
# LINUX KONTÉNEREKRE ÉPÜLŐ VIRTUALIZÁCIÓS MEGOLDÁSOK VIZSGÁLATA

Készítette: Moldován István

Budapest, 2016. április

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

1117 Budapest, Magyar tudósok krt. 2.
Telefon: 463-4187, Fax: 463-3107
E-mail: admin@tmit.bme.hu • http://www.tmit.bme.hu

# BEVEZETÉS

A felhő alapú rendszerek alapját ma a virtuális gépek (Virtual Machines - VM) képezik. Segítségükkel képesek vagyunk rugalmas szolgáltatási környezetet kialakítani, amely alkalmazkodni tud dinamikus igényekkel szemben a VM-ek könnyű skálázhatósága és a menedzsment infrastruktúra terhelés megosztási képességei révén. A virtuális gépek kényelmes absztrakciót nyújtanak, de ennek az ára, hogy nagyobb erőforrás igényük van, mint egy tisztán hardveres („bare metal") megoldásnak.

Egy, a virtuális gépek rugalmasságát és a „bare metal" konfigurációk gyorsaságát ötvöző Linux konténer (container) technológia lehet a következő lépés az iparág evolúciójában. Ez egy gyorsan fejlődő, operációs rendszer szintű virtualizációt nyújtó technológia, amely az elmúlt években érte el népszerűsége eddigi csúcsát, nem kis részt a Docker konténerek megjelenésének köszönhetően.

A Docker konténerek megjelenése óta számos olyan projekt indult, amely a konténerekre alapozva megkísérel egy felhő alapú rendszerekhez hasonló rugalmasságú, de a jelenlegi virtualizációs megoldásoknál lényegesen erőforrás-hatékonyabb rendszert létrehozni. Bár a konténerek és virtuális gépek sokban hasonlítanak, a VM-eknél már megszokott menedzsment eszközök és infrastruktúra átültetése időbe telik.

# LINUX KONTÉNEREK

## MODERN DEFINÍCIÓ

Egy Linux konténer egy operációs rendszer szintű virtuális környezet, mellyel képesek vagyunk egy gépen több izolált Linux rendszert futtatni. Egyebek mellett saját fájlrendszere, hálózati és folyamat stack-je van, viszont a kernelen osztozik a host géppel és a többi futó konténerrel (szemben a tradicionális értelemben vett virtualizációval), aminek köszönhetően lényegesen kisebb többletmunkával, gyorsabban létrehozhatóak, mint egy virtuális gép.

Egy konténert a Linux kernel izolációt biztosító funkcióival lehet létrehozni, de nem kernel szintű fogalom. A használt kernelfunkciókhoz interfészt nyújtó LXC[1] az, ami a mai értelemben vett konténer fogalmát megalapozta (ezért is hívjuk őket LinuX Container-eknek).

A két legfontosabb használt kernelfunkció:

**namespace isolation (névtér izoláció):** segítségével minden folyamatnak saját homokozót adhatunk, egymás névtereibe nem képesek belátni, így a konténer szempontjából csak önmaga létezik. Külön kezelhetőek a következő névterek:

- IPC: folyamatok közötti kommunikáció
- Network: hálózati eszközök, stack-ek, portok
- Mount: csatolási pontok
- PID: folyamat ID-k
- User: felhasználó és csoport ID-k
- UTS: host és domain nevek

**cgroups (control groups)**[2]: hardveres erőforrások folyamatok közti kiosztását vezérli, képesek vagyunk korlátozni vele az egyes konténerek által használt CPU, memória, merevlemez, hálózati és egyéb erőforrásokat.

## KONTÉNEREK EVOLÚCIÓJA

Ebben a fejezetben az operációs rendszer szintű virtualizáció fejlődésének relevánsnak tartott mérföldkövei kerülnek ismertetésre.

### chroot

Az operációs rendszer szintű virtualizáció és így a konténerek ősének is a chroot[3 ] Unix parancs és chroot jail (börtön) tekinthető, amivel képesek vagyunk egy folyamat által látott gyökérkönyvtárat megváltoztatni. Ezzel a fájlrendszer izoláció részben megvalósul, de csakis az, és ráadásul egy root felhasználó könnyen kijátszhatja. Az újabb megoldások már nagyobb izolációra és biztonságra törekedtek.

### FreeBSD Jails

A FreeBSD börtönök[4] voltak a következő lépés. Egy ilyen börtön jellemzői:

- saját könyvtárfa, amiben lévő folyamat onnan nem képes kijutni

- saját hosztnév

- saját IP cím (sokszor alias egy meglévő hálózati interfészre)

- egy parancs, tehát egy futtatható állomány, amelyet a börtön gyökeréhez viszonyítva adunk meg

Ezek mellett saját felhasználói és root fiókjai vannak. A root fióknak nincs jogosultsága a börtön környezetén kívüli rendszerhez nyúlni. Ezekkel a börtönökkel képesek vagyunk például ugyanazon szoftver több verzióját is telepíteni egy gépre. A biztonságot pedig segíti, hogy amennyiben egy jogosulatlan felhasználó egy börtönben futó szolgáltatás biztonsági résén keresztül rendszergazda hozzáféréshez jutna, akkor sincs veszélynek kitéve a rendszer börtönön kívüli része. A FreeBSD börtönök csakis a FreeBSD operációs rendszerben elérhetőek.

