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

Felhő alapú alkalmazások automatikus horizontális skálázásának vizsgálata

Diplomaterv

Készítette
Tutkovics András

Konzulens
Dr. Rétvári Gábor Ferenc

2021. december 19.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Tutkovics András*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. december 19.

Tutkovics András
hallgató

Kivonat

Az elmúlt években megfigyelhető egy jelentős szemléletmódváltás a fejlesztett és használt alkalmazások körében. Egyre több informatikai szereplő ismeri fel, hogy milyen előnyöket tud nyújtani a monolitikus alkalmazás cseréje mikroszolgáltatások architektúrára.

A frissen készülő alkalmazások egyre nagyobb része konténerizálva, felhőre optimalizáltan készül. Sokak számára vonzó alternatívát kínál a könnyű és költséghatékony belépésének és a skálázhatóságának köszönhetően. Az így elkészített alkalmazások üzemeltetése is új kihívásokat tartogat. Ezen igényekre jelent meg az azóta folyamatosan fejlődő és növekvő Kubernetes konténer orkesztrációs platform.

A dolgozatomban a mikroszolgáltatások skálázásának kérdéskörét mutatom be Kubernetes platform segítségével. A szükséges alapinformációk miatt ismertetem a Kubernetes felépítését, külön kitérve az általa biztosított automatikus skálázási megoldásokra.

Ezután megvizsgálom az új architektúrán megvalósított alkalmazások kiszolgálási mutatóinak alakulását, egyre növekvő terhelés mellett. A kapott eredmények rámutatnak, hogy automatikus skálázó nélkül ez a konstrukció egy bizonyos terhelés után drasztikus mértékben veszít a kiszolgálási teljesítményéből.

Mérésekkel ellenőrzöm, milyen megoldást biztosít a Kubernetes automatikus horizontális skálázója (HPA) illetve azt, hogy milyen helyzetekben nem tud optimális eredményt adni. Ezen limitációk csökkentésére teszek javaslatot, hogy az új architektúra védettebbé váljon a szűkös erőforrások mellett bekövetkező terhelés növekedésével szemben.

A dolgozatban szereplő állítások alátámasztásához készítettem egy környezetet, amivel tetszőleges szolgáltatásháló definiálása után, terhelés hatására adatokat tudunk gyűjteni és azokat ábrázolni.

Abstract

In recent years, a significant change of attitude has been observed in the development of applications and their usage. More and more IT professionals are recognizing the benefits of changing monolithic applications to a microservice based architecture.

An increasing number of newly created applications are directly containerized, and optimized for the cloud. For many, it offers an attractive alternative due to its cost-effective entry and scalability. The operation of applications created in this way also faces new challenges. The ever-evolving and growing Kubernetes container orchestration platform has emerged to meet these needs.

In my dissertation, I present the issue of scaling microservices using the Kubernetes platform. I will explain the basic information required for Kubernetes, with special reference to the automatic scaling it provides.

I will then examine the evolution of service metrics for applications implemented on the new architecture under increasing load. Based on the results we got without automatic scaling after a certain load this design drastically loses its service performance.

I use measurements to check the solution provided by Kubernetes automatic horizontal scaler (HPA) and the situations in which it cannot give optimal results. I propose a few alternatives to reduce these limits and the new architecture to be more protected by the growing load in and environment with scarce resources.

In support of the statements in the dissertation, I prepared a framework that, after defining any service mesh, generates traffic load, collects data, and displays a representation of the results.

1. fejezet

Bevezetés

1.1. Motiváció

Nagyobb távlatból tekintve az informatikai rendszerekre, számos tendenciát figyelhetünk meg a technológia változásával összhangban. Egy ilyen folyamat, hogy régebben jellemzően dedikált csapat, dedikált nyelven, dedikált alkalmazást, dedikált hardveres erőforrásra fejlesztett éveken keresztül. Ennek az eredményei a mostanra megbonthatatlanul nagyra nőtt monolitikus[4] rendszerek, melyeknek külön szervereket kell biztosítani, hogy azok hiba nélkül tudják kiszolgálni a beérkező igényeket. Több területen használnak még ilyen rendszereket megbízhatóságuk miatt, ugyanakkor továbbfejlesztésük bonyolult és költséges folyamat.

Anyagi megfontolások mentén beláthatóvá vált, hogy saját fizikai szervereket ilyen szolgáltatások számára nem éri meg fenntartani, mivel azok az idő túlnyomó részében nem használják ki a rendelkezésre álló erőforrásokat. További faktorokat jelentettek a folyamatosan fejlődő virtualizációs technológiák, melyek lehetővé teszik, hogy a fizikai rendszert absztrahálva, különböző virtuális környezeteket hozzunk létre azok felett. A megoldásnak köszönhetően lehetőség nyílt előre definiált lépésekkel azonos tulajdonsággal bíró rendszereket létrehozni igény szerint[30]. Természetesen mindezt úgy, hogy a fizikailag rendelkezésre álló erőforrások kihasználtsága is javult.

A korábban vázolt monolitikus architektúrát a mai napig sokan alkalmazzák, azonban folyamatos tolódás figyelhető meg a mikro-szolgáltatások irányába, melynek üzleti okai vannak. Aki nem tud lépést tartani a gyors fejlődéssel, az rövid időn belül lemarad és hátrányba kerül a versenytársaihoz képest. Az érem másik oldala viszont az, hogy aki időben észreveszi a lehetőséget, az hirtelen nagy előnyt tud szerezni. Erre jó példa az Amazon[34] és a Netflix[22] története is.

Egyre szélesebb körben terjednek el a konténerizációs technológiák és velük együtt az úgynevezett mikro-szolgáltatások[4]. A paradigma értelmében a korábbi nagyobb kódegységet szét lehet bontani több kisebb kódbázisra, ami számtalan előnyt jelent az alkalmazás fejlesztésekor. Mivel a kisebb egységeket könnyebb megérteni mint a teljes kódbázist, az új fejlesztők számára könnyebb becsatlakozni a fejlesztési folyamatokba. A kisebb elemekre bontott kód lehetőséget biztosít arra is, hogy az egyes komponenseket több nyelven és többfajta keretrendszerben fejlesszük, minden komponens számára az optimális fejlesztési környezetet kiválasztva. Végül, ha egy komponens teljes körű felülvizsgálatára van szükség, az nem érinti a rendszer többi részét, amíg a publikus interfészek változatlanok maradnak. Ezek által lehetőség nyílik egy kellően agilis rendszer kialakítására, ami könnyebben és gyorsabban tudja kiszolgálni az ügyfelek igényeit, ami üzleti előnyt jelent.

Ez a modern paradigma tartalmaz néhány kihívást is, ami a monolitikus alkalmazások esetében nem merültek fel. Egy ilyen kihívás kerül tárgyalásra a dolgozatban is. A tendenciaváltás legnagyobb szereplője a Kubernetes[15], ami lehetőséget biztosít a konténerizációs technológiák egyszerű és széles körű használatára.

1.2. Feladat meghatározása

A korábban említett paradigmaváltással lépést kell tartani az alkalmazás üzemeltetésekor is. Fontos szempont az elkészült, mikro-szolgáltatásokból felépülő hálózat skálázásának kérdése. Sokan választották ezt a modern megközelítést, mert ígérete szerint könnyen képes a beérkező kérések okozta terheléssel arányosan skálázódni. Ez egy triviális feladat egészen addig, amíg az egyes egységeket különálló, másokkal nem összefüggő részként kezeljük. Ezt megtehetjük és látszólag legtöbben meg is teszik komolyabb fennakadások nélkül, azonban a valóság ennél jóval komplexebb. Figyelembe kell venni, hogyan kapcsolódnak egymáshoz a mikro-szolgáltatások, azok milyen és mennyi erőforrást használnak, mi történik a beérkező kérésekkel. A sok szolgáltatás közül melyiket érdemes először skálázni, melyik okozza a kiszolgálások lassulását, milyen módosítással lehet a legtöbb felhasználót kiszolgálni. Látható, hogy pár paramétert bevéve az egyenletbe a komplexitás meredeken emelkedik. Különösen megnehezíti a feladatot, ha a rendelkezésre álló erőforrásokat (például: processzor, memória, hálózat, I/O) globálisan optimálisan szeretnénk elosztani, hogy a lehető legjobb kiszolgálást tudja biztosítani a rendszer.

A diplomamunka során ezt a kérdéskört kellett megismernem és a felmerülő kérdésekre választ keresni. Első lépésben szeretném bemutatni a használt környezetet, a Kubernetes felépítését illetve, hogy milyen skálázási megoldások léteznek és

melyiket hogyan támogatja. Konkrét példán keresztül bemutatom az automatikus pod horizontális skálázó megoldását.

Egy egyedi keretrendszert kellett összeállítani, ami képes tetszőleges szolgáltatáshálót létrehozni Kubernetesen belül és azt lekérdezésekkel terhelni. Ezáltal lehetőségünk nyílik szabadon konfigurálható körülmények között vizsgálni a Kubernetes automatikus horizontális skálázóját, illetve az irodalomban javasolt egyéb megoldásokat is. Az így nyert eredményekből következtetéseket tudunk tenni az egyes implementációk dinamikáját, költségeit, hatékonyságát illetően.

A feladat része a kapott mérési eredmények vizualizációja is, így könnyebben láthatóvá válnak a skálázási megoldások és a megoldások közötti különbségek is.

Az elkészített keretrendszer segítségével, mérési eredményekkel kell bemutatni a Kubernetes felett történő mikroszolgáltatások kommunikációját és figyelni hogyan változik a terhelés függvényében. Meg kell ismerni, hogy a beépített skálázó működése milyen megoldásokkal jár és milyen helyzetben ad az optimálistól eltérő kiszolgálást. Ilyen helyzetekre megoldási javaslatokat kell keresni és bemutatni azok működéseit.

1.3. A dolgozat felépítése

A dolgozat fejezetei úgy lettek sorba állítva, hogy azok a lehető legkövethetőbb módon mutassák be a kapott eredményeket. Ehhez viszont az általános részektől kell indulni és így mutatva be az egyre konkrétabb részleteket. Először szeretném ismertetni a Kubernetes platform architektúráját és az általa támogatott automatikus skálázást a 2. fejezetben. A munka kezdetén el kellett olvasni a témában készült korábbi kutatásokat, amelyeket a 3. fejezetben ismertetek. Ezután a 4. fejezetben bemutatom az elkészített keretrendszert, annak építő elemeit és a megvalósítás során hozott döntéseket. Az elvégzett mérések kerülnek ismertetésre az 5. fejezetben, beleértve a kapott eredmények értelmezését is. A mérési eredmények alapján pár megoldási javaslatot teszek a 6. fejezetben, vázolva az előnyeiket és hátrányaikat. Végül a 7. fejezetben összefoglalom a diplomamunkám eredményeit és kitérek az újonnan felmerült kérdésekre, továbbhaladási irányokra.

2. fejezet

Kubernetes

A diplomamunka keretén belül végzett feladataim jelentős részben támaszkodnak a Kubernetes (*K8s*) rendszerre, így annak alapvető ismertetése szükségszerű a dolgozat további megértéséhez. Egy rendkívül széleskörű platformról van szó, szerteágazó felhasználási területekkel, emiatt a rendszer teljes és mély ismertetésére helyett próbáltam a dolgozat szemszögéből ténylegesen szükséges információk körére szorítkozni.

2.1. Kubernetes motivációja és rövid története

Ahogy egyre többen kezdték megismerni és alkalmazni a konténerizációs technológiákat, úgy vált egyre hangsúlyosabbá az üzemeltetési oldalon is. A korábbi kihívások átalakultak a technológia változásával együtt. Ezek után már nem az egyes szerverek vagy virtuális gépek (*VM*) adminisztrációs terhe jelentette a kihívást, hanem annak a számos mozgó alkatrésznek a kezelése, amelyből egy modern alkalmazás felépült. Ugyanis a részegységekre bontásnak a következményeként sok kis konténert kell egyszerre felügyelni, ami jelentősen több feladattal jár, mint a korábbi VM alapú, monolitikus rendszerek kezelése esetén.

Ez a probléma motiválta a Google szakembereit is, hogy egy olyan új platformot fejlesszenek, ami képes a fenti kihívások megoldására. Eleinte a projekt a *Borg*[6] nevet viselte, amit aztán 2014-ben a Google nyílt forráskódúvá tett Kubernetes néven. A projektet a *Cloud Native Computing Foundation (CNCF)*[7] vette gondozásába. Innentől kezdve bárki szabadon elérheti és fejlesztheti is bele. Az elmúlt 6 év alatt hatalmas fejlődésen ment keresztül és már a 22.-ik kiadásánál tart.

Egészen fiatal rendszerről van tehát szó, azonban a VMware kutatásából[35] is sok érdekes dolog derül ki. Egyre többen térnek át a Kubrenetesre és futtatják benne a konténerizált alkalmazásaikat. Látható, hogy nem egy rövidtávon elmúló

trendről van szó és még mindig felívelő ágban van, a korábbi alternatívák fokozott kiszorításával. A kutatásban résztvevők jelentős része számolt be hatékonyság növekedésről, főként az erőforrás gazdálkodásban és a fejlesztési ciklusok rövidítésének tekintetében.

Ma már gyakorlatilag a Kubernetes a konténer orkesztráció szinonimája és egyeduralkodó a területén, azonban ez nem mindig volt így. Az alapvető problémát, hogy meg kell könnyíteni a konténerek kezelését sokan felismerték és több megoldás is született rá. A különböző eszközök és fejlesztések közötti versenyt konténer orkesztrációs háborúnak is nevezik[8], melynek egyedüli győzteseként a Kubernetes került ki.

2.2. Felépítése

2.2.1. Architektúra

Kubernetes alatt általában egy teljes klasztert szoktak érteni, ami sokrétű problémára jelent megoldást. Feladatai közé tartozik a konténerek elindítása, azok felügyelete, kezelése. Ezenfelül kezeli a konténerek közti hálózatot, irányítja a forgalmat. A feladatok ellátásában több különböző egység vesz részt, saját hatáskörökkel. Az alábbi fejezetben szeretném ezeket az egységeket bemutatni és átfogó képet adni a Kubernetes működéséről.

2.2.1.1. Csomópontok

A legtöbb klaszter több csomópontból (*node*) épül fel, mely egy teljes fizikai szervert vagy egy virtuális gépet jelent a gyakorlatban. Itt fognak futni a később részletesen tárgyalt kapszulák (*pod*[1]), mely a Kubernetes legkisebb egységei és egy vagy több konténert tartalmazhatnak. Ezért itt szükség van azoknak a szolgáltatásoknak, amik ezt lehetővé fogják tenni. A csomópontok konfigurációját a vezérlő sík (*control plane*) végzi, melyről szintén részletesen lesz szó a 2.2.1.2 alfejezetben. A 2.1 sorszámmal ellátott kódrészleten láthatjuk, hogy az általunk későbbiekben használt klaszter esetében milyen csomópontok vannak. Fontos, hogy az egyes node nevek különbözőek legyenek.

Feladatkörét tekintve két különböző típusú node lehet. Egyik, ami a vezérlő síkot biztosítja, az úgynevezett *master* illetve a másik, ahol csak kapszulák

[1]A diplomamunka során felváltva használom az angol és magyar nyelvű kifejezéseket, hogy kevésbé legyen szóismétlő; illetve az angol megfelelő a beszélt közegben gyakoribb, viszont írásban a ragozása nehéz feladat.

futhatnak, ezek a *worker* típusú csomópontok. Látható, hogy a példaként hozott kódrészletben a `dipterv1` névvel ellátott node van mester szerepben. Az utóbbi időben egyre jobban elmosódott a határvonal a két típus között. Ajánlottan a master komponensek külön, dedikált csomópontokon futnak, de szabadon konfigurálható és akár futtathatunk alkalmazás specifikus kapszulákat is rajtuk.

```
$ kubectl get nodes
NAME       STATUS   ROLES                  AGE   VERSION
dipterv1   Ready    control-plane,master   236d   v1.21.0
dipterv2   Ready    <none>                 236d   v1.21.0
dipterv3   Ready    <none>                 236d   v1.21.0
```

2.1. kódrészlet. Később használt klaszter csomópontjai

A kapszulák futtatásához a Kubernetesnek lehetővé kell tenni a podok indítását az egyes csomópontokon és a közöttük lévő kommunikációt. Ezért minden egyes node rendelkezik az alábbi komponensekkel:

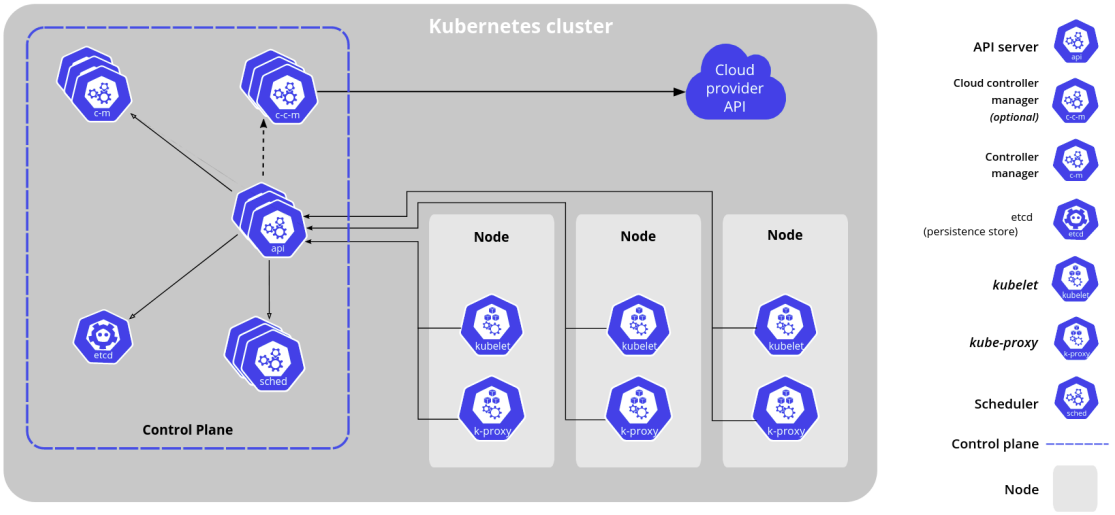
Kubelet Minden node rendelkezik egy *kublet* nevű ügynökkel (*agent*), mely feladata az adott csomópontra ütemezett konténerek rend szerinti futtatása. A futtatandó parancs a vezérlő sík felől érkezhet (a Kubernetes API szerverén keresztül), egy úgynevezett pod leíró kíséretében, amelyben lévő definíció alapján kezeli a *kubelet* a kapszulákat. Ez az alsóbb szinteken azt jelenti, hogy elindítja a szükséges konténereket a csomóponton. Ezen felül feladata, hogy az általa összegyűjtött információkat jelentse a vezérlő sík felé, ezzel lehetővé téve az új konténerek ütemezési döntését a csomópontokra.

Kube-proxy A *kube-proxy* szintén része minden csomópontnak, ahogy az a 2.1 ábrán is látszik. Ezen komponens feladata az adott node hálózatát karbantartani. Ezt úgy tudja megtenni, hogy operációs rendszer szinten hoz létre és kezeli a hálózati csomagokra vonatkozó szabályokat.

Konténert futtató környezet Mivel az egyes kapszulák konténereket tartalmaznak ezért szükség van egy olyan környezetre, ami képes futtatni ezeket a konténereket. Mai napig a legelterjedtebb a Docker, de a Kubernetes ezen kívül még több másikat is támogat.

2.2.1.2. Vezérlő sík

A Kubernetesen belül a vezérlő sík felelős a klaszteren belüli döntésekért. Feladata például meghatározni, hogy az újonnan létrehozandó kapszulák melyik cso-



2.1. ábra. Kubernetes klaszter elemei[13]

móponton induljanak el. A 2.1 ábrán is láthatjuk a vezérlő síkot kék szaggatott vonallal keretezve.

API szerver A klaszter központi egysége. Minden művelet rajta keresztül történik. Felelőssége a beérkező kérések hitelesítése és továbbítása a megfelelő rendszerelemek felé. Például a klaszter adminisztrációra használható `kubectl` parancs használatakor is a parancssori kliens összeállít és elküld egy üzenetet az API szerver felé, ami fogadja és válaszol rá.

Etcd Az *etcd* komponens is része a vezérlő síknak, ami egy elosztott módon működő[10] kulcs-érték párokat tartalmazó adatbázist valósít meg. Itt tárolja a klaszter összes saját tulajdonságát, például az egyes objektumok állapotait is. Ezáltal a korábbi példaként említett `kubectl` lekérdezés is ebből az adatbázisból kiolvasott értékeket fogja válaszul megkapni.

Ütemező Az ütemező (*kube-scheduler*) feladata, meghozni a döntést, hogy egy új pod melyik csomóponton kerüljön létrehozásra. Ez egy igazán izgalmas feladat, hiszen figyelembe kell vennie a rendszer jelenlegi foglaltságát, illetve a kapszula létrehozásakor külön meg lehet adni megkötéseket a node felé. Ilyen megkötés vonatkozhat a csomópont szoftverére vagy hardverére is.

2.2.2. Objektumok

A Kubernetes egyik erőssége, hogy az átlagos felhasználás esetében nem kell törődni a rendszer felépítésével, hiszen elég deklaratív módon létrehozni az erőfor-

rásokat, objektumokat. Minden egyéb feladatot a Kubernetes a felhasználók elől elrejtve hajt végre.

Pod (kapszula) A Kubernetesben megjelenő legkisebb logikai egység. Általában egy konténert tartalmaz, de lehetőségünk van több konténer részletes specifikálására is. Általánosságban elmondható, hogy rendszeresen létrejönnek és rendszeresen törölve is lesznek ezek az objektumok, tehát nem tartós életűek. Ezen tulajdonság miatt a rendszer egészére is célszerű egy dinamikusan változó környezetként tekinteni. Az egyszerű pod kezelése nehéz, mert az állandó változása miatt nincs statikus elérése, nem képes magát skálázni vagy érzékelni az esetleges hibákat. Ezen problémákra később látjuk a Kubernetes által nyújtott megoldásokat.

ReplicaSet *ReplicaSet* felelős az általa kezelt kapszulák számának monitorozásáért. Ennek értelmében, ha az egyik pod meghibásodik és megáll, akkor helyette egy újat fog létrehozni. Ezáltal biztosított, hogy mindig a megfelelő számú egység fogja fogadni a beérkező kéréseket és nem kell manuálisan monitorozni a státuszukat. A *ReplicaSet* megoldást nyújt a podok öngyógyulására azáltal, hogy újraindítja a kapszulákat egy esetleges hiba esetén, illetve a megadott replikaszámmal képes azokat skálázni.

Deployment Mivel *kapszulák* túl kicsi részei a teljes alkalmazásnak és életük sem kiszámítható ezért nem kifizetődő ilyen módon kezelni a rendszerünket. Erre találták ki a *Deplomen*t objektumot, ahol meg tudjuk adni, hogy milyen *Podok* jöjjenek létre, illetve beállíthatjuk a hozzájuk kapcsolódó *ReplicaSet* értéket is. Ezáltal lehetőségünk van absztrakt szinten, deklaratív módon megadni a kívánt rendszer tulajdonságait.

Service A korábban említett objektumokkal már meg tudunk valósítani bizonyos funkciókat, viszont ezt szeretnénk a klaszteren belül és kívülre is elérhetővé tenni. Erre találták ki *Service* objektumot. Ezen keresztül könnyen el tudjuk érni az azonos szolgáltatást nyújtó *kapszulákat* és nem kell az alkalmazás logikában számontartani az ő elérhetőségeiket. Ez ugye különösen nehéz feladat lenne, hiszen a készenléti idejük is elég változó lehet, mivel folyamatosan jönnek létre és törlődnek.

