Docker Swarm Mode

<< Back to Docker main

Contents

Bevezet?

Docker swarm mode

A docker swarm mode az 1.12-es verziótól része a docker-engine-nek, tehát már nem egy külön komponens. Az egyik leg szembeötl?bb különbség, hogy nem docker konténerekben fut, hanem része a docker-nek. ... ...

...

Fontos fogalmak

IPTV

Az IPVS (IP Virtual Server) egy szállítási rétegbeli load-balancer implementáció, amit "Layer 4 LAN switching"-nek is hívnak. Az IPVS része a Linux kernelnek. Közvetlen a Netfilter kernel szolgáltatásra épül. Az IPVS a küls? kéréseket a valós bels? TCP és UDP szolgáltatásokra irányítja a konfiguráció alapján, így egyetlen küls? IP címmel több bels? szolgáltatás is elérhet? akárcsak csak az apache név alapú virtuális hosztok esetén. Azonban van egy fontos különbség a Layer 7 HTTP load balancer-ekhez képest. Mivel a szállítási rétegben fut, nem képes http session alapú node választásra. (Természetesen egy TCP kommunikáción belül ugyan az a végpont fogja megkapni a csomagokat különben semmi értelme nem lenne.)

routing mesh

A swarm -on futó szolgáltatások portjait az úgynevezett **routing mesh** tartja nyilván. Ha egy szolágáltatást egy adott porton el kell hogy érjünk a swarm-on kívülr?l, akkor be kell regisztrálni a portot a routing mesh-be.
Az alábbi portokat kell kinyitni a VM-ek között még a swarm létrehozása el?tt:

- 7946 TCP/UDP for container network discovery.
- 4789 UDP for the container ingress network.

Tip
A **docker-machine**-el KVM-re létrehozott gépeken minden port nyitva van

Swarm cluster létrehozása

A **docker-machine** paranccsal már nem lehet közvetlen swarm-hoz kapcsolódott VM-eket létrehozni. Els?ként lére kell hozni a docker ready VM-eket, majd azokra belépve, már a dedikált swarm kezel? parancsokkal tudjuk felépíteni a cluster-t. (docker node, service és stack)

A swarm mode cluster-t egy bash szkripttel fogjuk létrehozni. 3 manager-t és 3 worker node-t. Els?ként a három manager virtuális gépet hozzuk létre. Ezután a mg0-ás gépen inicializáljuk a cluster-t, majd az m1 és m2 node-okat manager-ként beléptetjük a cluster-be. Ha ez megvan, akkor létrehozzuk a három worker node-t és azokat worker-ként léptetjük be a cluster-be.

Note
Mindig páratlan számú manager node-ot kell létrehozni, hogy a leader választó algoritmus nehogy zátonyra fusson

```
#!/bin/bash

#Create managers
for i in 0 1 2; do
        docker-machine create -d kvm --kvm-network "docker-network" --kvm-disk-size "5000" --kvm-memory "800"  mg$i
done

#Init cluster
docker-machine ssh mg0 docker swarm init --advertise-addr $(docker-machine ip mg0)

#Join managers
MANAGER_TOKEN=`docker-machine ssh mg0 docker swarm join-token -q manager`
WORKER_TOKEN=`docker-machine ssh mg0 docker swarm join-token -q worker`

for i in 1 2; do
        docker-machine ssh mg$i docker swarm join --token $MANAGER_TOKEN $(docker-machine ip mg0) --advertise-addr $(docker-machine ip mg$i)
done

#Create workers
for i in 0 1 2; do
        docker-machine create -d kvm --kvm-network "docker-network" --kvm-disk-size "5000" --kvm-memory "800"  worker$i
        docker-machine ssh worker$i docker swarm join --token $WORKER_TOKEN $(docker-machine ip mg0) --advertise-addr $(docker-machine ip worker$i)
done
```

Tip
A KVM helyett itt használhattunk volna Amzaon EC2-es driver-t is, pont ugyanígy létrehozta volna az egész cluster-t pár perc alatt. Részletek itt: Docker Swarm on AWS

A **--advertise-addr** paraméterre azt mondjuk meg, hogy az újonnan létrehozandó swarm node a swarm-management node-to-node kommunikációra melyik interfészét használja a VM-nek (ha több is van). A swarm node ezen az interfészen (alhálózaton) fogja magát reklámozni a swarm cluster-ben, a többi node az itt megadott interfészen fogja keresni. A swarm management node-to-node kommunikációt biztonsági okokból mindig VM internal hálózaton kell bonyolítani, tehát a **--advertise-addr** paraméternek mindig egy VM internal hálózati interfészt kell megadni.

A VM-eket mindig úgy kell létrehozni, hogy legalább két hálózatra csatlakozzanak. Legyen egy VM internal hálózat, ami a publikusan nem érhet? el, csak a guest-ek látják rajta egymást, és legyen egy második hálózat, ahol a VM-ek kilátnak a netre, és akár a publikus hálózatból elérhet?ek. A "**docker-machine create**"-el olyan VM-eket hoztunk létre, amikre ez teljesül:

- **eth0**:192.168.123.0/255.255.255.0 - (**docker-network**) Azt a hálózatot mi definiáltuk, ez kilát a publikus internetre. (Ezt a hálózatot a KVM a "**forward mode=nat**" paraméterrel hoztuk létre, ezért lát ki a publikus net-re, lásd KVM#Add new network cikket a részletekért.) Ha az ingress load balance-olt hálózaton akarunk elérni egy service-t akkor az adott node ezen IP címét kell használni.
- **eth1**:192.168.42.0/255.255.255.0 - (**docker-machines**) Ezt a hálózatot a KVM driver hozta létre automatikusan a VM internál kommunikációra, tehát minden node-nak az **eth1** interfész IP címét kell megadni az **--advertise-addr** paraméterben. Szerencsére a "**docker-machine ip node-name**" parancs pont ezt az ip-t adja vissza.

Részletek itt: Docker Machine#Create machines with KVM (ugyan ezen az IP-n is m?ködik az ingress hálózat lokálisan, távolról ez nem elérhet?)

KVM driver esetén ha nem az alapértelmezett OS-t akarjuk használni, akkor a **--kvm-boot2docker-url** kapcsolóval kell megadni a .ISO helyét. Én a rancherOS-t próbáltam ki és m?ködött. Innen lehet letölteni: https://github.com/rancher/os (közvetlen link: https://releases.rancher.com/os/v1.5.0/rancheros.iso)

```
docker-machine create -d kvm --kvm-boot2docker-url "/home/adam/Downloads/rancheros.iso" --kvm-network "docker-network" manager1
```

```
# virsh net-list
 Name                 State      Autostart     Persistent
----------------------------------------------------------
 default              active     yes           yes
 docker-machines      active     yes           yes
 docker-network       active     yes           yes
```

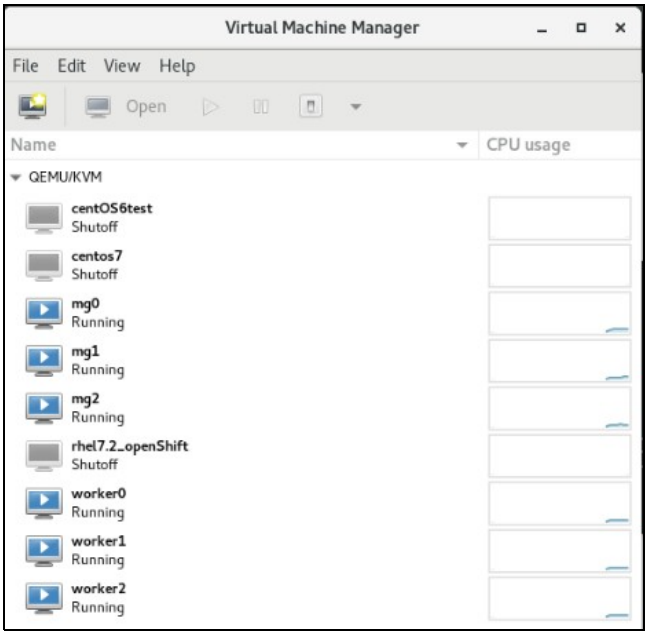
Ha lefutottak a script, nézzük meg a keletkezett VM-eket els?ként **docker-machine** szemszögb?l:

```
# docker-machine ls
NAME      ACTIVE   DRIVER   STATE     URL                         SWARM   DOCKER        ERRORS
mg0       -        kvm      Running   tcp://192.168.42.41:2376            v18.05.0-ce
mg1       -        kvm      Running   tcp://192.168.42.79:2376            v18.05.0-ce
mg2       -        kvm      Running   tcp://192.168.42.154:2376           v18.05.0-ce
worker0   -        kvm      Running   tcp://192.168.42.162:2376           v18.05.0-ce
worker1   -        kvm      Running   tcp://192.168.42.74:2376            v18.05.0-ce
worker2   -        kvm      Running   tcp://192.168.42.136:2376           v18.05.0-ce
```

Majd nézzük meg a **virsh**-val is.

```
# virsh list
 Id    Name                           State
----------------------------------------------------
 2     mg0                            running
 3     mg1                            running
 7     mg2                            running
 8     worker0                        running
 9     worker1                        running
```

```
10     worker2                        running
```