## Solaris Zones

A Solaris operációs rendszerben a zónák "Sun15"[5] töltik be az izoláció szerepét. Egy Solaris rendszernek van egy globális és maximum 8192 „non-global" azaz nem globális zónája. A globális zóna maga az alap operációs rendszer példánya. Egy zóna működésében nagyon hasonlít a FreeBSD börtönökére. Megjelenésükkor újdonságnak a hardveres erőforrások zónák közötti szétosztása számított, továbbá hogy a ZFS[6] fájlrendszer nyújtotta snapshot (pillanatkép) és klónozási képességekkel gyorsan lehetett klónozni egy zónát. A Solaris Zones használata a Solaris operációs rendszer 10-es és 11-es verziójára korlátozódik, viszont maguk a zónák lehetnek a Solaris operációs rendszer különböző „brand"-jei.

## OpenVZ

Az OpenVZ[7] egy nyílt forrású konténer technológia. Egy foltozott Linux kernelen fut így viszonylag sok operációs rendszert támogat. Azon kívül, hogy a névterekkel megegyező szintű izolációt nyújt újdonság még az élő migráció és az azt lehetővé tevő ellenőrző pontok

(checkpoint). Tehát képesek vagyunk egy futó konténert lényegi leállás nélkül áthelyezni egy másik szerverre. A hardveres erőforrások kiosztására saját ütemezési algoritmusokat használ és zónákhoz képest precízebben feloszthatóak. Bár az OpenVZ lényeges előrelépés volt az operációs rendszer szintű virtualizációban, népszerűségét korlátozza, hogy a fejlesztők nem erőltették a hivatalos Linux kernelbe kerülését és így az implementációban nagyon hasonló LXC lett (legalábbis ideiglenesen) a de facto konténer sztenderd.

## LinuX Container (LXC)

Az LXC[1] 2014 februárja óta hivatalosan is a Linux kernel részét képezi. Implementációjának két legfontosabb része a korábban valamelyest már kifejtett névterek és cgroups, ahol a névterek felelnek az izolációért, a cgroups pedig a hardveres erőforrások megosztásáért. Ezen kívül ma már támogatja az AppArmor és SELinux profilokat és a Seccomp policy-ket. Kapcsolódó projekt az LXD[ 8 ], ami egy LXC konténerekre épülő daemon, mely egy REST API-n keresztül szolgáltatja annak funcióit.

## Docker

A mai egyik legnépszerűbb konténer technológia a Docker[9], ami eredetileg LXC-re épült, de 2014. Március 10. óta már a saját konténer motorja (execution driver), a libcontainer[10] az alapértelmezett. A Docker tehát több mint konténer motor; egyben infrastruktúrát (Docker Hub) és szabványos konténer formátumot (Docker Image) ad. Az eredeti célja hordozható konténer sablonok létrehozása, ez alapján konténerek futtatása és sok rá épülő projekt továbbra is elsősorban erre használja. Időközben magasabb szintű funkciók is beépítve elérhetővé váltak és így már a Docker eszköztár önmagában is kvalifikálhat menedzselt szolgáltatási környezet kialakítására.

## Rocket

A Docker beépített funkcióinak és eszközkészletének bővülésével (cluster menedzser, overlay hálózat) egyre inkább általános célú platform lett, mint szimplán egy konténerek futtatására és sablonjaik menedzselésére szolgáló háttéralkalmazás, mint azt az eredeti Docker manifesto

meghatározott. Ezen fejleményekre válaszul született 2014 decemberében a CoreOS által fejlesztett Rocket[11] vagy rkt, aminek célja egy modulárisabb és biztonságosabb konténer futtató környezet (runtime), bárminemű extrák nélkül. A későbbiekben kifejtett Kubernetes kompatibilis vele.

### Open Container Initiative

2015. Június 22-én Open Container Initiative [12] (OCI) alatt Docker, CoreOS és egy sor másik érdekcsoport összefogott és elkezdett közösen kidolgozni egy nyílt konténer sztenderdet.

## A KONTÉNEREK ELŐNYEI ÉS HÁTRÁNYAI

A konténerek virtuális gépekkel való összehasonlításakor gyakran felhozott előnyök és hátrányok a következők.