Custom Resources (CR) A Kubernetes API lehetőséget biztosít számunkra, hogy tetszőlegesen kibővítsük. Így lehetőségünk van saját objektumokat is létrehozni, illetve ahhoz saját kontroll-logikát más néven kontrollert írni. Ehhez először egy

saját erőforrás leírót kell létrehozni (Custom Resource Definiton - CRD), ami tartalmazza az általunk kívánt erőforrás definícióját. Ezzel lehetőséget kapunk, hogy tetszőlegesen komplex leírásokat hozzunk létre és azt egy logika deklaratív módon kezelje.

2.2.3. Erőforrások

Lehetőségünk van a konténerek erőforrás használatához különféle megkötéseket tenni. Megadhatunk olyan szabályokat, amik maximalizálják a konténer számára használható erőforrásokat. Ezen szabályokkal elérhetjük, hogy egy esetlegesen hibásan implementált vagy kártékony alkalmazás a klaszter működésére nagymértékben kihasson az erőforrások mohó használatával. Másik jellegű szabályozás, amikor előre lefoglaljuk az alkalmazás által igényelt minimális erőforrásokat. Kritikus alkalmazások esetében hasznos lehet, mivel ebben az esetben garantálni tudjuk az erőforrások rendelkezésre állását a rendszer többi részétől függetlenül. Ezeket `request` és `limit` értékeknek nevezzük. Az igényelt erőforrásnál használhat a futó konténer többet is, azonban ez már függ a többi konténertől is. Limitáció esetén viszont csak a megadott mennyiség áll a rendelkezésére. Fontos jól megválasztani az értékét, hiszen túl alacsony memóriahasználat mellett lehet, hogy el sem tud indulni a konténer és hibát fog jelezni vagy az alacsonyra állított processzorhasználat miatt az alkalmazásunk lassulása tapasztalható.

Leginkább a processzor- és memóriahasználatra szoktak ilyen megkötéseket létrehozni, de lehetőség van más erőforrásokat is kezelni. A konténerek és podok létrehozásakor a *kubelet* ütemezője figyelembe fogja venni a megadott paramétereket és ez alapján választja ki, hogy melyik csomóponton fog futni.

2.3. Skálázás

A felhő alapú infrastruktúra egyik legjelentősebb előnye, hogy az alkalmazásunk képes adaptálódni a külvilág felől érkező kérésekhez. Ez azt jelenti, hogy ha több felhasználót kell egyszerre kiszolgálni, akkor a rendszer automatikusan növeli a kiszolgálásra fordított erőforrások mennyiségét. Ezzel a megoldással elérhetjük, hogy a végfelhasználó ne vegyen észre minőségbeli csökkenést és a kevésbé intenzív időkben pedig nem foglalunk feleslegesen erőforrást, ami az üzemeltetőnek is jól belátható anyagi érdeke.

Alapvetően két különböző skálázási módszert lehet elkülöníteni. Az egyik a vertikális, míg a másik a horizontális skálázás. Ezekről a későbbiekben bővebben lesz szó.

2.3.1. Horizontális skálázás

A két skálázási mód közötti különbséget mutatja a 2.2. ábra. Jobb oldalon látható megoldás az úgynevezett horizontális skálázás (másik nevén: scaling out). Ebben az esetben a megnövekedett igények kiszolgálásához több azonos egységet hozunk létre. Az összes egység azonos erőforrás felhasználással rendelkezik és a beérkező kérések köztük kerülnek szétosztásra. A megoldás egyik előnye, hogy könnyű alkalmazni és a Kubernetes rendszere is alapértelmezettként támogatja. Hátránya abból fakadhat, hogy a több fogadó egység miatt azonos felhasználótól érkező forgalom más egységnél kerülhet feldolgozásra, amire bizonyos alkalmazások esetén külön figyelni kell.

2.2. ábra. Vertikális és horizontális skálázás

2.3.1.1. Automatikus horizontális skálázás

Egy egyszerű példán keresztül szeretném bemutatni a Kubernetes beépített, automatikus horizontális skálázóját (HPA). Az automatikus skálázónak meg lehet adni, hogy milyen célértéket szeretnénk kapni. Például, hogy a futtatott pod által felhasználható CPU mennyiség milyen szinten legyen kihasználva. Jelenleg ilyen megkötést a memória és processzor felhasználásra lehet tenni, de tetszőlegesen létrehozhatunk saját metrikát is.

Az algoritmus folyamatosan lekérdezi a metrikák aktuális értékét és a 2.1 egyenlet alapján meghatározza az éppen szükséges replika számot. Ezzel a számított ér-

tékkel frissíti a pod replika számát, amit így a replikációért felelős kontroller észlel és megpróbálja elérni a kívánt állapotot. Ezzel módszerrel megvalósítható fel- és leskálázás is.

$$desiredReplicas = \left\lceil currentReplicas * \frac{currentMetricValue}{desiredMetricValue} \right\rceil \tag{2.1}$$

A folyamat szemléletesebb, ha konkrét példán nézzük meg működését. Ehhez először létre kellett hozni egy alkalmazást, amit tudunk majd skálázni. Ehhez egy egyszerű webszervert használtam, ami minden beérkezett kérés esetén egy CPU intenzív műveletet hajt végre. Miután létrejött a szükséges *Deployment* és *Service*, utána létre lehetett hozni az automatikus skálázót. Ennek a forráskódja látható a 2.2 kódrészlet tetején. Be lehet állítani, hogy mi legyen a minimális és maximális replika, ami között lehetősége van skálázni. Továbbá definiálni kell egy célértéket is, ami alapján a skálázási döntéseket meg tudja hozni. A példában látható, hogy 50%-os CPU felhasználást szeretnénk elérni. Fontos, hogy alapvetően a metrikákat a skálázó egy úgynevezett metrika szervertől gyűjti be, amit külön el kell indítani, mert alapértelmezettként nem fut a Kubernetesben.

```
$ cat hpa/hpa.yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50

$ kubectl apply -f hpa/hpa.yaml
$ while sleep 0.01; do wget -q -O- http://localhost:30100; done

$ kubectl get horizontalpodautoscalers
NAME        REFERENCE              TARGETS   MINPODS  MAXPODS  REPLICAS  AGE
php-apache  Deployment/php-apache  179%/50%  1        10       1         4m53s
...
php-apache  Deployment/php-apache  81%/50%   1        10       4         6m35s
...
php-apache  Deployment/php-apache  42%/50%   1        10       7         10m
```

2.2. kódrészlet. Automatikus horizontális skálázás folyamata

A skálázó létrehozását megtehetjük a kódrészleten látható második parancs kiadásával. Az erőforrás létrejötte után ki is lehet próbálni. Ehhez egy `bash` szkript segítségével állandó forgalmat generálunk és figyeljük, hogyan változik a kapszulák száma és ezzel összefüggően az egyes egységek CPU felhasználása. Kezdetben 1

darab pod végezte az összes beérkező kérés kiszolgálását, ami így a célértéknél 3-4-szer több processzort használt. A korábban mutatott képlet alapján, a felső egész részt vesszük és a jelenlegi replikaszám 4-szeresére skálázunk.

2.3.2. Vertikális skálázás

A skálázási megoldások közül a másik megoldás a 2.2. ábra bal oldalán látható. Ezt vertikális skálázásnak (másik nevén: scaling up) hívnak. Ebben az esetben a kiszolgáló egységek számát nem módosítjuk, hanem az általuk felhasználható erőforrások mennyiségét növeljük. Ilyen példa, amikor plusz memóriát rakunk a számítógépbe, vagy erősebb processzorra cseréljük a meglévőt.

Előnye a megoldásnak, hogy a korábbi félévek munkái alatt azt figyeltük meg, hogy a vizsgált alkalmazásaink ezzel a stratégiával azonos erőforrás felhasználás mellett jobb eredményeket értek el.[31]

Hátránya, hogy a jelenlegi implementáció szerint minden skálázásnál le kell állítani a futtatott egységet, ami bizonyos szolgáltatás esetén nem túl előnyös, hiszen ezen idő alatt kevesebb egység végzi a beérkező kérések kiszolgálását, illetve azt is jelenti, hogy legalább két pod futása szükséges hozzá.

2.4. Szolgáltatás minőségi osztályok

Következőkben szeretném bemutatni a Kubernetes klaszteren belül megjelenő szolgáltatásminőségi osztályokat, illetve azok működését. Három különböző minőségi osztály létezik jelenleg a Kubernetes implementációjában[14]. Ezen osztályok és hatásaik ismerete hasznos tudást biztosítanak az egyes mérések megtervezéséhez és a kapott eredmények értelmezéséhez.

A Kubernetes minden egyes kapszulához rendel egy minőségi osztályt, amit a beállított erőforrás specifikációkból fog meghatározni. Tehát, az egyes podok besorolását csak implicit módon lehet definiálni. Az ütemező számára hasznos információt fog nyújtani, amikor dönteni kell az egyes podok csomóponthoz történő rendeléséről vagy megszüntetéséről.

A kapszulán belüli konkrét osztály meghatározását a 2.3 kódrészleten látható módon lehet megtenni. A *kubectl* alkalmazás *get* és *describe* parancsával is lekérdezhető, de én az előbbit választottam, mert legtöbb esetben bővebb kimenetet kaphatunk, mint a *describe* paranccsal, ugyanis ilyenkor megkapunk minden információt, amit az adott erőforrással kapcsolatban a Kubernetes az *etcd*-ben tárol.

A kimeneten látszik, hogy az érintett erőforrás egy pod, amiben egy darab konténer fut. A kimenet szükségtelen részét töröltem és csak a számunkra releváns részt hagytam meg. Látható, hogy egyedül a memória értékre van megkötés, hogy mi legyen a konténer számára maximálisan használható memória mennyiség, ami a példában 170 megabájt. Ezzel szemben a pod létrehozásánál létezik megkötés, hogy milyen processzor és memória mennyiségeket szeretne közvetlenül lefoglalni és fixen saját használat alatt tartani. Könnyen leolvasható, hogy ez a memória esetén 70 megabájt és 100 mCPU processzort jelent. A számunkra másik fontos részlet már a *status* alatt található. Itt azok az információk szerepelnek, amiket nem mi definiáltunk és adtunk meg a létrehozás előtt, hanem az erőforrás működése közben változtak. Többek között ilyenek például, hogy mikor indult el a kapszula, hányszor kellett újraindítani, milyen belső IP címen érhető el illetve a számunka fontos *qosClass* érték is. Látható, hogy a konkrét példában ez "*Burstable*" értéket kapott.

```
$ kubectl get  pod -n kube-system coredns-558bd4d5db-h8l7z -o yaml

apiVersion: v1
kind: Pod
...
spec:
  containers:
  - ...
    resources:
      limits:
        memory: 170Mi
      requests:
        cpu: 100m
        memory: 70Mi
    ...
status:
  ...
  qosClass: Burstable
  ...
```

2.3. kódrészlet. Adott kapszula minőségosztályának vizsgálata

A következő alfejezetekben részletesen bemutatom, az egyes osztályok tulajdonságait, és mi alapján lesznek a kapszulák osztályozva. Láthatjuk, hogy milyen módon van lehetőségünk az egyes alkalmazások között priorizálni, ezt a Kubernetes hogyan fogja kezelni. Mindezzel együtt érthetővé válik a korábbi példán látott besorolás is.

2.5. Garantált minőségű osztály

A garantált (*quaranteed*) minőségű szolgáltatás osztály élvezi a legtöbb előnyét a Kubernetes ütemezőnek. Akkor kerülhet ebbe az osztályba egy adott pod, ha több kritériumnak is megfelel. A kapszulán belül, minden egyes konténerre érvényesnek

kell legyenek, az alábbi megkötések. Ezek vonatkoznak a rendes alkalmazás konténerre és az init konténerekre is.

- Konténeren belül meg van adva az igényelt és maximálisan használható processzor mennyisége.
- Konténeren belül meg van adva az igényelt és maximálisan használható memória mennyisége.
- Az igényelt erőforrások és a maximálisan használható erőforrások értékei megegyeznek.

A fenti kritériumokhoz kapcsolódóan fontos megjegyezni, azt az érdekes esetet, ami akkor áll elő, ha egy konténerhez mindössze az erőforrások limitációját adjuk meg. Ebben az esetben automatikusan kerül beállításra az igényelt erőforrás értéke, ami megegyezik a limitációval. Emiatt hiába nem adunk meg expliciten értéket az igényelt erőforrásokhoz, de kitöltjük a limit értékeket, akkor is garantált szolgáltatás osztályba fog kerülni az adott pod.

Ezen kapszulák olyan csomópontokra kerülnek ütemezésre, amik rendelkeznek elegendő erőforrással, illetve a bennük futtatott konténerek lesznek a kevés erőforrás miatt utoljára leállítva.

2.6. Börsztölhető minőségű osztály

A börsztölhető (*burstable*) kapszulák besorolását elég széleskörűen lehet meghatározni. Leginkább azt lehet mondani, hogy ide tartoznak azok a podok, amik valami miatt nem teljesítik a garantált szolgáltatás osztályba tartozó kritériumokat, azonban valamilyen erőforrás foglalás meg van adva. Tipikusan ilyen szituáció, amikor különböző értékek vannak megadva az igényelt és a maximálisan használható erőforrások mennyisége között. Akkor is ide kerül besorolásra egy kapszula, ha több konténer közül valamelyik nem teljesíti a garantált osztály elvárásait.

Ezen podokat igyekszik az ütemező a rendelkezésre álló információi alapján a számukra legjobb csomópontra osztani, de nem garantálható ennek a sikere. Ez abból fakad, hogy legtöbb esetben kevesebb információval rendelkezik róluk, mint a garantált osztályban, ahol minden érték adott volt.

Amikor elfogynak a csomópont által használható erőforrások és nincsen több legjobb szándék minőségű osztályba tartozó pod, akkor ezen podok kerülnek leállításra. Ezzel is védve a garantált osztályú kapszulákat.

2.7. Legjobb szándék minőségű osztály

Legalacsonyabb prioritási szinttel a legjobb szándék (*best effort*) minőségi osztály rendelkezik a felsoroltak közül, amire már a neve is utal. Ide kerülnek az olyan kapszulák, ahol semelyik konténernek nincsen beállítva erőforrásra vonatkozó információ. Tehát nincs megkötés a maximális használatra és nincs megkötés a minimálisan szükséges erőforrásra sem.

Ebben az esetben rendelkezik az ütemező a legkevesebb információval a kapszuláról, ami miatt nem is garantálható, hogy olyan csomóponton kerül elindításra, ahol elegendő erőforrás áll az alkalmazás rendelkezésére.

Ebbe a kategóriába tartozó podok annyi erőforrást használhatnak, amennyi rendelkezésre áll az adott csomóponton. Emiatt felmerül a veszélye, hogy a többi alkalmazás elől fogja elhasználni a szükséges erőforrásokat. Ezzel magyarázható, hogy az osztályba tartozó podok lesznek először leállítva, amikor fogytán van a csomóponton elérhető erőforrások mennyisége.

A fentebb írtak miatt körültekintően kell eljárni az erőforrások szabályozása közben, hiszen ezáltal lehetőségünk van priorizálni az alkalmazásainkat.

3. fejezet

Irodalomkutatás

A diplomamunka során több publikációt is elolvastam, hogy tisztább legyen a kutatási terület. A következőben szeretném összefoglalva bemutatni az eddigi kutatások által vizsgált irányokat.

Több oldalról meg lehet fogni a skálázás és a szorosan hozzá köthető erőforrás elosztás területét. Minden kutatás kicsit más szempontból vizsgálja, más áll a középpontban.

A legátfogóbb ismertető a témában a *Cloud resource management: A survey on forecasting and profiling models*[28] című cikk. Szépen összegzi a téma megjelenésekor ismert kutatásokat és rendszerezi azokat. Strukturáltan összegyűjtötték, hogyan és miért lehet szükséges az egyes alkalmazások profilozása, milyen akadályokkal szembesülhetünk.

Az alkalmazás mélyebb megismerésével lehetőségünk nyílik előrejelzéseket adni, amivel az erőforrások optimális használatát érhetjük el. Kritikaként megfogalmazódik, hogy a legtöbb kutatás elég szűk területet érint és nagyon specifikus kérdéskört vizsgál, miközben figyelmen kívül hagyja a felhős környezet változékonyságát és azt a valóságnál statikusabb rendszerként kezeli.

Ötleteket is tudunk meríteni, hogy milyen mutatókat érdemes a mérések során figyelembe venni. Példaként említi a processzor, memória, I/O lemez késleltetések, válaszidő, áteresztő képesség mérését. Utóbbi kettő rendszeresen előfordul a szolgáltatási szint megállapodásokban (SLA), aminek a következményei a felhasználó által érzékelhető szolgáltatási szintben (QoE) is megjelenhet. Erre jó példa lehet, hogy ha egy online beszélgetés közben gyakran és nagy méretben változik a beszélgető felek közötti késleltetés az a szolgáltatás megítélésére is ki fog hatni. Ezek a tulajdonságok vannak összefoglalva a 3.1 ábrán, ami szintén a cikkben jelent meg.

Gyakran elfelejtődik, de a cikkben hangsúlyozva van, hogy a rendszer monitorozása, döntéshozás és közbeavatkozás önmagukban is jelentenek valamekkora

plusz költséget.

3.1. ábra. Alkalmazás profilozás elmetérképe[28]

Korábban már a Budapesti Műszaki és Gazdaságtudományi Egyetem Távközlési és Médiainformatikai Tanszékén (BME TMIT) is foglalkoztak Kubernetesben történő skálázással. Az *Adaptive scaling of Kubernetes pods*[3] címen megjelent kutatás keretén belül sikeresen implementáltak egy automatikus skálázót, ami képes kihasználni a vertikális és horizontális skálázásban rejlő előnyöket. Az algoritmus első részében megkeresi az egy pod számára ideális erőforrás-használatot, ami tekinthető a vertikális skálázásnak. Később pedig az így megtalált konfigurációval történik az alkalmazás horizontális skálázása. Ez a megvalósítás egybeesik a korábbi kutatás következtetésével[31], mely szerint általában jobb kezdeti eredményt kapunk VPA segítségével, azonban feltehetően minden esetben létezik egy felső határ, ami után már nem kapunk jobb kiszolgálási eredményeket.

Több cikk[3][32] is említést tesz arról, hogy a jelenlegi módszertan alapján nehéz eltalálni az optimális erőforrás értékeket és ezzel együtt a skálázást. Ez főként amiatt van, mert jelenleg a skálázók kevés logikával és metrikával működnek, ezért indításukkor, kézzel kell megadni a szükséges értékeket. Ez igaz az erőforrás felhasználásra is. Ebben az esetben is az alkalmazás üzemeltetője adja meg indítás idejében a paramétereket, amik befolyásolják a kiszolgálás minőségét is.

Érdekes kérdés, ami irodalomkutatás során fogalmazódott meg, hogy a skálázó optimális döntéséhez a lehető legtöbb adatra van szükség, ami bizonyos alkalmazási területek körében problémás lehet. Ha az alkalmazás tulajdonosa nem szeretné, hogy megfigyeljék milyen egyéb szolgáltatásokkal van kapcsolatban, mennyi lekérdezést szolgál ki, kik használják, milyen arányban oszlik meg a terhelés a rendszere egységei között, akkor a végső döntés is kevesebb információ ismerete alapján kerül meghozásra. Érdekes kérdéskör az is, hogy hol van a határa a személyes adatoknak és mennyire lehet ezeket figyelembe venni az erőforrások elosztási döntésekben.

Több megoldás bontakozott ki a tanulmányozott cikkek alapján. Léteznek olyan megoldások, amik a rendszert fehér dobozként (*white box*) kezelik, amiben minden paraméter olvasható és befolyásolható, míg vannak olyanok, amik fekete dobozként (*black box*) modellezik és az alkalmazás belső működése számunkra ismeretlen marad[20]. Az első esetben van lehetőségünk analitikus modelleket gyártani és ezeket alapul véve jósolni, javaslatokat tenni. Utóbbi esetben, jó kiindulási alap lehetnek a gépi tanulásos modellek. Itt megjegyezhető, hogy számon tarthatjuk a modellünk bizonyos paraméterek melletti bizonytalanságát mint tényezőt és ezeken helyeken több mérést végezve javítható a modell bizonyossága[20].

Elterjedt kutatási terület, hogyan lehet ilyen jellegű feladatok megoldásában segítségül hívni a neurális-hálózatokat, illetve egyéb gépi-tanulásos modelleket. Ilyen modell lehet például a Markov-láncok, ami képesek a valós rendszerekben megfigyelhető periodicitásokat figyelembe venni[36]. Ilyen ismétlődés több nagyságrendben is elképzelhető, például egy napon belül is létezik, de létezhet nagyobb távlatban, például az évszakkal összefüggően is[32].

4. fejezet

Rendszer felépítése

A munkám során vizsgált kiszolgálási és skálázási kérdések megválaszolásához létre kellett hozni egy keretrendszert, ahol elvégezhetőek a szükséges szimulációk. Feladatom volt egy olyan környezet kialakítása, ahol tetszőleges szolgáltatáshálót létre lehet hozni, és teszt forgalommal terhelni azt. Ebben a fejezetben szeretném bemutatni az elkészített rendszert és az azt alkotó egyes elemeket. A fejezet végén részletezésre kerül egy teljes mérés folyamata és a kapott eredmények feldolgozásának lépései.

4.1. Rendszer részei

Az elkészült rendszer három fő komponensből áll. Ezek együttesen képesek tetszőleges tulajdonsággal rendelkező szolgáltatás hálózatokat megvalósítani, azt Kubernetes alatt elindítani, forgalmat generálni és mérés során adatokat gyűjteni.

Az egyes részekről bővebben is lesz szó, azonban most átfogóan ismertetem a rendszert, ami a 4.1 ábrán látható. Minden elem megalkotásánál lényeges szempont volt, hogy az elkészült környezetben könnyen lehessen méréseket indítani és a lehető legtöbb paraméter konfigurálható legyen.

