



TANSZÉKVEZETŐ

DIPLOMATERVEZÉSI FELADAT

Nánási Dániel

mérnök informatikus hallgató részére

Felh. alapú drónvezérlés

Az ötödik generációs (5G) mobil távközlési rendszerek megvalósítása során a korábban hardverfüggő szoftvermodulokat felh. alapú virtualizált funkciókkal helyettesítik. Egy ilyen felh. rendszerben rugalmasabb és erőforrás-takarékosabb üzemeltetést lehet elérni, de azokat a kapcsolatokat, amelyek szigorú minőségi elvárásoknak (Quality of Service ó QoS) kell megfelelniük, bonyolultabb kezelni. Különösen igaz ez a távvezérlési feladatokat (pl. drónvezérlés, ipari robotkarok vezérlése, jármű- ipari alkalmazások, stb.) biztosító kapcsolatok esetében.

A dolgozat célja egy olyan keretrendszer tervezése és megvalósítása, amely képes a drónvezérlés során dinamikusan változó környezetben, az alkalmazás szigorú késleltetési elvárásainak megfelelő megoldást nyújtani.

A hallgató feladata:

- Adjon áttekintést a modern felh. alapú rendszerekről, különös tekintettel az 5G rendszerekben is alkalmazott konténer menedzsment platformokra. Válasszon ki egy felh. rendszert, döntését indokolja.
- Tekintse át a drónvezérlést támogató szoftver megoldásokat, válasszon ki egy olyan környezetet, mellyel távvezérlési funkciók megvalósíthatók!
- Tervezzen meg és valósítsa meg egy, a munkája során használható felh. alapú tesztrendszert!
- Tervezze meg a kiválasztott vezérlést támogató környezet illesztését a kiválasztott felh. rendszerbe, hajtsa végre az integrációt és valósítsa meg egy minta eszköz vezérlését a felh. b. l!
- Tervezze meg és implementálja a távvezérlési funkciók dinamikus elhelyezését megvalósító módszert, amely a drónvezérlés QoS igényeinek megfelelő módon oldja meg a funkciókat végrehajtó modulok indítását a felh. rendszeren belül!
- Alakítson ki tesztkörnyezetet a távvezérlési funkciók elosztására, vizsgálja meg a rendszer viselkedését, és a funkciók elosztásának hatását az eszköz működésére, értékelje a megoldást.
- Munkáját részletesen dokumentálja!

Tanszéki konzulens: Simon Csaba, egyetemi docens

Budapest, 2020. március 10.

Dr. Magyar Gábor
tanszékvezető





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

Felhő alapú drónvezérlés

DIPLOMATERV

Készítette
Nánási Dániel

Konzulens
Dr. Simon Csaba

2020. december 18.

Tartalomjegyzék

Kivonat	5
Abstract	6
1. Bevezetés	7
1.1. Előzmények	7
1.2. A feladat célja	7
1.2.1. Feltételrendszer	7
1.2.2. Tömegkiszolgálás	8
1.2.3. Durva modell	9
1.2.4. Sáv szélesség	9
1.3. Feladat indoklása	9
1.3.1. Gyártóipar	9
1.3.2. Szállítás	10
1.3.3. Mezőgazdaság	10
1.3.4. Szórakoztatóipar	10
1.4. Használt kifejezések	12
2. Felhőrendszer kiválasztása	14
2.1. Virtualizációs alapok	14
2.1.1. Hypervisor	14
2.1.2. Konténer	14
2.1.3. Docker	15
2.2. Docker Compose	16
2.3. Docker Swarm	16
2.4. Mesos	16
2.5. Kubernetes	17
2.5.1. Architektúra	17
2.5.2. Komponensek	17
2.6. Keretrendszer meghatározása	18
3. Robotvezérlés környezete	20
3.1. Robotirányítás	20
3.2. Robot operációs rendszer - ROS	20
3.3. Kommunikáció - Mavros, Mavlink	21
3.4. Vezérlő - PX4	21
3.5. Szimulációs környezet - Gazebo	22
3.6. Együttes működés	23
4. Drónvezérlést tesztelő környezetek	25
4.1. Képfelismerésen alapuló drónvezérlő automatizált drónvezérlő scenario	25