**Előnyök:**

- nagyobb hardver kihasználtság
- rugalmasabb erőforrás allokáció
- könnyűsúlyú
- gyorsabban lehet létrehozni

**Hátrányok:**

- csak Linux alapú rendszereket tudunk konténerbe helyezni, nem módosíthatjuk a konténer kernelét
- nehezebb biztonságossá tenni, mivel sokkal közelebb van a kernelhez
- a menedzselésükhöz használt technológiák még nem olyan kiforrottak

- szorosan összefüggő komponensek egy konténerben kezelése még körülményes tud lenni, mindent külön konténerbe helyezni pedig fölösleges overhead-del járhat

# KONTÉNER ALAPÚ MENEDZSMENT SZOFTVEREK ÖSSZEHASONLÍTÁSA

Egy szolgáltatás rendszer megvalósításához a kiválasztott technológiáknak az alábbi elvárásoknak minimum meg kell felelniük:

1. Konténerek létrehozása, hordozható formátumba lementése.

2. Hálózati funkciók konténerek összekötéséhez, a bennük futó szolgáltatások kivezetéséhez.

3. Dinamikus skálázhatóság.

A Docker révén nagyon sok kisebb-nagyobb projekt indult, aminek célja arra magasabb szintű funkciók építése, ilyenek az ütemezés, hálózatba kötés, skálázás, terhelés megosztás és egyéb menedzsment szolgáltatások.
A [13] hivatkozásnál található egy a közösség által karbantartott gondolati térkép (mind map) a Docker (és nemrég már az Open Container) ökoszisztémáról. A gondolati térkép „Scheduler/Orchestration/Management/Monitoring" ága alatt levő nyílt forrású projekteket áttekintve a legaktívabb projekt a Google által kezdeményezett Kubernetes[14], mely saját elmondásuk szerint: „egy nyílt forrású platform applikációs konténerek automatikus telepítésére, skálázására és üzemeltetésére egy számítógépfürtön belül.[15]. Közelebbről megvizsgálva a Kubernetes képes ellátni mind a három fenti feltételt az alábbiak szerint: Az 1. feltételt implicit teljesít minden a listán, mivel azt maga a Docker már tartalmazza. A 2. feltételhez a Kubernetes Service-ei és a kapcsolódó SDN (Software Defined Networking) képességei kielégítik. A 3. feltételre pedig a „horizontal pod autoscaling" feature-je ad egy lehetséges megoldást.

## KUBERNETES *ALAPFOGALMAK*

A Kubernetes[16] számos olyan fogalmat használ, ami más informatikai területeken mást jelent vagy egyáltalán nincs jelen. Ezek közül a dokumentum többi részének megértéséhez szükségeseket ismertetjük ebben a fejezetben.

**Node:** Egy node egy számítógép, amin fut a Kubernetes node ágens (Kubelet).

**Kubelet:** A Kubelet a node-on futó ágens, mely pod-ok és konténereik indításáért és egészségéért felel.

**Pod:** A pod-ok a Kubernetes hierarchia legkisebb logikai egységei, melyek egy vagy több konténerből állnak. Egy pod célja, hogy a szorosan összefüggő applikációk (értsd konténerek) egy közös kontextusban futhassanak. Ezt többek között úgy éri el, hogy az elemei osztoznak egyes névtereken, és ha szükséges a felcsatolt könyvtárakon is (de ettől még az erőforrás-kiosztást meg lehet adni konténerenként is). A pod ilyen implementációja hasznos lehet, ha olyan segítő alkalmazásokat akarunk futtatni valami mellet, mint például egy naplózó, frissítéskezelő, helyi gyorsítótár menedszer és hasonlók.

**Label and selector:** A label-ek (címkék) kulcs/érték párok, amelyek bármikor szabadon hozzárendelhetőek a logikai egységekhez. Céljuk a felhasználó szempontjából hasznos információk tárolása nem előre meghatározott struktúra szerint, akár organizációs célokból, akár selector-okkal (kiválasztó vagy szelektor) való használathoz. Implicit nem bírnak jelentéssel a rendszer szempontjából, helyette a szelektorok által nyernek értelmet, melyekkel meghatározhatjuk, hogy milyen címkékkel rendelkező egységekre vonatkozzon egy utasítás vagy beállítás.

**Replication controller:** Egy replikációs kontroller felelőssége, hogy adott számú pod replikát tartson életben minden időpillanatban. A monitorozott pod-ok halmazát címkékkel adjuk meg, amennyiben több vagy kevesebb a feltételeknek megfelelő pod fut, akkor az adott számra hozza a többlet törlésével vagy újak létrehozásával. Azt, hogy az újonnan létrehozott pod milyen legyen a kontrollernek megadott pod sablon (template) határozza meg. Fontos, hogy a már futó konténer nem kell, hogy megegyezzen a sablon pod-dal, azt csak az új pod-ok létrehozásához használja.

**Service:** Mivel a pod-ok természetüknél fogva rövid életűek és mindegyiknek egyedi IP címe van, így felvetülhet az igény, hogy egy változó számú pod által támogatott szolgáltatást egy cím alatt tegyük perzisztensen elérhetővé. Ennek előnye, hogy ha egy service mögött álló összes

pod-ot kicseréljük (frissítés, skálázás vagy más okból), akkor is ugyanúgy érjük el a szolgáltatást. Ezt az absztrakciós funkciót látják el a service-ek.