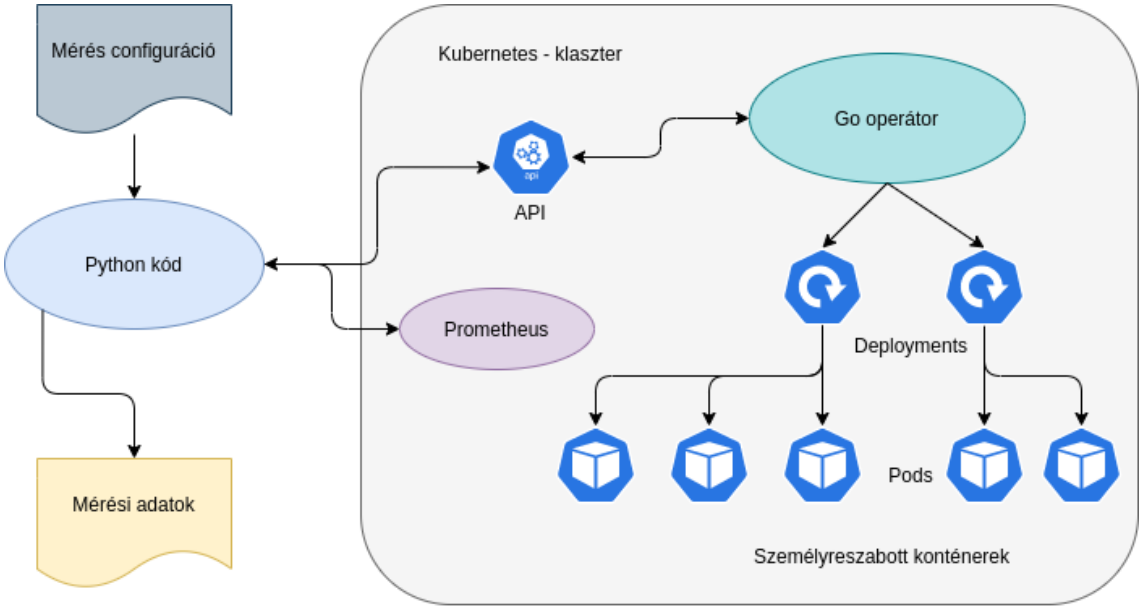
A mérések paramétereit egy konfigurációs fájlban tudjuk megadni, ahonnan egy Python program olvassa be és vezényli le a mérések elvégzését. Először a Kubernetes API-n keresztül létrehoz egy *ServiceGraph* objektumot, ami nem egy beépített típus. Az objektum értelmezéséhez és megfelelő kezeléséhez létre kellett hozni egy operátort, ami képes egy ilyen definíció szerint létrehozni a szükséges erőforrásokat (*deployment, pod, service, HPA*).

Az elindításra kerülő kapszulák tartalmaznak egy alkalmazást, amely számára az operátor indításkor át tudja adni szükséges paramétereket. A Go nyelven megírt alkalmazás képes a beérkező kéréseket fogadni és az indításkor megadott paramété-

rek alapján műveletet végrehajtani a hatására. Elég speciális igényeink vannak az alkalmazást illetően, ezért azt külön implementálni kellett.

Miután elkészültek a kért objektumok és képesek már kiszolgálni a klaszteren kívülről érkező igényeket a Python szkript elkezd számukra forgalmat generálni. A terhelés lejárta után ki kell nyerni a mérés során keletkezett adatokat. A feladat elvégzésében segítségünkre lesz a Prometheus[12] alkalmazás, ami gyűjti és exportálja a klaszterből érkező metrikákat.

Az elkészült rendszerhez szükséges részek forráskódjai megtalálhatóak a GitHub felületén[2].



4.1. ábra. Rendszer áttekintése

4.2. Korábbi munkák

A korábbi projekttárgyak keretén belül már foglalkoztam hasonló kérdéskörrel, így már volt az egyes eszközök használatához tapasztalat és implementáció is, amiből ki lehetett indulni. Önálló labor tárgyban megismerkedtem a Go programozási nyelvvel, és elkészítettem egy Kubernetes operátort, ami képes egyedi erőforrás definíciók feldolgozására és ez alapján beépített objektumokat létrehozni. A munka elején ezt tovább kellett bővíteni, hogy képes legyen skálázót is létrehozni valamint javítani kellett a megbízhatóságán is.

Korábbi tanulmányaim során elkezdtem készíteni egy Go nyelven íródott alkalmazást, melynek célja azonos volt a mostani projektben foglalttal. A megkezdett alkalmazás is egy kellően szabadon konfigurálható paraméterezéssel rendelkező web-

szerver volt, hogy tetszőleges módon tudja kiszolgálni a beérkező kéréseket. A korábbi állományból kiindulva kellett fejleszteni a funkcionalitásán, hogy valósághűbb eredményeket tudjon szolgáltatni.

4.3. Operátor

Fontos szempontnak tekintettük, hogy egyszerű konfigurációs fájl alapján létre lehessen hozni a szolgáltatáshálót, amit szeretnénk tesztelni. A Kubernetes rendszer nem tartalmaz olyan erőforrást, ami számunkra ezt a funkcionalitást alapvetően támogatná. A feladat számunkra az lesz, hogy egy közös konfiguráció alapján több különböző fajta, már beépített erőforrást hozzunk létre. Már az orkesztrációs platform fejlesztésénél felkészültek arra az esetre, hogy sok egyedi igény fog megjelenni a különböző használatból fakadóan, ezért már fejlesztés közben fontos szempont volt a Kubernetes könnyű kibővítése. A mögöttes gondolat az, hogy nagyon változó igények és funkcionalitások jelenhetnek meg a felhasználási körülményektől függően, amiket nem célszerű a központi egységbe implementálni, hiszen akkor a rendszer könnyen átláthatatlanná és inkonzisztensé válna. Helyette egyedi erőforrásokat (CustomResource) hozhatunk létre és ezen erőforrások mellé megadhatunk különböző vezérlőket. Ezt csinálják az operátorok, amivel a fentebb vázolt feladat a diplomamunkában is megoldhatóvá vált.

4.3.1. Implementáció

A klaszter operátorral való kibővítése egy gyakran alkalmazott megoldás, amikor összetettebb alkalmazást vagy logikát kell implementálni. Olyan szinten elterjedt megoldásról van szó, hogy több nyelv és keretrendszer közül lehet választani[19]. Létezik továbbá egy külön weboldal a korábban megírt és publikált operátorok böngészésére, ami kiindulási pontként szolgálhat a forráskódok tanulmányozására.[26] A Kubernetes alkalmazási lehetőségei között ezt a megoldást külön mintaként említik.[5]

Én a feladat megoldásához az Operator-SDK[25] keretrendszerét használtam fel. A keretrendszerrel könnyen írhatunk Kubernetes operátorokat. Ezt megtehetjük Helm, Ansible vagy Go segítségével is. A felsorolt opciók közül a Go programozási nyelv adja a legnagyobb rugalmasságot, mivel lehetőségünk van nagyon mély szinten belenyúlni a klaszter működébe. Én is ezt a megoldást választottam a korábbi projekttárgy keretén belül és sikerült megszerezni az alapvető ismereteket. Felada-

tom volt a korábban megírt alkalmazás hibáinak javítása és új funkciókkal történő bővítése.

Sajnos limitált számú segédanyag érhető el és azok is többnyire ugyanazt a példaalkalmazást mutatják be, ezért az elején nehéz belekezdeni. Egyedül a korábban mások által megírt operátorok forráskódja tudott jó kiindulásként szolgálni.

Több megoldás is létezik az elkészült operátorunk futtatására. Futtathatjuk a klaszteren kívülről, mint alkalmazást, illetve a klaszteren belülről is. Utóbbi esetben készíteni kell az alkalmazásunkból egy Docker képfájlt, ami aztán külön kapszulában fog futni. Illetve klaszteren belül lehetőségünk van egy úgynevezett operátor életciklus kezelővel (Operator Lifecycle Manager - OLM) is megtenni ezt. Az utóbbi megoldás a legfejlettebb, úgy lehet elképzelni, mint egy külön csomagkezelőt az operátorok számára, ami a klaszteren belül fut. A félévben én a klaszteren kívüli megoldást használtam, mivel az operátort is folyamatosan fejleszteni kellett a különböző igények szerint, illetve a projekt kapcsán nem releváns az operátor futási környezete, cserébe több idő marad a mérésekre koncentrálni.

4.3.2. Használata

A 4.1 kódrészlet mutat egy példát a szolgáltatásháló definíciójára. Számunkra elég végiggondolni, hogy milyen szolgáltatásokat szeretnénk, hogyan kövessék egymást, milyen erőforrást biztosítsunk számukra. Miután ezek megvannak, készítünk belőle egy *yaml* dokumentumot.

Látható, hogy az `apiVersion` és `kind` értékek egyediek, általuk tudunk hivatkozni a saját erőforrásunkra. Szintén meg kell adni pár metainformációt, mint például az objektum neve és névtere. Számunkra legérdekesebb beállítások a `spec` szekció alatt találhatóak. Itt tudjuk felsorolni, hogy milyen szolgáltatásokat szeretnénk majd a hálózatba. Be tudjuk állítani, hogy hány replikával fusson egy-egy szolgáltatás, illetve hogy milyen portokon tudjuk majd őket elérni. Ezek az információk azért lesznek fontosak, mert ez alapján fog az operátor létrehozni Kubernetes Deploymenteket és Serviceket. Ha szeretnénk skálázást is beállítani az adott szolgáltatásra akkor azt is megtehetjük, ha megadjuk a `hpa` konfigurációs paramétereket. Ezek hiányában nem kerül létrehozásra, és fix replikaszámmal fog üzemelni.

Továbbá definiálni kell, hogy az adott alkalmazás számára milyen végpontok létrehozása szükséges. Az itt megadott végpontok meghívása esetén lesz képes válaszolni a beérkező kérésekre. Például a *front-end* szolgáltatás a `/instant` és `/chain` lekérdezésekre fog válaszolni. A válasz gyorsasága és a közben felhasznált erőforrás mennyiséget is lehetőségünk van szabályozni. Az itt megadott paraméterek továb-

bításra kerülnek majd a futtatott konténerekhez így ők fogják helyileg érvényre juttatni a szabályokat. Ezen paraméterek értelmezése miatt volt szükséges egy egyedi alkalmazás implementálása.

A bemutatott kód csak részlete a teljes forrásállománynak, mert elég repetitív ezért nem szerettem volna a helyet foglalni, de a GitHub oldalán megtalálható a teljes kód.

```
apiVersion: dipterv.my.domain/v1beta1  # Erőforrás csoportja
kind: Servicegraph                  # Egyedi erőforrás típusa
metadata:                           # Metainformációk
  name: servicegraph                # Objektum neve
  namespace: customNamespace        # Névtér, ahol az objektum létezik
spec:                               # A szolgáltatásháló konfigurációja
  nodes:                            # Milyen csomópontokat tartalmazzon
    - name: front-end               # Csomópont neve
      replicas: 2                   # Alapból hány replika induljon belőle
      port: 80                      # Milyen porton figyeljen a Pod
      nodePort: 30000               # Legyen-e NodePort típusú service
      hpa:                          # HPA specifikus specifikációk
        utilization: 80             # Kitűzött CPU felhasználás
        min_replicas: 2             # Minimális replika szám HPA-nak
        max_replicas: 5             # Maximális replika szám HPA-nak
      resources:                    # Erőforrás request / limit értékek
        memory: 100                 # Memória felhasználás
        cpu: 100                    # Biztosított processzor mennyisége
      endpoints:                    # Alkalmazás végpontok, amire figyel
        - path: /instant            # Lekérdezés útvonala
          cpuLoad: 10               # Ennyi CPU intenzív műveletet fog végezni
          delay: 1                  # Legalább ennyi időbe telik a kiszolgálás
        - path: /chain              # Újabb lekérdezés specifikáció
          cpuLoad: 20
          delay: 2
          callouts:                 # Meg lehet adni, hova kérdezzen tovább
            - url: back-end:80/profile   # Meghívja a back-end /profile végpontot
            - url: db:36/get        # Utána meg a db /get-től kérjen adatokat

    - name: back-end                # Újabb csomópont definíciója
      replicas: 3                   # Ennyi replikaszámmal induljon
      ...                           # Nincs megadva HPA mező, akkor nem lesz HPA
      endpoints:                    # A szolgáltatás által biztosított endpointok
        - path: /profile
          cpuLoad: 30
          delay: 2
    ...
```

4.1. kódrészlet. Saját szolgáltatásháló definiálása

A szolgáltatásháló definíciójának leírása után, annak kezelése nem jelent gondot. Az új erőforrás Kubernetesben történő regisztrációja után a futó operátorunk már képes értelmezni a megadott leírót. Ezáltal felhasználói oldalról egy beépített típusnak megfelelően tudjuk kezelni a saját `Servicegraph` objektumunkat is. Egy példa indítás látható a 4.2 kódrészletben, amikor a *kubectl* parancson keresztül meghívjuk a Kubernetes API-t és továbbítjuk a hivatkozott konfigurációs fájlban leírtakat számára. Innen kerül feldolgozásra az operátor által, ami létrehozza a

kívánt erőforrásokat a klaszteren belül. A második sorban láthatjuk a visszaküldött üzenetet, miszerint sikeresen létrejött az új objektumunk.

```
$ kubectl apply -f path/to/servicegraph.yaml
servicegraph.dipterv.my.domain/servicegraph created
```

4.2. kódrészlet. Szolgáltatásháló indítása

4.4. Alkalmazás konténer készítése

A korábban látott módon lehetőségünk van tetszőleges szolgáltatásokat elindítani, azonban hogy az adott szolgáltatásoknak további paraméterek tudjunk átadni, olyan alkalmazás kell, ami tudja kezelni őket. Például: milyen végpontokra figyeljenek, milyen kiszolgálási idővel dolgozzanak, mennyi erőforrást használjanak fel. A feladat megoldására létrehoztam egy Go alapú webalkalmazást, mely induláskor átveszi a szükséges paramétereket. A webszerverhez beérkező kérések pedig a megadott paraméterek által specifikált módon fognak végrehajtódni.

A teljes kódbázis részletes ismertetése túlmutatna a dolgozat keretein, ezért ettől szeretnék eltekinteni. A mérések során használt kódok és a Docker képfájl készítéséhez szükséges minden forrás, erőforrás és instrukció megtalálható a diplomamunkához tartozó verziókövető rendszerben[2]. Azonban szeretném kiemelni a számomra izgalmasnak ítélt részeket és említésre méltó megoldásokat.

Az alkalmazás elkészítéséhez fel tudtam használni a korábbi egyetemi tárgyak alatt elkészített forráskódot, melynek továbbfejlesztése szükséges feladat volt. A korábbi verzióban *html* alapú válaszüzenetet kaptunk vissza az alkalmazástól, ami megjelenítve számunkra könnyebben áttekinthető, azonban automatizáláshoz nehezen használható. Így módosítani kellett a kódot, hogy a visszaadott üzenet *json* alapú legyen, ezzel segítve az eredmények későbbi feldolgozását.

Lehetőségünk van a *http* kérés kiszolgálása közben további webalkalmazások felé lekérdezéseket indítani. Ez a funkció lehetőséget ad, hogy a klaszteren belül tetszőleges szolgáltatáshálókat valósítsunk meg, azonban fejleszteni kellett rajta. Lecseréltem ezen lekérdezéseket aszinkron hívásokra, hogy a válaszra várakozás közben elvégezhetővé váljon a saját csomópontban szükséges számítások végrehajtása.

4.4.1. Processzor-használat skálázása

Egyik legfontosabb feladata az egyes podokban elindított alkalmazásnak, hogy fel tudja dolgozni a számára átadott paramétereket. Kintről kell átadni minden olyan

értéket, ami meghatározza a működési módját. Többek között meg lehet adni, hogy az egyes végpontokra érkező kérések esetén mekkora processzorigényt generáljon. Ezen funkció implementálása bizonyult a legérdekesebb feladatnak.

Az alkalmazás első funkcióiban nem kapott még jelentős figyelmet, mert a nagy rendszer összeállításán volt a fókusz. Emiatt egy egyszerű processzorigényes algoritmust implementáltam, ami a feladatát tökéletesen ellátta. Az általam választott algoritmus az Eratoszthenész szitája[27] volt, ami képes megkeresni egy adott számnál kisebb összes prímszámot. Az algoritmusban szereplő lépések nem nehezek, azonban több egymásba ágyazott *for* ciklust tartalmaz, ami így beláthatatlan ideig de használja a rendelkezésre álló processzort.

A megoldás gyengeségét az adta, hogy nem lehet finomhangolni és a Kubernetesben létező mCPU mértékegység szerint konfigurálni az alkalmazás egységünket. Amikor a rendszer fejlesztésének előrehaladott szakaszában ez a probléma ismét előkerült, megoldási javaslatot kellett keresni.

A megoldás alapját az úgynevezett szoros ciklusok (*tight loop*[24]) módszertan adta.

A mögöttes elgondolás, hogy 1000 mCPU felhasználást akkor kapunk, ha a megfigyelt konténer egy teljes másodpercig birtokolja és ezen idő alatt folyamatosan használja is az erőforrást. Viszont előre meghatározni, nem lehet, hogy mennyi műveletet végezzünk, hiszen ezt befolyásolja a Kubernetes klaszter mögötti infrastruktúra. Tehát elképzelhető, hogy azonos algoritmus futása eltérő terhelésként jelenik meg az adott rendszerben értelmezett 1 CPU egység miatt.

Ennek következménye, hogy ha a Kubernetes mCPU mértékegységét akarjuk használni, akkor az alkalmazás tényleges működtetése előtt szükséges egy kalibrációs tesztet végezni.

Az alkalmazás az indításakor futtatott függvényében egy másodpercig egy egyszerű tight loop algoritmust futtat és figyeljük, hogy ezen idő alatt hány iterációt tud megtenni. Amennyiben az ez információ adott, könnyen tudunk generálni megadott terhelést bármilyen környezetben. Ehhez az kell, hogy a beérkező kérés esetén a korábban kiszámolt iterációk számához arányosan az igényelt processzor-fogyasztással megegyező iterációt tegyen meg.

A végeredményül előállt megoldás által lehetőségünk van finomhangolni és kontrollálni az egyes lekérdezések hatására keletkező processzor terhelést és annak idejét is.

4.4.2. Elkészült alkalmazás használata

Az elkészült alkalmazást fel kellett készíteni, hogy a Kubernetesben tudjuk futtatni. Ehhez Docker képfájlt készítettem belőle és miután elláttam a megfelelő címkékkel feltöltöttem[1] a Docker Hub oldalra, ami egy ingyenes és publikus konténer adattár.

```
$ kubectl describe pod front-end-54b6ffc64c-cd8kl
...
    Command:
      /app/main
      -name=front-end
      -port=80
      -cpu=100
      -memory=100
      -endpoint-url=/instant
      -endpoint-delay=1
      -endpoint-call=''
      -endpoint-cpu=10
      -endpoint-url=/chain
      -endpoint-delay=2
      -endpoint-call='back-end:80/profile_db:36/get'
      -endpoint-cpu=20
...

$ kubectl run shell --rm -it --image nicolaka/netshoot -- sh
~ # curl front-end:80/instant
{"service":"front-end","host":"front-end","config":true,"endpoint":"/instant","
    cpu":10,"delay":1,"calloutparameter":"''","callouts":[""],"actualDelay
    ":16967552,"time":"2021-05-13T16:55:02.951824587Z","requestMethod":"GET","
    requestURL":{"Scheme":"","Opaque":"","User":null,"Host":"","Path":"/instant
    ","RawPath":"","ForceQuery":false,"RawQuery":"","Fragment":"","RawFragment
    ":""},"requestAddr":"10.0.1.49:43678"}
```

4.3. kódrészlet. Konténer futtatása és válasza

A 4.3 sorszámú kódrészleten egy, az elkészült Docker konténert futtató kapszula leírásának részlete látszik. Az indításhoz a korábban látott, 4.1 kódrészlet konfigurációját használtam fel. Érdemes megfigyelni, hogy az egyes paraméterek hogyan képződnek le a konténerhez. Például átadásra kerül a szolgáltatás neve, hogy melyik porton fog figyelni, melyik végpontok hívására kell válaszolnia és a hozzá tartozó egyéb paraméterek.

A kódrészleten látott második paranccsal létrehozunk egy újabb kapszulát a klaszterben. Látható, hogy interaktív módban indítottuk el, így kaptunk egy felületet, ahonnan lekérdezést lehet indítani a *front-end* kapszula felé. Ez jó megoldás abban az esetben, ha ki szeretnénk próbálni, hogyan működik az alkalmazásunk. A `kubectl run` parancs alatt látható, hogy a *curl* alkalmazás kérésére milyen választ kaptunk. A válasz *json* formátumban tartalmazza a lekérdezés főbb paramétereit. Látható, hogy melyik végpontra érkezett, milyen konfiguráció van beállítva a végponthoz.

4.5. Mérés vezénylése

Az elkészített operátorral és konténerizált alkalmazásunkkal tetszőleges szolgáltatáshálót képesek vagyunk létrehozni. A következő megoldandó kihívás a mérések kezelése volt, mivel törekedtem a precíz eredményekre. Fontos szempont volt, hogy a mérések elvégzése a lehető legegyszerűbben és rugalmas módon tudjon megtörténni. A feladat megvalósításához a Python nyelvet választottam, hiszen könnyen lehet benne szkripteket fejleszteni illetve komoly számítást nem fog végezni így nem számottevő a plusz erőforrás-felhasználása, más alacsonyabb szintű nyelvekhez képest sem.

A mérés megkezdése előtt készíteni kell egy konfigurációs fájlt. Erre egy példa látható a 4.4 kódrészleten. A mérést vezénylő alkalmazás az itt megadott paraméterek alapján fogja végezni a mérést. Többek között meg kell adni, hogy a mérések milyen szolgáltatáshálózatokon kerüljenek elvégzésre. Lehetőség van többet is megadni, így egy indítással akár az összes számunkra érdekes esetet le tudjuk szimulálni egyszerre. Továbbá meg kell adni, hogy milyen kérés per másodperc (*QPS*) értékekre vagyunk kíváncsiak. Fontos megadni, hogy milyen forgalmat és hova szeretnénk generálni. Ez látható a `Load` részen belül. Megadjuk a mérés idejét, IP címet és portot, ahol a szolgáltatásunk fogadni fogja a kéréseket.

```
Name: example-measurement        # Name of the measurement
Servicegraphs:                   # Service graph specifications
  - /path/to/service_graph1.yaml # Measured services config file
  - /path/to/service_graph2.yaml # Can add more graphs
Result_location: /path/to/result # Where to write results
Cluster:                         # Not yet used
  Name: Dipterv                  # Clustername
  IP: 152.66.211.2               # CLuster / node IP address
QPS:                             # query per seconds
  From: 0                        # QPS lower bound
  To: 200                        # QPS upper bound
  Granularity: 10                # QPS increment by this number
Load:                            # Load specification
  Preheat: 0                     # warmup time in seconds
  Time: 180                      # measurement time in seconds
  ServiceIP: 152.66.211.2        # IP for our service
  ServicePort: 30000             # port for our service
  ServicePath: to-backend        # path to our service
  ServiceQuery: ''               # URI's query fragment
Prometheus:                      # Prometheus values
  IP: 152.66.211.2               # Prometheus IP address
  Port: 31090                    # Prometheus port number
```

4.4. kódrészlet. Mérés konfigurációja

A forgalom generálását egy külön alkalmazás végzi, a *Vegeta*[33]. Természetesen többféle forgalom generáló alkalmazás használata is lehetséges és a rendszer fejlesztése közben kettőt is kipróbáltam. A korábbi döntés a *Fortio* generátorra esett, mert

könnyen használható, egyre szélesebb körben elterjedt, Go nyelven íródott és ideális döntésnek tűnt a méréseinkhez. Azonban a folytatott mérések közben számos nehezen értelmezhető jelenséget produkált, ezért került sor a váltásra.

Számos alkalmazást megnéztem, hogy minél körültekintőbb döntést tudjak hozni. Ebben nagy segítségemre volt egy összegyűjtött alkalmazás lista[9], így nem kellett megkeresni az egyes eszközöket, hanem csak össze kellett hasonlítani és kipróbálni azokat. Végül a döntés a *Vegetára* esett, mert hasonló funkcionalitásokkal rendelkezik, mint a korábbi eszközünk, ami jelentősen megkönnyítette a komponens cseréjét.

Egy-egy terhelés szimulációjához meg kell adnunk pár paramétert az alkalmazás indításához. Ilyen például az elérni kívánt QPS érték, a terhelés időtartama, a megcélzott végpont, az eredmények mentési neve és helye. Ezeket a paramétereket a korábban ismertetett konfigurációs fájl alapján állítjuk elő.