4.2.	Azonos konténerrel egy drónvezérlés virtuális környezetben	26
4.3.	Két VM-en több drón szimulációja és vezérlése	27
5.	QoS sztohasztikus becslése	29
5.1.	Tömegkiszolgálási modell	29
5.2.	Várakozási idő várható értéke	30
6.	Kialakított Kubernetes alapú felhő	31
6.1.	Kubernetes technológiái	31
6.1.1.	K8S	31
6.1.2.	K3S	31
6.1.3.	Kind	31
6.1.4.	MiniKube	32
6.1.5.	Miért K3S?	32
6.2.	Földi scenario felkészítése a menedzselt felhőrendszerbe	32
6.3.	Kubernetes virtualizált telepítése Multipass VM-eken	32
6.4.	Konténer registry, lokális és központi	33
7.	Drónirányítás mint szolgáltatás a Kubernetes felhőben	36
7.1.	Szimpla 4 podos megvalósítás	36
7.2.	Konténerek hálózata, Service és Deployment	37
7.3.	Szolgáltatás külső elérése	38
7.3.1.	NodePort	38
7.3.2.	Ingress, szolgáltatások szétválasztása	39
7.3.3.	ROS Port forwarding	39
7.3.4.	LoadBalancer	40
7.4.	Teljes klaszter létrehozása N drónnal	42
7.4.1.	Kauzalitási probléma	42
7.5.	Deploy szkriptek	42
8.	Drónkapcsoló állomás Kubernetes felhőben	45
8.1.	Felhasznált technológia	45
8.1.1.	Python	45
8.1.2.	Kubernetes könyvtár alkalmazása	46
8.1.3.	Docker könyvtár alkalmazása	47
8.1.4.	Naplózás	47
8.2.	Kivitelezési lehetőségek és várható kapcsolási idők	47
8.2.1.	Node váltás és új címhirdetés	48
8.2.2.	Node váltás proxy mögött	48
8.2.3.	Széleskörű rendelkezésre állás	48
8.3.	Megvalósítás: DaemonSet	48
8.4.	Mérési adatok kinyerése	49
9.	Távoli Robotvezérlés Optimalizálása	51
10.	Mérések	53
10.1.	Node váltás ideje meghibásodott Node esetén	53
10.2.	Válaszidő különbség esetén a jobb Node kiválasztása	54
10.3.	Terheltségi válaszidő viszonya	54
10.4.	Késletetés hozadéka 5G közeghozzáférés esetén	58
Összegzés		58

Ábrák jegyzéke	60
Kódrészletek jegyzéke	61
Irodalomjegyzék	61

HALLGATÓI NYILATKOZAT

Alulírott *Nánási Dániel*, 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, 2020. december 18.

Nánási Dániel
hallgató

Kivonat

A diplomaterv a mai felhőrendszerekről és azokban robot irányítási megoldásaiban mélyed el. Megmutatja a mai rendszerek felhőrendszerek szolgáltatás alapú technológiáit és bevezeti az olvasót a konténeralapú szolgáltatásmenedzsmentbe. Leírást ad a felhasználói igényekről és a felhőrendszereknek a jövőbeli fejlesztési lehetőségeiről. Áttekintést ad a közösségi fejlesztések által nyújtott nagy méretű erőforrás menedzsment megoldásoknak a mikéntjére, azok hibájára és bemutatja tesztelésüket. A dolgozat ismerteti és feldolgozza a mai robotirányítási megoldásokat, kifejezett prioritást tekintve a drón irányításnak, azoknak ipari felhasználásáról és megvalósíthatóságáról. Betekintést ad miként valósítható meg egy 5G kommunikációra alapuló robotirányításra tervezett felhőrendszer. Megmutat egy nagyobb számú robotot központi konténer alapú vezérlő és jelfeldolgozó rendszer megvalósítást. Továbbá kifejti ezeknek a tervezési és implementációs lépéseit és a távvezérlési funkciók dinamikus elhelyezését megvalósító módszert. Szimulációt mutat nagyszámú robotvezérlés és annak kamerajelének feldolgozására a kialakított rendszerben. Ennek tükrében összeveti a drónvezérlés QoS (Quality of Service) igényeinek feltételeit és a megvalósított rendszer funkcióinak ezen feltételrendszerre szabott tervezési megoldást ad a felhő rendszeren belül! Összefoglalja az elvégzett munkát, a szimuláció eredményeit és a QoS feltételrendszert a kialakított tesztrendszerben.

Abstract

The thesis delves into today's cloud systems and the control solutions of individual robots. It demonstrates today's systems with cloud-based service technologies and introduces readers to container-based service management. Provides a description of user needs and cloud systems. Overview and a community development of large-scale resource management solutions is a way of failing and presenting their testing. The thesis describes and processes robot control solutions with explicit priority for drone guidelines, industrial use, and applicability. Provides insight into how to implement a cloud system designed for robotic control based on a 5G communication system. It focuses for larger number of robots in the operation of a central container-based control and signal processing system. In addition, it shows the design and implementation steps as well as the dynamic placement of remote control functions in the system design methodology. It analyses results of simulations in a system created to process a large number of robot controls and their camera signals. In order to improve quality, in order to facilitate the use of QoS (Quality of Service) and to facilitate the operation of the established system, the conditionalities usually becomes available. Summarize the work, simulation, development and QoS conditions of the test system.

1. fejezet

Bevezetés

Ebben a fejezetben betekintést nyújtok az olvasó számára a feladatról, mint problémáról, hogy az miért is indokolt és milyen iparágakat érinthet, mi a pontos célja a feladatnak és mik a főbb kritériumok. Tisztázni szeretném a képet az olvasóban, hogy mik is a projekt előzményei amely alapján kialakult a feladatspecifikáció és mi is az a motiváció ami pontosan ezt a projektet eredményezte.

