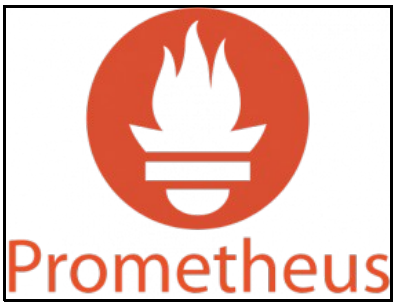
Metrics and Monitoring in swarm

<< Back to Docker main



Contents

Mi az a metrika

A metrika egy fajta speciális loggolás, amit a metrikát szolgáltató rendszer nem egy log fájlba ír, hanem biztosít egy HTTP API-t, amin keresztül a metrikát feldolgozó szolgáltatás le tudja azt periodikusan kérdezni. A metrika név-érték pár, aminek a jelentése bármi lehet ami id?ben változik. A metrika listában minden egyes metrika jelentését a metrikát szolgáltató rendszernek kell definiálnia. Pl egy node-on lév? elérhet? maradék memóriát jelképezheti a következ? metrika:

```
node_memory_MemAvailable
```

A metrikát begy?jt? rendszer ezt periodikusan lekérdezi a metrikát szolgáltató rendszerbt?l és a lekérdezés id?pontjával együtt beírja az adatbázisába, így minden metrika úgynevezett id?sort alkot a metrika adatbázisban. Pl:

```
 ID?PONT         METRIKA NEVE       ÉRTÉK
  10:12   node_memory_MemAvailable   18
  10:13   node_memory_MemAvailable   17
  10:14   node_memory_MemAvailable   11
```

A metrika értéke lehet szám, string vagy logikai érték is.

A metrikákat minden rendszerben, szervezetben vagy swarm cluster-ben egy központi egység kérdezi le és gy?jti össze a saját adatbázisában. A metrikát összegy?jt? rendszerben aztán lekérdezéseket írhatunk fel a metrikákra, aminek az eredménye alapján aztán automatizált folyamatokat

indíthatunk be, mint pl a swarm cluster méretének megváltoztatása, alert küldése sms-ben, emailben. Tehát a metrikák a hagyományos log fájlokkal ellentétben a baj valós idej? detektálására, vagy sokkal inkább a baj elkerülésére szolgálnak. Ebb?l kifolyólag a logokkal ellentétben a metrikák élettartalma nagyon rövid, tipikusan olyan lekérdezéseket szoktunk írni, ahol a lekérdezett metrikák kora nem több mint 10 perc, de inkább pár perc, hiszen itt mindig valaminek az id?ben történ? változására vagyunk kíváncsiak. Így nem olyan kritikus a metrikák elvesztése mint a hagyományos log fájloké, amik tipikusan offline elemzésre szolgálnak, ha a baj már bekövetkezett.

A metrikákat úgynevezett **time-series** (id?sor) adatbázisban kell letárolni (**TSDB**), vagyis egy adott metrikához nyilván van tartva minden lekérdezéshez az akkor kapott érték. Ez a speciális struktúra ugyan letárolható lenne hagyományos adatbázis kezel?kben is, de nagyon nem lenne hatékony a bennük való keresés. Léteznek direkt erre a speciális adatmodellre készült adatbáziskezel?k, amik rettent? hatékonyan tudnak keresni a time-series adatokban. Egy adott metrika tárolását egy listaként lehet elképzelni, ahol a lista elemek az id?bélyegekkel vannak indexelve, és a listaelem tárolja az adott id?pillanathoz (amikor a lekérdezés történt) a metrika értékét. A f?bb **time-series** adatbázis kezel?k:

- InfluxDB:
- Prometheus
- Graphite
- OpenTSDB
- KairosDB

...

Fontos, hogy a time-series db számára a metrika csak egy név-érték pár (egy string amihez tartozik egy érték), tehát a TSDB nem értelmezi a kapott metrikát, a lekérdezéseket úgy kell megírni, hogy legyen értelme a háttérrendszerre vonatkozóan. Tehát mikor megtervezzük a lekérdezéseinket a TSDB-ben, akkor els?ként a metrikát szolgáltató rendszer specifikációját kell megnézni, hogy az milyen metrikákat szolgáltat magáról, és melyiknek pontosan mi a jelentése.

A **time-series** adatbáziskezel? (**TSDB**) folyamatosan gy?jti a különböz? komponensek metrikáit, és minden egyes begy?jtés után ki fogja értékelni a különböz? metrikákra felírt lekérdezéseket, amik általában abból állnak, hogy egy time-series lekérdezés eredményét összeveti egy értékkel vagy logika változóval, és a végeredmény vagy igaz vagy hamis. Ha a végeredmény igaz, akkor a time-series adatbáziskezel? riasztást fog generálni (beindít egy automatizált folyamatot), ha a kiértékelés hamis, akkor meg nem fog semmit csinálni. A riasztás hatására küldhetünk emial-t, végrehajthatunk egy bash script-et... stb.

TSDB használatával lehet a swarm cluster egészségét automatizált módon monitorozni vagy akár dinamikusan változtatni a worker node-ok mennyiségét. Pl felírhatunk különböz? szabályokat a node-ok leterheltségére. Ha a node-ok valamilyen metrika mentén túlságosan leterheltek, akkor újabb node-okat állítunk automatikusan üzembe, ha meg a terhelés túl alacsony ugyan ezen metrika alapján, akkor meg bizonyos node-okat megszüntetünk.

Mi innent?l kezdve csak a Prometheus-ra fogunk fókuszálni.

Prometheus felépítése

Id?sorok (time series)

A TSDB-ben id?sorokat tárol el az adatbázis. Miden végpont minden metrikájához tárol egy id?sort. Lekérdezéskor elmenti a metrika új értékét a lekérdezési id?pontjával együtt Pl az alábbi példában 1 másodpercenként kérdezi le a **roxy_http_request_total** értékét:

```
proxy_http_request_total
1. 2018.08.19 13:12:01  -  23
2. 2018.08.19 13:12:02  -  23
3. 2018.08.19 13:12:03  -  24
4. 2018.08.19 13:12:04  -  25
5. 2018.08.19 13:12:05  -  27
```

Fontos, hogy az id?sorokat a Prometheus végpontonként tárolja. Ha pl 20 swarm worker-em van, és mind a 20 "beszámol" a **proxy_http_request_total** nev? metrikáról, akkor ezek nem lesznek összemosva egy közös id?sorba, hanem húsz különböz? id?sort (time series) fognak képezni.

```
NODE1: roxy_http_request_total
1. 2018.08.19 13:12:01  -  23
2. 2018.08.19 13:12:02  -  23
3. 2018.08.19 13:12:03  -  24

NODE2: roxy_http_request_total
1. 2018.08.19 13:12:01  -  6
2. 2018.08.19 13:12:02  -  7
3. 2018.08.19 13:12:03  -  10
```

A Prometheus adatbázisban a lekérdezéseket a Prometheus saját nyelvén, PromQL-ben kell megírni.

Címkék, plusz dimenziók

A Prometheus szabványú metrikában további dimenziókat lehet bevezetni minden metrikához úgynevezett metrika címkékkel, amiket a metrikát szolgáltató rendszer (pl egy apache) hozzáf?z a metrika nevéhez. A címkék tehát tovább specializálnak egy metrikát, pl egy http proxy a *proxy_http_request_total* nev? metrikával mondhatja meg, hogy a lekérdezési id?pontjáig hány kérés érkezett a proxy-hoz. De ezt tovább specializálhatja címkék bevezetésével. Az alábbi példában a **method** és a **status** címéket használta a proxy a *proxy_http_request_total* metrika finomításához. Az alábbi példában tehát a metrika értéke nem az összes request-re vonatkozik, csak azokra amiket GET-el kértek le, és amiknek 200-as volt a státusza.

```
proxy_http_request_total{method="GET", status="200"} 13
```

A valóságban ez úgy nézne ki a metrikát szolgáltató rendszer által gyárott metrika listában, hogy sorba jönne az összes variáció egymás után, pl:

```
...
proxy_http_request_total{method="GET", status="200"} 13
proxy_http_request_total{method="GET", status="500"} 12
proxy_http_request_total{method="POST", status="200"} 30
proxy_http_request_total{method="POST", status="300"} 20
...
```

Fontos, hogy a címkének is a metrikát szolgáltató rendszer ad jelentést, a time-series adatbázis kezel? számra (a mi esetünkben Prometheus) a metrika és a benne lév? címkék is csak név érték párok. Azonban a címke és annak az értéke is részei a metrika nevének. Tehát a metrikát az összes címéjével együtt felfoghatjuk egy string-nek, aminek van egy értéke. A time-series adatbázisokban a címkék segítségével nagyon trükkös lekérdezéseket lehet felírni, amiket a time-series adatbázis nagyon hatékonyan meg tud keresni.

Bár err?l már beszéltünk, itt most külön kiemelem, hogy mit nevezünk a Prometheus-ban time series-nek. Minden egyes metrika minden címke el?fordulási fajtájával külön id?sort képez.

Beépített címkék és metrikák

https://prometheus.io/docs/concepts/jobs_instances/
Azt már említettük, hogy a Prometheus nem mossa össze a különböz? végpontoktól begy?jtött mintákat, külön-külön id?sorban menti el ?ket. De hogy éri ezt el. Úgy, hogy automatikusan két beépített címkét illeszt minden egyes begy?jtött metrikához:

- **job**: A Prometheus config-ban konfigurált job neve, ami azonos konténereket fog össze. Pl cAdivsior vagy node-exporter konténerek.
- **instance**: Egy job-on belül a példányneve, amit IP:port párossal ír le a Prometheus ha csak nem küld mást magáról a végpont.

Pl az alábbi metrikát a cAdvisor egyik konténere küldi. A Prometheus config-ban a job neve "cadivisor" (lást Prometheus konfig cím? fejezetet)

```
container_cpu_system_seconds_total{id="/",instance="10.0.0.12:8080",job="cadvisor"}
```

Ezen felül még 4 beépített id?sort is automatikusan létrehoz a Prometheus minden egyes végponthoz, tehát minden instance-hoz:

```
up{job="<job-name>", instance="<instance-id>"}
```

1 if the instance is healthy, i.e. reachable, or 0 if the scrape failed.

```
scrape_duration_seconds{job="<job-name>", instance="<instance-id>"}
```

Lekérdezés ideje

```
scrape_samples_post_metric_relabeling{job="<job-name>", instance="<instance-id>"}
```

the number of samples remaining after metric relabeling was applied.

```
scrape_samples_scraped{job="<job-name>", instance="<instance-id>"}
```

the number of samples the target exposed.

Honnan jön a metrika

Metrikát magáról nagyon sok rendszer tud szolgáltatni, pl a *Traefik* reverse proxy, vagy ahogy azt majd látni fogjuk, akár a docker daemon is képes metrikákat szolgáltatni saját magáról. Ezen felül nagyon sokféle metrika exporter is elérhet?, amik OS szinten is képesek metrikákat szolgáltatni. Általában a metrika lekérdezésére egy http interfészt biztosít a metrikát adó rendszer, amit a /**metrics** URL-en lehet elérni. A http interfészek esetében PULL metrika begy?jtésr?l beszélünk, vagyis a Prometheus (vagy bármelyik másik TSDB) a konfigurációja alapján periodikusan (pár másodpercenként) meghívja a megfelel? URL-t, ahol visszakapja az aktuális metrika listát (név-érték párokat), amit beír az adatbázisba. Létezik PUSH alapú metrika gy?jtés is.

Általában nem várjuk el a swarm service-ekt?l hogy magukról metrikákat szolgáltassanak, sokkal inkább metrika gy?jt? rendszereket szokás beépíteni a cluster-be. A két legelterjedtebb metrika gy?jt?:

- **cAdvisor**: Ez egy adott Docker démonról képes metrikákat begy?jteni (a Docker démonon futó konténerekr?l szolgáltat információt). Az összes konténer keretrendszert támogatja, nem csak a Docker-t.
- **node-exporter**: OS szinten gy?jt metrikákat (pl CPU és Memória használat, szabad disk hely, stb..). Ez nem Docker (vagy konténer) specifikus komponens, akkor is használható ha a rendszerünknek semmi köze a konténeres világhoz.

Készíthetünk olyan alkalmazást ami metrikákat szolgáltat magáról. Ehhez a Prometheus 4 nyelven is ad API-t aminek a segítségével nagyon egyszer? Prometheus szabványú metrikákat szolgáltatni: **Go, Java, Python, Ruby**.
https://prometheus.io/docs/instrumenting/clientlibs/

Tehát továbbra sem a Prometheus szolgáltatja metrikát, a Prometheus csak összegy?jti azt, de a kezünkbe ad egy API-t, amivel a saját rendszerünkbe nagyon könnyen építhetünk metrika szolgáltató interfészt.

Java-ban nagyon egyszer?en szolgáltathatunk metrikákat az alkalmazásunkból a Prometheus client library-val. Van hozzá Maven dependency. Abba az osztályba, ami a metrikát szolgáltatja, egy statikus konstruktorral inicializáljuk a Prometheus metrika gy?jt?t. Az alábbi példában inicializálnunk egy **my_library_request_total** metrikát, ami fölé a http metrika listában egy #-val oda lesz írva a help szöveg: "Total request" (ezt nem veszi figyelembe a Prometheus). A metrikához hozzáadtuk a **method** nev? címét.

```
class YourClass {
  static final Counter requests = Counter.build()
     .name("my_library_requests_total").help("Total requests.")
     .labelNames("method").register();

  void processGetRequest() {
    requests.labels("get").inc();
    // Your code here.
  }
}
```

Ahányszor meghívjuk a **processGetRequest()** metódust, a fenti **method="get"** megcímkézett változatához hozzá fog adni egyet. (Ez egy counter típusú metrika, err?l részletesen olvashatunk majd a **Metrika típusok** cím? fejezetben.