Miután megterheltük a rendszert és megkaptuk a kérések kiszolgálásával kapcsolatos statisztikákat a *Vegetából*, szükséges még a rendszer erőforrás-felhasználását célzó metrikákat is begyűjteni. Erre a `Prometheus` rendszerén keresztül van lehetőségünk. A szoftver támogatja az API hívásokat, így lehetőségünk nyílik könnyen lekérdezni az általa gyűjtött statisztikákat.

```
# Prometheus query parameters
cpu_query = {"query": "container_cpu_usage_seconds_total{image!='', \
                       namespace=~'default|metrics', \
                       container!='POD'}",
    "start": str(start_time),
    "end": str(end_time),
    "step": "2",
    "timeout": "1000ms"
}

# Get parameters from config
prometheus_ip = config["prometheus_ip"]
prometheus_port = config["prometheus_port"]

# Assemble prometheus base query URL
url = "http://" + str(prometheus_ip) + ":" + str(prometheus_port) + "/api/v1/
    query_range?"

# Assemble full query
cpu_full_query = url + urllib.parse.urlencode(cpu_query)

# Get results from Prometheus
cpu_res = json.loads(requests.get(cpu_full_query).text)
```

4.5. kódrészlet. *Prometheus* rendszer használata Python kódból

A metrikák lekérdezésére a 4.5 kódrészlet ad egy példát. Meg kell adnunk, hogy milyen értékekre vagyunk kíváncsiak. A látható példában ez a konténerek által felhasznált CPU mennyisége, továbbá kikötéseket teszünk, hogy csak a *Default* vagy

Metrics névtérben futó konténerek érdekelnek. Meg kell adni a lekérdezés kezdeti és vég idejét, amíg a generált kéréseket kiszolgálta. Össze kell állítani a `Prometheus` elérhetőségét, amihez a mérés elején megadott konfigurációt vesszük alapul. Ha megvan az előkészített API hívás, akkor a kódrészletben látott módon meg kell hívni azt. A kapott válasz `json` formátumban érkezik, így később mi is így kezeljük.

A korábban látott megoldással egyéb adatokat és metrikákat is lekérdezhetünk. Jelenleg négy értéket gyűjtünk:

- Konténerek processzor felhasználásait külön-külön.
- Konténerek memória felhasználásait külön-külön.
- Az összes futó pod, ami részt vesz a mérésben.
- A futó konténerek száma típus szerint. (például: külön amik a front-end szolgáltatást valósítják meg és külön amik a back-end szolgáltatást)

A `Prometheus` könnyen kiterjeszthető rendszer így közel tetszőleges metrikákat lehet általa gyűjteni.

A mérés végén miután összegyűjtöttük az összes keletkező adatot, beleértve a `Prometheus` és `Vegeta` rendszereket is illetve az eredeti konfigurációt, azokat perzisztálni kell. Erre a legkézenfekvőbb módszer az adatok kiírása `json` fájlba. Ez azért is előnyös, mert szkriptek segítségével könnyen olvasható és feldolgozható formátumot biztosít az eredmények kiértékeléséhez.

4.6. Eredmények ábrázolása

Az egyes mérések kimenetei *json* fájlok, melyek ember általi értelmezése és ezáltal a következtetések levonása nehéz feladat lenne ebben a formában. Szükségessé vált tehát egy külön alkalmazás implementálása, ami az általunk kapott eredményeket könnyen átlátható formában ábrázolja.

A feladat megoldására szintén a Python nyelvet választottam, mert könnyen lehet benne fájlokat kezelni, egyes objektumokat módosítani és rendelkezik grafikus megjelenítő modullal. Fontos szempont volt az alkalmazás könnyű használata ezért bizonyos paramétereket akár parancssorban, indításkor is meg lehet adni. Ezáltal az alkalmazás működését tudjuk befolyásolni a konkrét forráskód módosítása nélkül.

Az alkalmazás indításakor meg kell adni, hogy hova vannak mentve a mérések során készült kimeneti *json* fájlok, melyek feldolgozása szükséges. Itt egy-egy mérés a forgalomgenerátor által, adott terhelés mellett, adott ideig történő szimulációját

értjük. Ebből következik, hogy egy teszteset futtatása több ilyen értelemben vett mérést tartalmaz, hiszen egy skálán megy végig a rendszer, ahol a minimális QPS értéktől megadott lépésközzel halad a maximális QPS értékig. Egyesével elkezdi beolvasni a mérési eredményeket és minden fájlból készít egy "tisztázott" eredményt. Ebben már csak az adott grafikon kirajzolásához nélkülözhetetlen értékek kerülnek tárolásra. Például előfordul, hogy a Prometheus eredményében szerepel olyan pod is, ami nem vett részt az adott mérésben, mert még a korábbiból maradt ott. Az ilyen eseteket észlelni kell és kiszűrni az értékeit, hogy ne okozzanak anomáliákat.

4.7. A mérés folyamata

Ebben az alfejezetben szeretném a korábban bemutatott alkotóelemeket egymás után sorba tenni és az általuk végrehajtott egy-egy lépést részletesebben bemutatni, egy szimuláció indításától kezdődően a kapott eredmények feldolgozásáig.

A bemutatott lépések az elkészült rendszer működésének használatára koncentrálnak, emiatt a Kubernetes operátor működése nem szerepel fajsúlyos mértékben a leírtak között. Az operátor működését ebben az esetben tekinthetjük egy beépített szolgáltatásnak.

A lépések könnyebb követhetőségét segíti a 4.2 ábra. A mérés elindításához szükségünk van egy konfigurációs állományra, amiben a kívánt szimulációhoz szükséges paraméterek kerültek definiálásra. Ebben a konfigurációs fájlban kell megadni azt vagy azokat a szolgáltatáshálót leíró *ServiceGraph* erőforrás *yaml* fájlokat, amiken a szimulációkat futtatni szeretnénk. Több ilyen szolgáltatás háló is megadható, ebben az esetben többször fog lefutni a szimulációnk azonos paraméterekkel.

Az elindított Python alkalmazás először ellenőrzi, hogy létezik-e parancssorban megadott konfigurációs állomány. Amennyiben nem volt megadva, a program futása közben is lehetőségünk van azt behivatkozni, ráadásul ebben egy automatikus kiegészítő funkció is segítségünkre van. Ha ez megtörtént, akkor egy segédmodul segítségével beolvassuk a paramétereket és eltároljuk egy Python szótárban, hogy később könnyebb legyen az adott értéket felhasználni.

Ezután minden megadott szolgáltatásháló esetén el kell végezni a következő lépéseket. Először ki kell számolni, hogy milyen QPS értékű terhelések generálása lesz szükséges. Ha van még ilyen QPS érték, amire nem lett elvégezve a szimuláció, akkor megigényli az éppen aktuálisan vizsgált szolgáltatáshálót a Kubernetes klaszterben. Ez alapján az operátor elkezdi létrehozni a szükséges erőforrásokat. Eközben az alkalmazásunk folyamatosan figyelemmel követi, hogy az igényelt kapszulák állapota futó legyen, hiszen csak a rendszer teljes elindulása után lehet a méréseket elkezdeni.

Miután elindultak a podok bizonyos előmelegítést végez a rendszer, ha a konfigurációs fájl alapján szükséges. Ez a funkció akkor lehet hasznos, ha nem szeretnénk rögtön a friss rendszeren méréseket végezni, hanem adunk neki egy kis időt és terhelést, hogy felvegyen egy alapállapotot. Ebben az esetben viszont figyelni kell arra, hogy a Prometheusban megjelenő adatok nem valósidejűek és ez befolyásolhatja a mért eredményeket. Emiatt a dolgozatban bemutatott méréseknél ezzel a lehetőséggel nem éltem. A funkció hiba nélküli alkalmazásához további fejlesztések szükségesek.

Az előmelegítési fázis után következhet a valódi terhelés generálása. Ehhez jelenleg a parancssoros alkalmazást, a Vegetát használom. Miután a konfigurációban beállított ideig a megadott QPS értékkel történő terhelés befejeződött, szükséges egy kicsit várni. Erre amiatt van szükség, hogy a Prometheushoz is megérkezzenek a mérés során generált statisztikák.

A mesterséges késleltetés után le kell futtatni az egyes Prometheus lekérdezéseket. Itt megaddható, hogy milyen felbontással, mettől-meddig és milyen értékeket kérdezzen le. A rendszerből érkező egyes *json* válaszokat össze kell fűzni, illetve a mérő eszköz által generált statisztikákat is érdemes kezelni.

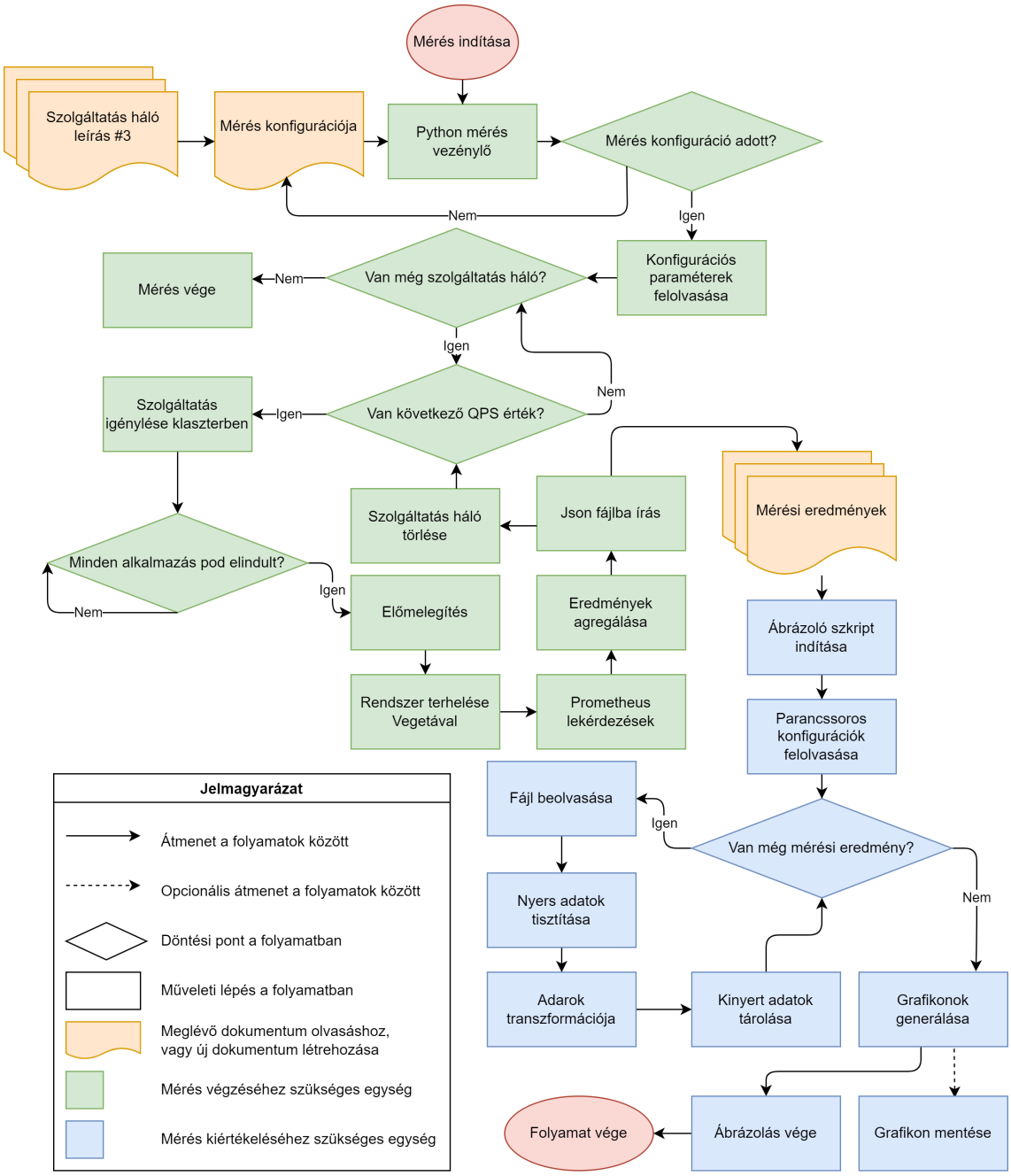
A különböző forrásokból összeállított kimeneteket egy nagy fájlba kell kimenteni, hogy a későbbiekben is fel tudjuk azokat használni. A fájl mentése után törölhető a szolgáltatás háló, hogy a következő QPS értékkel történő mérésnél ne zavarjon be a korábbi rendszer és az azon végzett mérések, így a lehető legtisztább eredményeket kapjuk. Tehát ezt minden terhelési értékre és szolgáltatási hálón végig kell járni és az egyes kimeneti eredményeket letárolni.

Eddig tart a mérést vezénylő Python alkalmazás hatásköre és feladata. Amikor előálltak a kimenetek, akkor készen állnak a további feldolgozásra, azonban ezt már egy következő alkalmazás fogja végezni. Ezért is szerepel más színnel a 4.2 ábrán, ahogy azt a bal alsó sarokban található jelmagyarázat is tartalmazza.

Az ábrázolás folyamata a korábban bemutatott lépésekhez képest lineárisabb, emiatt könnyebben is értelmezhető. Az ábrázoló szkript indításakor szintén bizonyos paraméterek megadása kötelező, hiszen meg kell adni, hogy melyik mappában szereplő mérési eredményeket szeretnénk grafikonra vinni. A megadott mappában található és értelmezhető *json* típusú fájlokat fogja az alkalmazás egymás után beolvasni.

A beolvasott fájl igazán nagyméretű és rendezetlen adathalmazt tartalmaz. Emiatt szükséges a nyers adatokat tisztítani és csak a számunkra értelmezhető információkat megtartani. Az így kapott adatokat transzformálni kell, hogy a későbbiekben az alkalmazás könnyebben tudja ábrázolni.

A kinyert és előkészített adatokat el kell tárolni, amíg minden mérési eredményt tartalmazó fájlon megtörténik ez a lépés. Ennek a folyamatnak a végére előáll az ábrázolható adatok együttese. Ezeket különböző módon tudjuk ábrázolni és megjeleníteni. A kész grafikonokat lehetőségünk van lementeni, de ez csak egy opcionális lépés, emiatt szerepel az ábrán szaggatott vonalas nyíllal. A megjelenített grafikus ablak bezárása után végére is értünk a folyamatnak.



4.2. ábra. Szimuláció futtatásának és kiértékelésének lépései

5. fejezet

Mérések

Ebben a fejezetben szeretném bemutatni a diplomamunka keretein belül elvégzett méréseket és a hozzájuk szükséges egyéb ismereteket. Először bemutatom milyen lehetőségek voltak a környezet kialakítására és a mérések közben használt eszközök, szoftverek verzióit. Ezután következik a mérések bemutatása és a kapott eredmények értékelése. Az eredmények kiértékelése során kiderül, hogyan viselkedik forgalomszabályozás nélkül egy szolgáltatásháló és ezen mennyire segít a beépített horizontális skálázó.

5.1. Mérés környezete

A feladat elején főként implementációval kellett foglalkozni így nem kapott jelentős szerepet a Kubernetes klaszter. Ezért a félév első felében elég volt lokálisan futtatni, amit én Minikube segítségével tettem meg.

Amikor a fejlesztési rész kezdett véglegesedni kellett egy rendes környezet, a végleges mérésekhez. Ehhez több lehetőséget is számításba vettem.

- **BME Cloud** - Egyik lehetőség az egyetemi felhő volt. Itt létre lehet hozni virtuális gépeket, amiket aztán klaszterbe lehet szervezni. Hátránya viszont, hogy hosszabb távra nehezen lehet gépet igényelni, rendszeresen le is állítják ami könnyen okozhatja egy-egy mérés elvesztését, hiszen több órán keresztül is futhat.

- **Klaszter a felhőben** - Kézenfekvő megoldás lehet igényelni egy teljes, menedzselt klasztert. Erre több opció is van, csak hogy a legnagyobbakat említsem: Google, Amazon, Microsoft. Ezek a minőségi szolgáltatások azonban havidíjasok lennének, és bizonyos tekintetben kevésbé rugalmasak.

- **Tanszéki infrastruktúra** - A BME TMIT tanszék is rendelkezik egy telepített Kubernetes klaszterrel, így ez is felmerült opcióként. Előnye a megoldásnak, hogy a telepítést nem kell elvégezni, ezzel gyorsítva a munka haladását. Hátrányként viszont felmerült, hogy többen használják a rendszert és előre le kell foglalni.

- **VM igénylés a Schönherz kollégiumtól** - A Villamosmérnöki és Informatikai Karhoz tartozó Schönherz kollégiumban lévő Kollégiumi Számítástechnikai Körtől lehet tanulmányokhoz és egyéb projekthez kapcsolódóan virtuális gépeket igényelni. Az igénylés leadása után lehetőséget kaptam három virtuális gép használatára egészen a projektfeladat végéig.

A fenti opciók közül számomra a negyedik volt a legszimpatikusabb így kaptam is három teljesen új virtuális gépet. Telepítéshez a Debian operációs rendszert választottam, mert stabil, megbízható és széles körben támogatott. A virtuális gépek tulajdonságai az 5.1 táblázatban láthatóak. Hasonló erőforrásértékekkel rendelkeznek, annyi különbséggel, hogy csak az egyik gépnek van publikus címe. Továbbá a táblázat tartalmazza az egyes csomópontok nevét és a klaszteren belüli szerepét is.

Tulajdonság	VM 1	VM 2	VM 3
CPU (mag)	4	4	4
Memória (GB)	4	4	4
Tárhely (GB)	10	10	10
Csomópont neve	dipterv1	dipterv2	dipterv3
Külső IP	152.66.211.2	∅	∅
Belső IP	10.151.103.1	10.151.103.2	10.151.103.3
K8s szerep	control-plane,master	worker	worker

5.1. táblázat. Használt csomópontok tulajdonságai

5.1.1. Klaszter előkészítése

A virtuális gépekből klasztert kellett szervezni, amire különböző megoldások léteznek.[18] Ezek közül két különbözőt szeretnék kiemelni:

1. **Kubespray** - *Ansible* felhasználásával előre definiált lépéseket hajt végre. Pár soros konfigurációt kell megadni számára, ami a nélkülözhetetlen paramétereket tartalmazza és ígérete szerint minden mást megold.

2. **Kubeadm** - Az előzőhöz képest a *kubeadm* sokkal manuálisabb lehetőséget biztosít a telepítéshez. Segítségével le tudjuk generálni s szükséges kulcsokat a klaszterhez, és képes új csomópontokat a klaszterhez kötni.

Nem szándékosan de kipróbáltam mindkét megoldást. Vonzó volt ugyanis a Kubespray, hogy könnyen és automatikusan telepít minden szükséges függőséget. Sajnos a magas szintű telepítése miatt hiba esetén nem sok hasznos információ áll rendelkezésre annak javítására. Miután eredménytelenül zárult minden próbálkozás, újratelepítettem a virtuális gépeket és áttértem a Kubeadm használatára.

A telepítés menetét nem részletezném, mert a hivatalos weboldalon[16] látható lépéseken kellett végigmenni. Az elkészült klaszter egy mester csomóponttal és kettő kiszolgáló csomóponttal rendelkezik. A fő csomópont rendelkezik egyedül külső hálózatról elérhető, publikus IP címmel, a többit csak belső címen lehet elérni. Ez némiképpen nehezítette a telepítés folyamatát, részben emiatt sem működött a Kubespray megoldása.

A klaszter elkészülte után még pár függőséget ki kellett elégíteni, hogy a lokálisan összeállított mérő keretrendszer működni tudjon. Külön telepíteni kellett egy konténer hálózati interfész (*CNI*) bővítményt. Ez alapból nem része a Kubernetesnek így külön kell telepíteni, hogy a különböző csomóponton futtatott konténerek tudjanak egymással kommunikálni. A választás a *Cilium* szoftverre esett, mert szerettem volna jobban megismerni, széleskörűen használható és jól dokumentált. Telepítése után lehetőségünk van teszteseteket futtatni, ami igazolja a klaszterben a csomópontok és kapszulák kommunikációját.

A 4.5 szakaszban leírt módon a mérések során gyűjtött adatok jelentős részét a Prometheus rendszere gyűjti össze és tőle kérdezzük le. Emiatt az éles méréseket végző klaszterben is telepíteni kellett. Ebben a Helm, Kuberneteshez készített csomagkezelő megoldása segített. A `prometheus-stack` csomagot kellett telepíteni annyi kiegészítéssel, hogy kintről is elérhetővé tegyük a felületet és szolgáltatásait. Ezt úgy érhetjük el, hogy a létrehozandó `Service` típusát `NodePort`-ra állítjuk és a következetesség miatt adunk neki egy portszámot. Az így létrehozott környezet rendelkezik egy webes felhasználói felülettel is, ahol tetszőleges lekérdezéseket indíthatunk.

5.1.2. Verziók

A rendszer és mérések összeállításához több különböző komponenst kellett integrálni. Az 5.2 táblázat összefoglalóan tartalmazza az egyes környezetek és használt eszközök verzióit. Mindig törekedtem a legfrissebb verzió használatára, hiszen ezek tartalmazzák a legtöbb funkciót és a legtöbb hibajavítást, azonban a környezet összeállítása után már nem végeztem verziófrissítést, nehogy eltérést okozzon a mérési eredményekben.

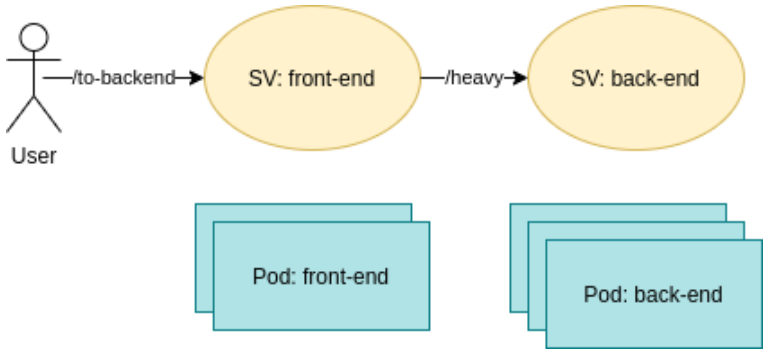
Szoftver	Verzió
Cilium	1.9.5
Debian	10 (Buster)
Go	1.16.3
Grafana	7.5.3
Kubectl	1.21.0
Kubernetes	1.21.0
Minikube	1.17.1
Operator-SDK	1.4.0-32
Prometheus	2.24.0
Python	3.7.3

5.2. táblázat. Használt eszközök és szoftverek verziói

5.2. Tesztmérés

A rendszer összeállítása után lehetővé vált konkrét mérések elvégzése. A rendszer működését bizonyítandó és az egyes elemek további finomhangolása miatt kezdetben egy tesztmérésen kísérleteztem. A mérésnek nem volt célja a hasznos eredmények gyűjtése, csak az elvárt működés bizonyítása. Főleg ezen mérés alatt derült ki, hogy a korábban használt forgalomgenerátor alkalmazást is le kell cserélni, ezzel is közelebb kerülve a valósághű eredményekhez.