**Endpoint:** Egy service végpontjai azok a pod-ok, amelyek kiszolgálják azt, pontosabban azok, amelyek címkéi kielégítik a service szelektorát.

**Volume:** A konténerekben lévő fájlok elvesznek a konténer megszűnésekor. Ha azt akarjuk, hogy egyes fájlok mégis megmaradjanak azokat a konténerek fájlrendszerébe csatolható volume-okba helyezésükkel tehetjük meg. Természetesen ez visszafelé is működik, vagy ha több konténer között meg akarunk osztani egy közös mappát.

## HÁLÓZATI MODELL

Ez a fejezet a Kubernetes dokumentáció hálózati modellt[18] és service-eket[19] leíró részei alapján készült.

A Kubernetes hálózati modell a következő problémákat próbálja megoldani:

1. konténer-konténer kommunikáció
2. pod-pod kommunikáció
3. pod-service kommunikáció
4. külső-belső kommunikáció

Az 1. pontot úgy oldja meg, hogy az egymással szorosan kommunikáló konténereket közös pod-ba és ez által közös hálózati névtérbe helyezi, így azok szimplán a localhost alatt képesek elérni egymás portjait. Ez persze magával vonja, hogy egy pod-on belül lévő konténereknek osztoznia kell a portokon.

A 2. ponthoz elvárja, hogy minden pod-nak egyedi IP címe legyen, a pontos módját nem határozza meg, de elsősorban az SDN megoldásokat támogatja. A végeredmény, hogy logikailag minden pod (és node) ugyanazon a hálózaton elérhető közvetlen címzéssel.

A 3. és 4. pont hátterében az úgynevezett kube-proxy áll. Ennek pontos működésének kifejtése előtt érdemes tisztázni, hogy egy service megvalósítása tulajdonképpen nem más, mint egy virtuális IP cím, egy regisztráció a kube-proxy applikációknál és egy sor tűzfal bejegyzés a node-okon.

Egy service létrehozásakor annak megadunk egy virtuális IP címet és portot. Minden node-on fut egy kube-proxy applikáció, ami új service felvétele esetén nyit egy tetszőlegeset a node fizikai portjai közül, amin hallgatózni fog. Ezek után beállítja a node tűzfalát (iptables), hogy a service virtuális címére és portjára érkező csomagokat irányítsa át erre a fizikai portra. Végül, az így megkapott csomagokat továbbküldi az adott service-t kiszolgáló végpont pod-ok valamelyike felé (alapértelmezésben round-robin szerint választva).

### Overlay hálózat (flannel)

A Kubernetes több SDN technológiát is támogat, ezek közül az egyik leghasználtabb a flannel[20] által nyújtott overlay hálózat. A flannel overlay hálózat segítségével képesek vagyunk a cluster összes pod-ját úgy kezelni, mintha azok egy nagy alhálózaton lennének. Ehhez az kell, hogy a node-okon futó flanneld daemon-ok tisztában vannak a logikai hálózat felépítésével és az erre vonatkozó információkat szinkronban tartják egymással egy elosztott kucs-érték táron keresztül (ez esetben etcd[14]). Minden node-nak van saját flannel0 interfésze és ahhoz tartozó egyedi IP cím alhálózata, és ebből az alhálózatból kap újabb alhálózatot a docker0 virtuális bridge, aminek a tartományából végül a pod-ok/konténerek fognak címeket kapni. Így egy pod virtuális címéből következik, hogy melyik node-on van.

Amikor egy pod küldeni akar egy csomagot egy másik node-on lévő pod-nak, az a csomag először a flanneld daemon-nál fog kikötni, amely tudván a hálózat felépítését, becsomagolja az egészet még egy csomagba, amit a cél pod-ot futtató node-nak címez, forrásnak pedig a saját node-ját állítja. Ezek után az új csomag a megszokott módon eljut a cél node-hoz. A cél node a csomagot először a flanneld daemon-jának adja, ami kicsomagolja az eredeti csomagot és azt továbbküldi a saját flannel0 interfészén, amire csatlakozó docker0 bridge-en keresztül végre eljut a csomag a megfelelő pod-hoz.