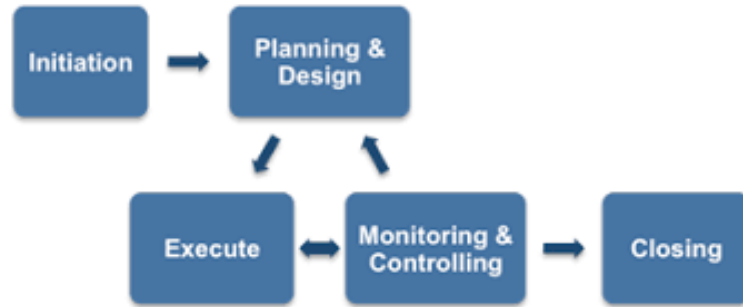
1.1. Előzmények

A témám címe felhő alapú drónvezérlés, aminek tükrében többféle ágazatból is belecsöppenhettem ennek a projektnek a feladatköreibe. Ugyanis a felhőtechnológiák, a robotvezérlés és az irányítástechnika spektrumot is lefedi a feladat, azonban én a felhőtechnológiák felől közelítem meg. Villamosmérnök BSC 5. félévében témalabor keretei között csatlakoztam a Távközlési és Médiainformatika Tanszék (TMIT) Felhő alapú hálózatok szakmai műhelyéhez, ahol azóta is minden félévben végzek projektfeladatot. Ehhez a motivációm egyszerű volt, mivel már az alapképzés eleje óta érdekelnek azon architektúrák, ahol erőforrás menedzselés és központi irányítás alatt, például master-slave módon működnek. A felhőalkalmazások ipara is egy ilyen terület, ahol bizonyos szolgáltatásokat minél szélesebb körben szeretnénk értékesíteni igény szerint, azonban valamilyen központi vezérlés működteti automatikusan az erőforráskiosztást, hozzáférést. A műhelyben végzett feladataim között felmerült a Kubernetes elosztott konténerkezelő rendszer, a Kubless serverless architektúra, felhő alapú beszédfelismerő rendszerek és az OpenStack virtuális gép menedzselő rendszer, amihez egy archiválási megoldást fejlesztettem szakdolgozatként. A felhő alapú drónirányítás téma a diplomatervem kezdetén került szóba, előtte nem foglalkoztam mélyebben robotirányítással és vezérléssel, így remélhetőleg ezekből a területekből is nagy tapasztalatot fogok szerezni a feladat befejeztén.

1.2. A feladat célja

1.2.1. Feltételrendszer

A feladat sokféleképpen általánosítható, hiszen egy felhőrendszerből igen sokféle ipari folyamatot lehet irányítani, csupán a megvalósítást kell kivitelezni. Egy drón irányítása sem különbözik, lehetne a cél robotkar vezérlése, gyártósor ütemezése, automatikus kötőpályás közlekedési eszköz irányítása vagy akár önvezető autók feletti vezérlés. Azonban a témámban nem egy drón irányítását, hanem több, akár tíz-húsz, de akár százat is elérheti az a tesztet amire szeretnék megoldást adni. Egy ilyen rendszert irányító felhő architektúrában rugalmasabb és erőforrás takarékosabb üzemeltetést lehet elérni. A számí-



1.1. ábra. Kritikus mérföldkövek folyamata [6]

táskapacitás optimalizáció nem minden amit ebben a helyzetben az ipar megkíván, hanem azokat a kommunikációs kapcsolatokat, amelyek szigorú minőségi elvárásoknak (Quality-of Service – QoS) kell megfelelniük, ezért bonyolultabb kezelni. A feladat feltérképezni a modern felhőrendszereket, amelyekkel meg lehet valósítani efféle iparban is használható vezérlési technikát QoS feltételek mellett. Tehát a projektnek három kritikus tervezési mérföldköve van. Ezeknek kapcsolata látszik a 1.1. ábrán.

1. Kiválasztani a megfelelő felhőrendszert, amiben megvalósítható $R \in [1, 100]$ drón, vagy általánosan, valamilyen vezérelhető eszköz (robot) irányítása.
2. Áttekinteni drónvezérlést támogató szoftver megoldásokat, kiválasztani egy olyan környezetet, mellyel távvezérlési funkciók megvalósíthatók. Megtervezni ebben a felhőrendszerben több drón irányítását és jelfeldolgozását, automatikusan és biztosítva, hogy hiba esetén visszaálljon a működő állapot.
3. A meglévő rendszer analízálása, optimalizálása és egy QoS feltétel felállítása.

1.2.2. Tömegkiszolgálás

Tehát a dolgozat célja nem csak a megvalósítás, hanem R számú tízes-százas nagyságrendű robotok irányításának a QoS feltételét és validációját is megadja. Ezt a feltételt valamilyen $f_{QoS}(R, C) \in N^2$ függvény fogja megadni, melynek paraméterei R a robotok száma és C a számítási erőforrás. Ha számításról beszélünk, akkor a felhőszolgáltatások iparában kiválasztunk egy valamilyen optimális $c = \text{RAM[GB]}/\text{VCPU}$ arányt mely az alkalmazásunkhoz megfelelő és annak $n \in N^+$, $C = c \cdot n$ egész számú többszöröse lesz a számítási kapacitás amire méretezünk. Vagyis valamilyen 2 dimenziós tervezési teret adunk meg a kívánt szolgáltatási minőség eléréséhez. Az ilyen jellegű feladatokban a szolgáltatás minősége általában a válaszidőt szokta jelenteni $n \cdot 10$ -es nagyságrendben. Természetesen ez nem egy túl pontos modell, de egy mérnöki kapacitástervezés számára kellő kiindulási alapot adhat méretezni a felhőrendszert, esetleg skálázhatósági lehetőségeket is figyelembe véve. Ezen kívül mivel nagyszámú eszközről beszélünk nem tekinthetünk el a közegátviteli technológiáról. Azt sejthetjük, hogy a mai elterjedt távközlő rendszerek nem alkalmasak akár száz fölötti felhasználóval, mondjuk robottal folyamatos kommunikációt tartani megfelelő QoS-el. Ezért megnézzük, hogy a mai modern felhőrendszerekkel és az 5G között milyen kapcsolatot lehet alakítani és mik a feltételek egy ilyen rendszerben való üzemeltetéshez.