A lekérdezésben így nézne ki:

```
#Total requests.
my_library_requests_total{method="get"} 23
```

A metrikát szolgáltató HTTP servlet elkészítésére több megoldást is kínál a Prometheus client API. Nézzünk egy példát az egyszer? Java HTTP servlet-re.

```
Server server = new Server(1234);
ServletContextHandler context = new ServletContextHandler();
context.setContextPath("/");
server.setHandler(context);

context.addServlet(new ServletHolder(new MetricsServlet()), "/metrics");
```

Note
A docker démon is tud magáról metrikákat szolgáltatni, de ez egyenl?re csak kísérleti jelleggel m?ködik ... leírni hogy kell ...

Metrika típusok, statisztika alapok

https://www.weave.works/blog/promql-queries-for-the-rest-of-us/

https://prometheus.io/docs/concepts/metric_types/

A Prometheus 4 féle metrika típust definál, de ezek csak az API szinten vannak megkülönböztetve, a Prometheus-ban már nincsenek (állítólag a következ? verzióban már meglesz, most tarunk a 2-es f?verziónál), ott nekünk kell tudni, hogy értelmes e amit felírunk szabály az adott metrikára.

Note
Ugyan a Prométeusz adatbázis nem tesz különbséget a metrika típusok között, mégis fontos megérteni a 4 alaptípus közötti különbséget, mert a Prometheus API igen is megkülönbözteti ?ket és egyedi név és formátum konvenciót alkalmaz rájuk. Csak akkor tudunk értelmes lekérdezést írni egy metrikára, ha tudjuk, hogy mit jelent az adott metrika.

Gauge (?e?d?)

A Gauge (mér?) a legegyszer?bb metrika a Prometheus-ban, ez egy egyszer? mér?szám, aminek fel és le is mehet az értéke, pl memória használat.

Counter

A számláló a második legegyszer?bb metrika fajta. Megmutatja, hogy a metrika lekérdezésének a pillanatában hány darabot számoltunk össze abból, amit a metrika jelképez, pl http lekérdezések száma egy webszerverben. A számlálónak csak n?het az értéke, vagy reset-kor visszavált 0-ra. Persze ennek betartását a Prometheus nem ellen?rzi, számára ugyan olyan név-érték pár a számláló típusú metrika is mint bármelyik másik. Ha a hivatalos Java API-t használjuk az alkalmazásunkban, akkor ez az API biztosítja ennek a betartását. Pl:

```
# TYPE builder_builds_failed_total counter
builder_builds_failed_total{reason="build_canceled"} 0
```

A counter típusú metrikák neve a konvenció szerint **_total** postfix-el van ellátva.

A számláló abszolút értékére nem szokás támaszkodni, mivel a service újraindulásakor a számlálón nullázódik, folyton jú nodo-okat indítunk el, vagy régieket állítunk le, a service-ek jönnek, mennek. Sokkal inkább az id?beli változása a lényeg, tehát olyan lekérdezéseket (gráfokat) praktikus felírni, ami csak egy adott id?szeletre vonatkozik, pl mindig csak az utolsó 5 percre. Lássunk két példát:

Az alábbi **sum(<metrika név>)** függvény az összes olyan már begy?jtött metrikának összegzi az értékét ahol a metrika neve és a címke az alábbi volt: *batch_jobs_completed_total{job_type="hourly-cleanup"}*. Azonban ha újraindul egy node, akkor a node-hoz tartozó batch_jobs_completed_total nev? számláló típusú metrika értéke nulláról fog újra indulni, így hamis képet láthatunk.

```
sum(batch_jobs_completed_total{job_type="hourly-cleanup"})
```

A rate() függvény a sum()-al ellentétben a változást számolja ki. Megmondja, hogy a megadott id?intervallumban 1 másodpercre levetítve mekkora volt a változás a számlálóban: **rate(<metrika név> [intervallum])**. Tehát a rate() egy csúszó ablakot használ, mindig visszanézve az id?ben. Ha pl 5 percre állítjuk az intervallumot, akkor legyen C1 az utoljára rögzített metrika érték, legyen C2 az 5 perccel ezel?tt rögzített els? metrika érték, akkor a rate() a következ? módon számolandó: **(C1-C2)/300** (Az osztással megkapjuk másodpercre levetítve a változás értékét)

```
sum(rate(batch_jobs_completed_total{job_type="hourly-cleanup"}[5m]))
```

Na de mit csinál akkor a sum?? a lekérdezéseket majd külön megnézzük

Histogram

http://linuxczar.net/blog/2017/06/15/prometheus-histogram-2/
https://statistics.laerd.com/statistical-guides/understanding-histograms.php
http://www.leanforum.hu/index.php/szocikkek/167-hisztogram-2
Fontos kifejezések:
distribution=eloszlás
latency=eltelt id? az input és az output között bármilyen rendszerben
frequency=gyakoriság
cardinality=számosság

Mi az a Hisztogram

A Hisztogram a gyakoriság eloszlását mutatja meg a mintának, amivel sokszor sokkal többre lehet menni, mint a hagyományos pl érték-id? diagramokból. A hisztogram egy minta elemzését segít? egyszer? statisztikai eszköz, amely a gy?jtött adatok (minta) alapján következtetések levonására ad lehet?séget. A hisztogram tulajdonképpen egy oszlopdiagram, amely X-tengelyén a tulajdonság osztályok (egy változó különböz? értékei), Y-tengelyén pedig az el?fordulási gyakoriságok találhatók. A hisztogram megmutatja az eloszlás alakját, középértékét és terjedelmét.

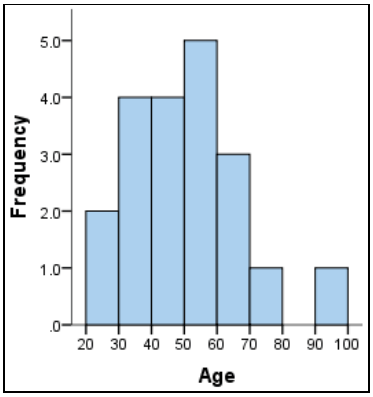
Nézzünk egy példát, hogy hogyan készül a hisztogram. Ha van egy mintám, amiben emberek korai vannak benne:

```
36      25      38      46      55      68      72      55      36      38
67      45      22      48      91      46      52      61      58      55
```

Ahhoz hogy ebb?l hisztogramot tudjunk készíteni (gyakorisági eloszlást), a minta elemeit úgynevezett osztályokba kell sorolni (**bins** vagy **buckets**), ami azt jelenti, hogy a folyamatos minta értékkészletet felvágjuk (általában egyenl? méret?) sávokra/osztályokra (x tengely), és megnézzük, hogy egy sávba hány minta elem tartozik (y tengely). A fenti példában az értékkészlet az emberek kora, amiben az osztályok, (amikbe be akarjuk sorolni a mintákat), legyenek 10 éves periódusok, és induljon 20-tól és menjen 100-ig, így összesen 8 osztályt kapunk. Fontos, hogy a Hisztogramban az osztályok (buckets) mindig összeérnek, nem lehetnek benne lukak (Az hogy 10-re vettük az osztály méretét, ez a mi egyéni döntésünk volt, bármilyen más felosztást is választhattunk volna). Most nézzük meg, hogy egy osztályba (bucket) hány elem kerül, vagyis hogy pl a 40-t?l 50-ig terjed? osztályba hány ember kerül bele. Láthatjuk, hogy a 40-50 osztályban a gyakoriság = 4.

Bin	Frequency	Scores Included in Bin
20–30	2	25,22
30–40	4	36,38,36,38
40–50	4	46,45,48,46
50–60	5	55,55,52,58,55
60–70	3	68,67,61
70–80	1	72
80–90	0	-
90–100	1	91

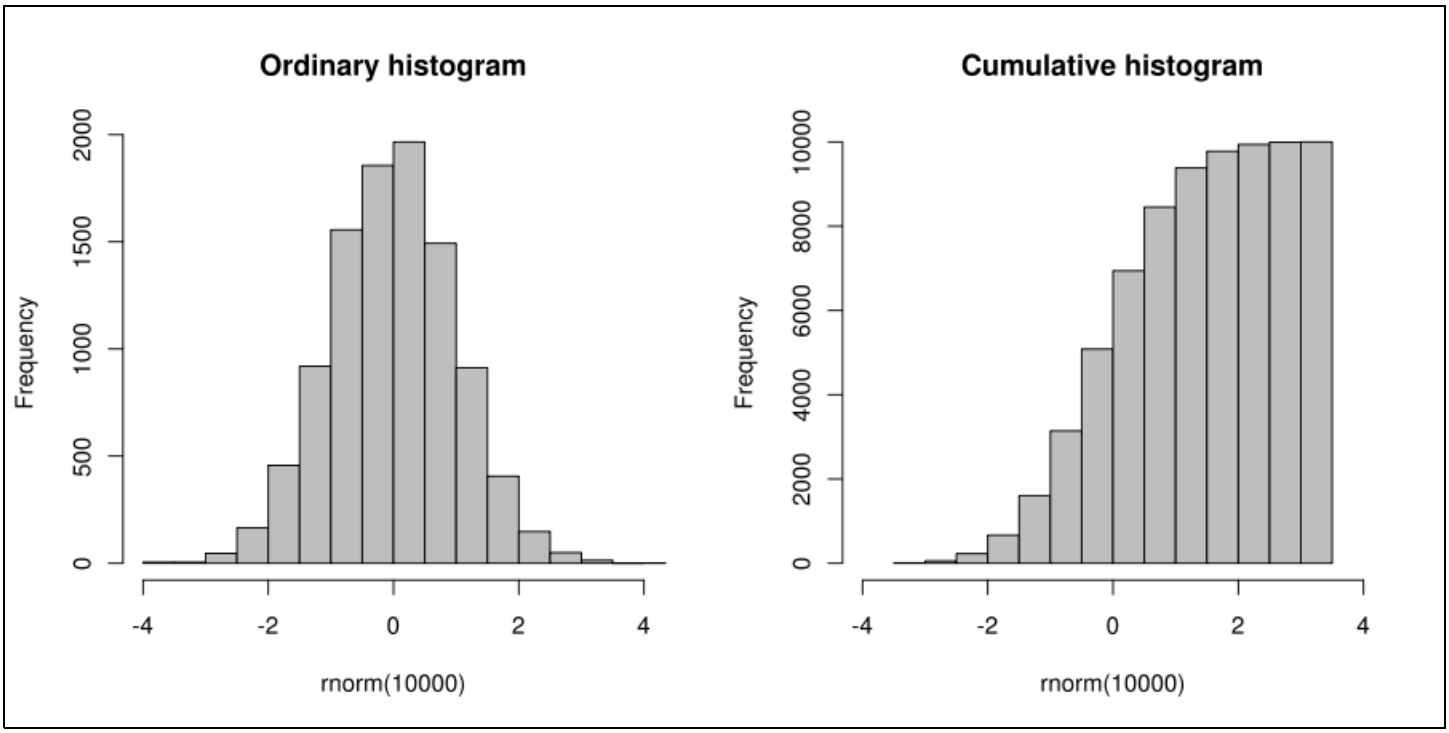
Ábrázoljuk a kapott eredményeket:



Osztályok (vödrök, bucket) meghatározása: Azt hogy egy osztály (vödör, bucket, bin) mérete mekkora legyen arra nincs ökölszabály. Ne legyen túl kicsi, mert akkor túl sok oszlop lesz a grafikonon, de ne is legyen túl nagy, mert akkor meg túl kevés, és az eloszlási görbét nem lehet majd jól látni. Ezt pl kísérleti úton lehet meghatározni.

Hisztogram fajták

- Normál Hisztogram: Ezt láthattuk a fenti példában. Minden egyes osztályhoz tartozó oszlop azt mutatja meg, hogy a mintából hány darab tartozik az adott osztályba (a példában hány ember tartozik egy adott id?sávba)
- Cumulative Histogram: itt az a különbség, hogy egy osztályhoz tatozó oszlop nem csak azt mutatja meg, hogy hány elem tartozik oda a mintából. A Cumulative Histogram-ban minden oszlop az összes el?z? oszlop összege (összes el?z? gyakoriság összege) + az adott osztályhoz tartozó gyakoriság



Note
A Prometheus cumulative Histogram-ot használ. Állítólag azért mert sokkal kevesebb er?forrásból lehet el?állítani a kumulatív hisztogramot mint a simát
https://www.robustperception.io/why-are-prometheus-histograms-cumulative

Apdex, egy speciális histogram

http://apdex.org/overview.html
https://en.wikipedia.org/wiki/Apdex
Az Apdex index-el alkalmazások teljesítményét lehet mérni. A teljesítményen itt felhasználói elégedettséget kell érteni az alkalmazás válaszidejének a szempontjából, hiszen azon túl hogy egy rendszer üzemel, elérhet? a felhasználó számára, semmi más nem számít, csak hogy elégedettek e a felhasználók a teljesítménnyel (válaszid?vel) vagy sem. Valójában semmi más nem számít.

A metrikákon alapuló mérésekkel nehéz általános következtetést levonni a felhasználói elégedettségr?l az Apdex kitalálói szerint. Tegyük fel, hogy pl egy swarm cluster-ben az összes node összes alkalmazásáról összegy?jtjük a válaszid?ket. De mit kezdjünk ezzel az adathalmazzal. Önmagában a válaszid?b?l nehéz elégedettséget levonni. Ha átlagoljuk a válaszid?ket, akkor ezzel lehet hogy elveszítünk olyan információkat, ami arra utalna, hogy nagy az általános elégedetlenség.