# SZOLGÁLTATÁSI KÖRNYEZET KIALAKÍTÁSA

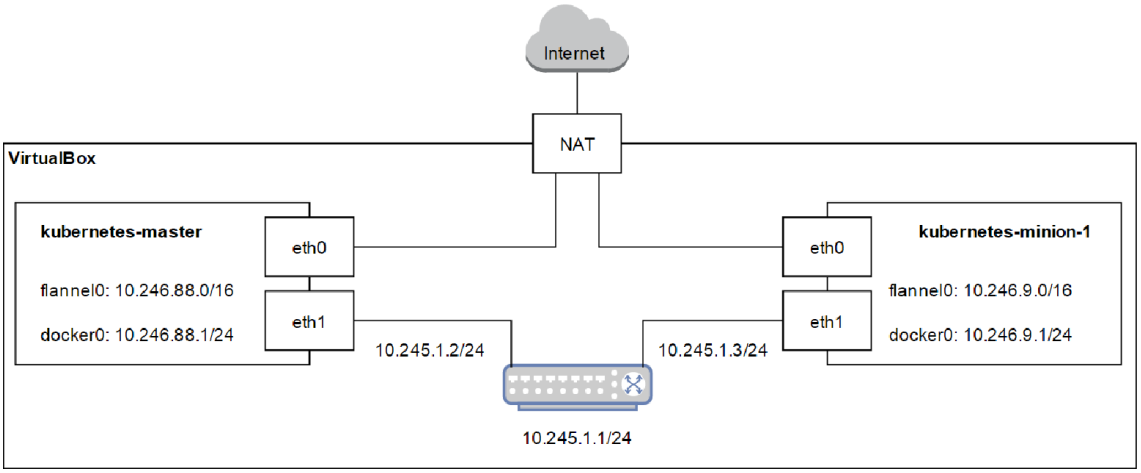
## KUBERNETES KÖRNYEZET KIALAKÍTÁSA

Az előző fejezetben leírtaknak megfelelően a Kubernetes eszközkészlete adódott a legalkalmasabbnak a szolgáltatási környezet kialakítására. A környezet könnyű hordozhatósága és reprodukálhatósága végett virtuális környezetben futatott megoldás lett kidolgozva. Erre a célra a VirtualBox[21] lett kiválasztva, mivel ingyenes és mind a Docker Machine, mind a Kubernetes dokumentációjában található a használatához útmutató és példák.

A környezet kialakításához használt szoftverek: Microsoft Windows 8.1 operációs rendszer, VirtualBox + Oracle VM VirtualBox Extension Pack v5.0.10, Vagrant v1.7.4, Kubernetes v1.1.1. A virtuális gépek létrehozása a Kubernetes dokumentáció[23] alapú telepítéshez kapcsolódó része alapján készült.

Ez alapesetben az 1. ábrán látható 2 virtuális gépet hozza létre (a használt címekkel felcímkézve). Tehát a telepítése egy Kubernetes mestert és egy node-ot hoz létre, amelyek egy virtuális bridge-en keresztül összekötve az egyik interfészükön a másikon pedig ki vannak NAT-

olva az internet felé. Az alapértelmezett operációs rendszer egy „vagrant box”-ként megadott Fedora 20 „Heisenbug”. Azonban az így létrehozott gépek felcsatolási problémákkal küzdenek




**1. ábra: Virtuális Kubernetes cluster**

és ennek következtében nem képes befejezni a telepítést. A megoldásnak a Fedora egy újabb verziójának használata bizonyult. Bár a rendszer így már felállt, de problémát okozott még a *minion-1* node alacsony memóriája, aminek következtében nagyon korlátozott volt a rajta létrehozható pod-ok száma. Ezt orvosolandó megdupláztuk a hozzárendelt memória méretét. A módosításokat is figyelembe véve a következő parancsokkal indult el a Kubernetes cluster:

```
cd <kubernetes repository klónja>
export KUBERNETES_BOX_NAME=box-cutter/fedora22
export KUBERNETES_PROVIDER=vagrant
export KUBERNETES_MINION_MEMORY=2048
./cluster/kube-up.sh
```