1.2.3. Durva modell

A feladat megvalósított terméke pedig egy valamilyen automatizált felhőrendszerbe ültetett drónirányító központ melynek segítségével N darab virtuális vagy fizikai drónt tudunk üzemeltetni, az előző bekezdésben taglalt QoS feltételek mellett. Fontos, hogy a drónirányító központban minden kommunikációt tudjunk kontrollálni és biztonságos állapotba helyezni a drónokat, hogyha tudjuk, hogy valamely biztosítandó alapfeltételnek nem tesz eleget pillanatnyilag a rendszerünk. Tehát amit meg kell valósítani a mérések mellett az nem más, mint egy felhőrendszerbe integrált szoftver, amely biztosítja

- előre definiált N darab drón irányítását a felhasználónak, a felhőből kivezetett interfészen,
- minden drónra vonatkozó kamerakép és irányítási parancsok átviteli sebessége, előre definiált konstans alapján,
- a drón és felhő közötti kapcsolatot egy előre definiált minimum sáv szélesség alapján,
- amennyiben az előző feltételeknek nem tesz eleget a rendszer, a biztonságos leállítást az összes drónnak.

1.2.4. Sáv szélesség

Ezen feltételek mellett felírhatunk pár egyszerű összefüggést egy durva modellre, aminek blokkdiagramja a 1.2. ábrán látható. Tudjuk, hogy lesz egy valamilyen még ki nem választott felhőrendszer, amely biztosan több node-ból fog állni, tehát több fizikai vagy virtuális gép fogja szolgáltatni a felhőrendszer platformját. A node-ok között feltételezzük, hogy nincs sáv szélességi korlát. A felhőrendszer szolgáltat valamilyen interface-t amint elérhető az integrált szolgáltatás a drónok számára. Ezen modellen következik, hogy a hálózati interface és a tényleges integrált applikáció közötti sáv szélességnek nagyobbnak kell lennie, mint a drón felé a sáv szélesség összessége:

$$BW_c \geq \sum_{i=1}^n BW_d_i$$

A drónok sáv szélességére felírhatjuk azt a feltételt, amely kimondja, hogy bármely drón sáv szélességének nagyobbnak kell lennie, mint a videófolyamhoz rendelt minimális sáv szélességnek és az irányításhoz megszabott minimális irányíthatósági sáv szélességnek.

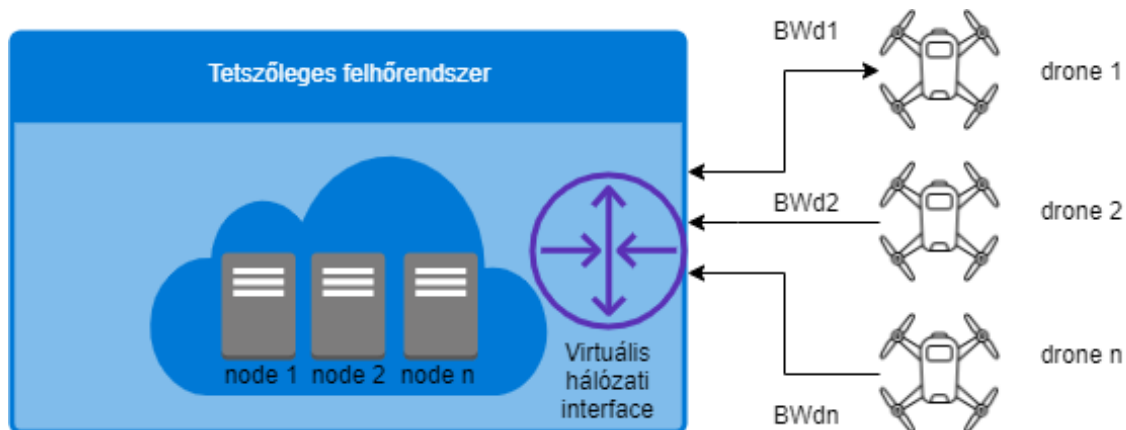
$$\forall d \in D : BW_d \geq BWV_{min} + BWC_{min}$$

Ahol BWV_{min} a videó minimális sáv szélessége, BMC_{min} a kontrollerhez rendelt minimális sáv szélesség, BW_d az adott drón és a felhő közötti sáv szélesség, D pedig a drónok halmaza.