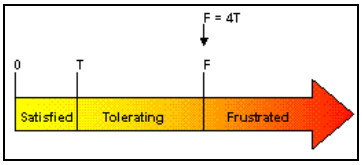
Az Apdex az összes metrikából nyert adatot egyetlen számmá alakítja 0 és 1 között, ahol 0=mindenki elégedetlen, 1=mindenki maximálisan elégedett. Ez a szám a rendszer teljesítmény, más szóval az Apdex értéke (Apdex value)

Az Apdex érték mindig a Target válaszid? függvénye, ami az általunk megállapított, szerintünk optimális válaszideje egy alkalmazásnak: $Apdex_T$
Tehát ha szerintünk egy alkalmazásnak a Target válaszideje 2 másodperc kéne legyen, akkor ezen alkalmazásra kiszámolt Apdex indexet így jelölik: $Apdex_2 = X$
Tegyük fel hogy az univerzumunk 3 alkalmazásból áll, amikre a következ? Apdex Target számokat határoztuk meg (optimális válaszid?k) és a következ? Apdex értékek jöttek ki a mérés alapján:

- Adatbázis: T=1, $Apdex_1 = 0.72$
- Email rendszer: T=8, $Apdex_8 = 0.62$
- Webshop: T=2, $Apdex_2 = 0.53$

A begy?jtött válaszid?kb?l egy speciális hisztogramot csinálunk, három nem egyenl? méret? vödörbe osztjuk az összes válaszid?t (emlékezzünk rá, hogy a histogram definíciója alapján nem kell hogy egyenl?ek legyenek a vödrök):

- **Satisfied**: Azon válaszok amik kisebbek, mint a Target válaszid?, vagyis gyorsabban kiszolgáltuk a felhasználókat. A response time 0 és T között van.
- **Tolerating**: A válaszid? nagyobb mint T, de még az elfogadható határon belül van. A válaszid? T és F (elfogadhatatlan) között van, ahol F mindig 4*T a definíció szerint.
- **Frustrated**: A válaszid? nagyobb mint F, vagyis az elfogadhatatlan kategória. A válaszid?k F és végtelen között vannak.



Az Apdex érték kiszámolásához a histogram értékeit kell összeadni a következ? módon: (Satisfied vödör darabszáma + a fele a Tolerating vödör darabszámának, és az egész osztva az összes darabbal.

$$Apdex_T = \frac{Satisfied\ count + \frac{Tolerating\ count}{2}}{Total\ samples}$$

Pl ha van 100 mintánk, ahol a Target id? = 3s, ahol 60 válaszid? van 3s alatt, 30 darab van 3 és 12 (4*3) között, és a maradék 10 van 12 fölött, akkor az Apdex:

$$\frac{60 + \frac{30}{2}}{100} = 0.75$$

Histogram a Prometheus-ban

https://prometheus.io/docs/practices/histograms/
https://prometheus.io/docs/concepts/metric_types/

A metrikák világában a hisztogramok általában válaszid?b?l és válasz méretb?l készülnek. A metrika base neve konvenció szerint megkapja a **_bucket** postfix-et. Ezen felül a vödör fels? határát pedig az "**le**" címke tartalmazza. Mivel a Prometheus kumulatív hisztogramokkal dolgozik, a vödör midig 0-tól az "**le**" címkében megadott értékig tartalmazza a minták darabszámot. Az alábbi példában a prometheus_http_request_duration_seconds hisztogram 0-tól 0.4s-ig terjed? vödörhöz metrikáját láthatjuk.

```
prometheus_http_request_duration_seconds_bucket{le="0.4"}
```

Fontos, hogy a Prometheus-ban minden egyes vödör egy külön id?sor. A hisztogramot a metrikát szolgáltató rendszerben el?re kitalálták, el?re rögzítették a vödrök méretét, ez fix, ez az id?ben nem változik. Pl a fenti **prometheus_http_request_duration_seconds** hisztogramban 9 vödröt definiált az alkotó, a legels? 0-tól 0.1-ig terjed, az utolsó meg 0-tól 120s-ig. Tehát az alkotó úgy gondolta, hogy az összes válaszid? 0 és 120 közé fog esni.
Minden egyes minta begy?jtéskor a metrikát szolgáltató rendszer elküldi a Prometheus-nak az aktuális, teljes hisztogramot, tehát a hisztogramot a Prometheus készen kapja, nem ? számolja ki. Ebb?l adódik, hogy minden egyes hisztogram vödör egy külön id?sort alkot, hiszen minden egyes lekérdezéskor változhat a hisztogram. Tehát a _bucket-el végz?d? metrikák egy hisztogram darabkái. A hisztogramot alkotó minták száma id?ben folyton n? ahogy egyre több request-et szolgál ki a szerver, úgy egyre több mintánk lesz, ebb?l kifolyólag majdnem minden vödör értéke is n?ni fog (kivéve az a vödör, ami olyan kicsi request id?t szimbolizál, amibe nem estek bele minták. Mivel a hisztogram kumulatív, az összes ett?l nagyobb válaszid?t szimbolizáló vödör értéke n?ni fog). Ahogy telik az id?, egyre több válaszid? értéke (mintája) lesz a hisztogramot szolgáltató rendszernek, tehát mindig egyre több mintából állítja el? a hisztogramot, a hisztogram rudacskái minden lekérdezéskor egyre nagyobbak.
A példában említett hisztogramot a következ? metrikák (vödrök) alkotják:

```
prometheus_http_request_duration_seconds_bucket{le="0.1"}
prometheus_http_request_duration_seconds_bucket{le="0.2"}
prometheus_http_request_duration_seconds_bucket{le="0.4"}
prometheus_http_request_duration_seconds_bucket{le="1"}
prometheus_http_request_duration_seconds_bucket{le="3"}
prometheus_http_request_duration_seconds_bucket{le="8"}
prometheus_http_request_duration_seconds_bucket{le="20"}
prometheus_http_request_duration_seconds_bucket{le="60"}
prometheus_http_request_duration_seconds_bucket{le="120"}
```

Speciális hisztogram metrikák
A **_sum** postfix-re végz?d? metrikában van az összes minta összege, de nem id?ben visszamen?leg, hanem az adott metrika begy?jtéskor aktuálisan kapott hisztogram-ban lév? minták összeg. Ezt egyfajta speciális számlálónak is felfoghatjuk, aminek az értéke szintén csak n?ni tud, hiszen mindig egyre több mintából állítja el? a hisztogramot a mintákat szolgálató rendszer (pl cAdviser). Egy esetben tud n?ni a _sum, ha negatív megfigyelések is lehetségesek, pl h?mérséklet esetén.

```
prometheus_http_request_duration_seconds_sum
```

A **_count** postfix-re végz?d? hisztogram metrika az összes minta darabszámát adja vissza. A fent leírt okokból ez értelem szer?en csak n?ni tud.

```
prometheus_http_request_duration_seconds_count
```

Kötelez?en kell legyen minden hisztogramban egy **le=+Inf** elem, aminek az értéke mindig megegyezik a **_count** metrikáéval.

```
prometheus_http_request_duration_seconds_bucket{le="+Inf"}
```

Summary (kvantilis)

Na ezt még egyáltalán nem értem. quantiles

A p-ed rend? kvantilis az a szám, amelynél az összes el?forduló ismérvérték p-ed része nem nagyobb, (1-p)-ed része nem kisebb. Például az $x_{2/5}$ kvantilis esetében az adatok 40%-a nem nagyobb, 60%-a nem kisebb a meghatározott kvantilisnél.

Meghatározásánál fontos az adatok sorrendbe való rendezése.

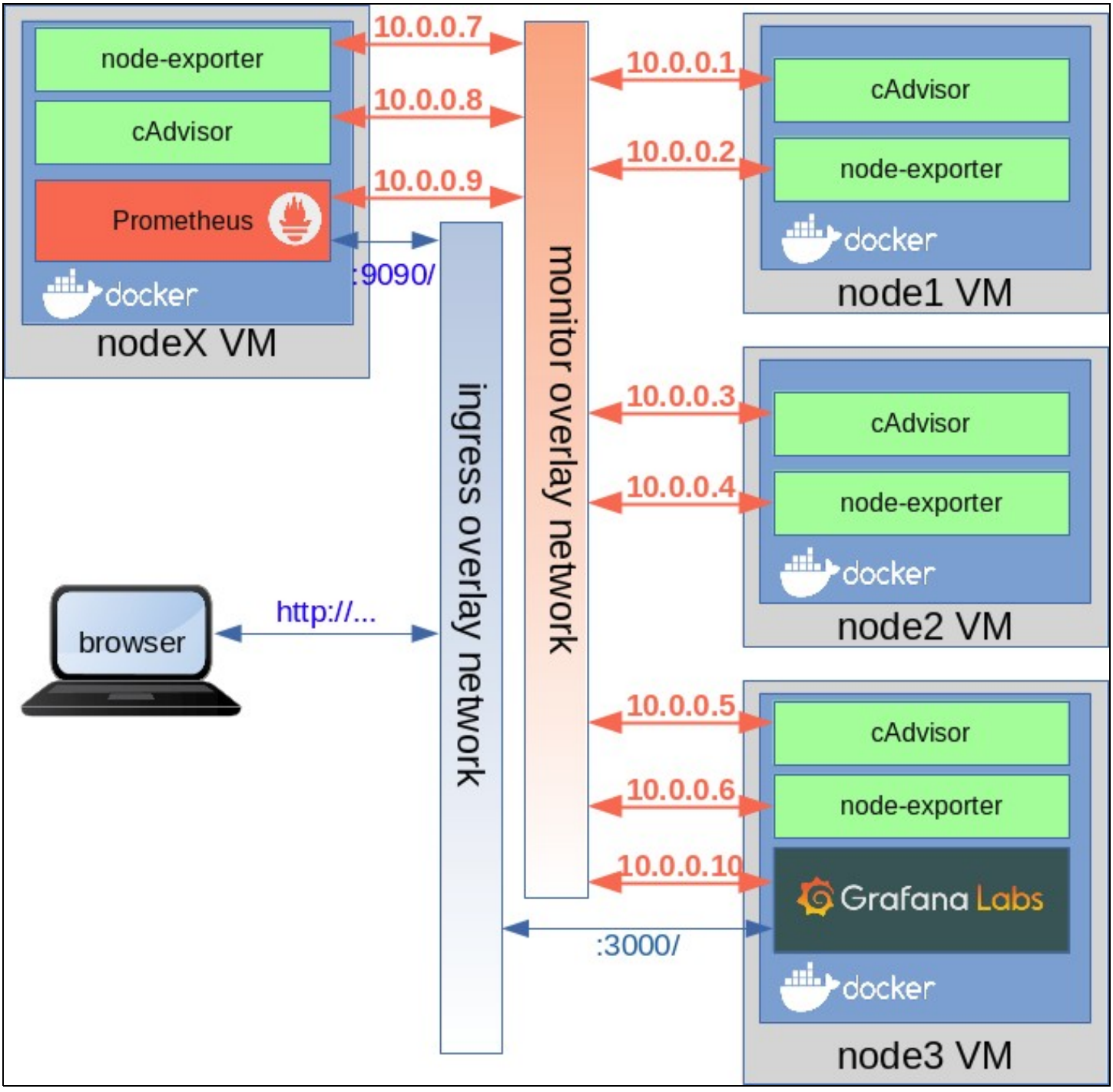
https://www.tankonyvtar.hu/hu/tartalom/tamop425/0027_MA3-7/ch01s07.html

https://povilasv.me/prometheus-tracking-request-duration/

Prometheus architektúra

Ha a pet szemléletet magunk mögött hagyjuk, akkor máris kézenfekv?, hogy számunkra irreleváns, hogy a swarm melyik node-ra fogja telepíteni a Prometheus-t (arra amelyik szerinte a legoptimálisabb), a lényeg, hogy garantáltan csak egy példány jöjjön bel?le létre. Minket csak a portja érdekel, a port segítségével bármelyik node IP címével el lehet majd érni a load-balancolt **ingress** hálózaton. Ugyan ez igaz lesz a Grafana-ra is. A **Grafana** az **ingress** hálózaton keresztül bármelyik node IP címével eléri majd a Prometheus-t, és mi is a böngész?b?l az **ingress** hálózaton bármelyik node IP címével elérjük majd a **Grafana** konzolt.

Ellenben fontos, hogy a Prometheus a **monitor** overlay hálózatra is csatlakozzon, hogy közvetlenül el tudja érni a node-okon futó **node-exporter**-t és a **cAdvisor**-t (emlékezzünk, hogy ezen konténereknek nem nyitottunk portot a host felé, így ezek kizárólag a **monitor** overlay hálózaton érhet?ek el). A Grafana már csak az **ingress** hálózatra kell hogy csatlakozzon, mert csak a Prometheus-ból végez majd lekérdezéseket.



Nem jó az ábra!!! A Grafana a monitor hálózatra is csatlakozik, ott éri el a Prometheus-t. Ezen felül az ingress-re is, ott érjük el kívülr?l. A Grafana ingress kapcsolatára a portot is rá kell írni, mert az publikálva lett!!

Itt leírni, hogy minden egyes node VM en indítani kell egy egy node-exporter-t és cAdvisor-t, és hogy melyik pontosan mit fog csinálni... ...

...

Note
Készíthetnénk speciális címkékkel ellátott swarm node-okat dedikáltan a Prometheus-nak és a Grafana-nak, de ez szembe menne a swarm személettel, vissza vinne minket a tehéncsordáról a háziállatos felfogás irányába. Majd a swarm tudja hogy hova a legjobb ezen konténereket rakni, ne mi akarjuk megmondani!

Cluster létrehozása

```
#!/bin/bash

#Create manager
docker-machine create -d kvm --kvm-network "docker-network" --kvm-disk-size "5000" --kvm-memory "800"  mg0


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

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


#Create workers
for i in 0 1 2 3; 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



# eval $(docker-machine env mg0)

# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS
4nnp0k53i8739t1wxrp874wxa     worker1             Ready               Active
lu4000w394kl622c3g211yg31     worker0             Ready               Active
nx6kmd9qj55yc4f3fwpbhq60q     worker2             Ready               Active
qvnnm8mvokpn662ooliqko3zv *   mg0                 Ready               Active              Leader
t2ng6ylwcgvq4ssx4cvf71lfi     worker4             Ready               Active

# docker network create -d overlay monitor

# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
...
akhr117qs46o        monitor             overlay             swarm
```

Node exporter

Ezt nem lenne muszáj konténerként telepíteni, futhat natívan is, de egyszer?bb ha konténerként fut, mert a swarm így automatikusan ki tudja rakni minden node-ra, nem nekünk kell. Ha konténerként fut, akkor a host OS szinte minden porcikáját fel kell mountolni a node-exporter konténernek, hiszen a le kell tudja kérdezni a host OS állapotát. Ez azért sem nagy biztonsági kockázat, mert ha swarm-ot futtatunk, akkor a hostok valamelyik virtualizációs környezetben fognak futni, és nagy valószín?séggel az egyetlen feladatuk a docker futtatása lesz.