A szimulációban két szolgáltatás vett részt, melyek az 5.1 sorszámú ábrán is láthatóak. Volt egy nodeport segítségével kintről elérhető *front-end* szolgáltatás, illetve egy csak bentről elérhető *back-end*. Azt a szituációt vizsgáltuk, amikor a *front-end* nem használ sok erőforrást, és két pod fut belőle. Ezzel szemben a *back-end* egy jóval erőforrás igényesebb szolgáltatás, viszont három egység fut belőle. A *front-end* minden felé érkező, `/to-backend` végpontra érkező kérést továbbított a *back-end* `heavy` végpontjára. A szolgáltatáshálóval elértük, hogy a második szolgáltatásunk a számára beállított, sok processzort igénylő műveletet hajtsa végre.



5.1. ábra. A mérésnek kitett szolgáltatások kapcsolata

A tesztmérés során kapott eredmények feldolgozása során kiderült, hogy a szimulációhoz felhasznált forgalomgenerátor alkalmazás nem az elvárt módon működik. Emiatt kellett az eredetileg használt *Fortio* alkalmazást cserélni az új alkalmazásra, ami a *Vegeta* lett. A *Fortio* minden egyes kérésre megvárja a visszaérkező választ és csak ezután tud újabb lekérdezést indítani egy-egy szimulált felhasználó számára. Ez a működés viszont nem fedi a valóságos terhelést, amikor az egyes lekérdezések tetszőleges számban érkezhetnek, nem kell megvárni a korábbi felhasználó kérésének kiszolgálását.

5.3. Tervezett mérések

A korábban bemutatott mérés alapján a rendszer képes szimulációkat futtatni és a kapott adatokat feldolgozni. Ezután meg kellett tervezni a diplomamunkához szükséges méréseket, amik a lehető legtöbb információval fognak szolgálni.

- **Szolgáltatásháló HPA nélkül** - Szükséges méréseket végezni, hogy megértsük a szolgáltatáshálók alapvető kiszolgálási működését. Az itt elvégzett méréseket lehet később referenciának használni.
    - **3 frontend egymás után, 1 backend** - A beérkező kérés kiszolgálása egy szolgáltatásokból álló láncon halad végig. A lánc eleje tartalmazza a gyors, nem erőforrás-igényes műveleteket és a lánc végén kerül sor a költséges műveletre.
    - **3 frontend egymás mellett, 1 backend** - Az előzővel hasonló architektúra, azonban a frontend szolgáltatást megvalósító alkalmazások egymás mellett helyezkednek el, így a láncolat hossza lecsökken és az általuk kiszolgált kérések száma növekedhet.
- **Szolgáltatásháló HPA segítségével** - A referencia mérések után meg kell nézni, milyen változásokat érhetünk el a beépített automatikus skálázóval.
    - **1 frontend, 1 backend egymás után** - A mérés során megfigyeljük, hogy a kezdeti konfigurációval hogyan változik a kezelt alkalmazáskomponensek száma. Ki fog derülni, maximálisan mennyi pod futhat egyszerre.
    - **Korábbi mérések eltérő konfiguráció mellett** - Az előző pontban kapott eredmények alapján a szükséges konfigurációk módosítása mellett figyeljük a szimuláció során kinyert eredményeket. Módosítjuk a kezdeti replikaszámot és a cél-processzorhasználati értéket is.

5.4. Elvégzett mérések statikus konfigurációval

A projektmunka jelentős részét tette ki, hogy méréseket kellett végezni a korábban bemutatott rendszerelemekkel rendelkező környezetben. A mérések darabszámát szemlélteti az 5.1 kódrészlet. Beépített Linux parancsok segítségével könnyen megkaphatjuk a projekt során készített és a kiértékeléshez használt *json* dokumentumok darabszámát. Ahogy a kódrészlet is mutatja először rekurzívan lekérdezzük az adott mappán és almappákon belül található adott kiterjesztéssel rendelkező fájlokat *(find)*. Majd az így kapott eredményt csővezeték segítségével hozzákötjük egy szintén beépített parancs *(wc)* bemenetére, ami megszámolja a kiírt sorok számát. Ahogy az a kódrészleten is látszik, ennek a kimenete 1872, ami azt jelenti, hogy a projekt jelenlegi állapotához kapcsolódóan ennyi különböző mérés született. Az egyes mérések itt egy-egy Vegeta terheléshez kapcsolódó konfigurációk, válaszidők és egyéb Prometheus-ból kapott eredményeket jelentik.

```
$ find -type f -iname \*.json | wc -l
1872
```

5.1. kódrészlet. Elvégzett mérések száma

A következő alfejezetekben szeretném bemutatni azon méréseket, amelyek nem tartalmaztak semmilyen automatikus mechanizmust, ezáltal nem tudták a terheléstől függően módosítani a konfigurációikat. A fő kérdés az volt, hogyan viselkedik a rendszer külső forgalomszabályozás (*flow control*) nélkül. Alapértelmezetten nincsen limitáció a klaszterbe érkező kérések számára illetve az egyes szolgáltatásokhoz érkező kérésekre sem, ezért lehetőségünk van elárasztani azokat. A szimulációk során folyamatosan növelve a beküldött kéréseket figyeljük a rendszer kiszolgálási paramétereit és stabilitását.

5.4.1. Költséghatékony frontendek és költséges backend láncban

Az első bemutatott mérésnél azt vizsgáltam, hogyan változik a rendszer működése, ha több költséghatékony alkalmazáson keresztül jut el egy erőforrás-igényes hátsó alkalmazáshoz. A szimulált szolgáltatáshálót alkotó elemek az 5.3 táblázatban bemutatott specifikációkkal rendelkeznek. Látható, hogy a három költséghatékony frontend egymásnak továbbítják a beérkező kéréseket, majd ezután jut el a backendhez. A könnyebb értelmezés miatt mindegyikből egy darab replika került elindításra. A frontend egységek egy-egy kérés kiszolgálásához nagyjából 10mCPU processzort igényelnek, és 1000mCPU volt a lefoglalt és a maximálisan felhasználható erőforrás

mennyiség számukra. Ezek alapján $\frac{1000mCPU}{10mCPU/kérés} = 100$ kérés kiszolgálására elegendő erőforrás van számukra allokálva. Ezzel szemben, a sor végén álló, költséges backend maximum 2vCPU-t használhat, és az egyes beérkező lekérdezések kiszolgálására 100 mCPU processzorra van szüksége. Ebből kifolyólag maximum $\frac{2000mCPU}{20mCPU/kérés} = 20$ kérés kiszolgálására van lehetőség másodpercenként.

Minden kérés a *front-end-1*-hez érkezik be és innen kerül továbbításra. Mivel a sor elején lévő egységek ötször több kérést tudnak kiszolgálni, ezért a szűk keresztmetszet a sor hátulján lévő *back-end-1* lesz. Emiatt minden beérkező kérésnek el kell jutnia a sor végére, miközben a sor elején lévő egységeken erőforrás-használatot indukál. Mire eljut a sor végére és feldolgozásra kerül, addig a beérkező *http* kapcsolat eldobásra is kerülhet a beállított két másodperces időkorlát miatt.

Név	front-end-1	front-end-2	front-end-3	back-end-1
Replikák száma	1	1	1	1
CPU használat (mCPU)	10	10	10	100
CPU limit (mCPU)	1000	1000	1000	2000
Memória limit (kB)	1000	1000	1000	1000
Továbbhívás	front-end-2:80	front-end-3:80	back-end-1:80	-

5.3. táblázat. Három költséghatékony frontend után egy költséges backend

Az ismertetett környezetben elvégzett mérésről készített grafikon látható az 5.2 ábrán. Megfigyelhető, hogy az egyre nagyobb beérkezett terhelés hatására folyamatosan növekszik a rendszer által felhasznált erőforrások mennyisége[5.2a és 5.2b]. Különösen érdekes, hogy a *back-end-1* által használt processzor mennyisége, a várt módon, 20 lekérdezés / másodperc érték környékén eléri a maximumot, onnan már nem emelkedik tovább, csak stagnál[5.2c]. Ezzel ellentétben az egyes frontend alkalmazások erőforrásigénye folyamatosan növekszik a mérés végéig.

Ezzel párhuzamosan a memória használata is érdekesen alakul. 18 QPS értékig látszólag nem változik érdemben, alacsonyan marad. Azonban a backend telítése után, minden egységnél elkezd növekedni[5.2b és 5.2d]. Ez azzal magyarázható, hogy a rendszerben lévő várakozási sorok, pufferek folyamatosan telnek meg, mivel a láncolat végén lévő költséges backend nem képes a beérkezések számával tartani a kiszolgálási számot.

Az alsó sorokban látható grafikonok segítségével leolvasható, hogy az átlagos válaszidők 20 QPS környékén elérik a két másodperces felső korlátot[5.2e], amivel szinkronban az időben kiszolgálásra kerülő kérések száma is drasztikusan elkezd

visszaesni[5.2f], majd utána a beérkező kérések elhanyagolható részét tudjuk csak időn belül kiszolgálni. Összességében megállapítható, hogy létezik egy olyan pont a szimulációban, aminél több beérkező kérés esetén csak az erőforrásokat használjuk és az érdemi kiszolgálás is hirtelen és drasztikus mértékben visszaesik. Ilyenkor elszállnak a késleltetések és a memória használat, valamint folyamatosan nő a processzor használata is.

5.4.2. Költséghatékony frontendek egymás mellett és költséges backend mögöttük

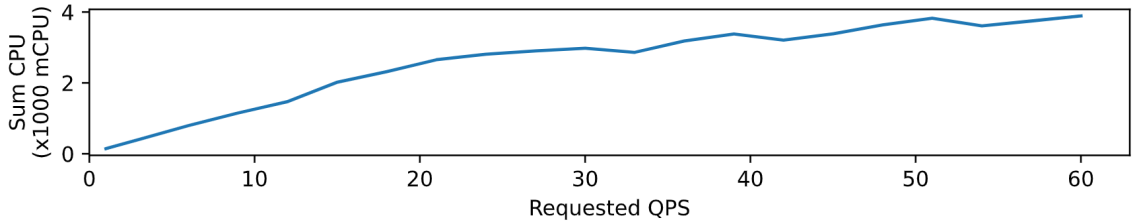
Az 5.4.1 fejezetben látott eredmények alapján felmerült a kérdés, hogyan változnak a kiszolgálási paraméterek, rövidebb szolgáltatásháló esetén. Emiatt a hálózat architektúráját módosítani kellett. A megkonstruált erőforrás-használatok és a rendszer felépítését az 5.4 táblázat tartalmazza.

Látható különbség, hogy ebben a mérésben nem hozunk létre egy hosszú láncot az egyes frontend alkalmazások egymás mögé kötésével, hanem ennél sokkal rövidebb lesz egy-egy kérés kiszolgálásának folyamata. A költséghatékony frontend után rögtön a költséges backendhez kerül továbbításra a beérkező kérés. Ezen kívül még egy fontos módosítás, hogy három darab frontend fog futni és továbbra is egy backend lesz hátul. A megadott értékek alapján kiszámolható az egyes egységek által körülbelül kiszolgálható kérések száma másodpercenként. 5.1. egyenlet alapján leolvasható, hogy a *front-end-1* áteresztő képessége 120 kérés, míg a *back-end-1* ennél jóval szerényebb 20 beérkező kérést tud kiszolgálni másodpercenként, mint ahogy az következik az 5.2 egyenletből.

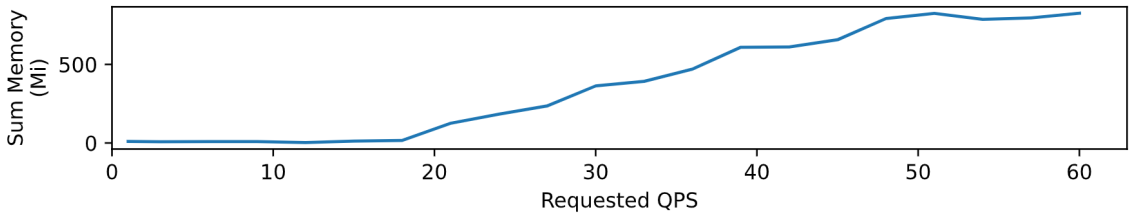
$$3(replika) \times \frac{1000mCPU}{25mCPU/kérés} = 120kérés/másodperc \tag{5.1}$$

$$1(replika) \times \frac{2000mCPU}{100mCPU/kérés} = 20kérés/másodperc \tag{5.2}$$

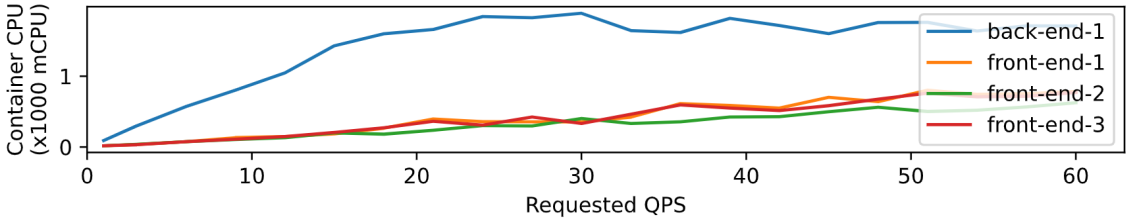
Jelen mérésben nincsen megadva semmilyen automatikus skálázó, ezért amikor az operátor az indításkor létrehozza adott replikaszámmal az egyes Kubernetes Deploymenteket, azok nem is fognak változni, függetlenül a beérkező kérések mennyiségétől. A bemutatott környezeten elvégezve a szimulációt a kapott eredmények láthatóak az 5.3 ábrán. A korábbi méréssel szinkronban, hasonló megállapításokat tudunk leolvasni. Látható, hogy a *back-end-1* telítése az elvárt 20 QPS körül megtörténik (illetve kicsit még előtte). Azonban ettől függetlenül a *front-end-1* által elhasznált processzor mennyisége folyamatosan növekedett[5.3c]. A legmagasabb



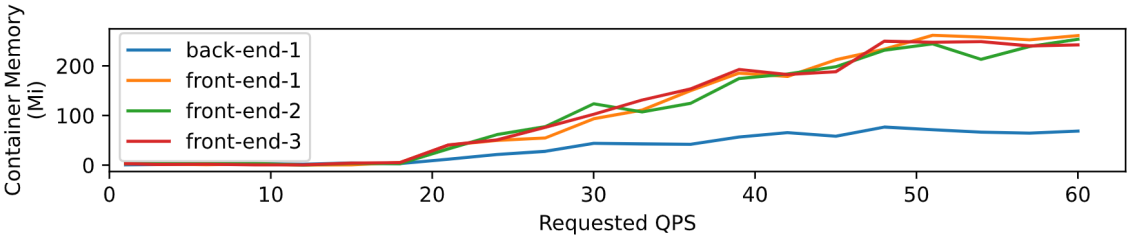
(a) Összes CPU felhasználás



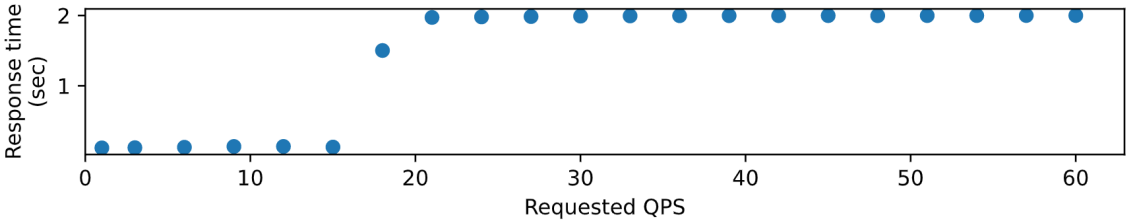
(b) Összes memória felhasználás



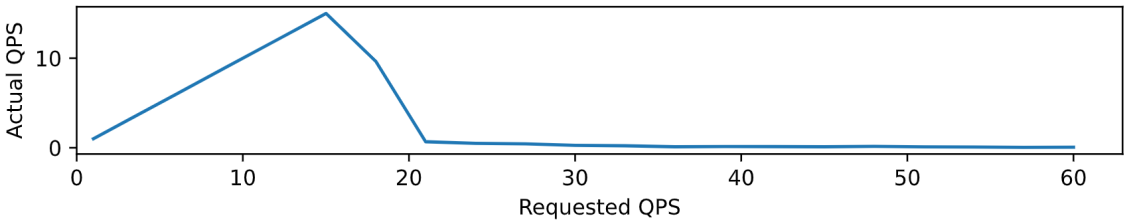
(c) CPU felhasználás konténerenként



(d) Memória felhasználás konténerenként



(e) Átlagos válaszidő



(f) Igényelt és sikeres kérések száma másodpercenként

5.2. ábra. Három költséghatékony frontend után egy költséges backend

Név	front-end-1	back-end-1
Replikák száma	3	1
CPU használat (mCPU)	25	100
CPU limit (mCPU)	1000	2000
Memória limit (kB)	1000	1000
Továbbhívás	back-end-1:80	-
HPA	-	-

5.4. táblázat. Költséghatékony frontend több replikával és utána egy költséges backend

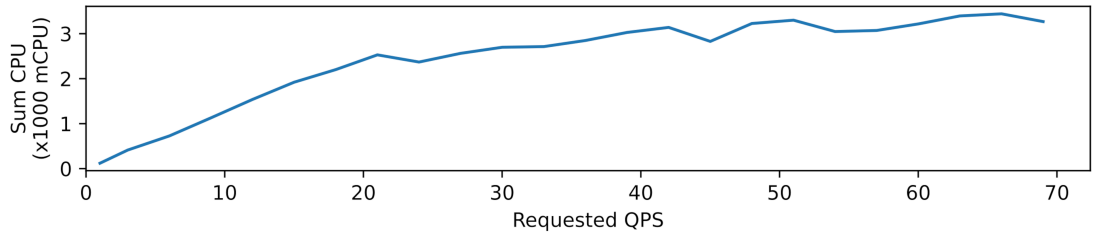
terhelésnél már olyan szintre emelkedett a három példány által összesen használt processzor erőforrása, mint a backend maximuma, tehát mintha 2 teljes virtuális CPU magot is lefoglaltunk volna számukra.

A költséges backend telítése után jelentősen megugranak a beérkező kérések kiszolgálásához szükséges késleltetések[5.3e] és az egyes egységek által használt memória mennyiségek[5.3b és 5.3d] is. A legalsó grafikonon látszik, hogy 20 QPS-nél már vissza is zuhant a sikeres kiszolgálások száma[5.3f], amivel szinkronban a kiszolgálások átlagos válaszideje is megugrik a maximumra, ami jelenleg 5 másodperc, mert a beállítások szerint ennyi idő után bontjuk a *http* kapcsolatokat.

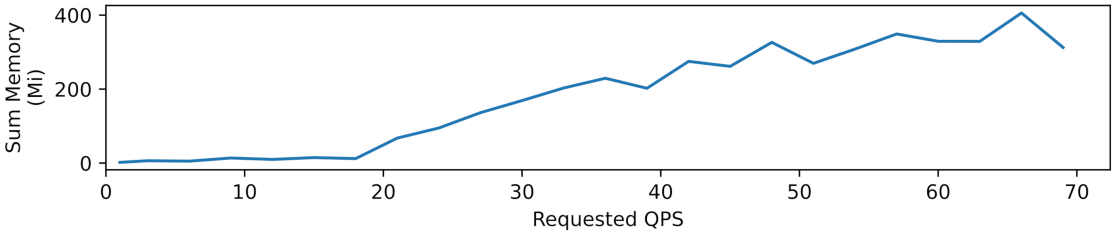
Röviden elmondható, hogy a beérkező kérések növelésével együtt fokozatosan nő az elhasznált processzor mennyisége és a sikeres kiszolgálások száma is. Azonban van egy pont, amikor a hálózatban lévő szűk keresztmetszet nem képes tartani a lépést a beérkező kérések mennyiségével. Ebben az esetben hirtelen visszaesik a kiszolgálás minősége, drasztikusan megugrik a várható kiszolgálási idő és ezzel együtt csökken a sikeres kiszolgálások aránya.

5.4.3. Tapasztalt probléma összegzése

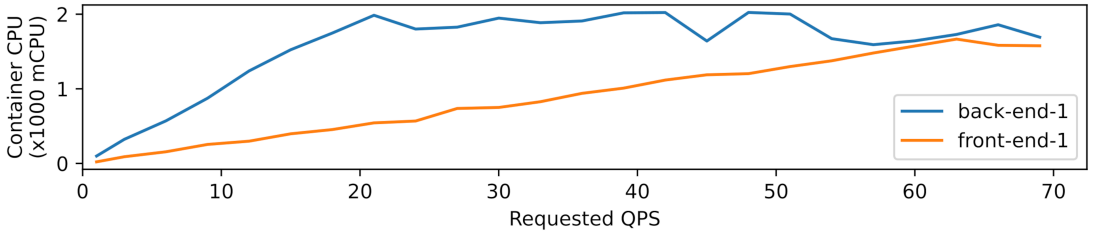
Az elvégzett mérések válaszul szolgáltak a kérdésre, hogy mi történik statikus konfiguráció esetén a szolgáltatásháló kiszolgálási teljesítményeivel, ha nagy mennyiségű kiszolgálandó kérés érkezik be. Látható, hogy forgalomszabályozás nélkül túlterhelődik a hálózatban szereplő szűk keresztmetszet, ami ezután drasztikusan kihat a teljes rendszer működésére. Mivel nincsen mechanizmus ennek az észlelésére, ezért nagyobb terhelés esetén a korábbi mennyiségű kérés kiszolgálása sem lesz lehetséges, ez okozza a sikeres kiszolgálások csökkenését.



(a) Összes CPU felhasználás

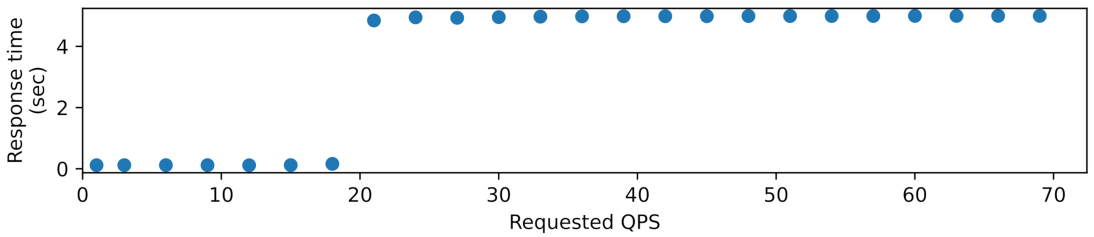


(b) Összes memória felhasználás

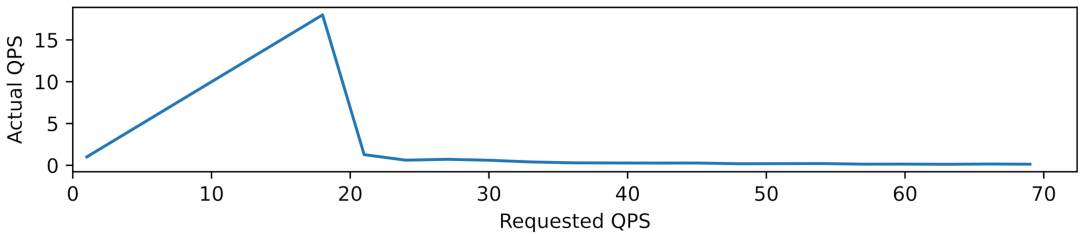


(c) CPU felhasználás konténerenként

(d) Memória felhasználás konténerenként



(e) Átlagos válaszidő



(f) Igényelt és sikeres kérések száma másodpercenként

5.3. ábra. Frontend három példányban és egy backend

A probléma orvoslására több megoldási irány is létezik. Ilyen például a rendelkezésre álló erőforrások optimálisabb elosztása, a túlterhelt egység skálázása vagy a hálózati forgalom szabályozása. Ha a beállítások dinamikusabbak lennének, akkor a frontend által lefoglalt és feleslegesen használt erőforrásokat át lehetne csoportosítani a backend részére, amivel növelhető lenne a rendszer globális maximum kiszolgálása.