1.3. Feladat indokoltsága

1.3.1. Gyártóipar

A nagyszámú eszközök irányítása számtalan szektorban elterjedt és alkalmazható. Akár beszélve a gyártó szektorról, ahol nagyon sok folyamatot robotok látnak el és legyen az bármilyen robot, az valószínűleg irányítható felhőből. Persze sokszor a gyártók egy drágább, de kapacitásban túlbiztosított lokális rendszerről irányítják a gyártórobotjaikat, mivel a felhős megoldások még annyira nem elterjedtek ebben a célfelhasználásban. Azonban egy ilyen megoldással rengeteg erőforrás megtakarítható. A 1.3. ábrán láthatóak az Ipar 4.0



1.2. ábra. A drónok és felhő kapcsolatának durva modellje

főbb komponensei és könnyen meggyőződhetünk róla, hogy egy mai innovatív ipari környezetben inkább a felhő alapú megoldásokat választják a skálázhatóság és a biztonságuk miatt. Ebben az esetben felhőről általában számítási kapacitásról beszélünk, azonban később kitérünk a szolgáltatások kategóriáira.

1.3.2. Szállítás

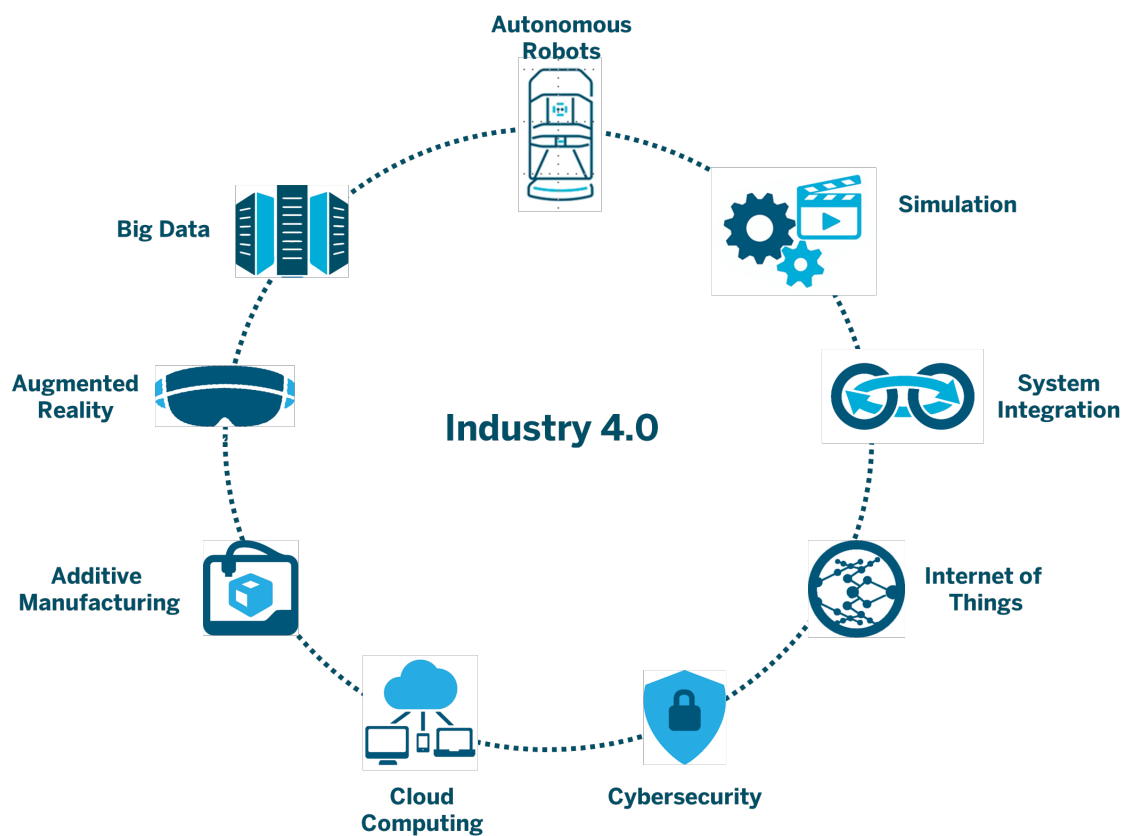
Nem csak a gyártóiparban lehet elképzelni sok robot irányítását, hanem például szállítás és utazás terén is. A Műegyetemhez legközelebbi metróvonal már évek óta önvezérlőként működik, ugyan egyelőre a hossza befejezetlensége miatt csak tíz körüli metrószerelvény működik egyszerre, azonban ez is egy olyan példája a robotirányításnak, amit minden nap észlelhetünk. Az Amazon házhozszállító cég, amelynek egyébként az AWS (Amazon Web Services) leányvállalata a világ egyik legnagyobb felhőszolgáltatója, már tesztel drónokat, amelyek betöltik a csomagházhozszállítás szerepet. A levegőben való csomagszállítás egyik problémája a 1.4. ábrán látható.