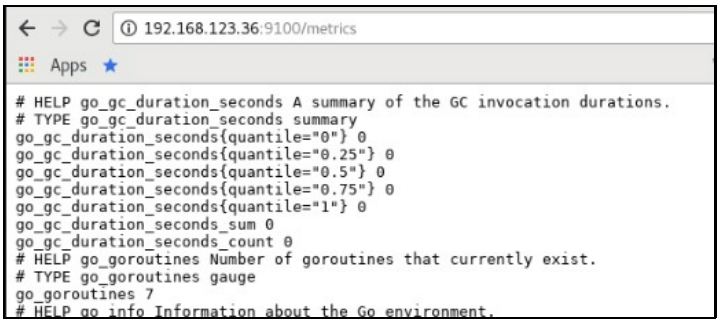
Els?ként swarm-on kívül tegyük rá a mg0-ra, és nyissuk ki a host OS felé a 9100-es portot, hogy lássuk, hogy milyen metrikákat ad. A fentiekben létrehoztunk egy monitor nev? overlay hálózatot. Erre mind a Node Exporer mind a cAdviser és a Prometheus is közvetlen fog csatlakozni egy egy network interfésszel, ahol közvetlen meg tudják egymást szólítani, tehát egyáltalán nem szükséges a Node Exporter vagy a cAdviser egyik portját sem publikálni az ingres, load balanc-olt hálózatban. Azonban ha nem publikáljuk a portájt, nem tudunk a böngész?b?l belenézni. Ezért miel?tt swarm service-ként telepítenénk, els?ként standalon konténerként fel fogjuk rakni az mg0 node-ra, úgy hogy publikáljuk a 9100 portját, ahol a metrikákat szolgáltatja, majd a böngész?vel meg fogjuk nyitni ezt a portot.

```
docker run -d \
-p 9100:9100 \
--name node-exporter \
--mount "type=bind,source=/proc,target=/host/proc" \
--mount "type=bind,source=/sys,target=/host/sys" \
--mount "type=bind,source=/,target=/rootfs" \
prom/node-exporter:v0.16.0 \
--collector.filesystem.ignored-mount-points \
"^/(sys|proc|dev|host|etc)($|/)"



# docker-machine ssh mg0 docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS                    NA
7f81d5a9243c        prom/node-exporter:v0.16.0     "/bin/node_exporter ?"   18 seconds ago      Up 18 seconds       0.0.0.0:9100->9100/tcp   no
```

A KVM-es környezetünkben két virtuális hálózat található. A **docker-machine** hálózat, ami egy guest-only network. Ezt a docker-machine KVM driver hozta létre a swarm internal management kommunikációra. A másik hálózat a **docker-network** (eth0), amit mi hoztunk létre, ez egy publikus hálózat, ezen keresztül látnak ki a node-ok az Internetre, és mi is ezen hívjuk meg az ingress hálózaton elérhet?, load-balance-olt swarm szolgáltatásokat. A **docker-machine ip** sajnos a guest-only IP címet adja vissza, nekünk viszont a másik kell:

```
# docker-machine ssh mg0 ifconfig | grep -A 1 eth0 | grep "inet addr"
          inet addr:192.168.123.36  Bcast:192.168.123.255  Mask:255.255.255.0
```

http://192.168.123.36:9100/metrics




```
# docker-machine ssh mg0 docker rm -f node-exporter
```

Service-ként így kell feltenni. Azért nem kell portot nyitni, mert a monitor nev? overlay hálózatra van kötve, ahova majd a Prometheus-t is rákötjük, így nincs arra szükség, hogy a host OS felé kinyissuk a 9100-as portot.

```
docker service create \
--name node-exporter \
--mode global \
--network monitor \
--mount "type=bind,source=/proc,target=/host/proc" \
--mount "type=bind,source=/sys,target=/host/sys" \
--mount "type=bind,source=/,target=/rootfs" \
prom/node-exporter:v0.16.0 \
--collector.filesystem.ignored-mount-points \
"^/(sys|proc|dev|host|etc)($|/)"
```

Létezik egy módosított változata is a node-exporter-nek, ami képes a swarm node nevét is belerakni a metrika címkéjébe, így nem csak egy IP címet látunk majd a Prometheus-ban/Grafana-ban, ahem egy beszédes node nevet, pl: mg0

```
docker service create --detach=false \
  --name node-exporter \
  --mode global \
  --network monitor \
  --mount type=bind,source=/proc,target=/host/proc \
  --mount type=bind,source=/sys,target=/host/sys \
  --mount type=bind,source=/,target=/rootfs \
  --mount type=bind,source=/etc/hostname,target=/etc/host_hostname \
  -e HOST_HOSTNAME=/etc/host_hostname \
  basi/node-exporter:latest \
  --path.procfs /host/proc \
  --path.sysfs /host/sys \
  --collector.filesystem.ignored-mount-points "^/(sys|proc|dev|host|etc)($|/)" \
  --collector.textfile.directory /etc/node-exporter/
```

A "**mode global**" miatt kerül rá az összes swarm résztvev?re.

```
# docker service ps node-exporter
ID                  NAME                                    IMAGE                          NODE
wu1gprh06sgj        node-exporter.y6pm0pw3l12d46lmb3qf4phg1   prom/node-exporter:v0.16.0   mg0
khoy23w1c48c        node-exporter.7cxts15b77dvsbm9gygn7846j   prom/node-exporter:v0.16.0   worker4
1bf9g1n3xheo        node-exporter.zg7qx4aowtlh239lw29gvsbuf   prom/node-exporter:v0.16.0   worker2
y5d1l42a6p16        node-exporter.utxvbnfxu5x5sydnhd1uj7n5b   prom/node-exporter:v0.16.0   worker1
ttx9wyggx4wt        node-exporter.y1y1wnhz8kq70smuuet8exdhh   prom/node-exporter:v0.16.0   worker0
```

cAdvisor

Mi az a cAdvisor ... + architectura ábra. ...

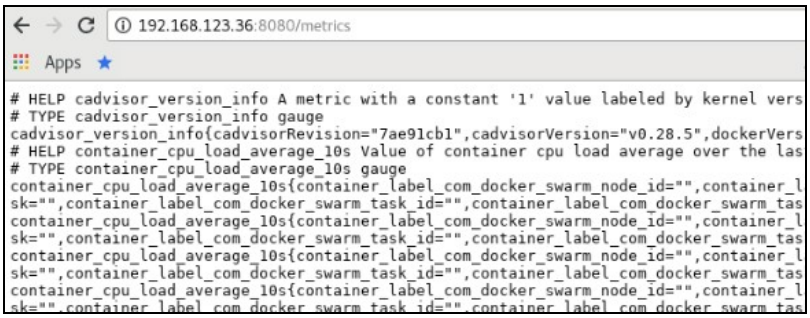
A cAdisor és a Node Exporter metrikái között van egy kis átfedés, mert a cAdivisor is szolgáltat pár OS specifikus metrikát.

A cAdivsor -t sem lenne muszáj konténerben futtatni, de így sokkal egyszer?bb. A lényeg, hogy tudja olvasni a host-on futó dockar démon állapotát, tehát a docker socket-et fel kell mountolni a cAdvisor konténernek.

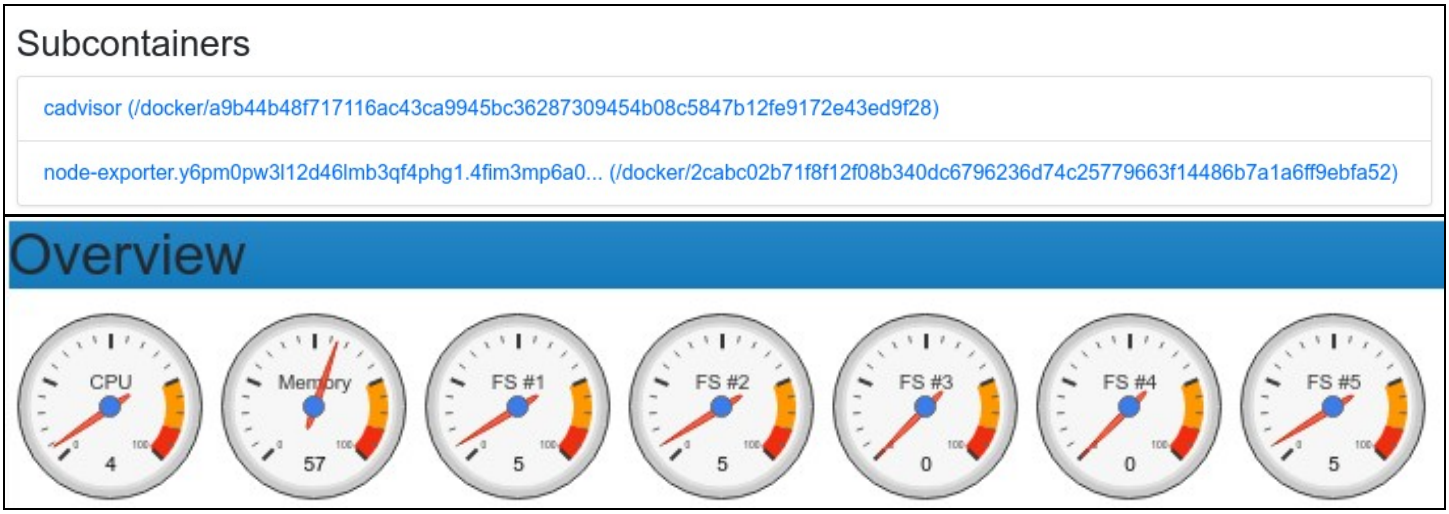
A kés?bbiekben, mikor majd swarm service-ként telepítjük a **cAdvisor**-t nem lesz rá szükség hogy a publikáljuk a 8080-ás portját az ingress, load-balance-olt hálózatra, mivel a **cAdvisor** konténerek is a monitor nev? overlay hálózatra fognak kapcsolódni, így a Prometheus el fogja közvetlen érni ?ket. Azonban mi el?bb meg akarjuk nézni böngész?b?l, hogy hogyan néznek ki a metrikák, amiket szolgáltat, ezért es?ként standalone docker konténerként fel fogjuk telepíteni ezt is az mg0 node-ra, úgy hogy publikáljuk a 8080 portját.

```
docker run -d --name cadvisor \
-p 8080:8080 \
--mount "type=bind,source=/,target=/rootfs" \
--mount "type=bind,source=/var/run,target=/var/run" \
--mount "type=bind,source=/sys,target=/sys" \
--mount "type=bind,source=/var/lib/docker,target=/var/lib/docker" \
google/cadvisor:v0.28.5
```

Mivel a cAdviser konténert is az mg0 node-ra tettük, ugyan azon az IP-n érhet? el mint az el?z? példában a Node Exporter.
http://http://192.168.123.36:8080/metrics



http://192.168.123.36:8080



A grafikus felületnek sok értelme nincsen, ugyanis a cAdvisor-t kifejezetten cluster-ek monitorozására találták ki. A grafikus felület csak egy node lelki világát mutatja, és mivel minket az összes node érdekel, ezért a grafikus felület használhatatlan valójában. Ráadásul ha swarm-ban futtatnánk a 8080 porttal, akkor az ingress load balancer miatt sosem tudnánk, hogy melyik node webes felületére csatlakoztunk be, tehát swarm módban tényleg használhatatlan. Mivel a Prometheus a "közös" overlay hálózaton közvetlen el fogja érni a cAdvisor konténert, ezért itt az ingress hálózat nem fog bezavarni.

Ezért úgy fogjuk swarm service-ként kirakni a cAdivos-ort, hogy a 8080-as portját nem is nyitjuk ki a host felé, ugyanúgy rárakjuk a **monitor** nev? overlay hálózatra, ahol a Prometheus el fogja érni közvetlenül a http://10.0.0.X:8080/metrics linken.

```
docker service create --name cadvisor \
--mode global \
--network monitor \
--mount "type=bind,source=/,target=/rootfs" \
--mount "type=bind,source=/var/run,target=/var/run" \
--mount "type=bind,source=/sys,target=/sys" \
--mount "type=bind,source=/var/lib/docker,target=/var/lib/docker" \
google/cadvisor:v0.28.5
```