Belátható, hogy egy valóságban előforduló problémáról van szó, amelynek vizsgálata indokolt.

5.5. Mérések automatikus horizontális skálázóval

A korábban végzett mérések alapján ki lehet jelenteni, hogy a podok statikus konfigurálásával nem mindig találjuk meg a rendszer ideális állapotát, sőt a jövőbeli terhelés pontos információja nélkül nagyon valószínű, hogy szuboptimális állapotban maradunk. A Kubernetes beépítetten támogatja a podok automatikus horizontális skálázását, mint ahogy az a 2.3.1.1 alfejezetben bemutatásra került. A továbbhaladás részeként szerettem volna megnézni, hogy milyen megoldást kínál a beépített skálázó és milyen limitációkkal rendelkezik. Következőkben bemutatásra kerülnek az új mérésekre történő átállás kihívásai, az elvégzett mérések és az általuk szerzett tapasztalatok.

5.5.1. Áttérés nehézségei

Az újfajta mérés több gondot is okozott. Egyik, hogy az eddigre bejáratott ábrázoló alkalmazás teljes módosítást igényelt, ugyanis ezen mérések jelentősen eltérnek a korábbiaktól. HPA esetében az egyes értékek az idő függvényében változnak, ellentétben a statikus méréseknél, amikor minden a beérkező terhelés változásával volt összefüggésben. További fejlesztést igényelt az operátor logikai oldala is, mert a kezdeti méréseknél kiderült, hogy instabil állapotot okoz, ha automatikus skálázó is szerepel a rendszerben. A jelenség onnan eredt, hogy minden egyes alkalommal, amikor az operátor változást észlelt és a logikája futtatásra került, ellenőrizte a ServiceGraph konfigurációban megadott replikaszámot. Amennyiben az igényelt replikaszám nem egyezett a valóságban futtatott számmal, akkor módosította azt a konfigurációban megadott értékre. Ezen logika miatt, amikor a skálázó szeretett volna több azonos podot elindítani egy adott alkalmazásból, akkor ezt észrevette az operátor és visszaállította a kezdeti értéket. A következmény pedig az lett, hogy a rendszer sosem került stabil állapotba és a skálázó döntései nem tudtak érvényre jutni.

A fenti probléma ideális példaként tud szolgálni az operátorok egy kevésbé kutatott és ritkán tárgyalt kihívására. Amennyiben több kezelő alá esik egy-egy Kubernetes erőforrás, akkor nagyon körültekintően kell eljárni az egyes feladatkörök elosztásában. Nem triviális észrevenni, ha direkt vagy indirekt módon, egymás döntéseire reagáltan futnak le az operátorok algoritmusai. Ezzel a rendszer egésze könnyen instabil vagy holtponti (*deadlock*) állapotba tud kerülni.

Egy-egy ilyen helyzetet felismerni aránylag körülményes, alapértelmezetten nincsen erre külön logika implementálva az orkesztrációs platformba. Emiatt ezen esetek detektálása, újabb operátor bevonása nélkül nem lehetséges. Azonban újabb logika bevonása, ami azonos erőforrások kezelését teszi lehetővé, tovább növelheti az indirekt egymásra hatások valószínűségét.

Az ismertetett hibákat fejlesztésekkel kellett megoldani, ami magába foglalta az ábrázoló alkalmazást és az operátor logikájának újragondolását is. A végleges implementációban a megadott replikaszám csak akkor kerül ellenőrzésre, ha automatikus skálázó nem kerül definiálásra az átadott konfigurációs fájl alapján.

5.5.2. Lokálisan mohó, globálisan szuboptimális

A korábban a 2.3.1.1 alfejezetben bemutatott HPA skálázó döntési mechanizmusát ismerve, sejthető volt, hogy milyen szituációkra viselkedhet érzékenyen. Maga az algoritmus, ami meghatározza az egyes podokból aktuálisan szükséges darabszámot egyedül az ő felelőssége alá tartozó konténereket veszi számításba. Ezen kívül nem rendelkezik egyéb ismerettel a többi szolgáltatást illetően. Például nem tudja, hogy milyen más alkalmazás komponensekkel van kapcsolatban az alá tartozó komponens illetve azt se, hogy mennyi erőforrás érhető el összesen a klaszterben.

A fenti tulajdonságok alapján a jelenlegi skálázó csak lokálisan optimális döntéseket tud hozni egy-egy alkalmazás egység számára. Azt kellett bebizonyítani, hogy ezen tulajdonsága miatt a rendszer összességét érintő, globális optimumot nem fogja megtalálni. Ehhez létrehoztam egy egyszerű szolgáltatás hálót és az egyes elemekhez tartozó horizontális skálázót. A konkrét értékeket az 5.5 táblázat tartalmazza. Látható, hogy a korábbi mérésekhez hasonlóan itt is egy frontend és egy backend alkalmazás szerepel. A kérések a frontendhez érkeznek be, ami mindegyiket továbbítja a backend felé. Az egyes komponens podok által lefoglalt és maximálisan használható erőforrások mennyisége megegyezik. Egy virtuális CPU magot használhatnak illetve egy megabájt memóriát. Annyi különbség van, hogy a frontend a beérkező kérésenként negyed annyi CPU használatot fog generálni, mint a backend.

Név		front-end-1	back-end-1
Kezdeti replika szám		1	1
CPU használat (mCPU)		25	100
CPU foglalás és limit (mCPU)		1000	1000
Memória foglalás és limit (kB)		1000	1000
Továbbhívás		back-end-1:80	-
HPA	Minimum replika	1	1
	Maximum replika	5	5
	Cél CPU használat	70%	70%

5.5. táblázat. Költséghatékony frontend és költséges backend - HPA skálázóval

A korábbi mérésektől eltérően ebben az esetben az operátorunk a konfiguráció alapján fog létrehozni automatikus horizontális skálázót is. Azonos szabályok kerültek alkalmazásra a két egységhez kapcsolódóan. A megadott paraméterek leírják, hogy a komponensekből legalább egy replikának futnia kell de legfeljebb öt, illetve a processzorhasználati cél is adott. Ebben az esetben a HPA törekedni fog a 70 százalékos foglaltságra, ami azt jelenti, hogy a maximálisan elérhető processzorhasználat 70 százaléknál magasabb használat esetén fogjuk megkezdeni a skálázást.

A generált terhelés hatására a rendszernek másodpercenként hetven beérkező kérést kellett kiszolgálnia. Az egyes *http* kapcsolatokra öt másodperces időkorlát volt, amin belül ha nem érkezett válasz, akkor az adott kérést sikertelennek tekintjük. A teljes szimuláció a szolgáltatásháló létrehozásától számított tíz percig, azaz hatszáz másodpercig tartott.

Mérés során parancssorból is jól látható a skálázó működése. Ezt mutatja be az 5.2 kódrészlet, amin belül két parancs kimenetét is láthatjuk a mérés kezdetét követő öt és feledik percben. Első sorban lekérdezzük az éppen működésben lévő horizontális skálázókat. A parancs kimenetének második oszlopából kiderül, hogy az egyes skálázók az azonos névvel rendelkező deployment erőforráshoz vannak kapcsolva. Leolvasható, hogy skálázó döntései alapján öt és négy replikának kellene futni ebben az időpillanatban illetve, hogy mind a frontend mind pedig a backend túl lépte a számára előirányzott 70%-os processzor fogyasztást.

A kódrészlet második parancsán láthatjuk, hogy milyen podok és milyen állapotban léteztek ilyenkor. Megfigyelhető, hogy valóban szerepel a korábban a skálázó által meghatározásra kerülő négy darab frontend és az öt darab backend kapszula. Viszont, ami tanulságos, hogy közülük nem mindegyiknek sikerült az elvárt módon elindulnia. Erre abból lehet következtetni, hogy egyes podok állapot

mezője nem az elvárt *Running*, hanem *Pending* állapotban vannak. Amiatt van ez, mert ugyan a skálázó döntése alapján el kellene induljon az elvárt számú kapszula, azonban a Kubernetes ütemezője nem tudja őket elindítani. Látható, hogy három plusz három kapszula tudott rendesen elindulni, így rögtön hat virtuális cpu magot le is foglaltunk a klaszterben és nem tud új egységet lehelyezni. Ez a magyarázata, hogy míg a korábban elindított egységek (magasabb *AGE* értékkel) sikeresen elindultak, azonban a későbbieknél ez már nem mondható el.

```
$ kubectl get horizontalpodautoscalers
NAME         REFERENCE                TARGETS   MINPODS MAXPODS REPLICAS AGE
back-end-1   Deployment/back-end-1    100%/70%  1       5       5        5m34s
front-end-1  Deployment/front-end-1   80%/70%   1       5       4        5m34s

$ kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/back-end-1-7cb9d49594-7btf8     1/1     Running   0          2m48s
pod/back-end-1-7cb9d49594-fpgcj     0/1     Pending   0          108s
pod/back-end-1-7cb9d49594-lhjvl     0/1     Pending   0          108s
pod/back-end-1-7cb9d49594-rvsgk     1/1     Running   0          5m34s
pod/back-end-1-7cb9d49594-vdl7m     1/1     Running   0          3m48s
pod/front-end-1-6677c56849-26nmn    1/1     Running   0          5m34s
pod/front-end-1-6677c56849-26zr2    1/1     Running   0          3m48s
pod/front-end-1-6677c56849-8s9ph    0/1     Pending   0          48s
pod/front-end-1-6677c56849-fndnw    1/1     Running   0          2m48s
```

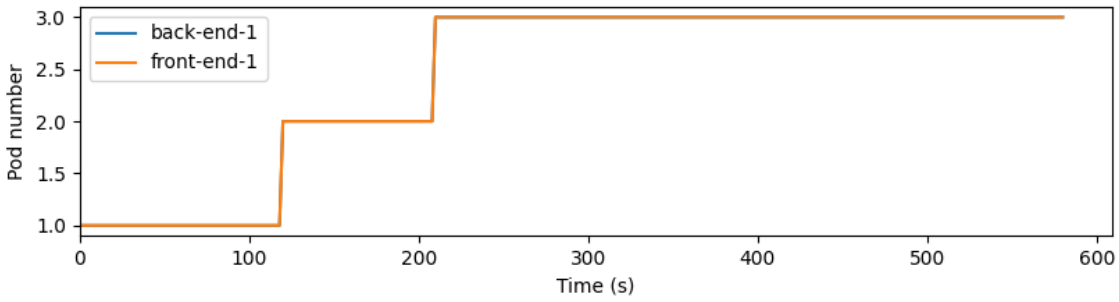
5.2. kódrészlet. Horizontális skálázóval történő mérés

A mérés eredményeit ábrázolva előkerült egy érdekesség is, amit az 5.4 ábrán láthatunk. A grafikonon egyszerre van ábrázolva a frontendből és a backendből futó egységek száma is. Erre utal a bal felső sarokban szereplő jelmagyarázat is, viszont a kék színnel rajzolt költségesebb backend nem is látszik. Ez amiatt van, mert a Prometheus adatai szerint egyszerre történtek meg a skálázások és ez az oka, hogy mindkét alkalmazásegységből három, három darab került elindításra. Értelemszerűen ez az erőforrások nem ideális elosztását vonja magával, hiszen azonos erőforrás került lefoglalásra és használatra a költségesebb illetve könnyebb alkalmazásrész számára is.

Elméleti optimum meghatározása A skálázó minősítéséhez meg kell határozni mi lenne az adott rendszerben ideális állapot. Melyik az az erőforrás-elosztás, amikor a legtöbb kérést tudja kiszolgálni. Ezt könnyű megtenni, ha rendelkezünk pár szükséges paraméterrel. Tudni kell a klaszter által kioszthatő erőforrás mennyisége, illetve azt, hogy abból hány darab pod indítható. Jelen esetben ez 6000 mCPU, amiből 6 darab kapszula kerülhet elindításra. Ezen felül tudni kell az egyes podok által maximálisan kiszolgálható kérések számát is. Ez is kiszámolható az 5.5 táblázatban megadott paraméterekből. Frontend számára 1 virtuális processzormag

áll rendelkezésre, és minden kiszolgált kérés igénye 25 mCPU. Ebből adódik, hogy egy frontend kapszula 40 kérés kiszolgálását tudja teljesíteni másodpercenként. Egy backend pod ezzel analóg módon $1000mCPU/100mCPU = 10$ kérés áteresztésére képes másodpercenként. Az elvégzett szimuláció esetén, mivel egy láncot alkotnak az alkalmazások, a leglassabb alkalmazás komponens fogja jelenteni a felső határt. Jelen példánál azt kapjuk, hogy 1:5 vagy 2:4 arányban elindítva a kapszulákat a kiszolgálás felső határa mindkét esetben 40 másodpercenkénti kérés lesz. Egyedül az a különbség, hogy ez a plafonérték a frontend vagy backend oldalán jelenik meg. Továbbá 1:4 arányban elindítva a szolgáltatásokat, szintén megkapjuk a 40 másodpercenkénti kérést, mert ez lesz mindkét egység felső határa. Ebben az esetben még a lefoglalásra kerülő erőforrásokon is sikerül spórolni, amit esetlegesen a rendszer másik pontjára lehet allokálni szükség szerint.

Tehát skálázó által alkotott megosztással a jelenlegi mérésben nem tudjuk maximálisan kihasználni a rendszerben lévő erőforrásokat. Pontosabban mondva kihasználjuk, mert a lefoglalt processzor mennyiségét jelentősen használjuk, azonban a kiszolgálás minőségén és mennyiségén ez nem látszódik.



5.4. ábra. HPA skálázóval történő mérés - 70% cél CPU használat mellett

A bemutatott eredményekből adódik a kérdés, hogy mennyire általános jelenségről van szó, vagy csak a skálázó számára átadott paraméterek módosításával lehet-e javulást elérni. Felmerült gondolatként, hogy a rendszer kezdeti állapotának változtatásával kaphatunk-e eltérő végeredményt. Ezekre a kérdésekre kellett válaszokat keresni, hogy megállapításokat tegyünk a skálázó működését illetően.

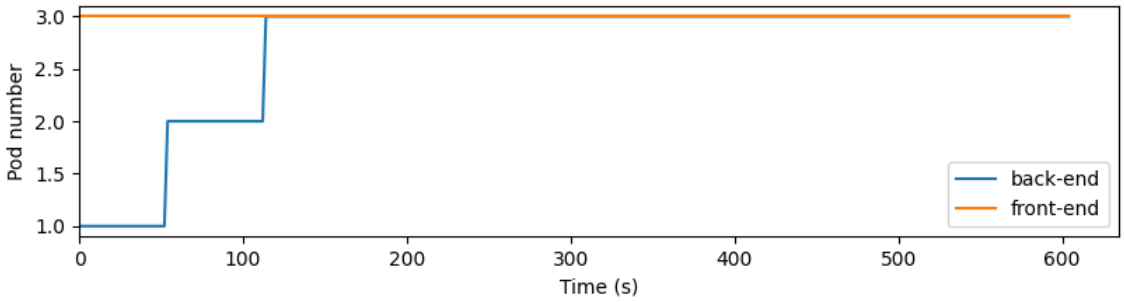
5.5.3. Mérés megismétlése, más indulási állapottal

További kérdés, hogy az 5.5.2 fejezetben bemutatott rendszer skálázása mennyire változik a kiindulási állapottól. Lehetséges, hogy az automatikus mechanizmus

indulása előtti állapotnak kihatása lehet a skálázási döntésekre és emiatt eltérő helyzetekből indítva a végeredmények is eltérőek lesznek.

A kérdés megválaszolásához több különböző helyzetből elindítva is elvégeztem a méréseket. Minden indításnál más replikaszámot adtam meg az egyes alkalmazáskomponensek részére és ilyen állapotban kezdte el a skálázó a monitorozásukat.

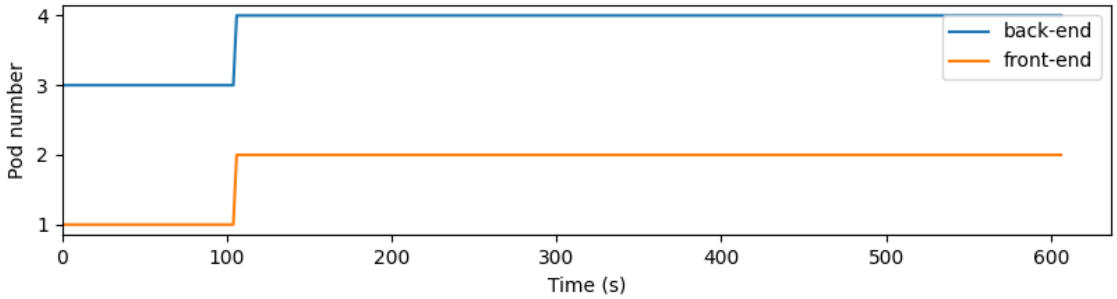
Első esetben három frontend és egy darab backend indult, az 5.5 táblázatban bemutatásra kerülő paraméterek mellett. A replikaszámok módosulását az 5.5 gafikon tartalmazza. Könnyen leolvasható, hogy a sárgával jelzett frontend podok mennyisége változatlan maradt a teljes mérés során, a skálázó döntései alapján nem volt indokolt a beavatkozás. Ezzel szemben a kék színű backend folyamatosan került felskálázásra, egészen addig, amíg ki nem egyenlítődött a két típusú egység replikaszáma. Ezzel együtt megkaptuk az eredeti mérés kimenetét, ahol szintén három, három egység került elindításra. Annyi különbség észrevehető, hogy ez a folyamat valamivel gyorsabban végig tudott menni és a szimuláció kezdetét követő második percben már a végleges replikák futottak.



5.5. ábra. HPA skálázóval történő mérés - 3 frontend és 1 backend kezdetben

Következő mérés elindítására az optimálisnak vélt erőforrás használathoz közelebbi állapotból került sor. Az 5.6 ábrán látható, hogy kezdetben egy frontend és három backend egység alkotta a rendszert. Ezután egy lépésben mindkét komponens replikaszáma növelésre került azonos időben. Ezáltal el is érték az aktuális klaszterben szereplő ideális eloszlást.

Az elvégzett mérések és a bemutatott eredmények alapján kijelenthető, hogy a Kubernetes beépített skálázó által adott végeredmény nem független a klaszter eredeti állapotától. A kapott erőforrás-allokációk az egyes alkalmazásegységek számára ideálisnak tartott állapottól való kezdeti távolságától nagymértékben függ. Mivel minden skálázó, csak a saját hatáskörébe tartozó kapszulákat kezeli és azonos időben hoznak döntést, ezért az optimális állapotnak még a szabad erőforrások lefoglalása előtt be kell állni, mert utána erre már nem lesz képes.



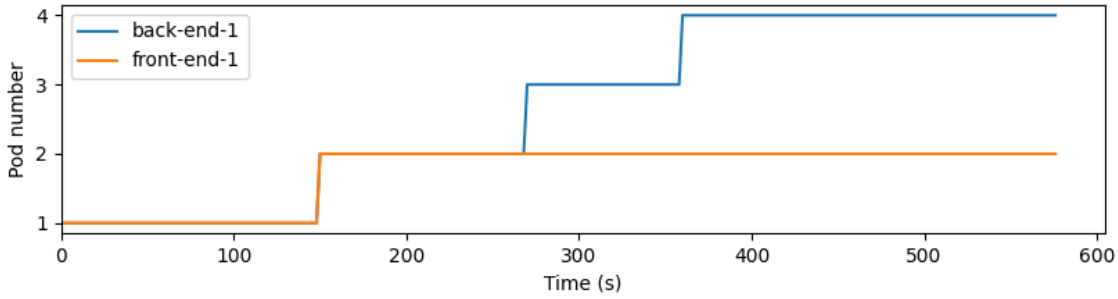
5.6. ábra. HPA skálázóval történő mérés - 1 frontend és 3 backend kezdetben

5.5.4. Processzorhasználati célérték konfigurálása

A korábbi mérek során felmerült, hogy azért történtek egyszerre a skálázások, mert a beállított cél processzorhasználati érték túl alacsonyan volt. Az ideális érték meghatározása függ az alkalmazástól, illetve a beérkező kérések ingadozásától is. Könnyen belátható, hogy minél nagyobb változásra számítunk, az időben annál nagyobb biztonsági tartalékot kell biztosítani az adott alkalmazásegységek részére. Amikor fontos, hogy kiesés nélkül lehessen kiszolgálni az összes beérkező és várhatóan még beérkező igényt, akkor az eredeti mérésnél állított 70%-os érték jó kiindulást jelenthet. Ezzel szemben a szimulációnk során a terhelés mértéke nem változott, a teljes mérés alatt azonos ütemben és azonos mennyiségű kérést intéztünk a hálózat felé. Emiatt megfontolandó volt nem a kiindulási állapoton, hanem a célértéken is változtatni és megnövelni azt. A megnövelt érték 90% lett, mert így kellően nagy részét kihasználjuk a lefoglalt erőforrásnak és mégis marad egy 10%-os biztonsági tartalék.

A lefutott szimuláció végeredménye látható az 5.7 ábrán. Leolvasható, hogy kezdetben egy kapszula futott a két alkalmazásból, és az első skálázásra együtt kerül sor. Ezután viszont a fronted erőforrás használata nem lépte túl az előirányzott értéket és emiatt további növelésre nem volt szüksége. Ezzel szemben a magas terheltségű backend képes volt fokozatosan skálázódni és a mérés végére négy darab replika futott belőle. Ezzel a beállítással sikerült elérnie az elméleti optimumot a rendszerünknek.

Összefoglalva, a HPA konfigurációjával és a rendszer kiindulási állapotától függ az alapértelmezett skálázó által megtalált állapot. Látható, hogy a rendelkezésre álló szűk erőforrások miatt fontos a körültekintő kezdeti konfiguráció, hiszen ezek hiányában csökken a kiszolgálások hatékonysága, feleslegesen használjuk és tartjuk lefoglalva a rendelkezésre álló erőforrásokat.



5.7. ábra. HPA skálázóval történő mérés - 90%-os cél CPU érték

5.5.5. Skálázás során jelentkező időkorlátok

A klaszter létrehozása közben csak a feltétlen szükséges konfigurációk kerültek beállításra és minden más pedig az alapértelmezett értéken maradt. Többek között a HPA működésének befolyásolását is meg lehet tenni akár a klaszter telepítése közben, akár pedig utána. Erre azért van mód, mert bizonyos paraméterek változók módosításával befolyással bírnak az algoritmus működésére[17]. A következőkben szeretnék bemutatni néhány ilyen paramétert.

- `horizontal-pod-autoscaler-initial-readiness-delay` - Amikor elindul egy új kapszula, akkor a skálázó még nem tudja kezelni azt, mert nem áll rendelkezésre kellő mennyiségű információ. Emiatt a kapszula létrejötte után a paraméter által megadott ideig úgy van kezelve, mintha még nem lenne készen, és nem kerül skálázásra sem. HPA specifikusan lehet állítani, az egyes alkalmazások egyéni szükségleteihez, de van egy központi értéke is. Alapértelmezettként harminc másodperc.

- `horizontal-pod-autoscaler-sync-period` - A Kubernetes skálázója nem folyamatosan gyűjti az adatokat és tesz beavatkozási lépésre javaslatokat, hanem időszakosan történik ez meg. Ezen periódus időtartamát tudjuk befolyásolni ezzel a kapcsolóval, ami alapértelmezetten 15 másodperc.