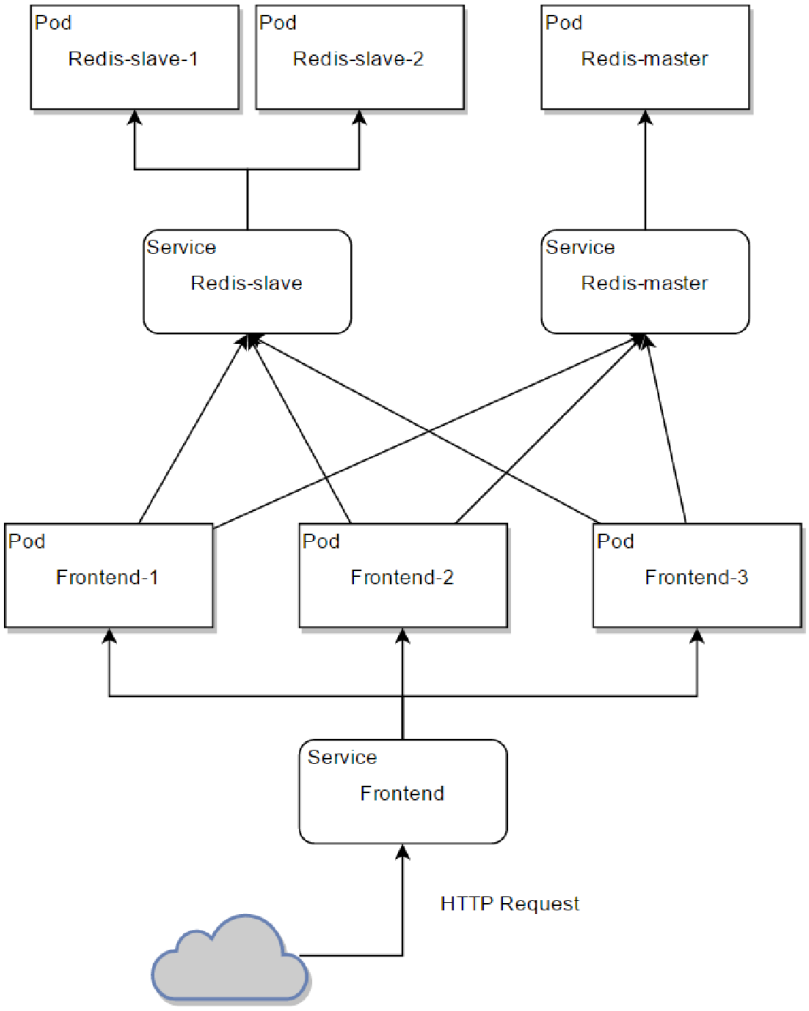
## KUBERNETES MULTI-KONTÉNER ALKALMAZÁS

A Kubernetes környezet helyes működésének verifikálásához és egyben a hálózatba kötött multi-konténer applikáció tesztként a dokumentáció PHP Server[24] példáját használtuk.
Az applikáció 3 fajta pod/konténerből áll:

- 1 db redis master
- 2 db redis slave
- 3 db web frontend

Mindegyik pod replikációs kontrollerek segítségével jön létre, majd service-ek segítségével lesznek elérhetőek kifelé és egymásnak. A Redis[25] egy népszerű (különösképp konténerben) memóriában tárolt NoSQL adatbázis. Ahogyan azt látni fogjuk a Redis master-slave felállást is támogat, ilyenkor a master-ben elhelyezett adatok úgymond replikálódnak a slave-ekben.

A frontend pod egy PHP szerver, amely olvasáskor a slave-ekkel, íráskor a master-rel kommunikál. Egy egyszerű online vendégkönyvet szolgáltat. Egy http kérés potenciális útvonala a 2. ábrán látható.



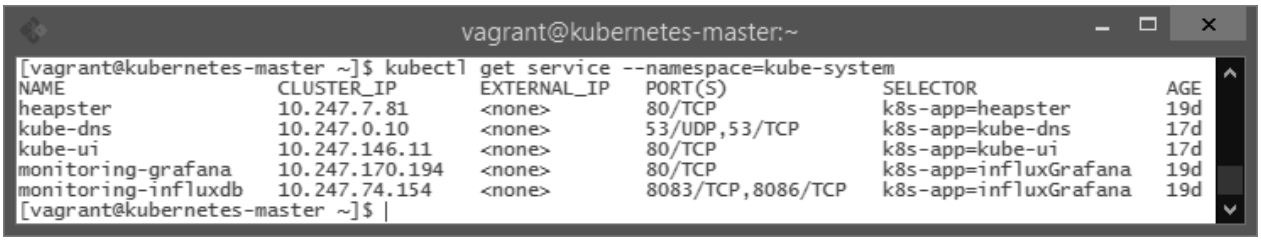

**2. ábra Egy http kérés lehetséges útvonala**

## *Előfeltételek*

Nulladik lépésként leellenőriztük, hogy az alapvető Kubernetes szolgáltatások futnak-e. Ezt a *cluster-info* paranccsal lehet lekérdezni. A futónak jelölt szolgáltatások és azok feladatai a következőek: **Heapster:** a konténer monitorozó és teljesítmény elemző, **KubeDNS:** a cluster saját DNS szervere, csakis a service-eknek van, **KubeUI:** a felhasználói felületért felel, **Grafana:**

a Heapster ágens által gyűjtött metrikák megjelenítésére szolgál, **InfluxDB:** a Heapster ágens által gyűjtött metrikák lementésére szolgál.

Azt, hogy milyen virtuális IP címeket kaptak ezek a szolgáltatások a kubectl get service –namespace=kube-system paranccsal lehet megnézni, kimenete a 3. ábrán látható.




Az elvártnak megfelelően az összes cím az overlay hálózat (*10.246.9.0/16*) tartományába esik. A webes felületet nyújtó szolgáltatások (Heapster, KubeUI, Grafana) mind a http (80) porton kommunikálnak, a KubeDNS a DNS szolgáltatási portot (53) használja, az InfluxDB pedig az alapértelmezett portját (8086).

*APPLIKÁCIÓ ELINDÍTÁSA*

## Redis master

Elsőként a Redis master replikációs kontrollerének konfigurációs fájlját definiáltuk. A fájlt úgy

**2. ábra: Kubernetes rendszerszolgáltatások virtuális címei**

paramétereztük, hogy egyetlen replikát hozzon létre, sablonként a DockerHub-on elérhető hivatalos Redis képfájlt állítottuk be. Azért érdemes még egyetlen példány esetén is replikációs kontrollerrel indítani, mivel így ha a pod valamilyen oknál fogva leáll, a kontroller automatikusan indít egyet helyette, továbbá a skálázást is leegyszerűsíti.