Végezetül listázzuk ki a swarm cluster node-jait els?ként az mg0-án, majd az mg1-en. Mind a két esetben ugyan azt az eredményt kapjuk. Láthatjuk, hogy jelenleg az mg0 a vezet?.

```
# docker-machine ssh mg0 docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
nl5mmm994ckimhe5vjazjjecs *   mg0                 Ready               Active              Leader              18.05.0-ce
vacts6xlgb6ufyx49vx6fxgt0     mg1                 Ready               Active              Reachable           18.05.0-ce
3llb2b0qh7oids0qhych9w73r     mg2                 Ready               Active              Reachable           18.05.0-ce
s7hp748qu6u4bb98doss31t4r     worker0             Ready               Active                                  18.05.0-ce
211o588k4qw2uymq6dl977mmt     worker1             Ready               Active                                  18.05.0-ce
hlzuosjp7wx6rxt0a66fms698     worker2             Ready               Active                                  18.05.0-ce

# docker-machine ssh mg1 docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
nl5mmm994ckimhe5vjazjjecs     mg0                 Ready               Active              Leader              18.05.0-ce
vacts6xlgb6ufyx49vx6fxgt0 *   mg1                 Ready               Active              Reachable           18.05.0-ce
3llb2b0qh7oids0qhych9w73r     mg2                 Ready               Active              Reachable           18.05.0-ce
s7hp748qu6u4bb98doss31t4r     worker0             Ready               Active                                  18.05.0-ce
211o588k4qw2uymq6dl977mmt     worker1             Ready               Active                                  18.05.0-ce
hlzuosjp7wx6rxt0a66fms698     worker2             Ready               Active                                  18.05.0-ce
```

GUI swarm management with Portainer

Több grafikus docker manager/monitoring eszköz is létezik:

- Shipyard (webes)
- Portainer (webes)
- Kitematic (vastag kliens)

A legegyszer?bb a **Portainer** használata, ami egyetlen konténert telepít fel a docker-be, képes távoli docker démonho is kapcsolódni, és van benne swarm mode támogatás is. (docker nélkül is futtatható)

Két lehet?ségünk van a Portainer futtatására:

- A Portainer konténerünket lokálisan futtatjuk, és távolról kapcsolódik valamelyik manager node remote API-jához, ahonnan a swarm adatokat is ki tudja olvasni, illetve módosítani is tudja a konfigurációt. A távoli kapcsolatot TLS autentikációval lehet megoldani. A Portainer webes felülete csak lokálisan lesz elérhet?.
- A Portainer-t swarm service-ként telepítjük fel azzal a megkötéssel, hogy csak manager noder-ra telepíthet?. A manager node-on az ott lokális docker socket-re fog kapcsolódni, ahonnan a swarm adatokat is ki tudja olvasni manager node-ról lévén szó. Ekkor a Portainer webes felülete globálisan lesz elérhet? mindenki számára.

Portainer telepítése swarm service-ként

A legkézenfekv?bb megoldás, ha a Portainer-t swarm service-ként a manager node-ok valamelyikére telepítjük föl. Az már mindegy, hogy melyik manager node-ra kerül, azt bízzuk a swarm-ra (cow személet), bármelyikre is kerül, lokális docker socket-re csatlakozva el fogja érni a cluster adatokat.

Az a gond, hogy a Portainer stateful szolgáltatás, a /**data** konténer mappában tárolja az adatait (állapotát). Staeful szolgáltatások kezelése egy kicsit nehézkes. A /data a mappát kéne legalább a guest VM-re lementeni, hogy ha a szolgáltatás újra indulna ne vesszenek el a Portainer adatok (állapot). Azonban ha a swarm egy másik manager nodre-ra újratelepítené, akkor fontos hogy ott is rendelkezésre álljon ugyan az a /**data** mappa tartalom (állapot), hogy ne vesszenek el a beállítások. Ehhez valamelyik docker volume plugin -t kell használjuk, lásd a Docker volume orchestration cím? cikket.

A swarm service-ként való futtatásnak az a nagy el?nye, hogy a Portainer életciklusát nem nekünk kell kezelni, ha az a manager node meghal, ahova a Portainer eredetileg telepítve volt, akkor a swarm automatikusan újra fogja telepíteni egy másik manager node-on. És mivel a Portainer /data mappáját a manager node-ok között egy közös perzisztens meghajtóra csatoltuk föl, miután a swarm újratelepíti a Portainer-t, az ott tudja folytatni, ahol a mások node-on abba hagyta.

Portainer telepítése

Telepítsük fel a **Netshare** docker volume plugin-t a Docker volume orchestration/Netshare fejezetben leírtak alapján. A Netshare segítségével felcsatolhatunk NFS megosztást közvetlen a Portainer konténerbe on-demand alapon, tehát a megosztás csak azon a node-on fog létrejönni, ahova a swarm a Portainer-t telepíti majd.

A portainer-t swarm service-ként fogjuk telepíteni. Ki fogjuk kötni hogy csak manager node-ra telepítheti a swarm, tehát biztos, hogy az el?bb létrehozott mg0, mg1 és mg2 valamelyikén fog landolni. Fontos, hogy megadjuk, hogy csak egy példány jöhessen bel?le létre. Két fontos mount-ot kell beállítani:

1. /**data**: Egy távoli NFS megosztást fogunk felcsatolni a Netshare plugin segítségével, ahol a Portainer a perzisztens adatait fogja tárolni.
2. /**var**/**run**/**docker.sock**: ezt a guest VM docker socket-jére kell rákötni, hogy a portainer hozzáférjen a docker/swarm adatokhoz és tudja is módosítani azokat.

```
eval $(docker-machine env mg0)

docker service create \
--name portainer \
--publish 9000:9000 \
--replicas=1 \
--constraint 'node.role == manager' \
--mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
--mount type=volume,src=192.168.42.1/home/adam/Projects/DockerCourse/persistentstore/portainer/data/,dst=/data,volume-driver=nfs \
portainer/portainer -H unix:///var/run/docker.sock
```

Ha a portainer-t swarm service-ként futtatjuk, akkor fontos, hogy command-nak is megadjuk a unix-docker socket használatát: **-H unix:///var/run/docker.sock**

Note
A command részben fontos, hogy a unix után három / jel van. Ebb?l aztán kett? lesz mire konténer lesz bel?le, nem pontosan értem, hogy ha escape-elni kell, akkor miért nem négy kell, minden esetre ha csak kett?t írunk, akkor a /run/docker.sock-ban fogja keresni a /var/run/docker.sock helyett

Nézzük meg melyik manager node-ra települt a portainer.

```
# docker service ps portainer
ID                  NAME                IMAGE                          NODE
jh04xrikkskh        portainer.1         portainer/portainer:latest     mg0
```

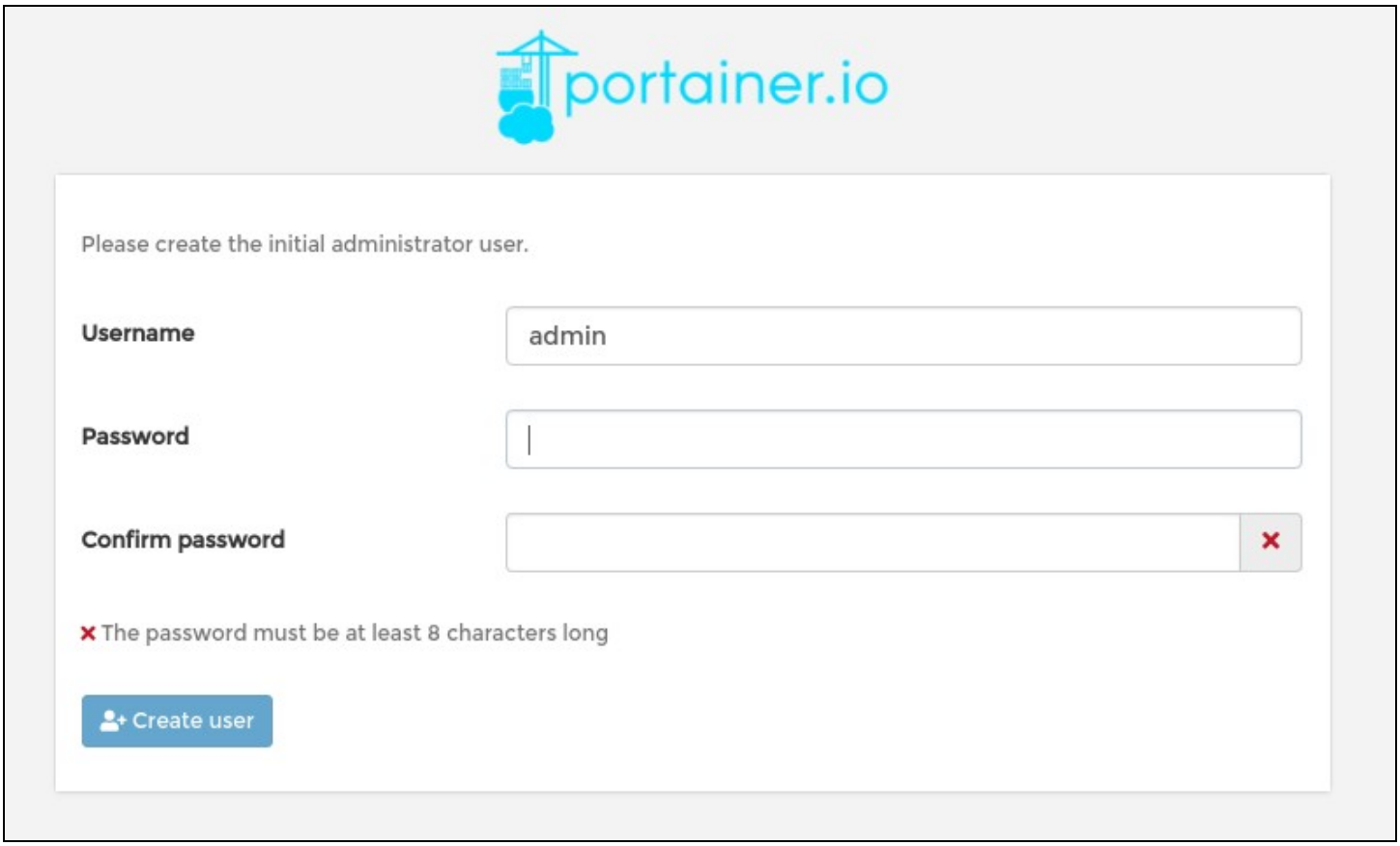
Láthatjuk hogy az mg0 manager node-ra került ki.

