável comparativamente à utilização de máquinas virtuais.

Tirando partido dos mecanismos referidos na secção anterior, o contentor oferece ao processo (ou processos) que sobre ele executam muitos mais do que o isolamento de memória e CPU que o SO oferece de raiz aos seus processos. Os contentores oferecem ambientes isolados, com todos os recursos (*file system*, redes, etc.) necessários à execução de uma dada aplicação ou (micro-)serviço.

Obtenção e produção de imagens

Imagens são *templates* a partir das quais se instanciam contentores. Incluem tipicamente uma hierarquia de pastas que se irá constituir como o sistema de ficheiros raiz (*root*) dos contentores instanciados, e metadados que descrevem a imagem e aspetos do contexto de execução (ex: pasta *home*, *user* associado, portos a expor ao *host*).

As imagens *docker* podem ser obtidas de servidores de repositórios denominados `Registries`, de que o Dockerhub é exemplo. É também possível construir uma imagem a partir do estado de um contentor *stopped* (a ver mais tarde), via o comando `docker container commit`.

A forma mais versátil de produção de imagens é, contudo, através da utilização de linguagem DSL expressa em ficheiros `Dockerfile`.

Vejamos um exemplo de DockerFile:

```
FROM alpine
LABEL purpose learning
CMD echo "built on `cat build_date.txt` "
RUN date > build_date.txt
```

A execução do comando

```
$ docker image build -t first .
```

produz uma nova imagem de nome `first` e gera o seguinte *output*:

```
Step 1/4 : FROM alpine
 ---> 49176f190c7e
Step 2/4 : LABEL purpose learning
 ---> Running in 01cf01775dc4
Removing intermediate container 01cf01775dc4
 ---> 01a8030af505
Step 3/4 : CMD echo "built on `cat build_date.txt` "
 ---> Running in 2c442c968008
Removing intermediate container 2c442c968008
 ---> 6f1a628374a1
Step 4/4 : RUN date > build_date.txt
 ---> Running in a9b755c5744a
Removing intermediate container a9b755c5744a
 ---> 92a628aec648
Successfully built 92a628aec648
Successfully tagged first:latest
```

Observa-se que cada passo da construção da imagem está associado a cada instrução presente no ficheiro Dockerfile.

- Todas as imagens são produzidas a partir de uma imagem base, especificada pela instrução FROM que aparece muitas vezes em primeiro lugar nos ficheiros `Dockerfile`. Neste caso é a imagem `alpine:latest` previamente obtida do `Dockerhub`.
- A instrução LABEL acrescenta *metadata* genérica à imagem na forma de pares chave/valor: o primeiro argumento é a chave, todos os outros correspondem ao valor.
- A instrução RUN modifica o *file system* da imagem com o resultado da execução do comando especificado.
- A instrução CMD representa *metadata* que define o comando a executar em cada instância (contentor) criada a partir da imagem.

O comando `docker image ls` lista as imagens existentes no repositório local. Tendo apenas as duas imagens referidas seria produzido o *output*:

```
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
first        latest    d8c68f2eb940   5 seconds ago   7.05MB
alpine       latest    49176f190c7e   4 weeks ago     7.05MB
```

Podemos finalmente criar e iniciar a execução de uma instância (contentor) da imagem `first` executando o comando `docker container run first`. Da sua execução seria produzido o *output*:

```
built on Wed Dec 21 14:08:01 UTC 2022
```

resultante da execução do comando especificado na instrução CMD.

O comando `docker system df` mostra o espaço ocupado pelo conjunto de imagens existentes. Neste caso o *output* (excerto) seria:

```
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          2         0         7.05MB    7.05MB (100%)
```

Observa-se que cada imagem ocupa 7,05M, o mesmo que o total de ambas as imagens. Este resultado é possível devido a um dos aspetos mais interessantes dos contentores `docker`: a partilha do conteúdo entre imagens, usando um modelo de sistemas de ficheiros organizado em camadas (neste caso o *file system* do tipo `overlay2`). Este *file system* permite montar numa pasta (`merged`), uma visão integrada de uma ou mais camadas *read/only* (`lower`) e de uma última camada *read/write* (`upper`). Neste caso a base (`alpine:latest`) é a camada *read/only* e a camada associada à imagem `first` é a camada *read/write*. São essas camadas *readonly* que podem ser partilhadas entre imagens, como veremos à frente.