1.3.3. Mezőgazdaság

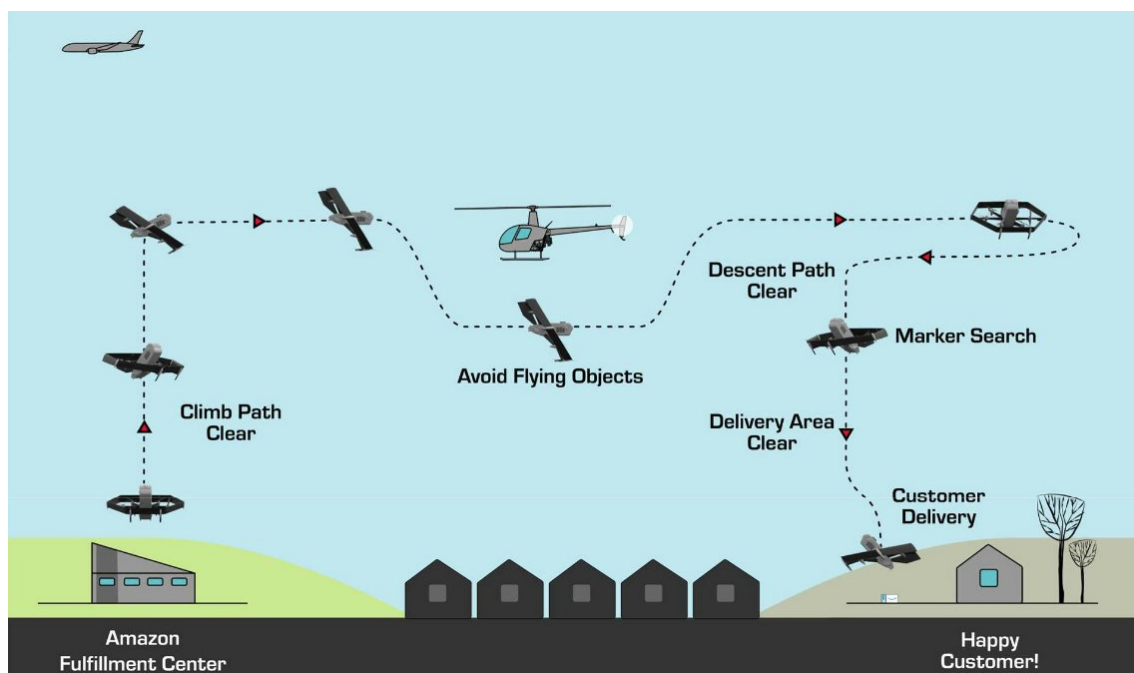
Az agrárvilágban is el lehet képzelni sok robotot kezdve a vetéstől az aratásig, azonban a drónoknak kifejezett szerepe akad a jövőben ebben az iparágban. Használnak ma már automatizált drónokat permetezésre, időszakos állomány megfigyelésre vagy akár kombájn útjának a felderítésére is. Ahogyan a gyártósortornál, itt is optimalizálhatunk több robot irányítás esetén felhőrendszerrel. A dolgozatban arra keresünk megoldást, hogy ezt milyen eszközrendszerrel érdemes tervezni.

1.3.4. Szórakoztatóipar

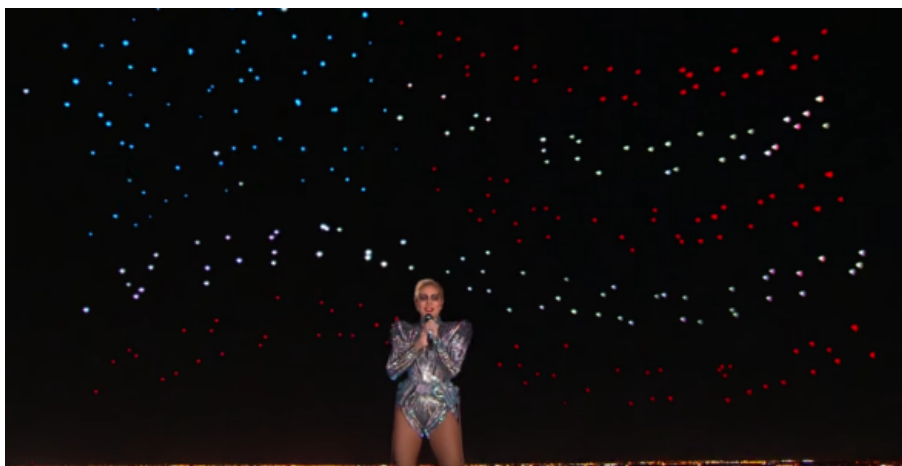
Idáig kiderült, hogy rengeteg iparágban alkalmazhatóak a tömegesen irányított robotok, amiket kreativitással könnyű bővíteni. Azonban a szórakoztatóiparban is megjelentek már a tömegesen irányított drónok. Egy jellegzetes példája ennek amerika legnagyobb sporteseménye a Super Bowl 2017-es döntője, ahol 300 drónt használtak fel az égboltra való fényfestéshez a szünetben lévő koncerthez. Ez az esemény egy pillanatfelvétele a 1.5. ábrán látható és az esemény technikai előkészületeiről a [5] cikkben lehet olvasni. Persze ebben a példában egy pár perces feladatról beszélünk az irányított robotok számára, a felhős megoldással pedig egy hosszútávú optimalizációt szeretnénk megadni.