Belépés a web-es konzolra

A 9000-as portot publikáltuk a swarm ingress hálózatán, ami azt jelenti, hogy a swarm node-ok bármelyikének az IP címén, a 9000-as porotn el fogjuk érni a Portainer webes konzolját:

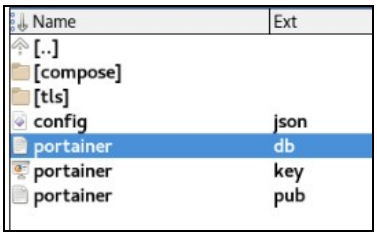
```
# docker-machine ip mg0
192.168.42.231
```

http://192.168.42.231:9000/#/init/admin



Els? induláskor be kell állítani az admin felhasználót.

A host gépen láthatjuk, hogy az NFS mappában megjelentek a portainer fájlok:



Portainer telepítése lokális konténerként

A következ?kben megnézzük, hogyan lehet távolról kapcsolódni a Portainer-el egy swarm cluster manager node-jához. Ez a megoldás több szempontból sem annyira el?nyös mint a swarm service-ként való telepítés, ezek a következ?k:

1. Egy dedikált manager-hez kapcsolódunk, ha az kiesik, ugrott az egész GUI management.
2. Csak azon a laptopon elérhet? a portainer, ahol összel?ttük a Remote kapcsolatot, tehát csak egy ember fér így hozzá, nem az egész csapat.
3. Nekünk kell kezelni a Portainer életciklusát, vagyis ha a Portainert konténer, vagy az azt futtató VM leáll, akkor nekünk kell kézzel újra indítani, vagy akár újra telepíteni a szolgáltatást.

Ezzel szemben el?nyök:

1. Egy Portainer példányból az összes swarm cluster-ünket kezelhetjük, mivel tetsz?leges számú remote rendszert lehet hozzáadni. Ennek az az el?nye, hogy a Portainer autentikációt és authorizációt csak egyszer kell beállítani.

A Portainer a távoli docker-hez a TLS API-n keresztül fog csatlakozni, amihez szükségünk van az ott futó docker démon publikus és titkos kulcsára. A KVM dirver-el készült docker-machine-ekre **boot2docker** operációs rendszer kerül feltelepítésre, ha ezt nem változtatjuk meg. A boot2docker-ben alapértelmezetten be van kapcsolva a TLS remote docker API (port: **2376**), és a titkosítatlan távoli hozzáférés ki van kapcsolva (port: 2375), tehát csak TLS-el lehet a manager-en futó docker démon-ra csatlakozni

Note
A példában a **mg0** manager node-ra fogunk kapcsolódni, de pont ugyan ezt az eredményt kapnánk az **mg1** és **mg2** manager-ekkel is.

TLS kulcsok és IP cím begy?jtése

Mikor a docker-machine létrehozta a manager node-okat, legyártotta azokat az ssh kulcsokat, aminek a segítségével a docker-machine be tud ssh-zni a VM-re jelszó és felhasználó név megadása nélkül (docker-machine ssh mg0). Ezeket a kulcsokat fogjuk mi is felhasználni, hogy a lokálisan futó **Portainer** hozzá tudjon kapcsolódni a (távoli) virtuális gépen futó docker démonhoz.

Adjuk ki **docker-machine env** parancsot, hogy megtudjuk, hol tárolja a lokális docker-machine környezetünk a távoli VM SSH kulcsait:

```
# docker-machine env mg0
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.42.41:2376"
export DOCKER_CERT_PATH="/root/.docker/machine/machines/mg0"
export DOCKER_MACHINE_NAME="mg0"
# Run this command to configure your shell:
# eval $(docker-machine env mg0)
```

Látható, hogy a kulcsok a **/root/.docker/machine/machines/mg0** mappában vannak

Nekünk három fájlra van innen szükségünk. A CA-ra, a certifikációnkra és a titkos kulcsunkra. A docker-machine a publikus kulcsot még telepítés közben felmásolta a VM-re.

```
# ll /root/.docker/machine/machines/mg0
total 316444
..
-rw-r--r-- 1 root root       1029 Jul 15 22:41 ca.pem
-rw-r--r-- 1 root root       1070 Jul 15 22:41 cert.pem
-rw------- 1 root root       1675 Jul 15 22:41 key.pem
...
```

Másoljuk a kulcsokat egy olyan mappába, ahol a böngész?t futtató user is eléri. (Ugyanis nagy valószín?séggel a root mappában jöttek ezek létre). A kulcsokat majd a Portainer webes telepítése közben majd tallózni kell.

A kulcsokon felül szükségünk lesz a master **mg0** IP címére is, ezt kell megadni a Porainer-nek:

```
# docker-machine ip mg0
192.168.42.41
```

A port, ahogy már írtuk, az alapértelmezett TLS docker remote port: **2376**
Ez az alapértelmezett remote port a docker démonnak. Szerencsére ez a port elve nyitva van a boot2docker operációs rendszerben.

Portainer telepítése lokális docker-en

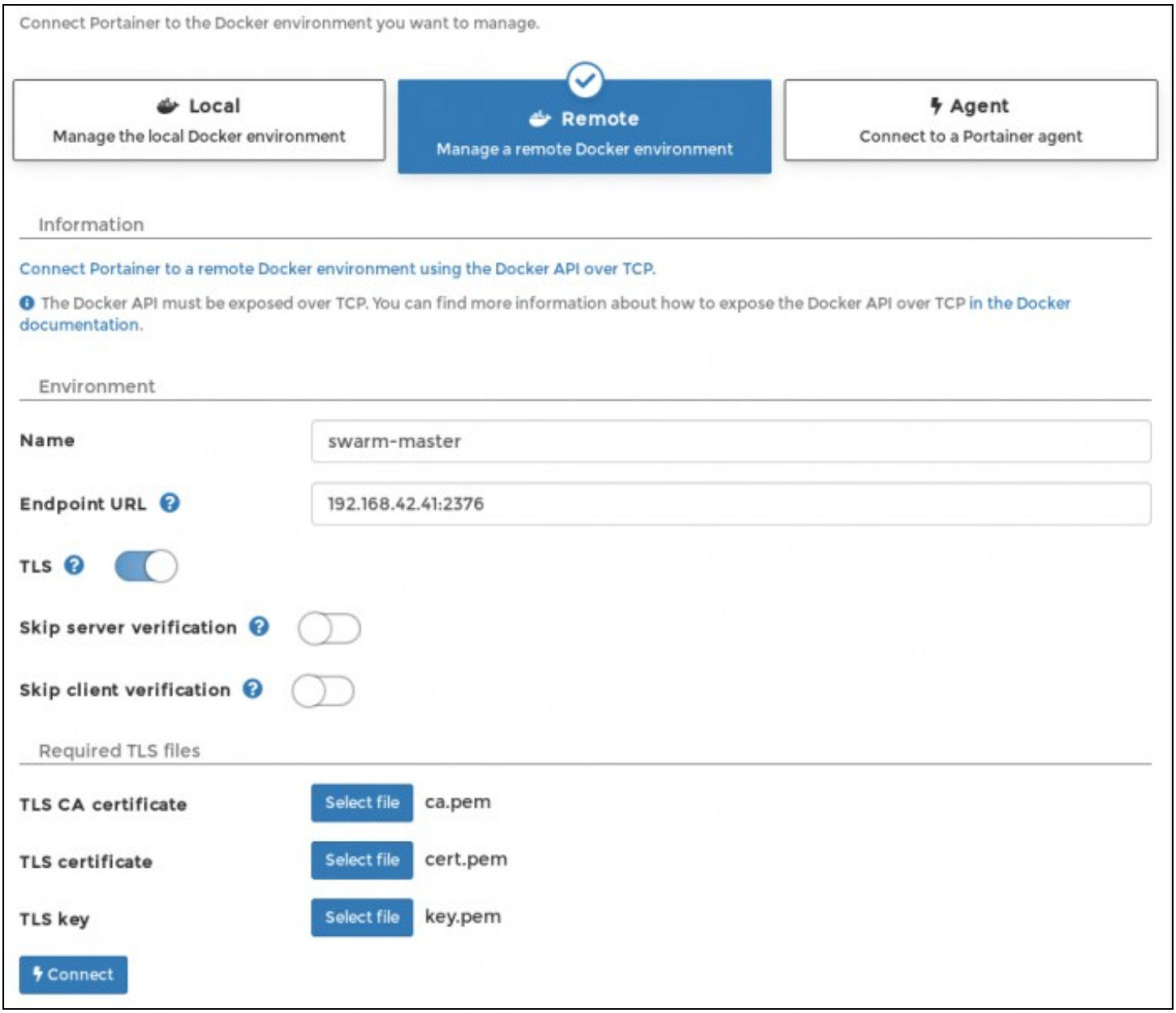
A Portainer-hez egyetlen egy image-et kell telepíteni: **portainer/portainer**

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

- A webes konzol a **localhost:9000** -as porton lehsz elérhet?:
- Ha a lokális docker démon-t is monitorozni akarjuk, akkor a docker démon socket-re rá kell kötni a portainer-t.