O comando `docker image inspect first` apresenta uma descrição da imagem, de que vemos um excerto a seguir:

```
[{
    "Id": "sha256:d8c68f2eb940a6e637148e7d28960c2241403e9a3fff3c1d7f7327a01af4c11f",
    ….
    "Parent": "sha256:5fc9fdf1956c20d7e15fc836ee50d6042c77e981df6c1a75e7aabc0ac96d52fe",
    ….
}]
```

A propriedade `Parent` remete para a imagem a partir da qual `first` foi produzida. Se observarem atentamente a listagem anterior, o *hash* 5fc… não corresponde ao *hash* do `Parent` expectável (`alpine:latest`). Isto porque esse `Id` é de uma imagem intermédia produzida no *build* da imagem `first`.

Se inspecionarmos essa imagem temos o excerto:

```
[{
   "Id": "sha256:5fc9fdf1956c20d7e15fc836ee50d6042c77e981df6c1a75e7aabc0ac96d52fe",
   ….
   "Parent": "sha256:8339a9a26ddd9a4000d3aea9e0dacc82391858460e154553778ef2acc3562c8d",
   ….
}]
```

E finalmente, inspecionando essa imagem `Parent`, temos o excerto:

```
[ {
   "Id": "sha256:8339a9a26ddd9a4000d3aea9e0dacc82391858460e154553778ef2acc3562c8d",
    …
    "Parent": "sha256:49176f190c7e9cdb51ac85ab6c6d5e4512352218190cd69b08e6fd803ffbf3da",
    …
}]
```

Onde vemos que a imagem `Parent` corresponde à imagem base `alpine:latest`. As outras imagens na cadeia são imagens intermédias que resultam do processo de *build*. A regra geral é a de que todas as instruções de um Dockerfile estão associadas a uma imagem, seja inicial, intermédia ou final. Neste caso as instruções LABEL e CMD estão associadas às imagens intermédias observadas. Algumas instruções (RUN, COPY, ADD e WORKDIR, a ver mais tarde), para além de produzirem imagens (intermédias ou finais) estão também associadas à criação de camadas (*layers*) do *file system* `root` da imagem final produzida.

Esta ideia da criação de uma imagem como uma cadeia de imagens intermédias (com *layers* associados) é uma caraterística muito importante do sistema Docker, tendo um conjunto de vantagens associadas. Para percebermos isso acrescentemos uma instrução ao nosso Dockerfile exemplo:

```
FROM alpine
LABEL purpose learning
CMD echo "built on `cat build_date.txt` "
RUN date > build_date.txt
COPY README ./
```

A instrução COPY copia o ficheiro README presente na pasta corrente do *host* para a pasta corrente (por omissão /) da imagem em produção. Realizando novamente o *build* da imagem `first` obtemos o seguinte *output:*

```
Step 1/5 : FROM alpine
 ---> 49176f190c7e
Step 2/5 : LABEL purpose learning
 ---> Using cache
 ---> 8339a9a26ddd
Step 3/5 : CMD echo "built on `cat build_date.txt` "
 ---> Using cache
 ---> 5fc9fdf1956c
Step 4/5 : RUN date > build_date.txt
 ---> Using cache
 ---> d8c68f2eb940
Step 5/5 : COPY README ./
 ---> 9a3f5f642f28
Successfully built 9a3f5f642f28
Successfully tagged first:latest
```

E criando e iniciando um novo contentor temos o *output*:

```
built on Wed Dec 21 14:08:01 UTC 2022
```

Se revirmos o *output* inicial notamos que é exatamente igual, o que pode ser inesperado considerando o comando associado à instrução RUN no Dockerfile anterior.

Ora isso acontece por boas razões. Se observarmos o *output* produzido pelo *build* verificamos que em todos os passos, com exceção do primeiro e do último, aparece a frase "`Using cache`".

Isto quer dizer que foram reaproveitadas todas as imagens (naturalmente a inicial, e todas as intermédias), sendo criada outra imagem, associada ao comando cópia. Este reaproveitamento das imagens anteriores (em cache) permite otimizar os tempos de *build*.