1.3. ábra. Ipar 4.0 komponensei [1]



1.4. ábra. Amazon házhozszállítás problémája drónnal [24]



1.5. ábra. Lady Gaga 300 drónnal a háttérben a 2017-es Super Bowl döntőjének félideje alatti koncerten [8]

1.4. Használt kifejezések

A hálózati-, virtualizációs- és robotiparban rengeteg rövidítés, mozaikszó és kifejezés létezik, amit főként csak azok ismernek, akik jelentősebben mélyültek el ezen iparág területén. Két csoportra bontom a diplomatervemhez használt kifejezéseket:

1. Azon szakmai kifejezések, amik elterjedtek a mérnöki szakmákban és mondjuk a BME VIK tetszőleges hallgatója, bármely, nem infokommunikációs szakosodás mellett is nagy valószínűséggel ismer és nem kell ismertetnem a tanulmányban. Pár példa a kategóriában, amikre külön nem térek ki, nem oldom fel a szövegben, ilyenek az IPv4, NAT, hálózati réteg, HTTP, CPU, for ciklus.
2. Azokat a kifejezéseket amik pedig a témához, szakterülethez kapcsolódnak, például a kifejezetten hálózati kommunikáció, virtualizáció, robot, irányítás iparágakba tartozó kifejezések, amiket nem általános mérnöki ismeretek, azokat a szövegben az első használatnál kifejtem, továbbá itt összegyűjtöm.

A tanulmányban használt speciális kifejezések:

- QoS (Quality of Service) - A szolgáltatás minőségének a biztosítása
- Virtuális gép (Virtual Machine, VM) - Virtualizált önálló teljes értékű operációs rendszer gazdagépen (host-on)
- KVM (Kernel-based Virtual Machine) - Kernelhez elosztott időben hozzáférő virtuális gép
- Host OS - Hosted virtualizáció esetén a gazdaoperációs rendszer
- Guest OS - Hosted virtualizált operációs rendszer a host OS fölött
- Hypervisor - A hardveren való virtualizációt megvalósító szoftver
- Node - A felhőrendszer egy fizikai eszköze
- Cluster/Klaszter - A felhőrendszerbe csatolt fizikai eszközök kapcsolata
- Volume - Szolgáltatás/VM háttértára

- Vertikális skálázás - Node hozzáadása a cluster-hez
- Horizontális skálázás - Node fejlesztés
- GCS (Ground Control Station) - Földi irányítóállomás
- Földi - Olyan alkalmazás ami nem felhőben fut
- Konténer Registry - Konténereket tároló és menedzselő rendszer

2. fejezet

Felhőrendszer kiválasztása

A számítás alapú felhőrendszereknek két alapja van, a teljes virtualizált önálló operációs-rendszer (VM, mint Virtual Machine) vagy pedig a konténerizált, önálló lebutított virtualizált operációs rendszer szolgáltatás szinten. Egy platform virtualizációja olyan virtuális gép létrehozását jelenti, amely egy valós operációs rendszerű számítógépként működik. A virtuális gépeken végrehajtott szoftverek elkülönülnek az alapul szolgáló hardverforrásoktól.

2.1. Virtualizációs alapok

A KVM (Kernel-based Virtual Machine, azaz kernel alapú virtuális gép), egy teljes virtualizációs megoldás amely képes kihasználni az újabb processzorokban rejlő hardveres virtualizáció képességet. Magában foglal egy betölthető kernel modult, alkalmas egy adott operációs rendszeren virtualizált környezetben egy másik operációs rendszert futtatni. Azaz miközben a gazda operációs rendszer fut a számítógépen addig képesek vagyunk egy másik vendégrendszert indítani virtuális környezetben. Ettől ez a megoldás sokkal gyorsabb.

2.1.1. Hypervisor

Szoftverben megvalósított menedzser a virtuális gépek számára. A gazdahardveren fut, lehetővé teszi a vendég gépek (guest OS) futtatását, memória kezelését és processzor ütemezését különböző algoritmusokkal. Két főbb típust különböztetünk meg.

Bare matel

Az első típusú hypervisorok közvetlenül a hardverre vannak telepítve. Tartalmazniuk kell saját operációs rendszerüket a rendszerindításhoz, a hardver futtatásához és a hálózathoz való csatlakozáshoz. Ilyen hypervisor pl. a Microsoft Hyper-V.

Hosted

A hosztolt hypervisorok olyan operációs rendszereken futnak, amely közvetlenül a hardverre van telepítve. Ebben az esetben szükség van egy gazda operációs rendszerre (host OS).

2.1.2. Konténer

A konténerizáció egy olyan megoldás, ahol a szolgáltatásokat külön-külön letisztult operációs rendszerre telepítjük és csak azokat a szoftvereket ami az adott szolgáltatáshoz

szükséges. A konténerizáció elszeparálása a Linux kernel namespace technológiáján alapul. Működésük alapján hasonlítanak a VM-ekre, úgy is lehet mondani, hogy egy olyan VM optimalizált megoldása, amin egy applikáció fut. Egy rendes operációs rendszeren futó számítógépes program képes megtekinteni az adott rendszer összes erőforrását (csatlakoztatott eszközök, fájlok és mappák, hálózati megosztások, CPU-teljesítmény, számszerűsíthető hardver- képességek). A konténeren belül futó programok azonban csak a konténer tartalmát és eszközeit látják el.