Note
Ha nem akarjuk a lokális daemon-t is monitorozni, akkor a **-v /var/run/docker.sock:/var/run/docker.sock** mount nem szükséges

Ha elindult a konténer, akkor válasszuk **Remote** lehet?séget



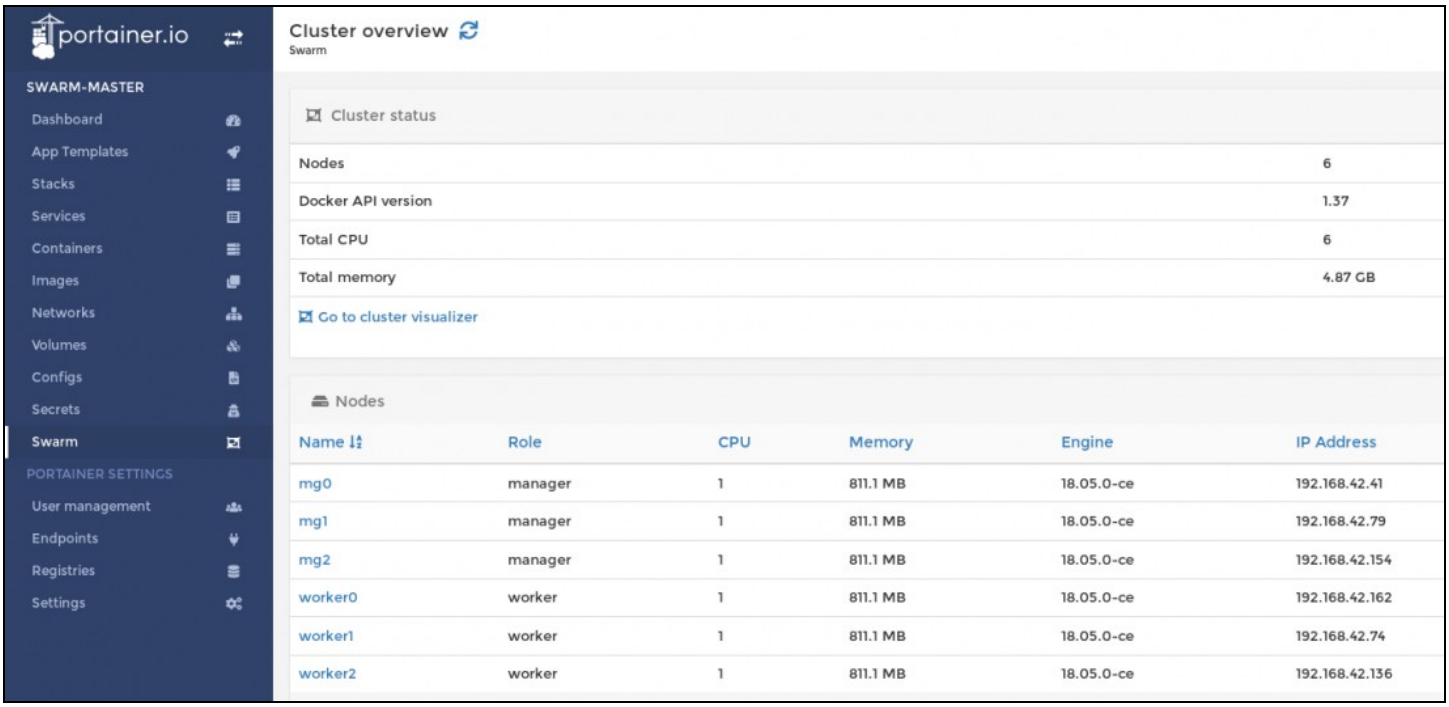
- Adjuk meg a manager-t futtató VM IP címét a 2376 portal (alapértelmezett TLS port)
- Kapcsoljuk be a TLS-t.
- TLS CA certificate: **ca.pem**
- TLS certificate: **cert.pem**
- TLS key: **key.pem**

Warning
Ha nem adunk olvasási jogot a docker-nek a kulcsokra, azt fogja kiírni a Portainer: "**Unable to create node**"

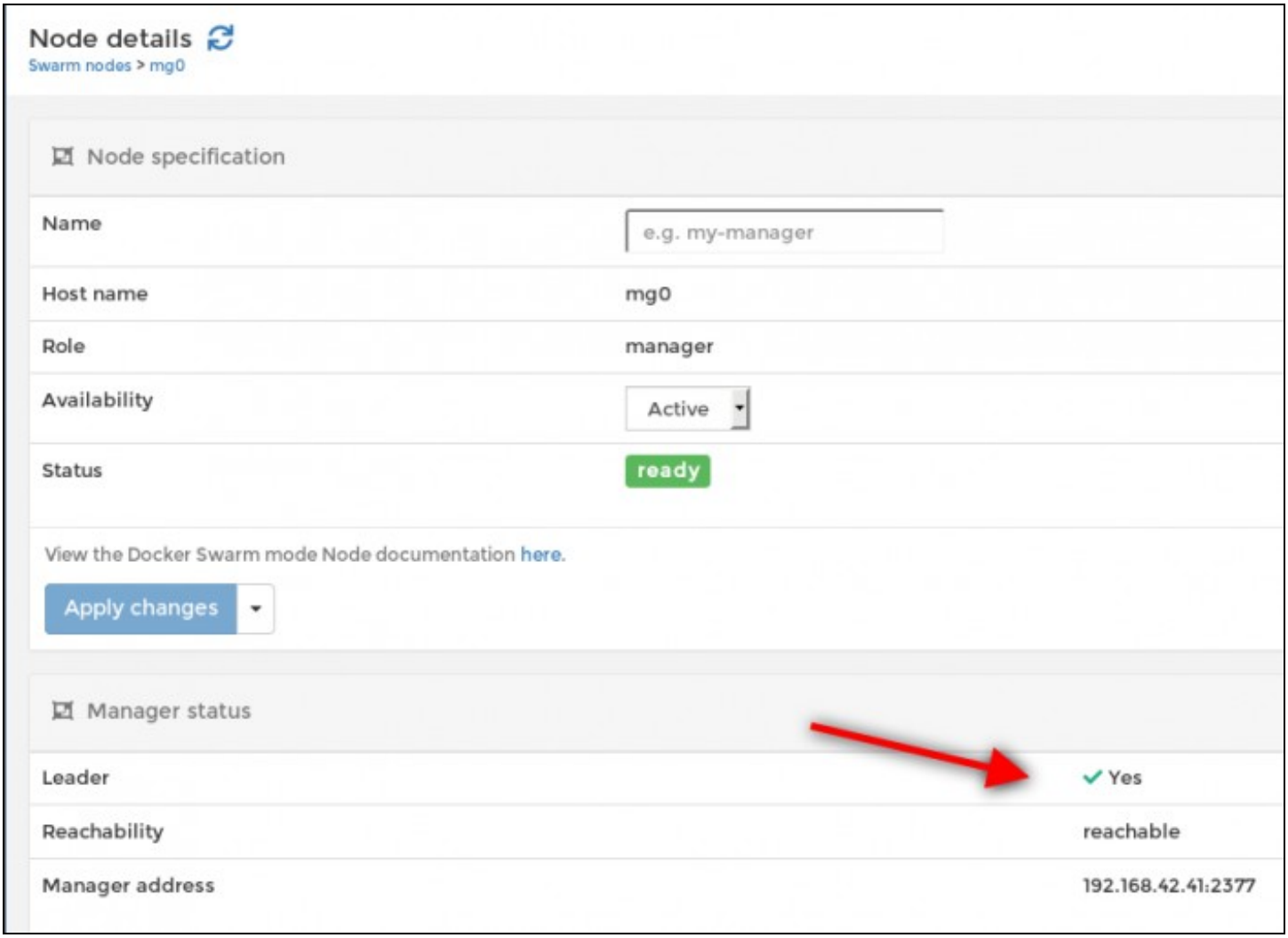
Majd mondjuk hogy connect. Ekkor bejön a desboard. Innent?l kezdve a távoli manager docker démonjához kapcsolódunk.

Cluster monitorozása

Nyomjunk rá a **Go to cluster visualizer** linkre, vagy a baloldali menüben a **swarm** menüpontra.