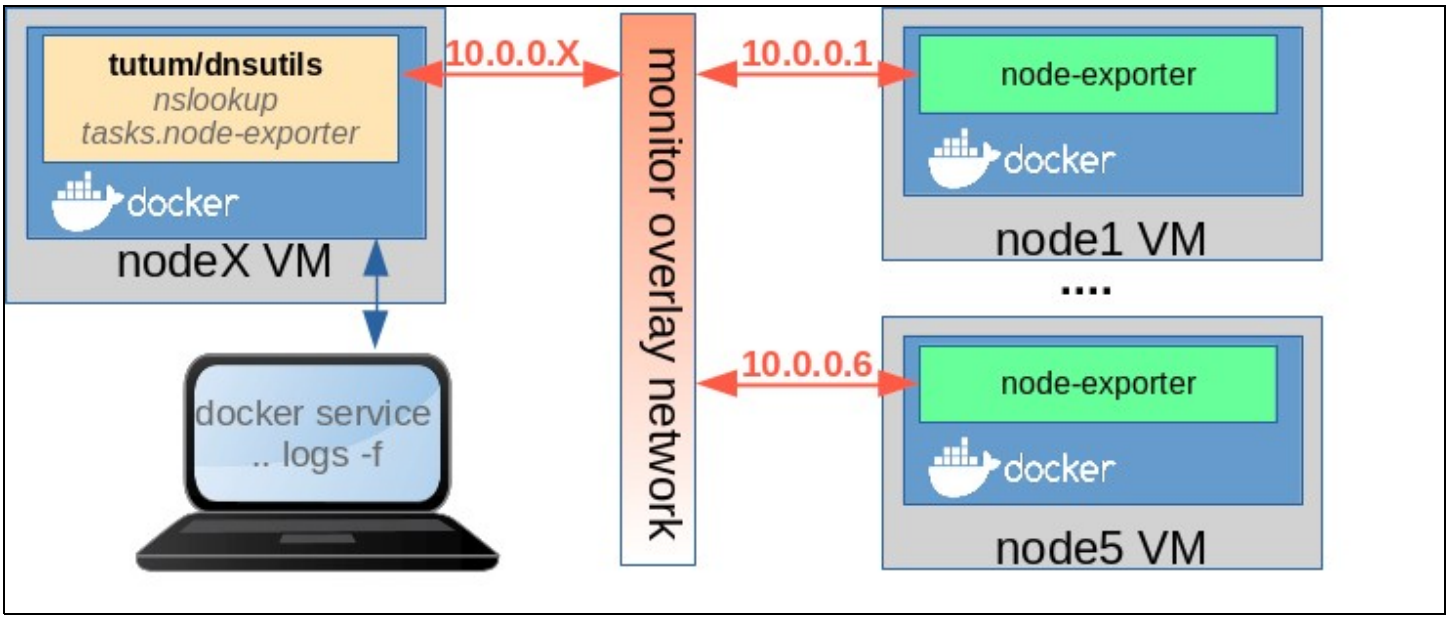
A "--mode global" miatt kerül ki az összes noder-ra, és ugyan úgy rákötöttük a **monitor** nev? overlay hálózatra.

```
# docker service ps cadvisor
ID                  NAME                                IMAGE                       NODE
sb6kszve3q9z        cadvisor.7cxts15b77dvsbm9gygn7846j   google/cadvisor:v0.28.5   worker4
4gi5aqnp790d        cadvisor.y1y1wnhz8kq70smuuet8exdhh   google/cadvisor:v0.28.5   worker0
uxy7ybpdgp0u        cadvisor.zg7qx4aowtlh239lw29gvsbuf   google/cadvisor:v0.28.5   worker2
dpddiisqmrtx        cadvisor.utxvbnfxu5x5sydnhd1uj7n5b   google/cadvisor:v0.28.5   worker1
5kalv9p21iav        cadvisor.y6pm0pw3l12d46lmb3qf4phg1   google/cadvisor:v0.28.5   mg0
```

DNS lookup

A Prometheus-nak tudnia kell az összes Node-Exporter és cAdvisor konténer **monitor** hálózatbeli IP címét, hogy le tudja kérdezni t?lük a metrikákat. Az IP címeket nyilván nem drótozhatjuk be a Prometheus konfigurációjába, mivel node-ok dinamikusan létrejönnek és megsz?nnek egy swarm cluster-ben. Szerencsére a swarm a **tasks.<service név>** dns lekérdezés hatására visszaadja a szolgáltatáshoz tartozó node-ok IP cím listáját azon az overlay hálózaton, ahol a lekérdezést végeztük (tehát egy olyan konténerb?l kell a lekérdezést végezni, ami ugyan arra az overlay hálózatra csatlakozik, mint a szolgáltatás konténerei. A Prometheus képes ilyen lekérdezéseket végrehajtani minden metrika begy?jtés el?tt, hogy tudja, hogy aktuálisan melyik node-któl kell begy?jteni a metrikákat.

Ahhoz hogy ezt demonstrálni tudjuk, szükségünk van egy olyan konténerre, ami szintén a **monitor** nev? overlay hálózatra csatlakozik, és van benne dns util program, pl **nslookup** vagy **dig**. Fontos, hogy ezt a konténert swarm service-ként futtassuk, hogy rá tudjuk csatlakoztatni a **monitor** overlay hálózatra, ahol a DNS lekérdezést szeretnénk eszközölni. A swarm a localhost-on biztosít egy beépített DNS szervert a konténerek számára. A demonstráció céljára tökéletes a **tutum/dnsutils** image, ami ugyan ahogy lefutott le fog állni, és a swarm már is újra fogja indítani, ett?l még minden egyes lefutáskor ki fogja nekünk írni a konténer log-ba a dns lekérdezés eredményét.



A következ? példában az összes node-exporter konténer IP címét fogjuk begy?jteni.

```
# docker service create --name util \
--network monitor \
--replicas 1 \
tutum/dnsutils nslookup tasks.node-exporter

# docker service logs -f util
...
util.1.zueyj5pydd0i@worker1    | Address: 10.0.0.17
util.1.zueyj5pydd0i@worker1    | Address: 10.0.0.18
util.1.zueyj5pydd0i@worker1    | Address: 10.0.0.15
util.1.zueyj5pydd0i@worker1    | Address: 10.0.0.11
util.1.zueyj5pydd0i@worker1    | Address: 10.0.0.16
```

Amint megjelenik a log-ban a node-ok listája, el is távolíthatjuk a folyton újrainduló **util** nev? service-t.

Prometheus

A Prometheus konténernek egyrészt csatlakoznia kell a **monitor** nev? overlay hálózatra, hogy le tudja kérdezni a **node-exporter** és a **cAdvisor** által begy?jtött metrikákat, másrészt publikálnia kell a 9090-ás portját az **ingress** (load-balance-olt) hálózaton. Fontos, hogy nem fogjuk megkötni, hogy melyik nodre-ra telepítse fel a swarm, csak azt, hogy egy példány jöjjön csak bel?le létre.

Prometheus konfiguráció

Fontos, hogy a Prometheus dinamikusan frissítse a **cAdvisor** és a **node-exporter** node-ok listáját, ne legyen beégetve a config-ba, hiszen ha a swarm mérete változik (pl dinamikus skálázás miatt) akkor fontos, hogy az új node-okat a Prometheus automatikusan hozzáadja a lekérdezend? node-ok listájához, vagy ha csökken a cluster akkor a kies? node-okat már ne vegye figyelembe. A Prometheus képes a fent bemutatott DNS lekérdezéssel frissíteni az aktuális node listát (**task.<service név>**, pl tasks.node-exporter), persze ehhez az kell, hogy ? is rajta legyen a **monitor** nev? overlay hálózaton, ahol a DNS lekérdezést eszközölni szeretné, ahogy ezt az el?z? példában láthattuk, csak a tutum/dnsutils helyett a Prometheus végzi majd a lekérdezést.

A Prometheus konfigurációs fájlja a /etc/pormetheus/prometheus.yml.

prometheus.yml

```
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: 'node'
    dns_sd_configs:
      - names: ['tasks.node-exporter']
        type: A
        port: 9100

  - job_name: 'cadvisor'
    dns_sd_configs:
      - names: ['tasks.cadvisor']
        type: A
        port: 8080

  - job_name: 'prometheus'
    static_configs:
      - targets: ['prometheus:9090']
```

A konfiguráció elég beszédes. Összesen három darab metrika forrást (job-ot) definiálunk. A node-exporter és a cAdvisor konténerek végpontját a docker DNS-böl kérdezni le (dns_sd_configs). A lekérdezés A osztályú DNS rekordokat ad vissza (type:A). A harmadik metrika szolgáltató a Prometheus saját maga, aminek fixen megadtuk az elérhet?ségét (static_config) a swarm szolgáltatás nevével.

Volume plugin használata

A Prometheus-nak a konfigurációs fájlját és az adatbázis mappáját a távoli NFS szerveren fogjuk tárolni, és a "Docker volume orchestration/Netshare" fejezetben bemutatott **Negshare** volume plugin-el fogjuk ezeket a Prometheus konténerbe mount-olni.

A Prometheus konfigurációs fájlja itt van: */etc/prometheus/prometheus.yml* Az /**etc**/**prometheus**/ mappában a yml fájlon kívül két további mappa is található. Nekünk az lenne a célunk, hogy a **prometheus.yml** fájl az NFS megosztásbol jöjjön. Sajnos volume-ként nem lehet egy fájlt mount-olni csak mappákat, csak a bind mount-al lehet követlen fájlokat mount-olni a konténerbe. Vagyis nem tehetjük meg, hogy csak a **prometheus.yml** fájlt cseréljük le, az NFS megosztást csak a teljes /**etc**/**prometheus**/ mappába tudjuk felcsatolni ezzel fejbe vágva az ottani többi fájlt/mappát. Nem tudunk mást tenni, mint hogy a teljes /**etc**/**prometheus** mappát átmásoljuk az NFS megosztásunkra, ahol majd lecseréljük a prometheus.yml fájlt.

Ehhez els?ként telepítsük föl a Prometheus konténert standalone módban mount nélkül, hogy onnan ki tudjuk másolni a /etc/prometheus mappa tartalmát. Els?ként nézzük meg mi van a /**etc**/**prometheus** mappában, majd a mappa tartalmát másoljuk át az NFS meghajtóra.

```
docker run -d --name prometheus -p 9090:9090 \
prom/prometheus:v2.3.2
```

A Prometheus image-ben bash nincs, de van sh:

```
# docker exec -it prometheus sh
/prometheus $ ls -l /etc/prometheus
total 4
lrwxrwxrwx    1 nobody   nogroup         39 Jul 12 15:08 console_libraries -> /usr/share/prometheus/console_libraries
lrwxrwxrwx    1 nobody   nogroup         31 Jul 12 15:08 consoles -> /usr/share/prometheus/consoles/
-rw-r--r--    1 nobody   nogroup        926 Jul 12 15:04 prometheus.yml
```

Láthatjuk, hogy a prometheus.yml mellett még két simlink is található, ezeket is át kell másolni.

Másoljuk át a mappa tartalmát, majd töröljük le a standalone Prometheus konténert, hogy fel tudjuk service-ként telepíteni.

```
# docker cp -L prometheus:/etc/prometheus /home/adam/Projects/DockerCourse/persistentstore/prometheus/
# chmod 777 -R /home/adam/Projects/DockerCourse/persistentstore/
# mv /home/adam/Projects/DockerCourse/persistentstore/prometheus/prometheus/ /home/adam/Projects/DockerCourse/persistentstore/prometheus/conf
# docker rm -f prometheus
```

Prometheus szolgáltatás indítása

```
docker service create \
--detach=false \
--name prometheus \
--network monitor \
-p 9090:9090 \
--mount "type=volume,src=192.168.42.1/home/adam/Projects/DockerCourse/persistentstore/prometheus/config,dst=/etc/prometheus,volume-driver=nfs
--mount "type=volume,src=192.168.42.1/home/adam/Projects/DockerCourse/persistentstore/prometheus/data,dst=/prometheus,volume-driver=nfs" \
prom/prometheus:v2.3.2
```

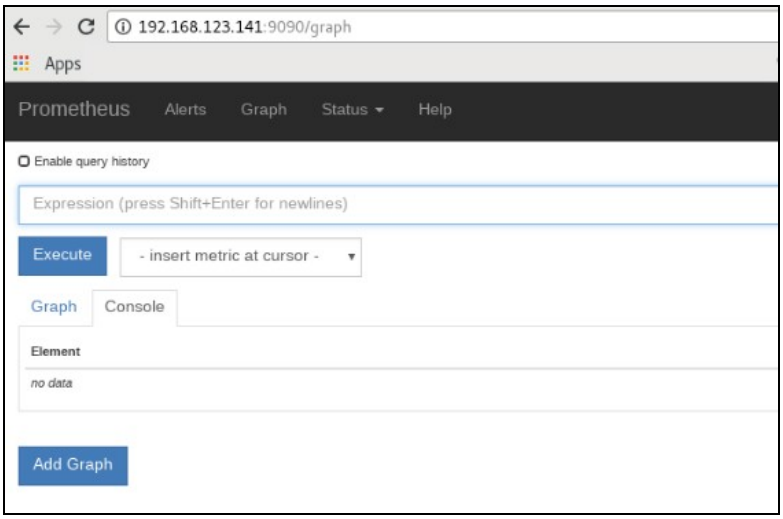
Nézzük meg, hogy melyik node-ra került:

```
# docker service ps prometheus
ID                  NAME                IMAGE                    NODE                DESIRED STATE       CURRENT STATE            ERROR
11bkjfeobwuk        prometheus.1        prom/prometheus:v2.3.2   mg0                 Running             Running 25 seconds ago
```