A possibilidade de reaproveitamento da cache depende das instruções. As instruções COPY (e ADD) produzem um *hash* do conteúdo de cada ficheiro copiado e do texto da própria instrução, que fica associado à camada em cache. Esse *hash* é recalculado em cada novo *build*. Se continuar idêntico ao armazenado, a camada em cache é aproveitada.

Para todas as outras instruções (RUN incluída), a imagem intermédia (e a respetiva camada em cache, se existir), só são reconstruídas se o *hash* do texto da instrução for diferente do *hash* anterior.

Como não foi o caso no segundo *build*, o ficheiro `build_date.txt` não foi reconstruído, aparecendo com o conteúdo original. A ideia, é claro, é otimizar o reaproveitamento dos conteúdos em cache, mas não permite à instrução RUN reavaliar comandos não idempotentes (como é o caso).

De qualquer modo é possível forçar essa reavaliação através da instrução ARG, que permite especificar argumentos a definir na execução do comando `docker build`. A seguir apresenta-se uma possível solução:

```
FROM alpine
ARG STAMP=1
LABEL purpose learning
CMD echo "built on `cat build_date.txt` "
RUN echo $STAMP > /dev/null && \
    date > build_date.txt
COPY README ./
```

Usando o argumento STAMP através do commando:

```
$ docker build --build-arg STAMP=`date +%s` -t first .
```

Resumindo, a construção por camadas dos *file system* `root` associados às imagens permite:

- Partilhar camadas entre imagens, minimizando o espaço em disco.
- Otimizar tempos de *build*, reaproveitando imagens intermédias e camadas em cache de *builds* anteriores.
- Otimizar o carregamento para o repositório local de imagens carregadas de Registry remotos, paralelizando a transferência de cada camada.

O repositório local nas versões recentes do Docker no Linux está organizado em pastas a partir de `/var/lib/docker`. A fig. seguinte apresenta a organização das diferentes entidades envolvidas: a cache com os conteúdos (pastas) de cada camada (**Data Cache**), os metadados dos *layers* associados (**Layers**) e das imagens, incluindo as imagens intermédias (**Images**), e as instâncias criadas (**Containers**).

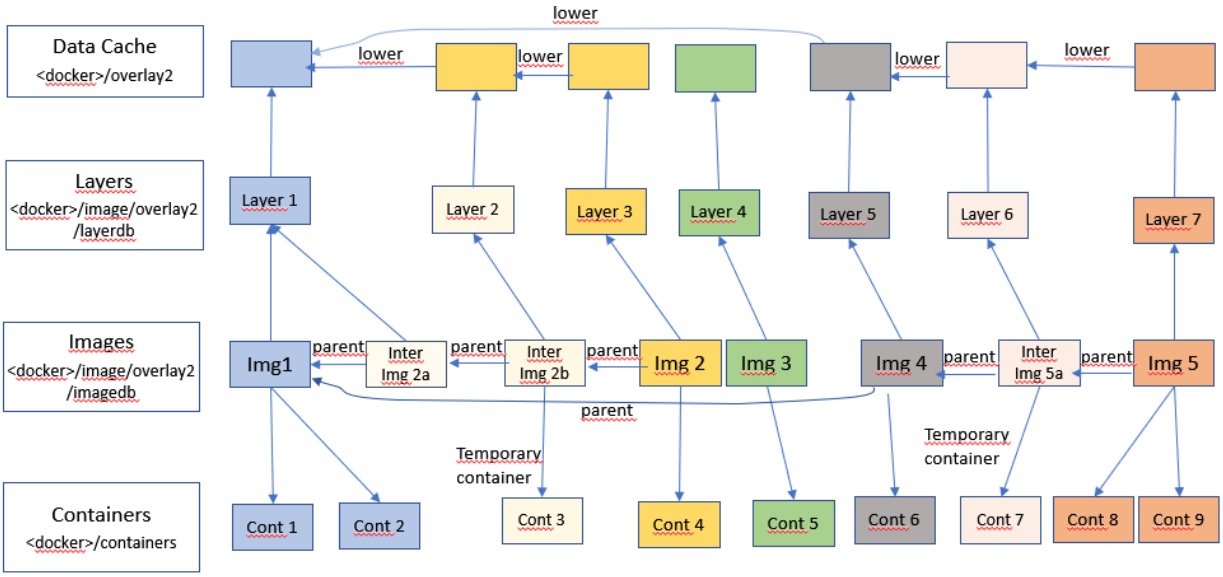

Figura 2 – organização da cache local de imagens e respetivos metadados

Gestão do ciclo de vida dos contentores

Os contentores têm o ciclo de vida apresentado na figura abaixo. Os nomes das transições de estado correspondem aos nomes dos subcomandos do comando `docker container` que as provocam.