Nézzük meg az mg0 node részleteit. Láthatjuk, hogy jelenleg ? a managerek vezet?je:



Service futtatása

https://blog.scottlogic.com/2016/08/30/docker-1-12-swarm-mode-round-robin.html

Service létrehozása image-b?l (swarm service)

A **docker service create** paranccsal egy darab docker image-b?l készíthetünk a swarm cluster-en futó szolgáltatást. Megadhatjuk, hogy hány példány jöjjön létre bel?le, de lényegében megegyezik a szintaxisa a **docker run** paranccsal.

Szolgáltatás definiálása

Hozzunk létre

```
# docker-machine ssh mg0 docker service create --name web --replicas 3 --mount type=bind,src=/etc/hostname,dst=/usr/share/nginx/html/index.ht
```

Tip
A docker 1.17-es verziójától kezdve, a **--mount** paramétert kell használni a **-v** (--volume) helyett a storage-ek kezelésére, a -v már elavult. Korábbi verziókban csak a swarm parancsokba lehetett használni, mostanra már a standalone docker parancsokban is ezt illik használni. A szintaktikája eltér a **-v**-t?l, ugyanis a **--mount** után **név=érték** párok következnek vessz?vel elválasztva szemben a **-v** három :-al elválasztott tagjával. Azonban swarm szolgáltatás esetén elve csak a **--mount** használható, a -v nem.

Monitorozás

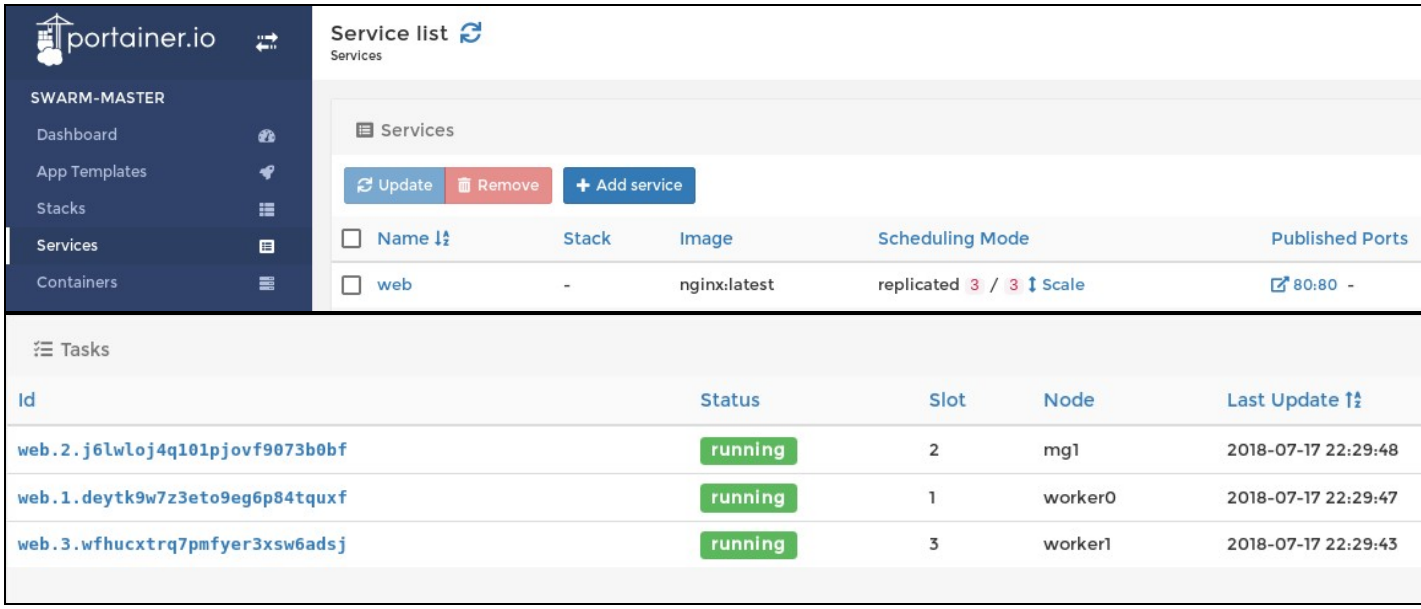
Listázzuk ki a swarm-unkon futó szolgáltatásokat. Ezt bármelyik manager-en kiadhatjuk:

```
# docker-machine ssh mg1 docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
yv47d25nc6dr        web                 replicated          3/3                 nginx:latest        *:80->80/tcp
```

Most listázzuk ki **ps**-el a szolgáltatás részleteit. Láthatjuk hogy a woker0, 1 és az mg1-re telepítette fel.

```
# docker-machine ssh mg1 docker service ps web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR
deytk9w7z3et        web.1               nginx:latest        worker0             Running             Running 14 minutes ago
j6lwloj4q101        web.2               nginx:latest        mg1                 Running             Running 14 minutes ago
wfhucxtrq7pm        web.3               nginx:latest        worker1             Running             Running 14 minutes ago
```

Nézzük meg a portainer-ben is az új szolgáltatásunkat a services menüpont alatt:

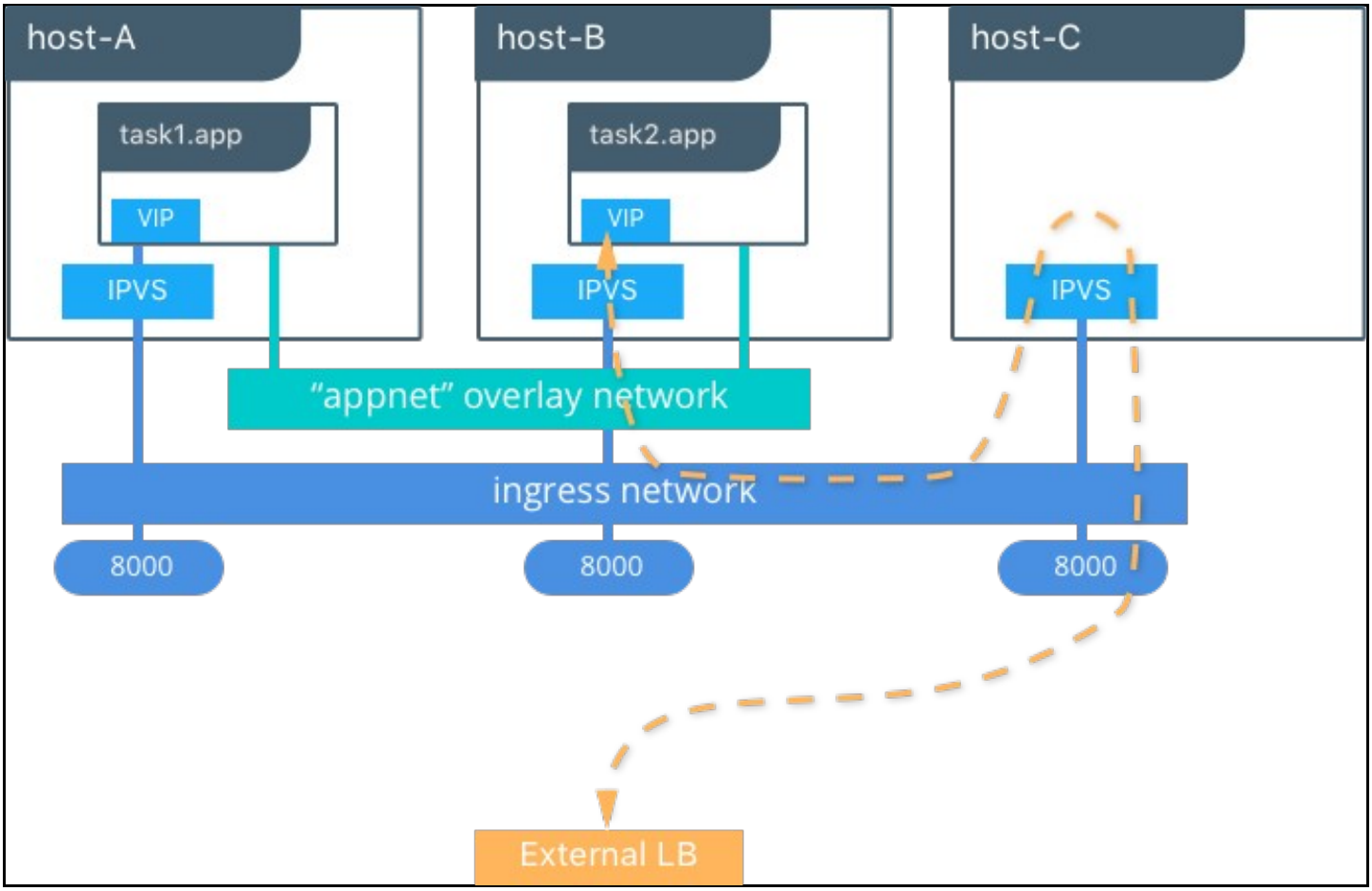


Load balancing

Nézzük meg a swarm nativ load balancert. Hívjuk meg a cluster bármelyik tagjának a publikus IP címét a 80-as porton:

```
# curl http://192.168.42.41:80
worker0
# curl http://192.168.42.41:80
worker1
# curl http://192.168.42.41:80
mg1
```

Minden egyes hívásra egy másik lábra fogunk beesni. Ez az elvárt m?ködés. Ha máshogy nem határozzuk meg a swarm szolgáltatás telepítése közben, akkor a natív Layer 4 szolgáltatás rétegbeli load balancer lesz bekapcsolva. Az összes swarm node-on fut a load balancer egy példánya, és ezek össze vannak kötve az úgynevezett routing mesh-el, az ingress hálózaton. Így teljesen mindegy melyik lábra esünk be, az a load balancer példány ami azon a lábon fut, ahova beestünk, át fogja irányítania a kérést annak a node-nak aki a "globális" load balanc algoritmus szerint a soron következ?. Ha másképp nem adjuk meg, akkor a load banac-olás round rubin, tehát mindig a soron következ?t adja.