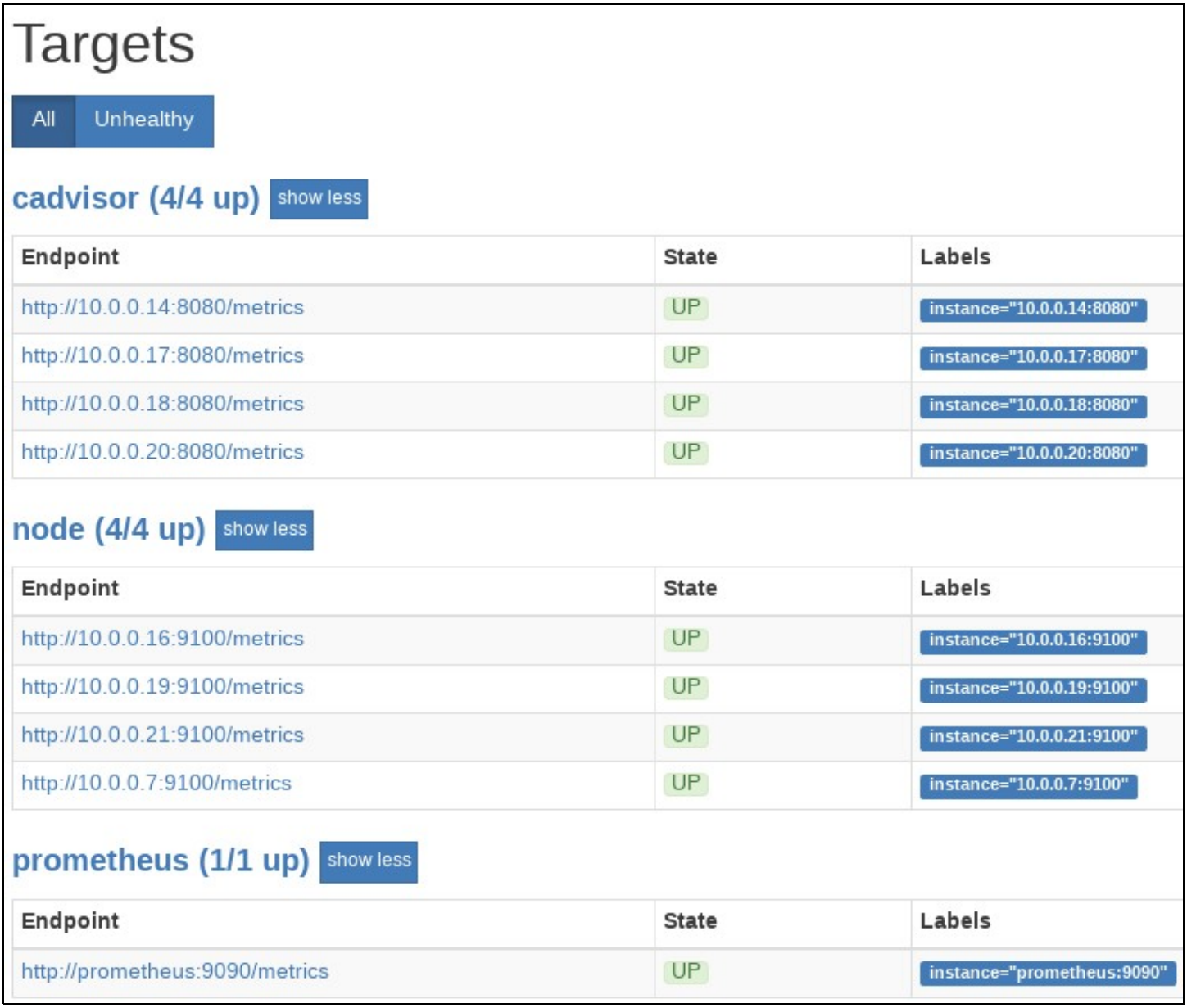
Innent?l kezdve a Prometheus webes konzol elérhet? bármelyik swarm node publikus IP címén a 9090-es porton az ingress hálózaton keresztül (**docker-network** hálózat, eth0 interfész). Kérdezzük le a worker0 IP címét. Sajnos a **docker-machine ip** parancs nem a publikus címet adja vissza, hanem a **docker-machine** hálózati címet, ami a swarm management kommunikációra szolgál a node-ok között.

```
# docker-machine ssh worker0 ifconfig | grep -A 1 eth0 | grep "inet addr"
          inet addr: 192.168.123.141  Bcast:192.168.123.255  Mask:255.255.255.0
```

http://192.168.123.141:9090/graph



Végezetül nézzük meg a Status/Targets képerny?n, hogy mind a 8+1 konténerhez sikerült a kapcsolódnia (4 node-exporter, 4 cAdvisor, 1 Prometheus)



Lekérdezések

https://prometheus.io/docs/prometheus/latest/querying/basics/#gotchas
A Prometheus-ban a lekérdezéseket PromQL nyelven kell írni. A lekérdezéseknek két legfontosabb eleme:

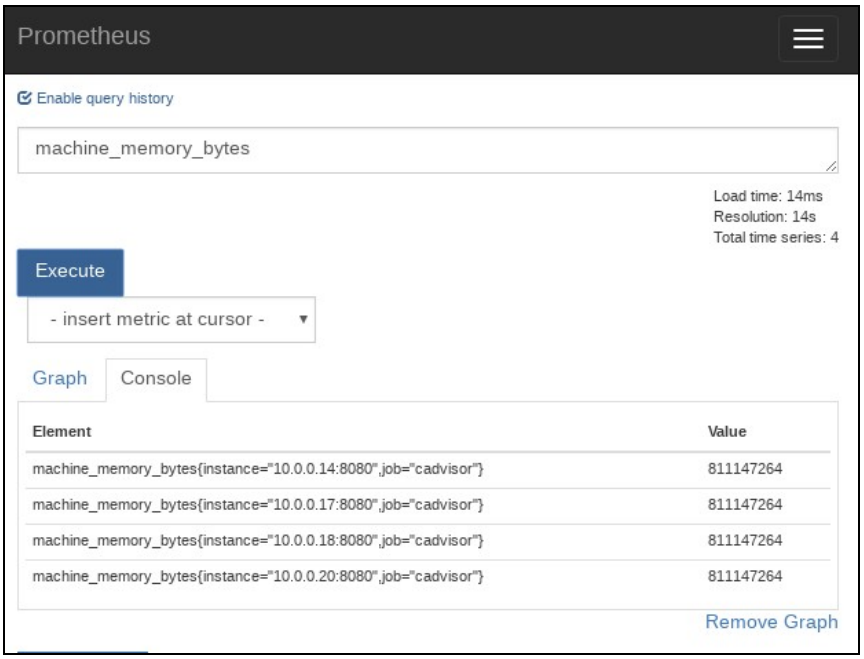
- Instant vector: A szelektronak megfelel? id?sorok egy-egy értékét adja vissza egy meghatározott id?pillanatra. Pl ha a selector a **machine_memory_bytes**, akkor az összes olyan id?sort vissza fogja adni, ahol a metrika alapneve machine_memory_bytes, vagyis az összes címe variánsát. Fontos, hogy minden egyes id?sorhoz csak egyetlen egy értéket fog vissza adni. Ha külön nem határozzuk meg, akkor ez a legutolsó lekérdezés eredménye lesz. Az instant vector lekérdezések jeleníthet?k meg gráfokon.
- Range vektor: A szelektor mellé megadunk egy id? intervallumot is. A szelektornak megfelel? id?sorhoz az összes értéket vissza fogja adni amit a megadott intervallumban rögzített. Az intervallum mindig a jelenben kezd?dik és az intervallum hosszával megy vissza az id?ben. Pl: ha a selector **machine_memory_bytes [5m]** akkor az összes illeszked? nev? id?sor minden felvett értékét vissza fogja adni amit az elmúlt 5 percben rögzített.

Selector-ok

Instant vektor választó

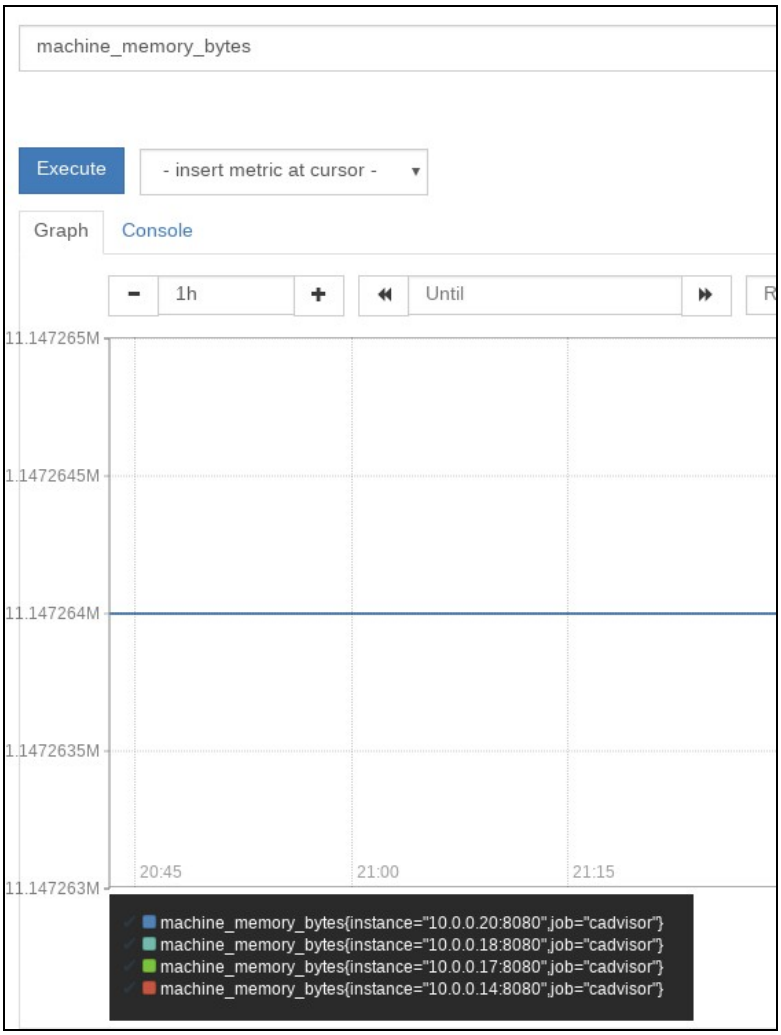
Ha magunk akarunk lekérdezést írni, akkor ezt a legegyszer?bb a Prometheus webes felületén összerakni a **Graph** képerny?n. A képerny? tetején lév? keres? mez?be (Expression) kezdjük el begépelni a keresett metrika nevét, ekkor fel fogja dobni az összes olyan metrikát, ami tartalmazza a beírt nevet. A legegyszer?bb selector, ha beírjunk egy metrika alap nevet, aminek vannak címke variánsai: **machine_memory_bytes**

Az Execute megnyomása után a **Console** fülön megjelenik a találati lista. A baloldali oszlopban van a beírt metrika név összes címke variánsa, a jobboldali oszlopban pedig a legutoljára rögzített értékük.



Láthatjuk, hogy összesen négy metrika felel meg a keresési kritériumnak, amit a négy cAdviser konténer szolgáltatott. Az utolsó lekérdezéskor mind a 4 node-on a memória használat 800 mega körül volt.

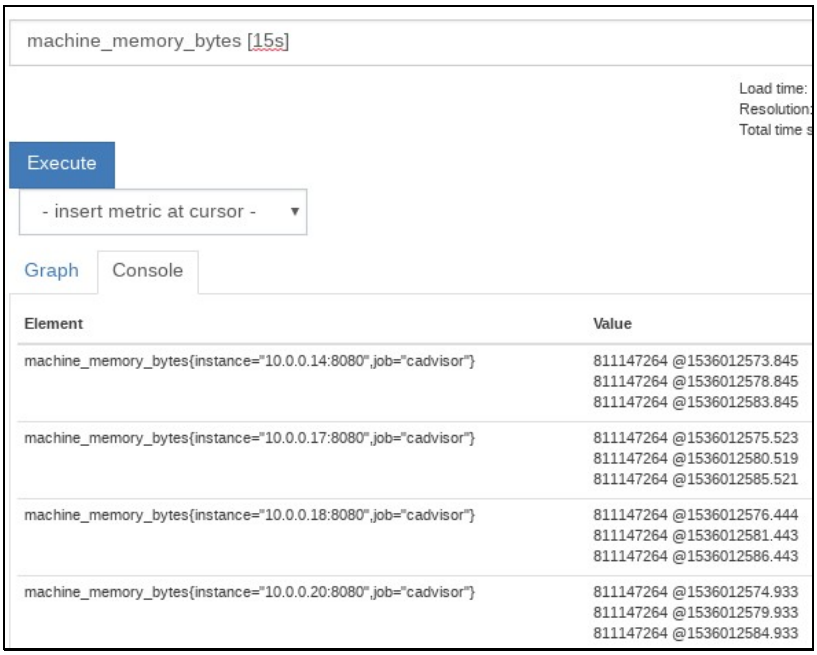
Ha átkapcsolunk a Graph fülre, akkor a Prometheus a Console fülön listázott metrikákhoz ki fog rajzolni egy gráfot, ahol az adott metrika értékeit láthatjuk egy órára visszamen?en, tehát itt nem csak a legutolsó értéket láthatjuk, hanem az utolsó egy óra összes értékét. Mivel a cluster-en még semmilyen valódi szolgáltatás nem fut, ezért a gráf nem túl látványos:



Mivel összesen 4 metrika találat volt az eredeti lekérdezésre, mind a négynek kirajzolja az 1 órás grafikonját (jelenleg a négy vonal teljesen egybe esik, nem a legjobb példa)

Range vektor választó

A range vektor lekérdezésben a metrika alapneve után oda kell írni az intervallumot kapcsos zárójelben, aminek az értékeire kíváncsiak vagyunk. Az eredmény a selector-ra illeszked? összes címke variáns összes értéke a megadott intervallumban csoportosítva metrika nevek szerint. Pl nézzük meg a **machine_memory_bytes** selector-ra illeszked? metrikákat az elmúlt 15 másodperce:



Láthatjuk, hogy mind a 4 megtalált metrikához (a négy cAdviser konténerb?l) 3-3 érték tartozik, mivel 5 másodpercenként mintavételez a Prometheus a konfiguráció alapján. A jobboldali oszlopban láthatjuk a metrika értéke után @-al elválasztva az id?bélyeget.

Fontos függvények és operátorok

rate function

A rate függvény megmutatja range vektorokra az egy másodperce jutó változást: rate(<metrika név>[intervallum hossz])

Els?nek nézzünk egy range vektort:

```
prometheus_http_request_duration_seconds_count{handler="/query",instance="prometheus:9090",job="prometheus"}[10s]
```

Mivel 5 másodpercre állítottuk a Prometheus adatbegy?jtését, ezért 10s-re visszamenve a jelenb?l, két minta lesz benne:

```
139 @1536269070.221
141 @1536269075.221
```

Láthat hogy a 10s ezel?tti begy?jtéskor a request-ek száma 139-volt, de szorgosan kattintgattam az ezt követ? 5 másodpercben, ezért a következ? begy?jtéskor már 141-volt. Vegyük ennek a range értékét, vagyis nézzük meg, hog 1 másodperce mekkora változás jutott:

```
rate(prometheus_http_request_duration_seconds_count{handler="/query",instance="prometheus:9090",job="prometheus"}[10s])
```

Az eredmény 0.5 lesz, tehát a 10s hosszú intervallumban 0.5-öt n?t a számláló másodpercenként.