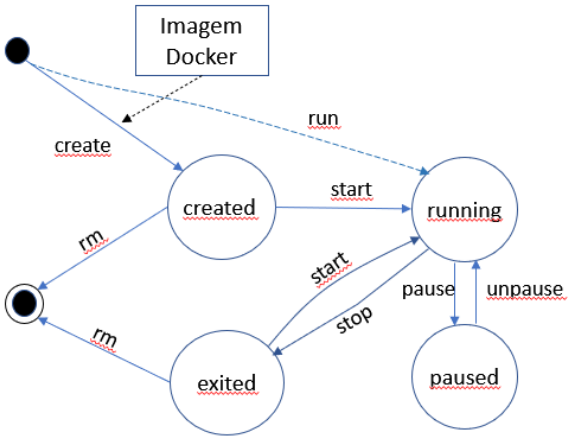

Figura 3 – Ciclo de vida dos contentores

Por exemplo, o comando

```
$ docker container create -t –name alpine1 alpine
2b106675a8f16be40ea6ef4e1a6a811d9e9ebd1fc358b0478b6077dd9cbf53c2
```

cria um novo contentor, instância da imagem `alpine:lastest` (carregada do Dockerhub caso ainda não exista na cache local), com o nome `alpine1` e o `id` (UUID) apresentado pelo comando.

O contentor pode ser colocado no estado *running* através do comando

```
$ docker container start -i 2b1
```

O último parâmetro poderia também ser o nome do container, neste caso `alpine1`.

Como sempre, quando não exista ambiguidade, a entidade a que os comandos se aplicam pode ser omitida (ficando `docker create`, `docker start`).

Para parar o contentor (colocando-o no estado `exited`) é usado o comando

```
$ docker container stop alpine1 ( ou o seu id)
```

É importante compreender que o estado `exited` é similar ao estado `created` no sentido que representa um contentor que não está em execução (não tem processos, apenas tem um *file system* e metadados associados). O contentor pode voltar ao estado `running` voltando a executar o comando start.

Como se vê no diagrama de estado da fig. 3 é também possível suspender (e retomar) a execução de um contentor usando os comandos (`pause` e `unpause`, respetivamente).

Finalmente, um contentor pode ser removido, desaparecendo do sistema:

```
$ docker container rm alpine1
```

Como é natural, tal como as imagens e outras entidades que iremos analisar à frente (*networks* e volumes), os contentores podem ser listados:

```
$ docker container ls -a
```

lista todos os contentores existentes, independentemente do seu estado (sem a opção `-a` apenas são listados os contentores em execução), e o comando:

```
$ docker container inspect ab2
```

Apresenta os metadados do contentor cujo `id` começa por `ab2`.

Dois comandos importantes para *debug* de contentores são o comando `exec`

```
$ docker container exec -it ab2 sh
```

Neste caso é criado um processo a correr o comando `sh` de forma interativa no ambiente de execução do contentor indicado, que terá de estar no estado `running`. Note-se que qualquer comando pode ser indicado para execução no comando `exec`, naturalmente desde que exista no *file system* do contentor.

O outro comando importante é o comando `logs`:

```
$ docker container logs ab2
```

Que mostra os *logs* (*output* enviado para o *standard error*) produzidos pelos processos do contentor indicado.

De forma interessante, o modelo de contentores e imagens é recursivo, no sentido em que os contentores são instanciados a partir de imagens, mas também são usados contentores na produção de imagens, de duas formas: novas imagens podem ser produzidas a partir de um contentor *stopped* usando o comando `docker container commit` e imagens são produzidas a partir do processamento de ficheiros de instruções (`Dockerfile`), que internamente cria contentores que produzem o estado do *file system* raiz das imagens intermédias e final.