Részletek a Routing mesh cím? fejezetben.

Scaling

A replikák számát a docker service scale paranccsal változtathatjuk meg. Növeljük meg 5-re:

```
# docker-machine ssh mg0 docker service scale web=5
web scaled to 5
```

Majd nézzük meg mi lett:

```
# docker-machine ssh mg1 docker service ps web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE                ERROR
deytk9w7z3et        web.1               nginx:latest        worker0             Running             Running 28 minutes ago
j6lwloj4q101        web.2               nginx:latest        mg1                 Running             Running 28 minutes ago
wfhucxtrq7pm        web.3               nginx:latest        worker1             Running             Running 28 minutes ago
7nwtcd43vva9        web.4               nginx:latest        worker2             Running             Running about a minute ago
kbz72e0wf1ba        web.5               nginx:latest        mg2                 Running             Running about a minute ago
```

A portanier-ben egy kattintással növelni tudjuk a szolgáltatás replika számát felfelé vagy lefelé a **scale** nyílra kattintva a **Services** listában.

Swarm stack

A **docker stack** szolgáltatások összessége, amik együttesen egy közöz applikációt alkotnak a cluster-en. Lényegében a **docker compose** swarm mode-ra adaptált változata. A stack-et ugyan úgy egy yml fájlban kell leírni, akárcsak a docker compose esetén, azonban a yml fájlban van egy extra szekció a szimpla docker compose-hoz képest, ez a **deploy**, amivel a swarm cluser-nek adhatunk utasításokat, mint pl, hogy hány példányban fusson egy adott image a stack-en belül.

Networking

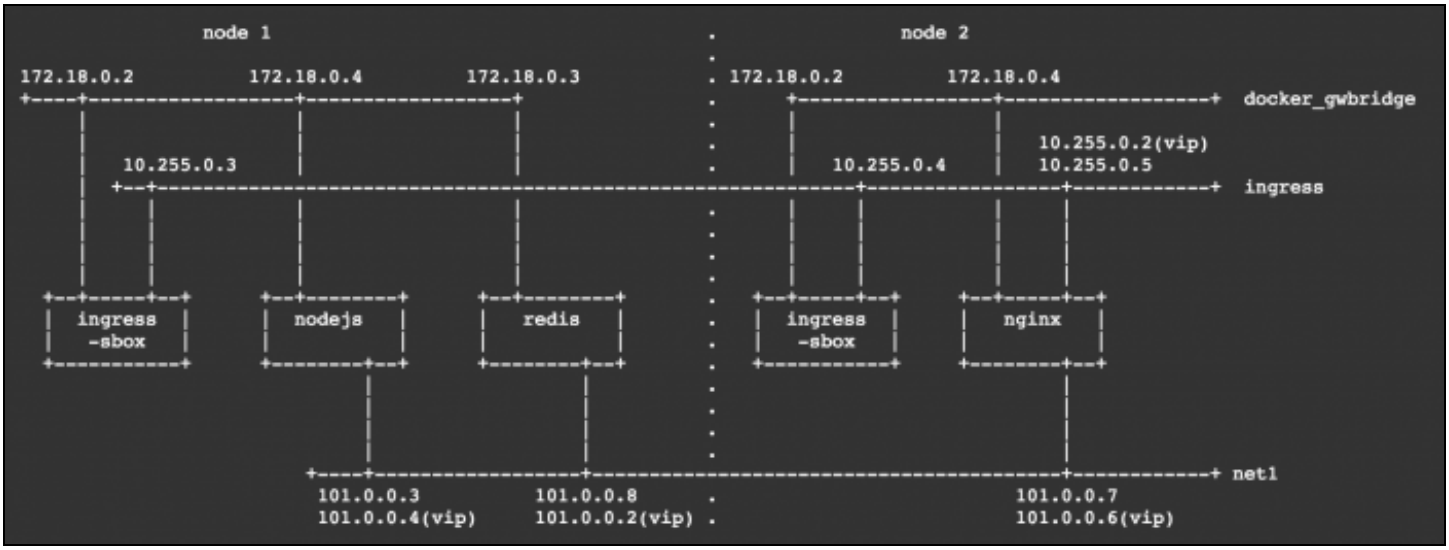
Hálózat típusok

Háromféle fontos hálózat típus van a swarm-ban az alap docker hálózatokon felül (ebb?l egyik a másik részlalmaza)
https://docs.docker.com/v17.09/engine/swarm/networking/

- **overlay network**: Ez egy hostokon átível? virtuális hálózat fajta, amit a docker emulál. (tehát nem Linux kernel funkció). A rá kapcsolódott konténerek számára transzparensen m?ködik, hiába vannak más és más fizikai hálózatokban, úgy látják, mind ha közös hálózaton lennénk. Az overlay hálózatot a docker az úgynevezett overlay driver-el hozza létre. Ezzel emulálja a hálózatot.

- **ingress networ**k: Ez egy speciális overlay hálózat, ebb?l csak 1 lehet egy swarm-on belül. Nem a neve, hanem a típusa számít. Ha mi nem hozzuk létre, akkor a docker fogja létrehozni a cluster inicializálásakor. Ezt load-balancolásra használja a docker. Ha bármelyik swarm node -ra érkezik egy kérés (akár olyanra is amin nem is fut konténer) akkor is továbbítja a kérést a megfelel? node-ra. A load-balancer-t a Linux kernelben található IPVS-el hozza létre, ami egy szállítási réteg beli load-balancer. A megfelel? konténer megtalálását (IP cím + port) az úgynevezett "routing mesh" végzi.

- **docker_gwbridge**: Nem teljesen értem mire jó. Egy fizikai hálózati elem, tehát nem a docker emulálja. Az overlay network és a node interfésze között hoz létre egy virtuális hidat. Ha nincs, akkor a docker létrehozza. A neve számít. Ha újat akarunk csinálni, akkor Linux operendszer szinten kell törölni, majd egy pont ilyen nev? új hálózatot kell létrehozni a megfelel? paraméterekkel.



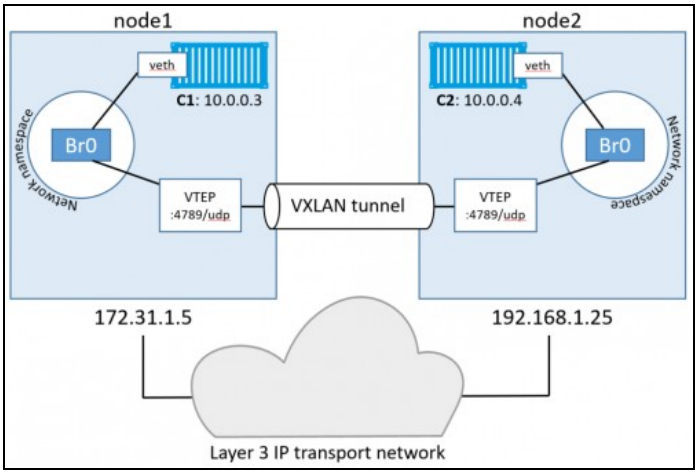
Overlay hálózatok

Átekintés

http://blog.nigelpoulton.com/demystifying-docker-overlay-networking/

- A beépített ingress overlay hálózaton felül mi is létrehozhatunk kézzel új overlay hálózatokat. Az overlay hálózatot össze kell rendelni swarm service-ekkel (ami swarm stack-en belül is lehet)
- Ha egy swarm service-t, egy overlay hálózathoz rendelünk, akkor az adott service összes konténere (akkor is ha távoli node-okon vannak) képes lesz egymással kommunikálni.
- Az overlay hálózatot a manager csak azokra a node-okra fogja kiterjeszteni, amik a megadott szolgáltatás konténereit futtatják. Egészen addig csak a manager-en létezik.
- Az overlay hálózat csak a node-on futó konténerb?l fog látszani. A konténerben létre fog jönne egy interfész, ami az overlay hálózatra csatlakozik, itt fog kiosztani egy IP címet az overlay network a konténernek. Tehát fontos, hogy a node gépen nem jön létre olyan interfész, ami az overlay hálózatra csatlakozok. A node gépen csak egy bridge jön létre.
- A docker a VXLAN tunel technológiát használja az overlay hálózatok létrehozására. Egy Layer 3 hálózaton hoz létre egy virtuális Layer 2 hálózatot.

Az alábbi ábrán a C1 és C2 konténereken jött létre overlay interfész, aminek az overlay bridge kiosztotta a 10.0.0.3 ill a 10.0.0.4 IP címét. Látható hogy a node1 és node2 hoszt gépeken csak a bridge található, azok nem kaptak overlay IP címéet. A két bridge-et a VXLAN tunnel köti össze.