Ha a **range vektorunk** selector-a nem csak egy metrikára illeszkedik, akkor a **rate** is több eredményt fog visszaadni, minden egyes range vektor találatra egyet.
Tegyük fel, hogy van két counter típusú metrikánk (az alap nevük megegyezik, csak a címkében különböznek)

```
example_metric{type="Y"}
example_metric{type="X"}
```

Nézzük az alábbi range vektort:

```
example_metric[10s]
```

Ennek az eredménye a következ? lesz, ha 5 másodpercenként mintavételezünk:

```
example_metric{type="X"}      1 @1536433370.221
                              2 @1536433375.221

example_metric{type="Y"}      5 @1536433370.221
                              10 @1536433375.221
```

Ha erre alkalmazzuk a **rate** függvényt, két eredményt kapunk:

```
rate(example_metric[10s])

{type="X"}    0.1
{type="Y"}    0.5
```

aggregation operators

Egy instant vektor összes találatára kijött metrika összegét mondja meg. Ezek a metrikák csak címkékben különbözhetnek egymástól, mivel a metrika alap nevét (a címke nélküli részt) kötelez? tejesen megadni. Az instant vektor-os keresés találatai a keresésben nem megadott címkék értékeiben különbözhetnek csak egymástól.

Pl: ha vannak ilyen metrikáim:

```
prometheus_http_request_duration_seconds_count{handler="...",instance="...",job="..."}
```

Ha az instant vektor keresésekor csak az alapnevet adom meg (prometheus_http_request_duration_seconds_count) akkor az összes címke variánst meg fogom találni:

```
prometheus_http_request_duration_seconds_count{handler="/query",instance="prometheus:9090",job="prometheus"}    1
prometheus_http_request_duration_seconds_count{handler="/graph",instance="prometheus:9090",job="prometheus"}    3
prometheus_http_request_duration_seconds_count{handler="/label/:name/values",instance="prometheus:9090",job="prometheus"}       4
prometheus_http_request_duration_seconds_count{handler="/metrics",instance="prometheus:9090",job="prometheus"}  10619
prometheus_http_request_duration_seconds_count{handler="/query",instance="prometheus:9090",job="prometheus"}    168
prometheus_http_request_duration_seconds_count{handler="/static/*filepath",instance="prometheus:9090",job="prometheus"} 8
```

De ha pontosítom a keresést a **handler="/query"** címével, akkor már csak két elem? lesz a találat:

```
prometheus_http_request_duration_seconds_count{handler="/query",instance="prometheus:9090",job="prometheus"}    1
prometheus_http_request_duration_seconds_count{handler="/query",instance="prometheus:9090",job="prometheus"}    168
```

Nam most, az összes aggregation operator ezen instant vektor találatokkal csinál valamit, pl a sum(..) ezen találatok értékét adja össze:

```
sum(prometheus_http_request_duration_seconds_count)    10841
```

Továbbiak:

- sum (calculate sum over dimensions)
- min (select minimum over dimensions)
- max (select maximum over dimensions)
- avg (calculate the average over dimensions)
- count (count number of elements in the vector)
- count_values (count number of elements with the same value)

Aggregation operation eredmény csoportosítása

Ha a by vagy a without kulcssavakat az aggregation operátor lekérdezés mögé írunk, és megadunk ott egy címke listát, akkor az eredmény a címke lista lapján lesz csoportosítva.

```
<aggr-op>([parameter,] <vector expression>) [without|by (<label list>)]
```

by (label list)
Ha a by mögé megadunk egy címkét, akkor az aggregation operator els?ként csoportokat fog képezni azokból a mintákból, ahol a címke(ék) megegyeznek, és azokra fogja végrehajtani az aggregálást. Nézzük az alábbi nagyon egyszer? példát, ahol a metrika alapnév=example_metric:

```
example_metric{job="A", type="X"}  = 1
example_metric{job="A", type="Y"}  = 2
example_metric{job="B", type="X"}  = 4
example_metric{job="B", type="Y"}  = 5
```

Ekkor a sum by nélküli eredménye:

```
sum(example_metric)
Element Value
{}              12
```

De ha hozzáadjuk a by (job)-ot, akkor két választ kapunk, egyet az A sum-ra, egyet a B-sum-ra:

```
sum(example_metric) by (job)
Element Value
{job="A"}      3
{job="B"}      9
```

without (label list)
A without-al pont az ellenkez?jét mondjuk meg, hogy mi szerint ne csoportosítson miel?tt össze adná a csoportok eredményét, tehát minden más szerint csoportosítani fog. A fenti példával ekvivalens eredményt kapunk, ha a **by (job)** helyett **without (type)** -ot írunk.

```
sum(example_metric) without (type)
Element Value
{job="A"}      3
{job="B"}      9
```

Lekérdezés példák

Átlag válaszid? az elmúlt 5 percben
A http_request_duration_seconds hisztogramnak van egy _sum és egy _count metrikája. Vegyük az összes _sum értéket az elmúlt 5 percben [5], majd vegyük ennek a

```
 rate(http_request_duration_seconds_sum[5m])
```

```
/
  rate(http_request_duration_seconds_count[5m])
```

Vizualizáció: Grafana

Ugyan a Prometheus-ban tudunk magunk lekérdezéseket írni, és bizonyos keretek között ezt a Prometheus meg is tudja jeleníteni, produkciós környezetben szükségünk van egy vizualizációs eszközre, ami a mi esetünkben a Grafana lesz. Grafana a piacvezet? Time Series DB vizualizációs eszköz. Out of the box támogatja az elterjedt TSDB-ket:

- Graphite
- Elasticsearch
- InfluxDB
- OpenTSDB
- KairosDB
- Prometheus

Telepítés

Volume plugin használata

A konténerek írható rétegét nem szabad írás intenzíven használni, írás intenzív alkalmazásokhoz (mint amilyen a Granafa is) volume-okta kell használni. Ehhez a már ismert **Netshare** volume plugin-t fogjuk használni nfs protokollal. Ezen felül a grafana konfigurációs mappáját is át fogjuk helyezni az on-demand volume megosztásra.

A Grafan adatbázis a konténeren belül a /**var**/**lib**/**grafana** mappában található. Ezt szimplán mount-olni fogjuk az NFS megosztás **grafana**/**data** mappájába. A konfigurációs állomány a /**etc**/**grafana** mappában van. Ezt fogjuk mount-olni az NFS megosztás grafana/config mappájába, de el?tte át kell oda másolni az /**etc**/**grafana** mappa tartalmát, ugyan úgy ahogy ezt a Prometheus telepítésénél is tettük. Els?ként feltelepítjük standalone docker konténerként, majd kimásoljuk bel?le a konfigurációs mappát:

```
docker run --name grafana \
grafana/grafana:5.2.4

# docker cp -L grafana:/etc/grafana /home/adam/Projects/DockerCourse/persistentstore/grafana/config/
# cd /home/adam/Projects/DockerCourse/persistentstore/grafana/config/grafana/provisioning/
# mv grafana/ config/
# chmod 777 -R config/
```

A használni kívánt datasource-okat (jelen esetben a Prometheus-t) vagy kézzel állítjuk be a Grafana webes konfigurációs felületén, vagy készítünk egy datasource yaml konfigurációs fájlt a /**etc**/**grafana**/**provisioning**/**datasource** mappába, így telepítéskor automatikusan létre fogja hozni a Prometheus adatkapcsolatot.

grafana/provisioning/datasources/datasource.yaml

```
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://prometheus:9090
```

A Grafana is a **monitor** nev? overlay hálózathoz fog csatlakozni, így közvetlenül el tudja érni a Promethues konténert. Mivel közös overlay hálózaton vannak, a Docker DNS fel tudja oldani a szolgáltatás nevét a konténer overlay hálózatbeli IP címére ha olyan rendszer indítja a névfeloldást, aki ugyan azon az overlay hálózaton van.

Ugyan így a **grafana**/**provisioning**/**dashboards**/ mappába el?re hozzáadhatunk dashboard-okat a Grafana-hoz, ami telepítés után azonnal rendelkezésre fog állni.

Service létrehozása

A Grafana-t ugyan úgy a monitor nev? overlay hálózathoz fogjuk csatlakoztatni. Két volume-ot fogunk felcsatolni a Netshare volume plugin segítségével, egyet a konfigurációnak, egyet pedig az adatbázisnak. Publikáljuk az **ingress** hálózatra a 3000-as portot.

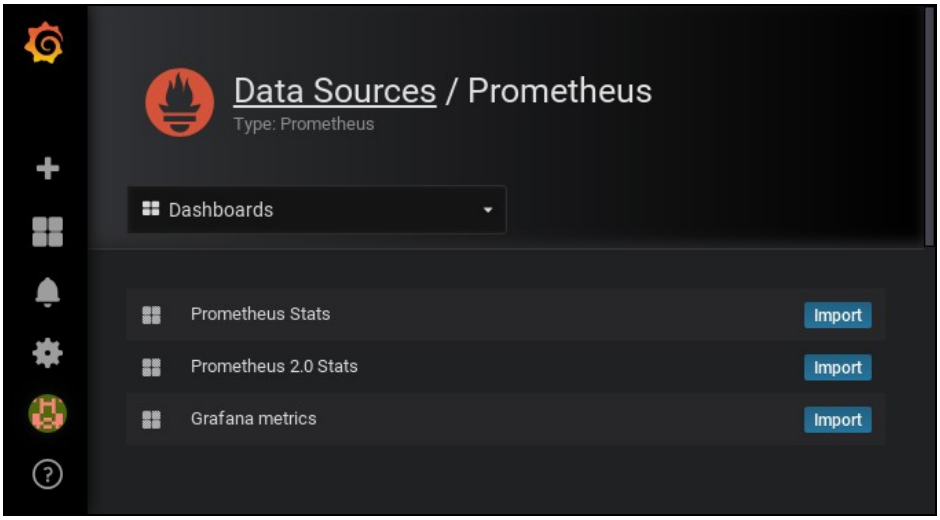
```
docker service create \
--detach=false \
--name grafana \
--network monitor \
--mount "type=volume,src=192.168.42.1/home/adam/Projects/DockerCourse/persistentstore/grafana/config/,dst=/etc/grafana,volume-driver=nfs" \
--mount "type=volume,src=192.168.42.1/home/adam/Projects/DockerCourse/persistentstore/grafana/data/,dst=/var/lib/grafana,volume-driver=nfs" \
-p 3000:3000 \
grafana/grafana:5.2.4

# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE                          PORTS
95918hlvh4u8        grafana             replicated          1/1                 grafana/grafana:5.2.4          *:3000->3000/tcp
cbg79ex3db2g        portainer           replicated          1/1                 portainer/portainer:latest     *:9000->9000/tcp
dzha5bvpjx67        node-exporter       global              4/4                 prom/node-exporter:v0.16.0
x035ni7e3qhi        prometheus          replicated          1/1                 prom/prometheus:v2.3.2         *:9090->9090/tcp
z60yg7cemg7p        cadvisor            global              4/4                 google/cadvisor:v0.28.5
```

Login to Grafana

Mindegy is melyik node-ra került föl, az mg0 IP címével nyissuk meg az el?bb publikált 3000 portot:

http://192.168.123.141:3000/login
A default user/password: **admin/admin**

Miután megadtuk az új jelszót els? belépéskor a settings képerny?n landolunk, ahol megjelenik az el?re hozzáadott Prometheus data source.



Adding dashboards

Terhelés a node-okon

Els?ként generáljuk egy kis forgalmat a node-okon, ehhez a **progrium/stress** docker konténert fogjuk használni.
https://github.com/progrium/docker-stress

```
docker service create \
--detach=false \
--mode global \
--name loadgenerator \
progrium/stress --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 30s
```

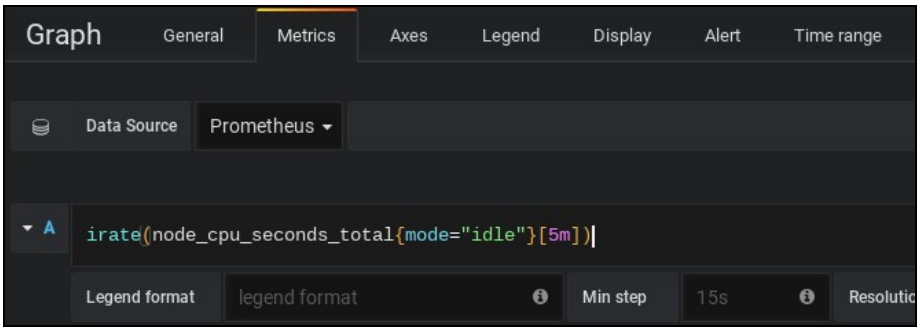
A progrium/stress mindig csak 30s-ig fog futni, de ahogy leáll a swarm újra fogja indítani, tehát pár perc után töröljük:

```
# docker service  rm loadgenerator
```

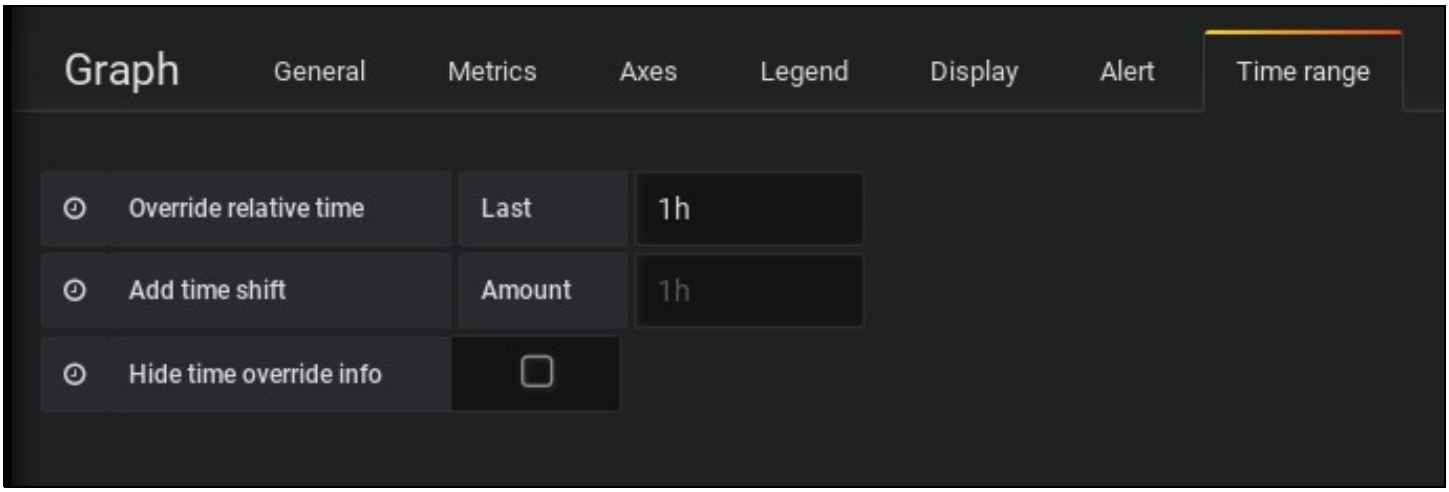
CPU idle grafikon

Menjünk a bal oldali "+" jelre, majd "Dashboard" majd Graph (bal fels? sarok). Data source-nak válasszuk ki a Prometheus-t, majd adjuk meg a következ? lekérdezést:

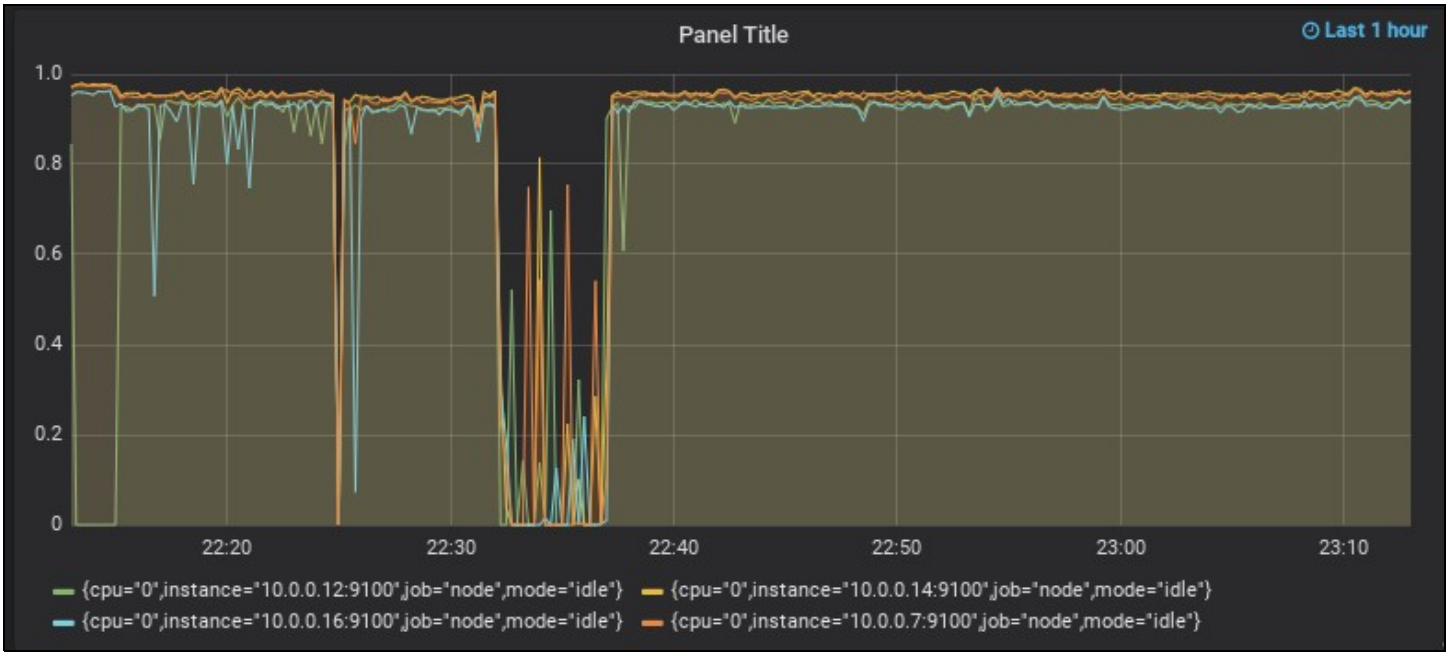
```
irate(node_cpu_seconds_total{mode="idle"}[5m])
```



Ezután a Time range fülön adjuk meg hogy 1 óra legyen a felbontás:



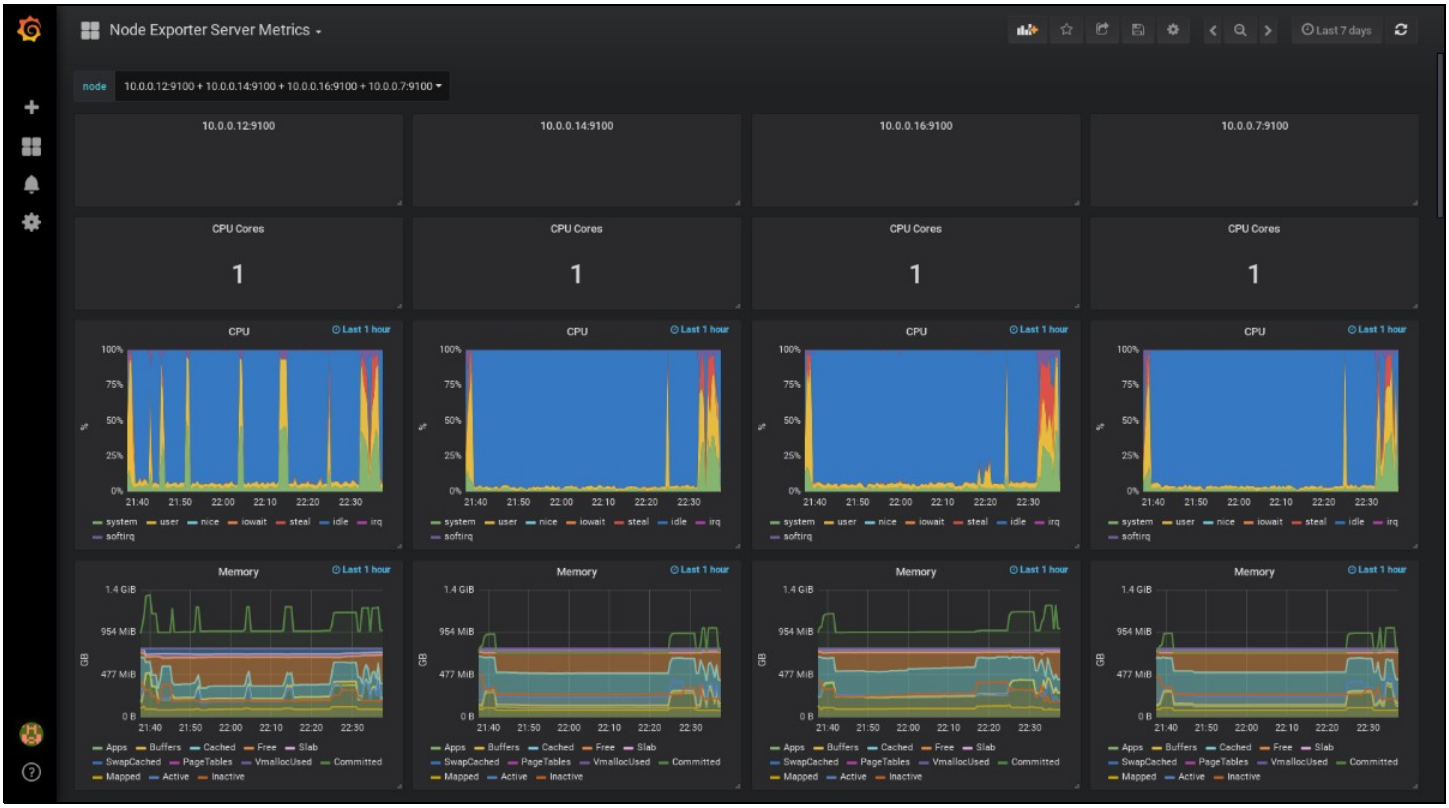
Ekkor mind a 4 node-ra mutatni fogja, hogy a CPU hány százalékban idle:



A jobb fels? sarokban lév? save ikonnal mentsük el.

Node Exporter Server Metrics

https://grafana.com/dashboards/405



Importálhatunk komplett Dashboard-okat, ami el?re van gyártva. A NodeExporter metrikákhoz pl több Dashboard is készült, ilyen pl a **Node Exporter Server Metrics**, ahol az összes node-ot akár egyszerre is láthatjuk. Az a baj, hogy a Node listában nem csak a Node Exporter-ek vannak, hanem az összes hoszt, aki a **monitor** nev? overlay hálózatra csatlakozik. A node exporter-hez ebb?l a hosszú listából csak 4 IP tartozik

Swarm stack

Az egész fentebb leírt architektúrát létrehozhatjuk swarm stack-ként egyetlen egy docker compose fájlal.

compose fájl

docker-compose.yml

```
version: '3'
services:
  cadvisor:
    image: google/cadvisor:v0.28.5
    networks:
      - monitor
    volumes:
      - "/:/rootfs"
      - "/var/run:/var/run"
      - "/sys:/sys"
      - "/var/lib/docker:/var/lib/docker"
    deploy:
      mode: global
      restart_policy:
        condition: on-failure
  node-exporter:
    image: basi/node-exporter:v1.15.0
    networks:
      - monitor
    volumes:
      - "/proc:/host/proc"
      - "/sys:/host/sys"
      - "/:/rootfs"
      - "/etc/hostname:/etc/host_hostname"
    command:
      - "--path.procfs=/host/proc"
      - "--path.sysfs=/host/sys"
      - "--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc)($$|/)"
      - "--collector.textfile.directory=/etc/node-exporter/"
    environment:
      - HOST_HOSTNAME=/etc/host_hostname
    deploy:
      mode: global
      restart_policy:
        condition: on-failure
  prometheus:
    image: prom/prometheus:v2.3.2
    ports:
      - "9090:9090"
    networks:
      - monitor
    volumes:
      - "prometheus-conf:/etc/prometheus"
      - "prometheus-data:/prometheus"
  grafana:
    image: grafana/grafana:5.2.4
```

```
    ports:
      - "3000:3000"
    networks:
      - monitor
    volumes:
      - "grafana-conf:/etc/grafana"
      - "grafana-data:/var/lib/grafana"


networks:
  monitor:
    driver: overlay

volumes:
  prometheus-conf:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/Projects/DockerCourse/persistentstore/prometheus/config
  prometheus-data:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/Projects/DockerCourse/persistentstore/prometheus/data
  grafana-conf:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/Projects/DockerCourse/persistentstore/grafana/config
  grafana-data:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/Projects/DockerCourse/persistentstore/grafana/data
```

Kiemelend?k:

- A **Netshare** NFS volume-okat csak a globális **volumes** szekcióban lehet definiálni, mert csak a globális volumes szekciónak van **driver** és **driver_opts** paramétere. A service definíción belüli **volumes** szekciónak nincs.
- A Netshare 0.35-ös verziójában bevezették a **share** paraméter kezelését. Korábbi verziókat még nem lehetett swarm stack-ben (compose) használni. A globális **volumes** szekcióban a volume forrását elvileg nem kell megadni, az a swarm-ra van bízva, hogy hol hozza létre, ezért nincs neki src paramétere. Az egyetlen erre használható paraméter a **driver_opts** share paramétere, amit a korábbi verziók még nem tudtak kezelni. (Emlékezzünk vissza, hogy a swarm service definícióban ez nem okoz gondot, mert ott a forrást is meg tudtuk adni)
- Mivel a globális **networks** szekcióban nem adtuk meg az external paramétert, a **monitor** nev? overlay hálózatot minden alkalommal létre fogja hozni a swarm stack telepítése el?tt, és le is fogja törölni ha töröljük a stack-et.
- A node-exporter command line paramétereibe köt?jelet kell használni, és nem kell dupla macskakörmöt használni a paraméter értékének megadásakor. A "&" jelet még egy "&" jellel kell escape-elni.

Stack telepítése

A stack-et az alábbi paranccsal hozhatjuk létre.

```
docker stack deploy --compose-file <yaml fájl név> <stack név>
```

A megadott stack nevet minden létrehozott szolgáltatáshoz hozza fogja f?zni prefix-ként, ezzel jelezve, hogy azok egy swarm stack részei. Még a monitor nev? overlay hálózat neve elég is oda fogja rakni a stack nevét.

Legyen a stack neve monitor:

```
# docker stack deploy --compose-file docker-compose.yml monitor
Creating network monitor_monitor
Creating service monitor_prometheus
Creating service monitor_cadvisor
Creating service monitor_node-exporter
Creating service monitor_grafana
```

Ezzel létrejött a monitor_monitor nev? overlay hálózatunk, ezen felül 4 swarm service, szintén a monitor prefix-el:

```
# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
...
nar4kl8o8tat        monitor_monitor     overlay             swarm

# docker stack ls
NAME                SERVICES
monitor             4

# docker service ls
ID                  NAME                    MODE                REPLICAS            IMAGE                          PORTS
fb3poq1m3my0        monitor_cadvisor        global              4/4                 google/cadvisor:v0.28.5
pidw51mgpc0e        monitor_node-exporter   global              4/4                 basi/node-exporter:v1.15.0
pu4z76b6oijq        monitor_grafana         replicated          1/1                 grafana/grafana:5.2.4          *:3000->3000/tcp
terni4ylw5ca        monitor_prometheus      replicated          1/1                 prom/prometheus:v2.3.2         *:9090->9090/tcp
```

Keressük meg az egyik node ingress hálózatbeli címét, hogy tesztelni tudjuk a Prometheus és Grafana konzolt:

```
# docker-machine ssh worker0 ifconfig | grep -A 1 eth0 | grep "inet addr"
          inet addr: 192.168.123.252  Bcast:192.168.123.255  Mask:255.255.255.0
```

Most lépjünk be a Prometheus-ba és a Grafana-ba. Mivel mind a kett? az NFS megosztásból szedi a beállításait és az adatbázisát is, már semmilyen beállításra nincs szükség, ezeket már korábban mind elvégeztük.

- Grafana: http://192.168.123.252:3000
- Prometheus: http://192.168.123.252:9090