Arquitetura

Como mostra a figura 4, o ecossistema Docker segue uma arquitetura *client/server* onde um servidor (`dockerd`) gere imagens (`Images`), eventualmente comunicando com um repositório remoto de imagens (`Registry`) para as obter, armazenando-as numa cache local, e as instâncias executáveis dessas imagens (`Containers`). O servidor expõe uma API Rest usada pela aplicação cliente (`docker - Docker Client`).

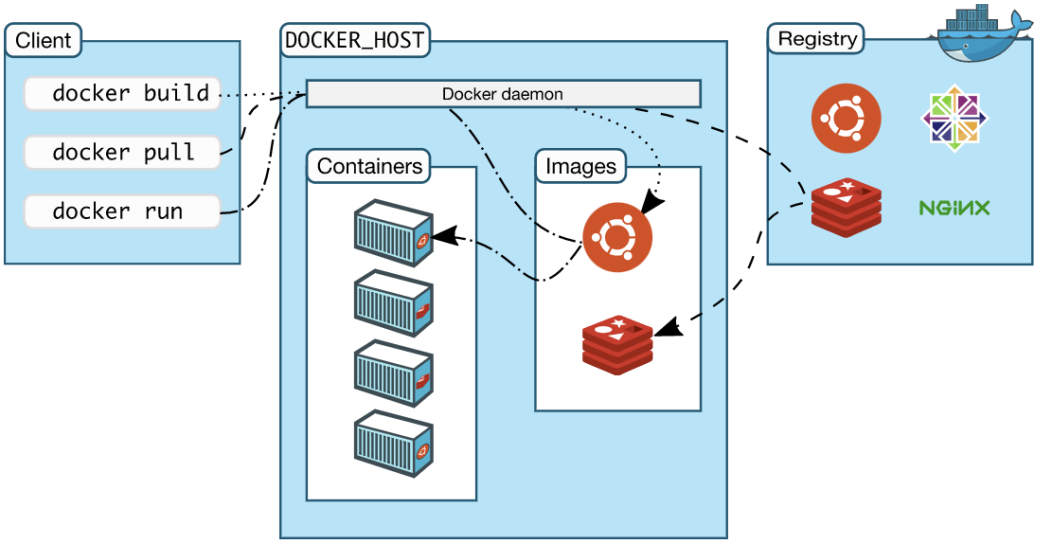

Figura 4 Arquitetura simplificada da infraestrutura Docker (obtida de https://docs.docker.com/get-started/overview/)

A arquitetura foi evoluindo para se tornar mais modular e suportar interoperabilidade com outros modelos de gestão de contentores que foram surgindo (ex: `kubernettes`). Essa modularização foi definida em função dos esforços de estandardização paralelos realizados pelo consórcio OCI (Open Container Initiative). Por exemplo a execução de contentores passou a ser processada por um *container runtime* independente (`runc`) e a gestão de imagens pelo componente `containerd`