A Redis master a 6379-es porton kommunikál, így azt meg lett adva a pod sablon részeként, hogy engedélyezve legyen, valamint, hogy mennyi erőforrást igényelhet ez a pod, így a Kubernetes ütemező jobban tud dönteni, hogy melyik node-on indítsa/indíthatja el.

Ezek után a kubectl create –f parancsnak paraméterül adva ezt a fájlt a rendszer létrehozott egy redis-master-gs8js nevű pod-ot.

A pod elindulása után, egy a címkéit szelektorként kapó service-t indítottunk el egy hasonlóan definiált fájl alapján, ügyelve arra, hogy a service leképezze a pod 6379-es portját a saját virtuális IP címéhez tartozó portjára.
A kubectl describe paranccsal ellenőriztük, hogy a service a 10.247.170.1:6379 virtuális IP címet és portot kapta, végpontja pedig a 10.246.9.5:6379 című pod.

### Redis slave

A Redis slave kontrollernek kettő replika lett beállítva. A master-hez hasonlóan itt is meg lettek adva az elvárt erőforrások, a használt port (szintén 6379) és a kiválasztáshoz hasznos címkék. Ezeken felül még az is, hogy a service discovery módja a DNS alapú legyen (a környezeti változós módszer helyett). Így képes lesz megtalálni a mastert-t a service neve alapján. Természetesen ehhez az is kell, hogy maga a slave úgy legyen bekonfigurálva, hogy azon a néven keresse a master-t; ezt a slave sablona már tartalmazza. A kontroller után a Redis slave service-t is elindításra került, az annak kiosztott cím és port *10.247.214.34:6379.*

### PHP Szerverek

A PHP szerver avagy frontend-nek 3 replika lett beállítva. A frontend is a cluster DNS szerverét használja a service nevek feloldásra, a sablonja a slave-hez hasonlóan már tartalmazza a többi service várható nevét. Az erőforrás-használatot a Redis pod-okhoz hasonlóan itt is megadtuk. Mind a pod-ot, mind a service-t úgy állítottuk be, hogy a http-nél megszokott 80-as porton kommunikáljon. A slave service virtuális címe és portja *10.247.79.217:80* lett.

## *MŰKÖDÉS ELLENŐRZÉSE*

### Pod IP cím kiosztás

Egy pod-nak kiosztott IP cím a flannel overlay hálózat tartományából kell, hogy kikerüljön és a pod hálózati interfésze ugyanezt a címet fogja használni. A *frontend*-e11xc pod-ot véve példának a *kubectl describe* paranccsal ellenőriztük a kiosztott IP címet, majd a pod-ba lépve az *ip address* parancs által listázott *eth0* hálózati interfész címét és a kettőt megegyezett. Az elvárt működésnek megfelelően, a *frontend*-e11xc pod az azt futtató *minion-1* node overlay

hálózatának tartományából (*10.246.9.0*) kapott címet és a szerint állította be az *eth0* interfészét *10.246.9.9/24*-re.

## Tűzfal

A service-ek elindulása után az elvárt működés, hogy minden service-hez létrejött tűzfal bejegyzés minden node-on, ahol a kube-proxy fut. Ezt a *minion-1* node-ba lépve, az *iptables –t nat –L* paranccsal kiolvasva a tűzfal beállításait ellenőrizhető. Példaképpen a *frontend* service *10.247.79.217* virtuális címének http portjára érkező kérések a *minion-1* node 39834-es portjára lettek átirányítva.

## Domain Name Service

Lekérdezve az egyik frontend pod környezeti változóit azt találtuk, hogy annak ellenére, hogy DNS feloldásra lett állítva be lettek állítva a kapcsolódó pok-ok elérhetőségei változókként is. Hogy ezt nem használja mi sem bizonyítja jobban, mint hogy sikertelen volt a névfeloldás annak ellenére, hogy az előfeltételek megnézésekor a **KubeDNS** running státuszt írt és service-ként is jelen volt. Ennek okának a DNS szerver replikációs kontroller beállítása bizonyult. A kívánt replikák (desired replicas) száma 0-ra volt állítva. A skálázási parancs kiadása után rövidesen elindult az általa használt 4 konténerből álló pod, mely egyúttal megjelent a service végpontjai közt. Ezek után már gond nélkül fel tudta oldani a *frontend* service virtuális címét az egyik redis slave-ből.

## Webfelület

Kkiolvasva a node tűzfal utasításaiból, hogy melyik portjára lett leképezve a frontend service a host gépről is elérhető a vendégkönyv szolgáltatást és így tesztelhető ez egy böngészőből is. A frontend-ek naplóiba nézve az látható, hogy a frontend-yiesi nevű pod-ra lett továbbítva ez a konkrét beírás. A redis master pod naplójában pedig jelen volt a változás lementésének bejegyzése, amely időbélyegei szerint a mentés 101 ezredmásodpercbe telt és ugyanabban a másodpercben indult, mint amelyben a frontend naplója jelezte a beérkezett üzenetet (a frontend naplója sajnos csak másodperc pontosságú).