Hálózat létrehozása

Overlay hálózatot csak a manager node-okon lehet létrehozni a **-d overlay** kapcsoló használatával. Hozzuk létre az **over-test** hálózatot.

```
# docker-machine ssh mg0 docker network create -d overlay over-test
```

Egészen addig, amíg nem rendeljük szolgáltatáshoz az overlay hálózatot, csak a manager node-okon lesz listázható. Figyeljük meg, hogy az mg0-án hoztuk létre, de az mg1-en is látszik:

```
# docker-machine ssh mg1 docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
idf7f51tp95i        ingress             overlay             swarm
...
6hig77pse2xt        over-test           overlay             swarm <<<
```

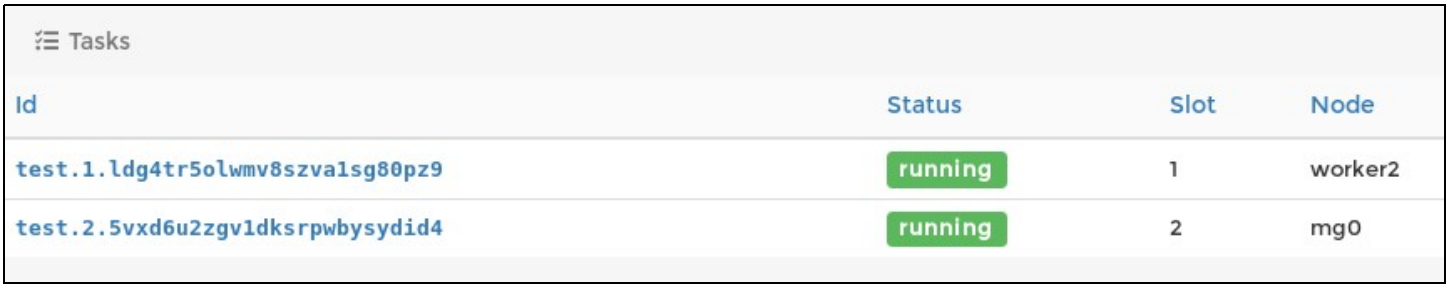
Viszont a worker1-en nem látszik. (Aki nem hiszi nézze meg)

Most hozzunk létre egy új szolgáltatást két replikával, és rendeljük hozzá az új **over-test** overlay hálózathoz. A hálózatot a --network kapcsolóval kell megadni. A szolgáltatásneve **test** lesz. (Természetesen a manager node-ok egyikén kell létrehozni az új szolgáltatást)

```
# docker-machine ssh mg1 docker service create --name test \
--network over-test \
--replicas 2 \
ubuntu sleep infinity
```

Nézzük meg a test szolgáltatás részleteit, hogy lássuk melyik node-okon fut:

```
# docker-machine ssh mg1 docker service ps test
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR
ldg4tr5olwmv        test.1              ubuntu:latest       worker2             Running             Running 50 seconds ago
5vxd6u2zgv1d        test.2              ubuntu:latest       mg0                 Running             Running 54 seconds ago
```



Láthatjuk, hogy létrejött a szolgáltatás két példánya az **mg0**-án és **worker2**-ön.

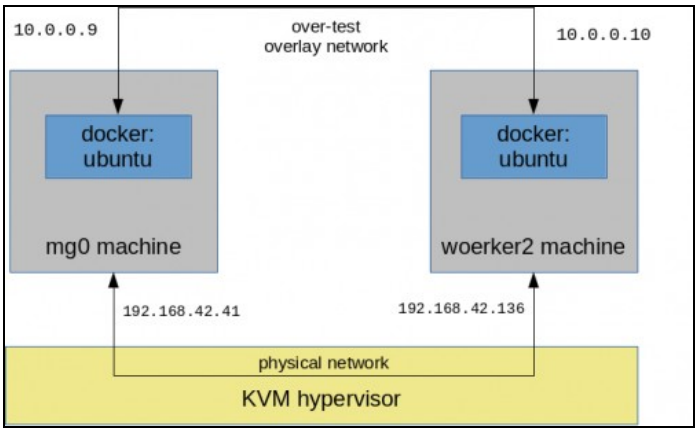
Listázzuk ki a **worker2** docker hálózatait. Láthatjuk, hogy a swarm a **worke2**-re is kiterjesztette az **over-test** overlay hálózatot.

```
# docker-machine ssh worker2 docker network ls
```

```
NETWORK ID          NAME                DRIVER              SCOPE
911906e5a269        bridge              bridge              local
5125832d68a8        docker_gwbridge     bridge              local
73a48e5dd720        host                host                local
idf7f51tp95i        ingress             overlay             swarm
ae8e1b7bd437        none                null                local
6hig77pse2xt        over-test           overlay             swarm <<<<
```

Tesztelés

Innent?l kezdve a **test** szolgáltatás két konténere elérik egymást az over-test hálózaton. Ezt demonstrálva, a test szolgáltatás m0-án futó konténeréb?l meg fogjuk pingelni a worker2-n futó konténert.



Próbáljuk ki. Derítsük ki a **worker2**-n futó ubuntu konténer **over-test** hálózati IP címét. Els?ként listázzuk az overlay hálózatunk részleteit:

```
# docker-machine ssh worker2 docker network inspect over-test
...
        "Name": "over-test",
        "Driver": "overlay",
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1"
        "Peers": [
                "Name": "c3651bad2030",
                "IP": "192.168.42.136" <<< ez itt nem az overlay cím, hanem a host címe!!!

                "Name": "41d7e29fc1fc",
                "IP": "192.168.42.41"
```

Láthatjuk, hogy a **over-test** overlay hálózatunk alhálózata két végpont csatlakozik, ezen felül láthatjuk hogy az overlay hálózatunk IP tartománya a 10.0.0.0/24. Az egyik peer a **worker2**, a másik az **mg0** (De ez nem innen látszik :))

Note
Fontos látni, hogy a **Peers** alatt felsorolt IP címek nem az overlay interfészek címe, hanem azoknak a VM-eknek (valódi gépeknek) a 'publikus' IP címei, ahol azok a konténerek futnak, akik csatlakoznak az **over-test** overlay hálózatra.

Ki kell deríteni, hogy mi a **worker2**-ön futó ubuntu konténer azon interfészének az IP címe, ami az **over-test** overlay hálózatra csatlakozik. Ehhez els?ként be kell ssh-zni a worker2 node-ra, majd az ott futó ubuntu konténerhez kell csatlakozni interaktív módon, hogy listázni tudjuk az interfészeit:

```
# docker-machine ssh worker2
docker@worker2:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
236f0a0cc334        ubuntu:latest       "sleep infinity"    35 minutes ago

docker@worker2:~$ docker exec -it 23 bash

root@239f2a1d9bc1:/# apt-get update && apt-get install net-tools
```

```
root@239f2a1d9bc1:/# ifconfig | grep 10.0.0.
        inet 10.0.0.10  netmask 255.255.255.0  broadcast 10.0.0.255
```

Tehát a worker2 node-on futó ubuntu konténer overlay címe: 10.0.0.10

Pingeljük meg az **mg0**-en futó ubuntu konténerb?l a worker2-n futó konténert. Ha megnézzük a mg0-en futó konténereket, láthatjuk a **test** szolgáltatáshoz tartozó ubuntu konténert:

```
# docker-machine ssh mg0 docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
426f0a0bb381        ubuntu:latest       "sleep infinity"    35 minutes ago
```

Ebb?l az ubuntu konténerb?l fogjuk pingelni a test szolgáltatás másik konténert, ami a worker2 gépen fut (ami elvileg egy fizikailag teljesen máshol lév? gép is lehetne)

Els?ként ssh-val be kell lépni az **mg0**-ra. Majd az ubuntu konténeren futtassuk interaktív módon a bash-t. Ez után fel kell telepíteni a ping parancsot, majd már futtathatjuk is a ping-et.

```
# docker-machine ssh mg0

docker@mg0:~$ docker exec -it 426f0a0bb381 bash

root@426f0a0bb381:/# apt-get update && apt-get install -y iputils-ping

root@426f0a0bb381:/# ping 10.0.0.10
PING 10.0.0.10 (10.0.0.10) 56(84) bytes of data.
64 bytes from 10.0.0.10: icmp_seq=1 ttl=64 time=1.07 ms
64 bytes from 10.0.0.10: icmp_seq=2 ttl=64 time=1.34 ms
```

Láthatjuk, hogy a **mg0**-án lév? konténer képes kommunikálni a **worker2**-n lév? másik **test** konténerrel, mivel mind a ketten ugyan ahhoz az overlay hálózathoz (is) csatlakoznak.

Ingress hálózat

Ahogy azt már láthattuk, az ingress network virtuálisan összeköti az összes swarm node-ot, még akkor is ha azok nem egy lokális hálózaton vannak. Annyi a megkötés, hogy a node-oknak el kell tudni egymást érni az alábbi portokon:

- 7946 TCP/UDP for container network discovery.
- 4789 UDP for the container ingress network.

Az ingress hálózat segítségével tudja megvalósítani a swarm a stateless load balancer funkciót alapértelmezetten round rubin módon.