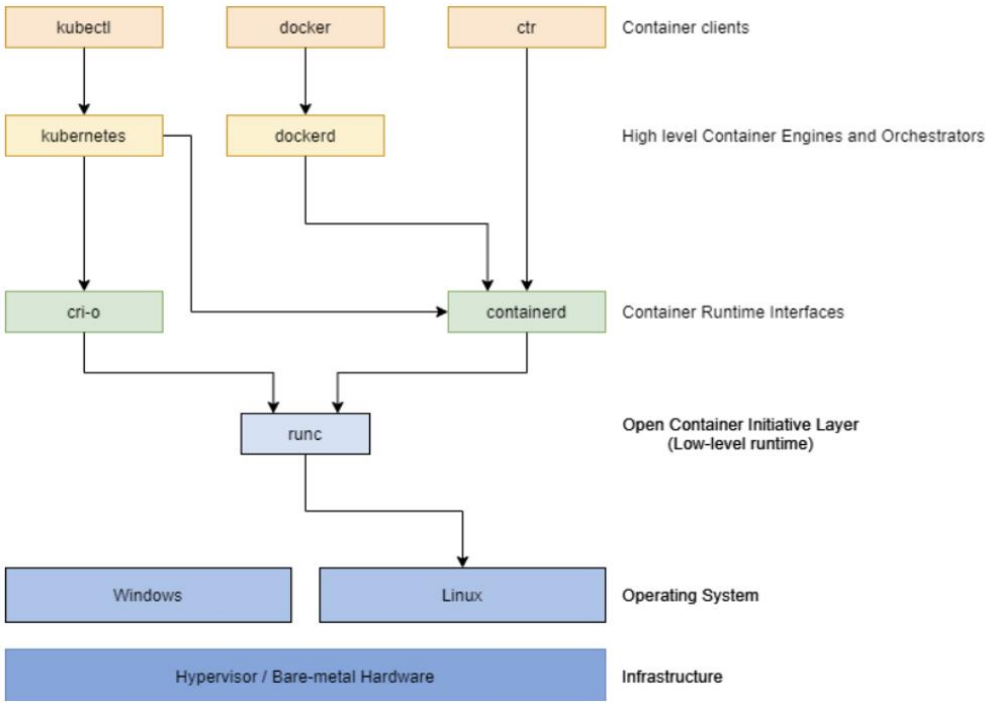

Figura 5 Componentes locais da arquitetura Docker no Linux e interoperabilidade retirado de https://earthly.dev/blog/containerd-vs-docker/

Networking

Um dos aspetos mais importantes na configuração e execução de contentores é a sua inserção na *intranet* virtual onde se executam os contentores locais.

A gestão das redes Docker é feita usando a entidade `network` da API, sobre a qual se podem aplicar as operações habituais (`create`, `rm`, `ls`, `inspect`, `prune`), bem como outras específicas (ex: `connect`) que permite ligar um contentor a uma rede de forma dinâmica.

Existem 3 redes pré-definidas, `bridge`, `host` e `none`, instâncias de *drivers* com o mesmo nome. A rede `host` representa exatamente o que o nome sugere, os contentores usam a rede do *host*, não existindo virtualização. Os contentores associados ao driver `none` apenas têm interface de rede de *loopback*, o que é útil quando se pretende proteger o contentor de acessos externos. A rede `bridge` constitui-se como a rede virtual onde os contentores coexistem por omissão. Cada contentor tem a sua interface de *loopback* e pode aceder aos outros contentores conhecendo o seu endereço IP. Por outro lado, as redes tipo `bridge` permitem que os contentores possam publicar portos de serviços associados que podem ser acedidos no *host* através de *port forwarding*.

É possível associar um contentor a uma rede específica através da opção `--network` do subcomando `run`. Por exemplo, a execução do comando:

```
$ docker container run -network none -t -it alpine
```

Cria e coloca em execução um contentor associado à imagem `alpine:latest` com um *shell* interativo como processo inicial. Nesse *shell* podemos ver que o contentor apenas tem a interface de *loopback*

```
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

É possível criar outras redes usando o comando `create` da entidade `network`

```
$ docker network create network mynet
```

Com o comando `docker network ls` listam-se as redes existentes:

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
78e249503c2f   bridge    bridge    local
a6ff9a2b135a   host      host      local
33a2993508a7   mynet     bridge    local
8ea94301e2d6   none      null      local
```

Vemos que `mynet` é uma nova rede do tipo `bridge`, com comportamento similar à rede `bridge` pré-definida, mas com uma diferença crucial. Os contentores associados à rede `mynet` têm acesso a um servidor `DNS` que permite obter o endereço `IP` de qualquer *host* na mesma rede a partir do seu *hostname,* do nome do contentor ou de um *alias*, sendo que todos estes nomes podem ser configurados na criação do contentor.

Consideremos a criação dos contentores em duas consolas distintas:

```
docker container run --hostname host_alpine1 --name alpine1 -it --network mynet  alpine
```

```
docker container run --hostname host_alpine2 --name alpine2 -it --network mynet  \
                        -p 8080:80 --network-alias alpine_service alpine
```

Observamos que o porto 80 no contentor alpine2 é publicado no porto 8080 do host.
Em qualquer das consolas podemos usar o servidor de DNS integrado para obter o IP de qualquer *host* da rede a que o contentor pertence, dado o seu nome (`--name`) , *hostname* (`--hostname`) ou o *alias* produzido pela opção `--network_alias`.

```
/ # nslookup alpine1 127.0.0.11
---
Name:   alpine1
Address: 172.18.0.2


/ # nslookup host_alpine2 127.0.0.11
---
Name:   host_alpine2
Address: 172.18.0.3

/ # nslookup alpine_service 127.0.0.11
---
Name:   alpine_service
Address: 172.18.0.3
```