A terhelés megosztás teszteléséhez egy terminálból aszinkron (az előző kérés befejeződésének megvárása nélkül) indítottunk 999 darab http kérést a frontend service felé. Ennek hatására azt tapasztaltuk, hogy miután a kérések lefutottak, a frontend pod-ok naplóiban mindegyikhez pontosan 333 darab http kérés jutott el, ami azt jelenti, hogy a terhelés megosztás a vártaknak megfelelően működik.

# Összefoglalás

A tanulmányban áttekintettük a Linux konténerek megvalósításait, valamint a Docker konténerek menedzselésére kiválasztott Kubernetes-t, mivel a már megvalósított funkciói a legjobban támogatják egy akár komplexebb szolgáltatás létesítését.

Megterveztük a Kubernetes környezet felépítését és hálózatba kötését, majd megvalósítottuk a rendszert. A Kubernetes tesztelésére egy több konténerben futó PHP-t szolgáltató web alkalmazást választottunk, amelyben a frontend konténerekhez egy terhelésmegosztón jutnak el a kérések. A kiváltott változások elosztott NoSQL (Redis) tárhelyet futtató konténerekben mentődnek le. Ennek helyes működését ellenőriztük és dokumentáltuk.

Értékelésünk alapján a Kubernetes jól átgondolt absztrakciói megkönnyítik egy szolgáltatás logikai tervezését, de az absztrakciók mögött lévő rendszer tervezése, telepítése és karbantartása viszonylag nagy hozzáértést igényel.

# Irodalomjegyzék


[1] Canonical Ltd. (2015, December) Linux Containers LXC. [Online]. https://linuxcontainers.org/lxc/

[2] Paul Menage, Paul Jackson, and Christoph Lameter. (2015, December) Kernel cgroups. [Online]. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[3] The FreeBSD Project. (2015, December) FreeBSD chroot. [Online]. https://www.freebsd.org/cgi/man.cgi?query=chroot&sektion=2&n=1

[4] Matteo Riondato. (2015, December) FreeBSD jails documentation. [Online]. http://www.freebsd.org/doc/handbook/jails.html

[5] Sun Microsystems. (2015, Október) Solaris zones documentation. [Online]. https://docs.oracle.com/cd/E53394_01/html/E54762/zones.intro-1.html

[6] Oracle Corporation. (2015, December) ZFS filesystem documentation. [Online]. http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/

[7] (2015, December) OpenVZ homepage. [Online]. http://openvz.org/

[8] Canonical Ltd. (2015, December) Linux Containers LXD. [Online]. https://linuxcontainers.org/lxd/

[9] Docker Inc. (2015, December) Docker homepage. [Online]. https://www.docker.com/

[10] Solomon Hykes. (2014, Március) Docker Blog libcontainer announcement. [Online]. https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/

[11] Alex Polvi. (2014, December) CoreOS Rocket announcement. [Online].

https://coreos.com/blog/rocket/

[12] Linux Foundation. (2015, December) Open Container Initiative. [Online]. https://www.opencontainers.org/

[13] Krishnan Subramanian. (2015, December) Open container ecosystem formerly docker ecosystem mind map. [Online]. https://www.mindmeister.com/389671722/open-container-ecosystem-formerly-docker-ecosystem

[14] (2015, December) etcd GitHub repository. [Online]. https://github.com/coreos/etcd

[15] Google Inc. (2015, December) Kubernetes homepage. [Online]. http://kubernetes.io/

[16] Google Inc. (2015, December) Kubernetes what is k8s introduction. [Online]. http://kubernetes.io/v1.1/docs/whatisk8s.html

[17] Google Inc. (2015, December) Kubernetes v1.1 documentation. [Online]. http://kubernetes.io/v1.1/

[18] Google Inc. (2015, December) Kubernetes networking. [Online]. http://kubernetes.io/v1.1/docs/admin/networking

[19] Google Inc. (2015, December) Kubernetes services. [Online]. http://kubernetes.io/v1.1/docs/user-guide/services.html

[20] (2015, December) flannel GitHub repository. [Online]. https://github.com/coreos/flannel

[21] Oracle Corporation. (2015, December) VirtualBox homepage. [Online]. https://www.virtualbox.org/

[22] Google Inc. (2015, December) Kubernetes vagrant based installation guide. [Online]. http://kubernetes.io/v1.1/docs/getting-started-guides/vagrant

[23] Mitchell Hashimoto. (2015, December) Vagrant homepage. [Online]. https://www.vagrantup.com/

[24] Google Inc. (2015, December) Kubernetes guestbook example. [Online]. http://kubernetes.io/v1.1/examples/guestbook/

[25] (2015, December) Redis homepage. [Online]. http://redis.io/