Bármelyik swarm node-on (akár manager, akár worker) kiadhatjuk a **docker network ls** parancsot, ugyan azt kell látni.

```
# docker-machine ssh mg0 docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b71223c228cd        bridge              bridge              local
410e603aa9c2        docker_gwbridge     bridge              local
53cd23b3594b        host                host                local
idf7f51tp95i        ingress             overlay             swarm <<<<
4fb53b47cf55        none                null                local
```

Ha mi magunk nem hoztunk létre új overlay hálózatot kézzel, akkor csak az ingress hálózat lesz swarm scope-ú, az összes többi hálózat lokális lesz

- overlay hálózatból többet is csinálhatunk, de csak egynek lehet **ingress** a típusa. Az alapértelmezett ingress hálózaton felül a **--network** kapcsolóval adhatunk meg további overlay hálózatokat a **docker service create** -nek.
- Ha az ingress hálózatnak nem megfelel?ek az alapértelmezett beállításai, pl nem jó az IP tartomány, vagy a service-ek egymás között kommunikációját is titkosítani a akarjuk, akkor sajnos ki kell törölni és kézzel létre kell hozni **--ingress** kapcsolóval.
- Ez a docker által létrehozott eszköz, nem a linux kernel hozza létre
- Egyszerre csak egy **ingress** típusú hálózat lehet létezhet egy swarm-ban, ezért fontos, hogy el?tt kitöröljük. A neve bármi lehet, nem csak ingress. A swarm nem a neve alapján fogja megtalálni, hanem a típusa alapján. Mivel csak egy lehet, ezért ez egyértelm?.
- az egyetlen hálózat swarm scope-al

```
docker network create \
  --driver overlay \
  --ingress \
  --subnet=10.11.0.0/16 \
  --gateway=10.11.0.2 \
  -- --opt encrypted \  <<<  a service-ek kommunikációját titkosítja
  --opt com.docker.network.driver.mtu=1200 \
  my-ingress
```

docker_gwbridge ??

- A docker_gwbridge egy valódi fizikai hálózat, egy virtuális bridge, amit a Linux kernel-ben hoz létre a docker. Ezt láthatjuk a VM-en kiadott ifconfig paranccsal
- viszont itt a név nagyon is számít. Mivel ez egy fizikai hálózat, a docker pont ilyen néven fogja keresni, mikor a swarm -ot létrehozza. Ha nincs készíteni fog egy újat, ha már van, akkor ezt fogja felhasználni.
- Ha módosítani akarnánk, akkor ki kell törölni Linux parancsokkal (pl: ip), majd kézzel létre kell hozni az új beállításokkal a **docker network create** paranccsal.

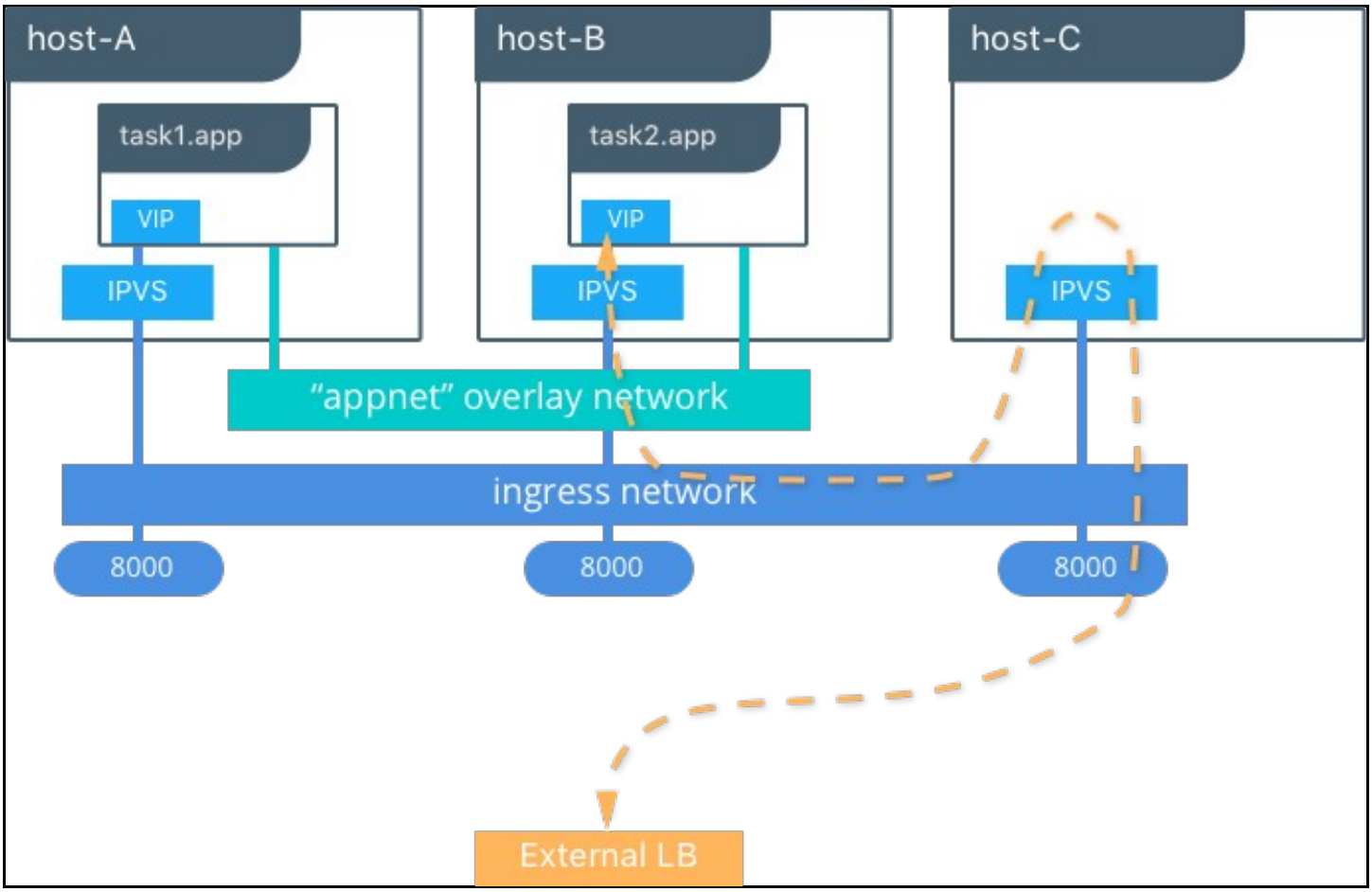
```
# docker-machine ssh mg0 docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b71223c228cd        bridge              bridge              local
```

```
410e603aa9c2        docker_gwbridge     bridge              local <<<<
53cd23b3594b        host                host                local
idf7f51tp95i        ingress             overlay             swarm <<<<
4fb53b47cf55        none                null                local
```

```
# docker-machine ssh mg0 ifconfig
...
docker_gwbridge Link encap:Ethernet  HWaddr 02:42:DF:B6:9C:AE
          inet addr:172.18.0.1  Bcast:172.18.255.255  Mask:255.255.0.0
          inet6 addr: fe80::42:dfff:feb6:9cae/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:19 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:1418 (1.3 KiB)
```

Load balancing

Routing mesh for stateless services



A dokker-ben van egy beépített "layer 4", szállítási rétegbeli, beépített load balnacer.

Ha a --publis kulcsszóval hozunk létre úgy szolgáltatást, akkor

- Ahogy azt már a bevezet?ben láthattuk, az úgynevezett routing mesh felel?s azért, hogy egy küls? portról elérjük a swarm szolgáltatást egy megadott porton.
- A **--publish** kulcsszóval kell regisztrálni a port mapping-et. A **published** a küls? még a **target** a bels? port.

Warning
A routing mesh-t logikus módon csak akkor tudjuk használni, ha minden host-on a service-nek csak 1 példánya fut. Különben ki kell kapcsolni a routing mesh-t (lásd lentebb)

```
$ docker service create \
  --name my-web \
  --publish published=8080,target=80 \
  --replicas 2 \
  nginx
```

Vagy meglév? szolgáltatáshoz meg lehet adni új port mapping-et a **--publish-add** kapcsolóval.

```
$ docker service update \
  --publish-add published=8080,target=80 \
  my-web
```

A **--publish** és **--publish-add** parancs alapértelmezetten csak tcp portot ad hozzá. A **protocol=udp** kapcsolóval lehet udp portot is hozzáadni

```
--publish published=53,target=53,protocol=udp
```

A routing mesh akkor van bekapcsolva egy adott service-re, ha a --publish kapcsolónak megadjuk ezt: **mode=ingress** Ez az alapértelmezett. Ha ki akarjuk kapcsolni a routing mesh-t, akkor ezt **host**-re kell állítani:

```
# docker service create --name dns-cache \
  --publish published=53,target=53,protocol=udp,mode=host \
  dns-cache
```

Auto scaling

A Docker swarm-ban nincs beépített auto scaling out of the box, nekünk kell implementálni, vagy használhatunk 3rd party eszközöket is. A Kubernetes-ben erre van egy remek beépített algoritmus, de a docker-swarm-ban is meg tudjuk ezt valósítani.

https://stackstorm.com/2017/06/22/autoscaling-swarm-aws-stackstorm/ https://github.com/sahajsoft/docker-swarm-service-autoscaler

https://docs.stackstorm.com/install/docker.html

Auto Scaling Docker Containers in Amazon ECS: https://www.codementor.io/jholub/amazon-ecs-auto-scale-docker-containers-6keydo24n ECS is an alternative to tools such as Kubernetes, Docker Swarm, or Mesos

https://github.com/gianarb/orbiter

https://prometheus.io/docs/prometheus/latest/installation/

https://docs.docker.com/config/thirdparty/prometheus/#use-prometheus

https://monitor.dockerflow.com/auto-scaling/