A fig. seguinte apresenta a relação entre duas redes bridge (a rede `default` e `mynet`) e a forma como o *host* pode aceder a portas de cada contentor via *port forwarding*

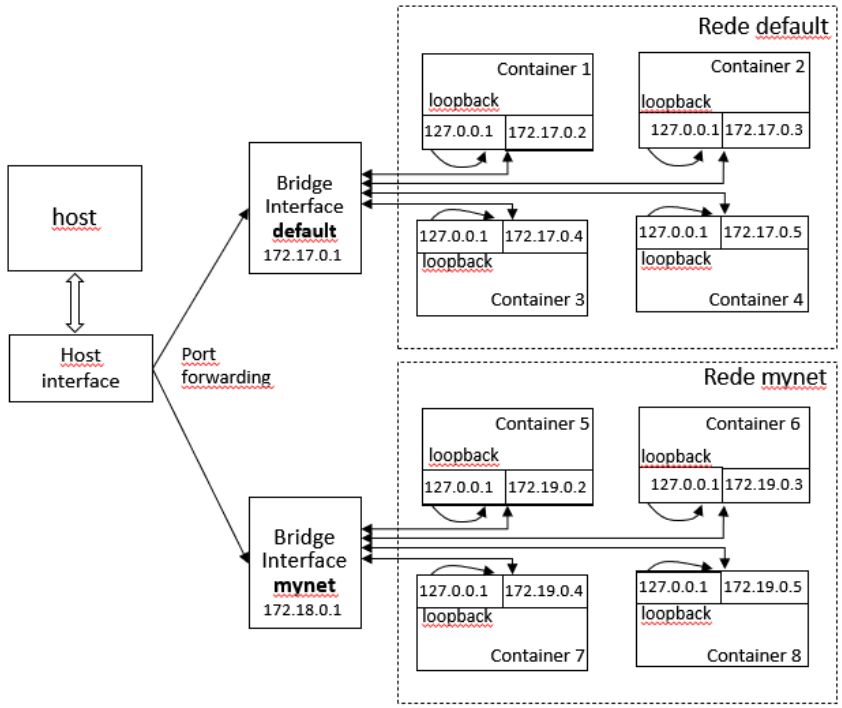

Figura 6 – Contentores em redes tipo bridge distintas e publicação de portos para acesso aos contentores pela rede do host

Volumes

Volumes são entidade do ecossistema Docker que representam pastas no *file system* do host acessíveis em contentores.

Os volumes são úteis por várias razões, em particular:

a) Permitem acesso controlado dos contentores a pastas no *file system* do *host*.
b) Partilha de ficheiros entre contentores.
c) Persistência de dados produzidos por um contentor após a sua remoção e posterior utilização por outro contentor.

Como entidades do Docker, os volumes têm um tempo de vida independente dos contentores onde são montados, e a API tem comandos sobre volumes que permitem as operações habituais a partir do cliente Docker (`create`, `rm`, `inspect`, `ls`).

Por exemplo, o comando:

```
$ docker volume create test_data
```

cria um volume de nome `test_data`, o que podemos confirmar com o comando:

```
$ Docker volume ls
DRIVER    VOLUME NAME
local     test_data
```

podemos associar volumes a contentores. O comando

```
$ docker container run -it -v test_data:/todo --name alpine1 alpine sh
```

Coloca em execução um contentor em que a pasta `/todo` monta o volume `test_data. Se entrarmos no contentor:`

```
$ docker exec -it alpine1 sh
```

Podemos acrescentar um ficheiro ao volume:

```
# echo “my data” > /todo/first.txt
```

Estes dados ficam associados ao volume, pelo que mesmo removendo o contentor, podemos aceder aos dados produzidos por este contentor noutro qualquer que monte o volume `test_data`.

Com esta utilização de volumes resolvemos a persistência dos dados produzidos pelos contentores e a partilha de informação entre contentores. Há também uma outra razão para a utilização de volumes - a não utilização de um *file system* de camadas permite um armazenamento mais eficiente dos dados, o que pode ser útil, por exemplo, para manter *logs* de processos.

Contudo, não é resolvido o cenário de acesso controlado a pastas já existentes no *file system* do *host*. Para isso é necessário criar *mount points* para pastas específicas no host, o que o Docker designa de *bind mounts*.