- `horizontal-pod-autoscaler-downscale-stabilization` - Nincs megkötés, hogy egy kapszulát csak egy HPA és egy metrika alapján kezelhet. Emiatt előfordulhat, hogy több skálázási javaslat is létezik azonos erőforráshoz. Ez inkonzisztenciához vezetne, ha mindig a legutolsó ajánlat kerülne érvényre. Ezért a végső döntésnél a fenti paraméter által megadott időt nézi, ami alapértelmezetten öt perc és ezen idő alatti legnagyobb érték lesz beállítva.

A paraméterek teljesen tetszőleges módon módosíthatóak, azonban figyelni kell, mert a klaszter teljes működösére kihatnak. Bizonyos esetekben szükséges lehet, ha a pod elindulása után több időt töltünk monitorozással és csak utána teszünk skálázásai javaslatot, ilyenkor módosíthatjuk az első megadott paramétert. Szintén adódhat olyan felhasználási terület, amikor a kapszulák terheltsége gyorsan ingadozik, ilyenkor érdemes lehet a szinkronizációs perióduson és a leskálázási időn is módosítani.

6. fejezet

Megoldási lehetőségek

Ebben a fejezetben szeretném bemutatni, hogy a korábban az 5 fejezetben látott mérések alapján milyen megoldási lehetőségek jöhetnek számításba. Természetesen minden bemutatott megoldásnak megvan a saját erőssége és gyengesége, amiket a következő alfejezetekben részletesen is ismertetek.

6.1. Feltárt probléma rövid összefoglalása

A megoldási lehetőségek bemutatása előtt szeretném röviden bemutatni a problémát és a felvetést, amire keressük a megoldást. A mérési eredmények alapján arra jutottam, hogy a mikroszolgáltatások közötti szabályozatlan kommunikáció miatt egy túlterhelt rendszer nem képes optimális működést biztosítani. Létezik egy átbukási pont, amikor a beérkező kéréseket az alkalmazás már nem képes időben kiszolgálni, mert az egyik komponense túlterhelt állapotba kerül. Ilyenkor a beérkező kérések továbbra is fogadásra kerülnek és elkezdődik a rendszerben lévő pufferek megtöltése. Ez egy öngerjesztő folyamatot indít meg, ahol a hosszabb sorbaállás, a folyamatos túlterheltség miatt aránylag egyre kisebb lesz a sikeres kiszolgálások száma. A hatékonytalan működést tovább rontja, hogy az előre beállított idő túllépése után bontjuk a kapcsolatot, azonban ezt az alkalmazás nem tudja lekezelni. Nem létezik implementált megoldás arra az esetre, hogy az ilyen kérések által keltett az egyéb alkalmazás egységek terhelését megszüntesse. Értelemszerűen ebben az esetben felesleges még a backend oldalán elvégezni az erőforrás intenzív feladatot, amikor az azt kiváltó eredeti kérés már eldobásra került.

A tapasztalt működés több irányból is megközelíthető. Egyik ilyen irány, hogy az egyes komponensek darabszámát automatikus skálázó segítségével határozzuk meg. Ezzel lehetőséget kapunk, hogy előre meghatározott cél processzorhasználati értékhez rendeljük a skálázási döntéseket. A megoldás észleli az esetlegesen terhelt

komponenseket és megpróbálja azokat horizontálisan skálázni. Amennyiben limitált erőforrással rendelkezünk a klaszteren belül, akkor elmondható, hogy léteznek olyan skálázási helyzetek, melyeket a Kubernetes jelenlegi skálázója nem tud ideálisan kezelni. Ez a működés onnan fakad, hogy a futó alkalmazásról nem rendelkezik globális ismerettel, csak az egyes független szolgáltatásokat kezeli a többitől függetlenül. Nem tudja megítélni, hogy a rendszer kiszolgálási paraméterek növeléséhez hova

Másik megközelítés, hogy a rendszerbe beérkező kéréseket szabályozzuk vagy pedig észleljük a túlterhelt állapotban lévő komponenseket és valamilyen logikával próbáljuk csökkenteni azok terheltségi szintét.

6.2. Lehetséges eszközök

További feladatom volt, hogy keressek olyan kiegészítést vagy javaslatot, ami feltárt hiányosságokat javítani tudná. A szimulációk futtatása alatt és források keresése közben számos megoldási lehetőség felmerült. A bemutatott eszközök elemzése során látni fogjuk, hogy kicsit más megközelítésből és a probléma más aspektusát célozva próbálja a feltárt hatásokat csökkenteni.

Az egyes eszközök értékelésénél az is fontos szempont volt, hogy a lehető legkevesebb módosítást kelljen végrehajtani az alkalmazás oldalán. Ez fontos, mert a megvalósítani kívánt feladat nem tartozik szorosan az alkalmazáshoz és ideális esetben teljesen transzparens módon működne. Természetesen több dolgot is mérlegelni kell az adott helyzetben ideálisnak ítélt megoldás kiválasztása közben.

Fontos azt is megfontolni, hogy egy az infrastrukturális részekre hatást gyakorló eszköz milyen információk alapján engedjük, hogy dolgozzon. Természetesen minél több információval rendelkezünk az adott alkalmazást illetően, annál közelebbi megoldásokat tudunk nyújtani az optimálishoz. Például, ha egy okos skálázónak rálátást adunk az alkalmazás hálózatára, az egyes egységek által használt erőforrásokra, a köztük lévő kapcsolatra és esetleg saját metrikákat is kivezetünk, akkor a skálázási döntés meghozatalában ezeket mind számításba tudjuk venni. Másik oldalról viszont aggályos lehet, ha ilyen szintű monitorozást és belelátást biztosítunk, mert ezáltal következtetéseket tudunk levonni a futtatott alkalmazásról. Ez pedig bizonyos esetekben az ügyfél érdekeit sértheti.

6.2.1. Beépített állapotjelzők

A Kubernetes kapszulák létrehozása közben, beépített módon lehetőségünk van a kapszula állapotáról jelzési pontokat meghatározni. Ezzel a megoldással sokrétűen használható funkciókat kapunk, amivel közvetetten több dolog befolyásolására nyílik lehetőségünk. Három jelzési módszerünk van, amit három különböző esetre találtak ki. Az alapvető felvetés onnan fakad, hogy különböző, gyakran előforduló helyzetekre adnak megoldási lehetőségeket különálló működésük által. Az egyes állapotjelzők ideális használati módját az alábbi szituációkkal szeretném szemléltetni.

1. **Életteli próba (liveness probe)** - Előfordulhat, hogy az alkalmazásunk valamilyen bemenetek következtében holtponti állapotba kerül, amit nehéz lenne kívülről észrevenni, viszont külső behatás nélkül nem tudna továbblépni belőle. Illetve egyéb okok miatt is kerülhet olyan állapotba, amikor a vizsgált alkalmazás nem képes ellátni a működését, mindezt különösebb hiba és kilépés nélkül. Szerencsére az ilyen esetek nem számítanak különösnek és hamar fel is lehet oldani az egység újraindításával, amit az életteli próbával tudunk kezelni. Ez egy teljesen ideális implementáció esetén nem fog gondot okozni, hiszen a kapszulák nem tartalmaznak, tárolnak fontos adatokat és állapotokat. Egy újraindítás idő- és költséghatékonyabb megoldás, mintha az egész rendszer funkcionális működését veszélyeztetnénk.

2. **Indítási próba (startup probe)** - Eshetőség, hogy az alkalmazás elindítása után még el kell végezni pár inicializációt, mielőtt a külső kérések kiszolgálását elkezdhetné. Ebben az esetben is tudatni kell az infrastruktúra részére, hogy a jelenlegi állapotában még nem áll készen a működésre. Nagy hibázási lehetőséget rejtene magába, ha erre a korábban látott életteli próbát használjuk, mert nehéz kiszámítani mennyi időt fog igénybe venni a komponens elindulása. Külön emiatt létezik az indítási próba, ami csak akkor fogja az adott kapszula állapotát készenléti státuszba sorolni, ha sikeresen lefutott a próba.

3. **Készenléti próba (readiness probe)** - Tegyük fel, hogy van egy szolgáltatásunk, ahol nagy fájlok feltöltésére van lehetőségünk. Ez egy időigényesebb feladat lesz, miközben az alkalmazásunkat nem szeretnénk további kérésekkel terhelni. Ebben az esetben meg kell várni a korábban érkező kérés kiszolgálását és csak utána van lehetőségünk fogadni a többit. Ilyen helyzetben tudjuk használni a készenléti próba jelzést. Amennyiben a próba sikertelen vol, tehát jelenleg nem engedhetünk új kiszolgálást, akkor a kapszula IP címe ki fog ke-

rülni erre az időszakra a hozzá tartozó *Service* alól, így forgalom se fog eljutni hozzá.

A fentebb bemutatott eszközök megkönnyítik a robusztus rendszerek építését. Segítségükkel az előforduló hibák gyors javítására tudunk fókuszálni, és nem cél azok mindenáron történő megakadályozása. A szemléletmód kihasználja, hogy az új konténerek elindítása általában pár másodperc alatt megtörténik. Továbbá ezen folyamat által egy öngyógyító tulajdonságot is kap a rendszer, amivel a megbízhatósága is javulni fog.

Ennek ellenére körültekintően kell ezeket az eszközöket is használni, hiszen rossz konfiguráció esetén mi magunk tehetjük működésképtelenné a rendszert. Például, hibás beállítás esetén az életteli próba hajlamos állandóan újraindítani az alkalmazást. Vagy a készenléti próba miatt egyes komponens egységek nem fognak annyi kérést kapni, amit azok valóban képesek lennének kiszolgálni.

Módszer tesztelése A következő feladat az volt, hogy a bemutatott állapotjelzők használatával ismét el kellett végezni a korábbi mérést és megnézni az elért hatását. Számomra a készenléti probléma volt a legérdekesebb, hiszen ez a megoldás ad lehetőséget arra, hogy addig ne érkezzen kiszolgálandó kérés az adott alkalmazás egységhez, amíg egy bizonyos állapot teljesül.

A mérés elvégzéséhez fel kellett készíteni a korábban megírt operátort, hogy a *Deployment* létrehozása közben egy készenléti próba definíciót is építsen bele. Valamint minden egyes alkalmazásegységnek rendelkeznie kell egy olyan végponttal, ami nem jelent számottevő többlet terhelést, és az ide érkező kérések kiszolgálása alapján tudja megmondani a rendszer jelenlegi terheltségét. Amennyiben a próba sikertelen, tehát a rendszer túlterhelt állapotban van, akkor nem szabad számára több kérést továbbítani, amíg nem csökken a kiszolgálásra váró kérések sora.

A módszer hatékonyságának vizsgálata érdekében arra gondoltam, hogy egy korábban már bemutatott mérést fogok megismételni, csak kibővítve az állapotjelzővel. Ehhez az 5.4.2 alfejezetben bemutatott mérést választottam. A szimuláció elvégzését követően kapott eredmények sajnos nem váltották be a hozzájuk fűzött reményeket, és nem javultak számottevő mértékben a kiszolgálási paraméterek. Összességben hasonló tendenciát lehetett megfigyelni, mint a korábbi, próba nélküli mérés esetén. A mérés elején folyamatosan emelkedett a kiszolgált kérések száma, azonban az eredeti méréssel azonos időben és helyen megindult a meredek visszaesés is. Ez amiatt lehetett, mert a beérkező kérések kiszolgálása továbbra sem tudott megtörténni az öt másodperces időkorláton belül, ahogy az a 6.1 ábra tetején is látszik. Az eredeti

6.1. ábra. Korábbi szimuláció megismétlése készenléti próbával

méréssel azonos tendenciát figyelhetünk meg és azonos helyen van az átbukási pont, amikor a rendszer terhelését növelve drasztikusan visszaesik a sikeres kiszolgálások száma. Ebben az esetben is azt látjuk, hogy a backend CPU felhasználása hamar egy plafonba ütközik, amin túl kitörni már nem tud. Illetve ezzel párhuzamosan a költséghatékony frontend továbbra is fogadja a beérkező kéréseket, ami miatt további erőforrásokat fog elhasználni. A korábbi méréseknél a backend részére 2000 mCPU és a frontend kapszuláknak pedig 1000 mCPU limitáció volt beállítva. Ezen limitek a mostani méréseknél sem változtak, azonban az ábráról tisztán leolvasható, hogy a felhasznált erőforrás mennyisége a korábbitól jelentősen elmaradt. Azt láthatjuk, hogy a beállított próba hatására a backend és a frontend által elhasznált erőforrások mennyisége lecsökkent. Az lehet erre a magyarázat, hogy a backend, amikor érzékelte, hogy kezd túlterhelődni, akkor egy kisebb időszakra szüneteltette a kérések fogadását, ami miatt arányaiban csökkent a mérések során elhasznált processzor mennyisége.

Sajnos azonban a beállított próbával sem sikerült elérni, hogy a kiszolgálási mutatók megfelelő mértékben javuljanak, hiszen a megismételt mérésben is egy megadott pont után visszaesett a sikeres kiszolgálások száma. Jelen állásban minden alkalmazás egység saját magának a terheltségét tudja monitorozni, ami miatt hiába van a sor végén lévő backend komponens leterhelve, ezt a frontend még nem érzékeli és továbbra is fogadni fogja a felé irányított kéréseket. Esetleg érdemes lehet további finomításokat bevezetni és egy saját szkripttel ellenőrizni a láncolat végén elhelyezkedő backend állapotát a frontend oldaláról is. Ezzel megoldhatóvá válna,

hogy a lassú komponenshez igazodva tudjuk a kéréseket fogadni már a kiszolgálási lánc elején.

Természetesen a vázolt megoldás is tartalmaz néhány kifogásolható aspektust. Az ilyen módon összeállított rendszer egy másik komponens állapotát monitorozza és ez alapján tudja a saját készenléti állapotjelzőjét állítani. Statikusan kell ezen próbákat beállítani és a terhelés változásával változhat a szűk keresztmetszet is. Például egy valódi forgalmat elemezve rendszeresen megjelenik, hogy periodikusan lesznek terhelve a komponensek. Nap elején elképzelhető, hogy a bejelentkeztető komponens lesz túlterhelt és szűk keresztmetszet, míg nap közben pedig az adatok validációjáért felelős szolgáltatás. Ezen változásokat pedig nehéz lehet lekezelni, egy egyszerűbb szkripttel.

6.2.2. Konténer specifikus metrikák alapján

Következőnek ismertetett megoldási javaslat szintén a beépített Kubernetes erőforrásokat és azok funkcióit kívánja felhasználni. Ezen megoldásoknak előnye, hogy a klaszter oldaláról nem igényelnek nagy mértékű többlet fejlesztést és támogatást, hiszen ezek a funkciók valószínűleg rendelkezésre állnak már a rendszerben vagy pedig könnyen telepíthetőek. A mérleg másik oldalán ezen megoldások az alkalmazás oldalán igényelnek fejlesztéseket vagy pedig a helyes konfiguráció okozhat kihívást.

Még korábban, a 2.3.1.1 alfejezetben bemutatott HPA skálázás esetén említésre került, hogy tetszőleges metrikák alapján is van lehetőségünk skálázni. Alapértelmezetten és leginkább a mérések alatt általunk is használt processzor erőforrás igény alapján történő skálázás a legelterjedtebb megoldás. Ezen metrikákat a klaszter automatikusan tudja gyűjteni, így a skálázók konfigurációja és elindítása egyszerű folyamattá tud válni. Illetve a legtöbb esetben, ha a klaszterünkben nincsenek szűkös erőforráshatárok aránylag jó működést eredményez.

A megoldási lehetőség hátránya, hogy ezen implementáció is lokálisan az egyes alkalmazásegységek skálázását fogja végezni. Így pedig, amikor az erőforrások allokációja nagymértékű nem képes a rendszer áteresztőképességét figyelembe venni.

Továbbra sincs dinamikus algoritmus a skálázó mögött, így az indítás pillanatában beállított konfigurációk alapján fogja meghozni a beavatkozási döntéseket. Emiatt a kezdeti konfigurációnál nagyon alaposan végig kell gondolni, hogy milyen szabályokat szeretnénk alkalmazni. Esetlegesen rosszul felmért metrikák alapján többlet erőforrásokat is lefoglalhatunk a kiszolgálási metrikák javulása nélkül.

A megoldás felvetése, hogy az alkalmazás logikai rétegében legyenek követve, mentve és exportálva olyan metrikák, melyek később az alsóbb szintű infrastruktúra

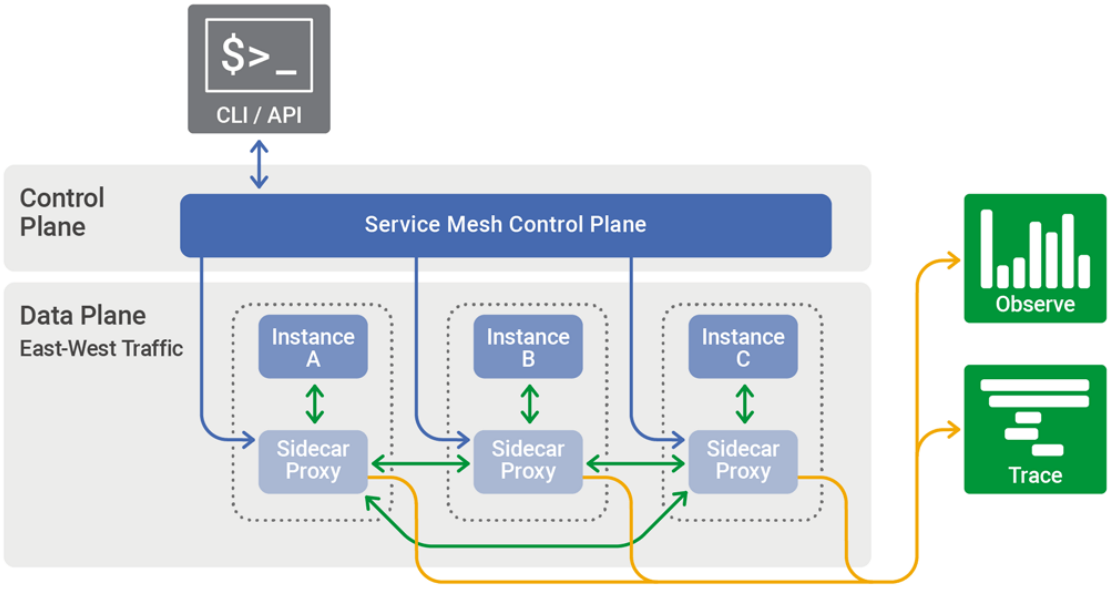
réteg működését fogják befolyásolni. Ezt szintén mérlegelni kell, hiszen nem tartozik szorosan az alkalmazás logikai részéhez illetve bizonyos mértékű többlet fejlesztést is igényel.

6.2.3. Szolgáltatás hálók által nyújtott lehetőségek

Több új kihívás is megjelent a mikroszolgáltatások elterjedésével, amivel szembe kellett nézni a fejlesztőknek és az üzemeltetőknek is. Ezen kihívások mentén született meg a szolgáltatás hálók (*service mesh*) fogalma. Fontos megjegyezni, hogy a fejezetben tárgyalt szolgáltatás háló kifejezés a dolgozatban szereplő többi előfordulásán túl egyéb megkötéseket is tartalmaz. Emiatt szeretném tisztázni, hogy a következőkben használt szolgáltatás háló alatt mit értünk.

A szolgáltatás háló egy lehetséges módja, hogy szabályozzuk az alkalmazásrészek közti kommunikációt. Külön infrastruktúra réteget tudunk létrehozni, amin keresztül láthatjuk és szabályozhatjuk a komponensek közti kommunikációt. Ezzel lehetőséget kapunk a kommunikáció optimalizációjára, amivel együtt az esetleges kieséseket is csökkenthetjük[29].

Az egyes alkalmazás komponenseket nevezzük egy-egy szolgáltatásnak, és a kommunikációjuk által alkothatunk belőlük hálózatot. Több megvalósítás is létezik, azonban a felépítésük nagyon hasonlít. Egy ilyen architektúra bemutatása látható a 6.2 ábrán.



6.2. ábra. Szolgáltatás hálót alkotó komponensek és működésük[11]

Az ábrán *Instance A/B/C* névvel ellátott alkalmazás egységek valósítanak meg egy-egy szolgáltatást az elképzelt mikroszolgáltatás architektúrában. Jól látható, hogy köztük nem történik semmilyen közvetlen kommunikáció, csak a mellettük lévő oldalkocsival (*sidecar*) vannak kapcsolatban. Ezek az oldalkocsik tipikusan egy konfigurálható proxyt tartalmaznak és az adott szolgáltatás mellett üzemelnek. Ezt a mintát nevezik a oldalkocsis mintának (*sidecar pattern*)[23]. Minden, a szolgáltatások által küldött és fogadott kérésnek keresztül kell menni ezen a proxyn, ami pedig így belelátást kap a csomagokba illetve azok továbbítására is ráhatása van. Ezáltal nyomon követhető és megfigyelhető lesz a komponensek közti kommunikáció. Ez önmagában is előnyös, hiszen könnyebben meg lehet tudni, hogy az egyes szolgáltatáshálók milyen kiszolgálást tudnak nyújtani, könnyebb kideríteni az aktuálisan legszűkebb keresztmetszetet. Természetesen a rendszerbe kötött proxykat kezelni is kell, amire egy külön réteg van, azonban ez már nagyobb mértékben specifikus az egyes megvalósításokra.

Az új architektúra is rendelkezik bizonyos megkötésekkel, amit az alkalmazása előtt figyelembe kell venni. Mivel új komponenseket hozunk be a kommunikációba, ezért a késleltetések is meg fognak nőni. Továbbá megfontolandó, hogy a proxyk működtetése is többlet erőforrás-használatot fog eredményezni. Természetesen növelni fogja a processzor foglaltságot és használtságot, valamint az alkalmazás futtatásához memóriára lesz szükség. További hátránya, ami egyben az előnye is, hogy egy új komplexitást hoz be az amúgy is összetett alkalmazás üzemeltetésbe. Ezért az architekturális döntés meghozatalában körültekintően kell eljárni.

6.2.4. Okos skálázó

A korábban bemutatott megoldási lehetőségeknél, 6.2.2 alfejezetben látottat leszámítva, a fő motiváció a beérkező terhelés szabályozása volt. Az eredeti problémánkat viszont több irányból is meg lehet fogni. Másik megközelítésben az erőforrások globálisan optimális elosztása a fő szempont, a megoldandó kihívás.

A következő megoldási javaslat ezt a megközelítést veszi alapul és bővíti ki a korábban látott javaslatokkal. A javaslat egy központi skálázó alkalmazás, ami folyamatosan monitorozza a klaszter teljes állapotát és az éppen futtatott alkalmazásokat is. A skálázási javaslatokat pedig ezen információk összességéből tudja meghatározni. Így figyelembe tudja venni az egyes kapszulák minőségi osztály besorolásuktól kezdve az egyes komponensek közti forgalmak megosztásán keresztül az esetlegesen exportált alkalmazás metrikákig.

Ezzel a megoldással lehetőségünk lenne az alkalmazás egységek kezelése helyett a teljes rendszer számára ideális döntéseket meghozni. Akár több névtérben futó rendszerek erőforrás allokációit is össze lehet hangolni, ami további optimalizálást jelentene.