A utilização de volumes pode ser especificada em Dockerfiles através da instrução `VOLUME <mount point>`, onde `<mount point>` representa a pasta no contentor que permite o acesso ao volume. Note-se que não é dado um nome ao volume. Neste caso, cada contentor instanciado a partir da imagem terá um volume próprio cujo nome é definido automaticamente pelo `docker` (usando um UUID). Embora tenham nome, estes volumes designados por volumes anónimos, pois o seu nome é apenas para consumo interno do `docker`. Os cenários de utilização de volumes anónimos (criados a partir da instrução `VOLUME`) são mais limitados, tipicamente servindo para criar pastas de ficheiros temporários ou logs.

Os volumes criados por comando ou automaticamente pelo Docker residem (na versão atual do Docker em Linux) na pasta `/var/lib/docker` (onde a pasta de cada volume tem o nome que foi dado ao volume).

Compose

Os contentores são muitas vezes usados para implementar aplicações organizadas em componentes isolados, associando cada componente a um contentor (temos assim aplicações multicontentor).

É possível orquestrar os contentores da aplicação, através da criação de *scripts* que tiram partido dos comandos apresentados nas secções anteriores. Contudo, isso significa "reinventar a roda" em muitos cenários similares e obrigar o administrador de sistemas a ter de lidar com diferentes linguagens de *script* e diferentes semânticas das operações.

O `docker compose` foi criado como uma solução genérica para simplificar a orquestração do ciclo de vida dos contentores de aplicações e serviços multicontentor dentro de um mesmo *host*.

Basicamente consiste num motor (*plugin* do cliente `docker`) que processa a descrição em ficheiro (`compose.yml`), expressa em linguagem `yaml`, das entidades (contentores, redes e volumes) associadas a uma dada aplicação multicontentor. Na fig. 7 apresenta-se um exemplo de orquestração através da linguagem `yaml` usada pelo `compose`. A descrição organiza-se em secções que correspondem a diferentes entidades `docker` (`services` – contentores), `networks` e `volumes`.

```
services:
  service1:
    image: image1
    ports:
      - 8000
    environment:
      - env1=env1_value
    volumes:
      - db1:/data
    networks:
      - net1
    depends_on:
      -  service2
  service2:
    image: image2
    ports:
      - 8080:4004
    networks:
      - tvsnet
  service3:
    image: image3
    networks:
      - net1
networks:
  net1:
volumes:
  db1:
```

Figura 7 - exemplo de compose.yml

A secção network define uma nova rede (net1) com as configurações por omissão idênticas à rede criada pelo comando docker network create net1 (rede tipo bridge com servidor DNS integrado), ou utiliza a rede já definida com esse nome.

A secção volumes cria o volume de nome db1, ou utiliza o volume já existente com esse nome

A secção services define o modo de criação de cada tipo de contentor usado na aplicação. Note-se que quase todas as propriedades de cada serviço representam opções de criação existentes na Rest API do docker engine.

A exceção é a propriedade depends_on que define uma relação de dependência de service1 a service2 que o compose tenta resolver ordenando a criação e execução dos contentores de acordo com as dependências. Note-se que os contentores não participam nesse protocolo, o compose simplesmente espera um pouco antes de lançar o próximo contentor dependente.

A descrição é executada através do comando executado na pasta que contém o ficheiro compose.yml

```
$ docker compose up -d
```

Este comando cria e coloca em execução contentores associados a cada serviço especificado. Esses contentores têm nomes que resultam da concatenação do nome da pasta que contém o ficheiro yaml, do nome do serviço e de um número de sequência. Assumindo que a pasta se chama test, então os serviços service1, service2 e service3 terão associados os contentores de nome test-service1-1, test-service2-1 e test-service3-1, respetivamente. A opção -d define a execução em *background* (sem ocupar um terminal) dos vários contentores.

Uma característica importante do compose é a possibilidade de escalar a aplicação para múltiplas instâncias de cada serviço. Por exemplo o comando

```
$ docker compose up -d –-scale service1=3
```

Replicaria 3 contentores associados ao serviço service1

A terminação controlada dos contentores é realizada através do comando

```
$ docker compose down
```

Também é possível parar os contentores associados a uma aplicação multicontentor através do comando docker compose stop (coloca todos os contentores no estado exited) e recomeçá-los com o comando docker compose start.