A bemutatott javaslatok közül ez a megoldás a legkomplexebb is. Ez egyértelműen a Kubernetes platform nyílt forráskódjának bővítésével járna. Egy ilyen projekt jelentős szakmai erőforrás ráfordításával járna, hiszen az újfajta skálázóhoz új erőforrásokat is létre kell hozni illetve a kezeléshez szükséges logikát implementálni is kell.

Nem elhanyagolható felvetés, hogy a skálázási algoritmus logikájának bonyolítása jelentős kockázatokkal járhat, ha annak paraméterezését a felhasználóra bízzuk.

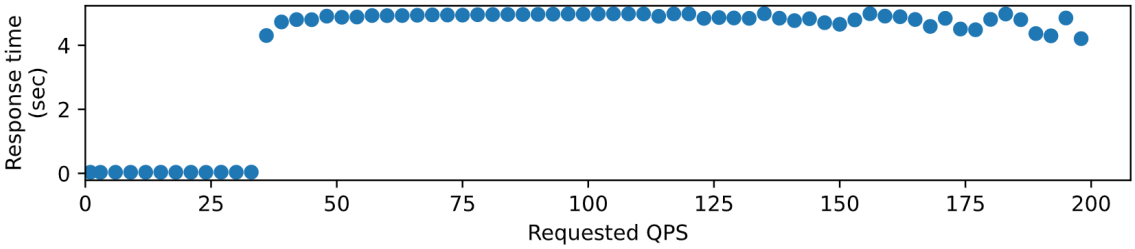
Analitikus modell Az okos skálázó optimális működéséhez szükséges jobban megvizsgálni a horizontális skálázás hatását. A vizsgálandó kérdés, hogy egy szolgáltatáshálóban elvégzett skálázás milyen kihatással bír a többi komponensre, és a teljes rendszer kiszolgálási értékei hogyan alakulnak. Meg kellett vizsgálni, hogy milyen összefüggés figyelhető meg a futtatott podok száma és az általuk kiszolgált másodpercenkénti kérések száma között.

Az alap feltevés látható a 6.1 egyenleten, ahol $f()$ függvény adja meg, hogy mennyi kiszolgálást tud teljesíteni az adott komponens az n-nel jelzett podok függvényében. Az $f()$ függvényt nem ismerjük, teljesen alkalmazásspecifikus, azonban a maximálisan elviselt terhelést a mérések alapján és n-et igen.

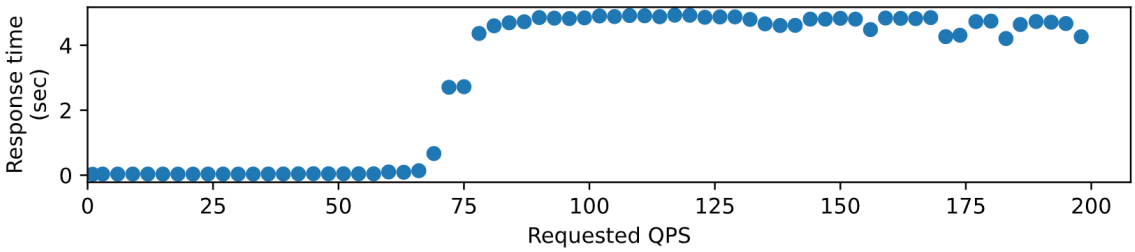
A feltevés az, hogy ezen ismeretekkel becsülhető a skálázás utáni kiszolgált másodpercenkénti maximális kérések száma, melyet Q' jelöl. Ezt úgy kaphatjuk, meg, hogy az ismert Q értékét elosztjuk az azt kiszolgáló kapszulák számával (n), amivel megkapjuk, hogy hozzávetőlegesen egy pod mennyi kérést tud kiszolgálni azonos konfigurációk mellett. Majd az így kapott eredményt felszorozzuk a skálázás utáni podok számával, ami n'.

$$\begin{aligned} f(n) &= Q \\ f(n') &= \frac{Q}{n} * n' = Q' \end{aligned} \tag{6.1}$$

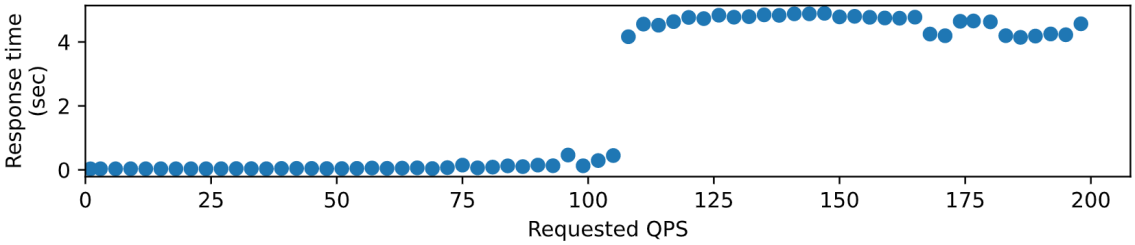
A tézis ellenőrzése szimuláción keresztül történt. A szolgáltatáshálóban egy gyors és erőforrás hatékony frontend szerepelt és utána pedig egy erőforrás-igényes backend. A backend számának növelésével szimuláltam a horizontális skálázás működését. A kapott eredmények láthatóak a 6.3 ábrán.



(a) Válaszidők egy frontend és egy backend egység esetén



(b) Válaszidők egy frontend és két backend egység esetén



(c) Válaszidők egy frontend és három backend egység esetén

6.3. ábra. Túlterhelt elem skálázása utáni kiszolgálási válaszidők

A legfelső[6.3a] grafikon mutatja azt az esetet, amikor egy darab backend konténer szolgálta ki az összes frontend felől érkező kérést. Leolvasható, hogy ebben az esetben körülbelül 35 QPS terhelés mellett sikerült időben kiszolgálni az össze beérkező kérést.

Amikor két példányban indult el a backend komponens, akkor az elvárt módon meg is ugrik az időben kiszolgálható lekérdezések száma. A grafikonról[6.3b] leolvasva az így kapott QPS érték 66 körül alakul, de nem éri el a 70-et.

Végül pedig az látható az alsó grafikonon[6.3c], amikor három példány fogadta a backend felé érkező kéréseket. Ebben az esetben nagyjából 108 kérés kiszolgálása vált lehetővé.

A kapott eredményekből látható, hogy visszaosztva az egyes méréseknél kapott maximális QPS értéket a futtatott backend podok számával megkapjuk, hogy egy kapszula 33-36 kérés kiszolgálását tudja elvégezni. Ez az érték kismértékben eltér az egyes szimulációkban, azonban kellően statikus marad végig, hogy modellként alkalmazzuk.

7. fejezet

Összefoglalás

7.1. Elvégzett munka

Szeretném röviden összefoglalni a diplomamunkában tárgyalásra kerülteket és ezáltal az elvégzett munkát is, hogy a korábban végigvezetett gondolatmenet részei egészben is megjelenjenek.

A fő vizsgálandó terület a terhelésfüggő HPA skálázó működése volt szolgáltatáshálókon. Ez a kérdés egyre több ágazatot érint, ahogy egyre többen választják az alkalmazásaik futtatására a felhőt, mint platformot. Bevezetésként felvázoltam ezen folyamatot, annak beazonosítható elemeit és a jelenlegi tendenciákat.

Ezután a technikai részletek és megvalósítás előtt bemutattam a diplomadolgozat központi szerepét játszó Kubernetes klaszter felépítést, és a benne elérhető erőforrások szabályozását és a beépített skálázóját.

Az érdemi feladatok előtt fel kellett mérni a szakmán belül elérhető korábbi kutatásokat. Ismertettem a főbb elméleti vonalakat és ötleteket gyűjtöttem a saját munkához.

A kérdések megválaszolásához sok mérést kellett végezni, melyhez nélkülözhetetlen volt implementálni egy erre alkalmas környezetet. A környezet több különálló részből állt. Létre kellett hozni egy saját erőforrást a Kubernetesen belül és implementálni egy kezelő logikát hozzá. Szükség volt egy tetszőlegesen konfigurálható alkalmazásra, ami előre beállított végpontokra érkező kéréseket szolgál ki. A mérés vezénylését és a kapott eredmények megjelenítését egy-egy külön alklmazás végzi.

Méréseken keresztül először megismertük, hogyan viselkednek a szolgáltatáshálók forgalomszabályozás nélkül egy túlterhelt rendszerben. Láttuk, hogy egy átbukási pont után a kiszolgálási paraméterek nagymértékben visszaesnek.

Következő kérdés, amire válasz született, hogy ezeket az eseteket mennyire csillapítja a HPA. Láthattuk, hogy szűkös erőforrások esetén nem minden esetben sikerül elérni az elméleti optimum pontot, és a végeredmény függ a rendszer eredeti állapotától és a jól beállított konfigurációtól. Amennyiben kellően nagy mennyiségű erőforrás áll rendelkezésre, akkor a HPA képes lenne a lokálisan optimális döntések által mindig növelve a szűk keresztmetszetet, egy globálisan is elfogadható állapotba kerülni.

A szolgáltatáshálók és a skálázó esetében megfigyelt limitációkra több megoldási javaslatot is adtam, melyek a probléma más-más aspektusát megragadva próbálják megszüntetni vagy csökkenteni azt. Ezen javaslatoknál bemutatásra került az adott eszköz előnye és hátránya is, hogy megfontoltabb döntési alapot kínáljon.

7.2. Dolgozatban nem vizsgált kérdések

A diplomamunka keretén belül sikerült elmélyülni a feladatban, miközben folyamatosan bontakozott ki a terület komplexitása. Sok feladatot sikerült megvalósítani és a legtöbb vizsgált kérdésre sikerült választ is kapni azonban a tanulmányok során kerültek elő olyan új kérdések is, melyeket a szerteágazó terület miatt nem sikerült felderíteni. Ezen kérdéskörök további vizsgálatot és kutatást igényelnek.

Az elvégzett mérések során bizonyos egyszerűsítésekkel éltem, hogy értelmezhető maradjon a kapott eredmény. Ilyen egyszerűsítés például az a megkötés, hogy a beérkező kérések száma egyenletes az időben, azonban ez nem minden rendszer esetén van így. Jól megfigyelhető periódusok jelentkezhetnek egy napon, hónapon vagy éven belül is. Több projekt is foglalkozik ezzel a jelenséggel és próbálják szintén keresni az erőforrások ideális használatát a historikus adatok elemzésével. Létezik egy fejlesztés, ami a horizontális pod skálázót szeretné ilyen irányba továbbfejleszteni[21]. A skálázó segítségével szintén érdemes lenne méréseket futtatni, azonban ennek előfeltétele, hogy dinamikusabban lehessen forgalmat generálni. Például az időben változóan illetve ezeket több alkalmazásvégpont irányába küldeni.

A dolgozatban elkészített mérések mögötti szolgáltatás hálók a leggyakoribb, egyszerű eseteket fedték le. Egy valódi mikroszolgáltatás architektúra ennél nagyságrendekkel több komponenssel rendelkezik, melyek közti kapcsolatok is összetettebbek. Jelenlegi keretrendszert tovább lehetne fejleszteni, hogy a komponensekhez érkező kérések megadott arányok mellett kerüljenek egyik- vagy másik további komponenshez továbbküldésre. Ezzel kicsit dinamikusabbá lehetne tenni a mostani rendszert, ami minden beérkező kérés esetén azonos kérés-válasz folyamatokat fogja elindítani.

További vizsgálandó kérdés, hogy a kapott eredmények más környezetben hogyan változnak meg. Például, más eredmény jöhet ki, ha több csomóponttal, nagyobb terhelés mellett, nagyobb erőforrás felhasználással történnek a tesztek. Sajnos az ilyen klaszterek bérlése drága különösen, hogy a méréseink szándékosan magas mennyiségű processzort használnak, ami alapja szokott lenni a bérelt infrastruktúra utáni számlázásnak.

7.2.1. Keretrendszer további használata

Az elvégzett munka jelentős részét kitevő keretrendszer lehetőséget biztosít további felhasználásra is. A diplomamunkán belül a feladata az volt, hogy fiktív paraméterekkel rendelkező szolgáltatás hálókat tudjunk elindítani és létrehozni. Ezen kívül egy igazán hasznos felhasználási mód lehet, ha már létező, üzemelő hálózatokat szeretnénk klónozni. Tehát létre tudunk hozni kisebb másolatokat az eredeti rendszerről, ami közel azonos kiszolgálási és erőforrás felhasználási paraméterekkel fog rendelkezni, mint a valódi. Ezután lehetőségünk van fiktív fejlesztéseket végezni az egyes komponensek átkonfigurálásával és az így kapott rendszert tesztelni. Ezáltal könnyebben lehet megalapozott döntéseket hozni, hogy az aktuális környezetben melyik szolgáltatást érdemes fejleszteni, melyikkel lehet érdemben befolyásolni az eredő nyereséget.

Továbbá az előzőleg bemutatott példán keresztül ki tudjuk próbálni az aktuális rendszerünket egy új környezetben is. Mindezt úgy tudjuk megtenni, hogy az alkalmazás egységeket alkotó képfájlok mozgatása nem szükséges, ezáltal nem csak gyorsabb lesz a tesztelés, hanem nem kell újabb felek számára elérhetővé tenni a megírt állományokat. Az így létrehozott mérések betekintést tudnak nyújtani, hogy egy esetleges infrastruktúra váltás milyen változásokkal jár a kiszolgálási paramétereket illetően.

Köszönetnyilvánítás

Nehéz szavakba önteni a dolgozat készítése közben feltörő érzéseket. Egyszerre van jelen a megkönnyebbülés, a hála, az alkotás szeretete és a kielégíthetetlen tudásvágy is. A diplomamunkám nem jöhetett volna létre számos támogatás nélkül, amit a körülöttem lévő emberek áldozatos munkájukból kaptam. Szeretnék köszönetet mondani, hogy az egyetemen eltöltött időm alatt a legkülönfélébb területeken tudást és élményt tudtam gyűjteni.

Szeretném külön kiemelni Dr. Rétvári Gábor konzulensemet, mivel az egyetemen végzett minden egyes projekttárgyat a felügyelete alatt készíthettem. Szakmai tanácsaival mindig tovább tudta gurítani a megakadónak tűnő feladatokat és végig iránymutatással segítette, hogy ne vesszek el az egyes részletekben és tudatosította, hogy a nagy képet figyelve jelenleg hol tartok. Mindezek már önmagukban köszönetet érdemelnek, de nem csak konzulensemmé vált Gábor, hanem a kötelező feladatokon túl mentorommá is. Számíthattam rá, amikor az egyes feladatok nehézsége és nagysága miatt motivációmat vesztettem illetve egyedülálló személyisége már önmagában motiváló. Megmutatta, hogy egyenességgel, nyitottsággal és örök jókedvvel milyen szélessé nyílhat a világ. Hálás vagyok érte, hogy együtt dolgozhattunk és tanulhattam tőle minden egyes konzultáció vagy találkozás alkalmával.

Köszönettel tartozom a családom támogatásáért, hogy egész életemben fontos szerepet szántak a taníttatásnak. Tudom, hogy jelentős áldozatokkal járt ez a folyamat számukra is, de remélem sikerült a legjobban kihasználni az eddigi lehetőségeket és hosszútávon meg fog térülni. Nekik köszönhetem, hogy támogatásukkal lehetőségem nyílt az egyetemi éveimet a kellő szabadsággal megélni, ezáltal is felfedezve illetve jobban megértve saját magam és a környezetem működését.

Végül pedig szeretném megemlíteni a Schönherz kollégium közösségét, ahol lehetőségem volt olyan dolgokat is kipróbálni, amik nem közvetlenül a szakmához, hanem a tágabban értelemben vett élethez hasznos tapasztalatokkal szolgáltak. Kipróbálhattam magam több csoport élén is, ezzel is tágítva a saját komfortzónám határát és ösztönöztek az alkotás és gondolkodás aktív folyamatára.

Irodalomjegyzék

[1] Tukovics András: Dockerhub - tuti/service-graph-simulator, 2020. URL https://hub.docker.com/repository/docker/tuti/service-graph-simulator. (felkeresve: 2021.05.13.).

[2] Tutkovics András: Github repo scaling msc, 2021. URL https://github.com/Tutkovics/Scaling_MSc. (felkeresve: 2021.05.05.).

[3] Markosz Maliosz Balla David, Simon Csaba: Adaptive scaling of kubernetes pods, 2020.

[4] Raman Bhadauria: Monolithic vs microservices architecture, 2021.08.17. URL https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/. (felkeresve: 2021.12.12.).

[5] Roland Huß Bilgin Ibryam: *Kubernetes Patterns*. 2019, O'Reilly Media.

[6] Kubernetes Blog: Borg: The predecessor to kubernetes, 2015.04.23. URL https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/. (felkeresve: 2021.05.02.).

[7] CNCF: Who we are | cloud native computing foundation, 2021. URL https://www.cncf.io/about/who-we-are/. (felkeresve: 2021.05.14.).

[8] Ryan Dawson: How did kubernetes win the container orchestration war?, 2020.06.28. URL https://hackernoon.com/how-did-kubernetes-win-the-container-orchestration-war-lp1l3x01. (felkeresve: 2021.12.12.).

[9] Denis Denisov (denji): Http(s) benchmark tools, testing/debugging, and restapi (restful), 2017.02.24. URL https://github.com/denji/awesome-http-benchmark. (felkeresve: 2021.11.21.).

[10] IBM Cloud Education: What is etcd? | ibm, 2019.12.18. URL https://www.ibm.com/cloud/learn/etcd#toc-raft-conse-6_wBECZ-. (felkeresve: 2021.05.13.).

[11] Owen Garrett Floyd Smith: What is a service mesh? - nginx, 2018.04.03. URL https://www.nginx.com/blog/what-is-a-service-mesh/. (felkeresve: 2021.12.05.).

[12] The Linux Foundation: Prometheus - monitoring system and time series database, 2014-2021. URL https://prometheus.io/docs/introduction/overview/. (felkeresve: 2021.05.05.).

[13] The Linux Foundation: Kubernetes components | kubernetes, 2021.03.18. URL https://kubernetes.io/docs/concepts/overview/components/. (felkeresve: 2021.05.13.).

[14] The Linux Foundation: Configure quality of service for pods | kubernetes, 2021.05.20. URL https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/. (felkeresve: 2021.11.14.).

[15] The Linux Foundation: Kubernetes documentation | kubernetes, 2021.07.20. URL https://kubernetes.io/docs/home/. (felkeresve: 2021.12.12.).

[16] The Linux Foundation: Creating a cluster with kubeadm, 2021.11.28. URL https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. (felkeresve: 2021.12.14.).

[17] The Linux Foundation: Horizontal pod autoscaler | kubernetes, 2021.12.09. URL https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. (felkeresve: 2021.12.11.).

[18] The Linux Foundation: Production environment - kubernetes, April 08, 2021. URL https://kubernetes.io/docs/setup/production-environment/#production-control-plane. (felkeresve: 2021.05.08.).

[19] The Linux Foundation: Operator pattern | kubernetes, Sep 07, 2021. URL https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#writing-operator. (felkeresve: 2021.11.01.).

[20] Nectarios Koziris Ioannis Giannakopoulos, Dimitrios Tsoumakos: Towards an adaptive, fully automated performance modeling methodology for cloud applications, 2018.

[21] Jamie Thompson (jthomperoo): Horizontal pod autoscaler built with predictive abilities using statistical models, 2019.12.09. URL https://github.com/jthomperoo/predictive-horizontal-pod-autoscaler. (felkeresve: 2021.12.05.).

[22] Christopher McFadden: The fascinating history of netflix, Jul 04, 2020. URL https://interestingengineering.com/the-fascinating-history-of-netflix. (felkeresve: 2021.05.06.).

[23] Masashi Narumoto: Oldalkocsi minta - cloud design patterns, 2021.08.11. URL https://docs.microsoft.com/hu-hu/azure/architecture/patterns/sidecar. (felkeresve: 2021.12.05.).

[24] NEWBEDEV: What is a "tight loop"? URL https://medium.com/blutv/qos-classes-of-k8s-pods-722238a61c93. (felkeresve: 2021.11.14.).

[25] Operator-SDK: Operator sdk, 2020.
URL https://sdk.operatorframework.io. (felkeresve: 2021.05.13.).

[26] Operator-SDK: Operatorhub.io | the registry for kubernetes operators, 2020.
URL https://operatorhub.io. (felkeresve: 2021.05.13.).

[27] Sebastian Pauli: Sieve of eratosthenes, 2021.11.12. URL https://mathstats.uncg.edu/sites/pauli/112/HTML/seceratosthenes.html. (felkeresve: 2021.12.12.).

[28] Carlos Becker Westphall Rafael Weingärtner, Gabriel Beims Bräscher: Cloud resource management: A survey on forecasting and profiling models. 2015.

[29] Inc. Red Hat: What's a service mesh?, 2018.06.29. URL https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh. (felkeresve: 2021.12.05.).

[30] Carlos Schults: What is infrastructure as code? how it works, best practices, tutorials, 2019.09.05. URL https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/. (felkeresve: 2021.12.12.).

[31] Tutkovics András (szakdolgozat): Felhő alapú alkalmazások teljesítményének kiértékelése és modellezése, 2019.

[32] Marco Chiesa Thomas Wang, Simone Ferlin: Predicting cpu usage for proactive autoscaling, 2021.

[33] Tomás Senart (tsenart): Vegeta: Http load testing tool and library., 2013.08.13. URL `https://github.com/tsenart/vegeta`. (felkeresve: 2021.11.21.).

[34] UKEssays: Impact of technology on amazon's business strategy, 8th Feb 2020. URL `https://www.ukessays.com/essays/business-strategy/impact-of-technology-on-amazons-business-strategy.php`. (felkeresve: 2021.05.06.).

[35] VMware: The state of kubernetes 2020. 2020.

[36] Zhiming Shen Sethuraman Subbiah Xiaohui Gu John Wilkes: Cloudscale: Elastic resource scaling for multi-tenant cloud systems, 2011.

Rövidítések és fordítások

A szakterület rövidítéseit és az előforduló angol kifejezéseit lehetőségemhez mérten próbáltam magyarítani illetve feloldani az első előfordulásukkor. Ahol találtam már előforduló magyar kifejezést, azt használtam, azonban az esetek nagy részében ez nem állt fent. Emiatt és a könnyebb kereshetőség érdekében összegyűjtöttem a rövidítéseket és az általam használt magyarításokat.

API Application Programming Interface / Alkalmazásprogramozási felület

BE Back-end

Black Box Fekete doboz

CNCF Cloud Native Computing Foundation

Control plane Vezérlő sík

CPU Processzor

CR Custom Resource / Saját Erőforrás

CRD Custom Resource Definition / Saját Erőforrás Leíró

Deadlock Holtponti állapot

FE Front-end

Flow control Forgalomszabályozás

HPA Horizontal Pod Autoscaler / Automatikus Horizontális Pod Skálázó

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

I/O Input and Output / Kimenet és Bemenet

K8s Kubernetes

Kube Scheduler Ütemező

kubectl Kubernetes parancssoros kezelője

Liveness probe Életteli próba

Node Csomópont

OLM Operator Lifecycle Manager / Operátor Életciklus kezelő

Pod Kapszula

QoE Quality of Experience / érzékelhető szolgáltatási szint

QPS Queries per Second

Readiness probe Készenléti próba

Service Mesh Szolgáltatás háló

SLA Service-level Agreement / szolgáltatási szint megállapodásokba

Startup probe Indítási próba

Tight loop Szoros ciklus

VM Virtuális gép

VPA Vertical Pod Autoscaler / Automatikus Vertikális Pod Skálázó

White Box Fehér doboz