2.1.3. Docker

A Docker a legelterjedtebb konténerizáció megoldás, 2013-ban kiadott technológia Windows és Linux operációs rendszerekhez. A Docker elsősorban Linuxra készült, ahol a Linux kernel erőforrás-elkülönítési jellemzőit (cgroup-okat), a névtereket és a fájlrendszert használja. Ezek lehetővé teszik a független konténerek egyetlen Linux példányán működését, elkerülve bármilyen összeférhetlenséget.

Hálózata

A docker konténereknek 3 féle hálózati interface-ük lehet,

- Host - kívülről elérhető
- Bridge - a docker konténerek elérik egymást
- None - nincs hálózatra kötve

Port forward

Docker konténerek létrehozásánál megadhatjuk a port átírányítást, így azonos alkalmazások sem akadnak össze.

```
$ docker run --name wp1 wordpress  
$ docker run --name wp2 -p 80:81 wordpress
```

2.1. kódrészlet. Példa két WordPress szolgáltatás párhuzamos indítására a 80 és 81-es portokon

Volume csatolás

Alapvetően nehézkes hozzáférni a konténer belső tárhelyéhez, azonban gazda OS alatt tudunk csatolni a szolgáltatáshoz tartozó könyvtárat a host OS fájlrendszeréhez.

```
$ docker run -v /usr/local/mywordpress:/wordpress wordpress
```

2.2. kódrészlet. Példa volume csatolásához

Példa docker konténer robotoperációsrendszerhez

A robotoperációs rendszerekről később lesz szó, azonban megadok egy példát ilusztrálva, hogyan alakítható egyszerű szolgáltatás docker konténerré, csupán egy Dockerfile nevű fájlra van szükség az applikáció főkönyvtárában (2.3. számú kódrészlet).

```
FROM ros:melodic-ros-base

RUN apt-get update && apt-get install locales -y
RUN locale-gen en_US.UTF-8
ENV LANG en_US.UTF-8

COPY . /catkin_ws/src/
WORKDIR /catkin_ws
RUN /bin/bash -c '. /opt/ros/kinetic/setup.bash; catkin_make'
RUN /bin/bash -c '. /opt/ros/kinetic/setup.bash; source devel/setup.bash'

CMD ["roslaunch", "bebop_gazebo bebop_moving_helipad.launch"]
```

2.3. kódrészlet. Példa alap robotoperációsrendszer konténerizációjához

A *FROM* parancsal a szülő konténert adom meg, amit DockerHub-ról automatikusan letölt, a *RUN* parancsokkal környezeti programokat telepítek, *COPY* az applikációhoz tartozó fájlokat másolja a konténerbe, *ENV* parancssal környezeti változót állítok, *WORKDIR* parancs a munkakönyvtár beállítása és a *CMD* utasítás a konténerizáltan futtatandó applikáció megadása.

2.2. Docker Compose

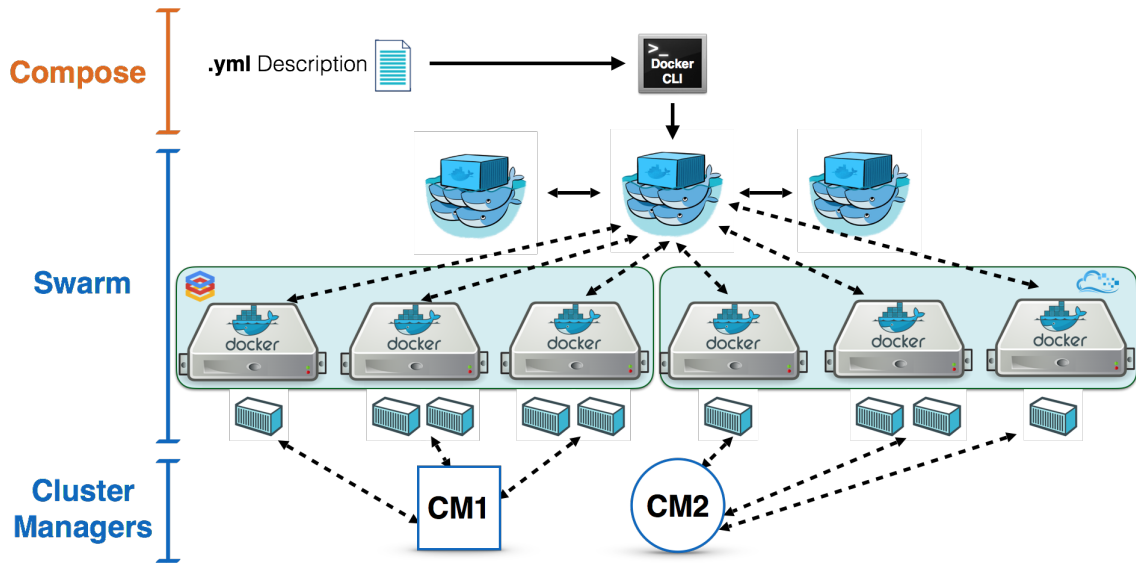
A Compose egy multi-konténer Docker eszköz különböző alkalmazások együttes definiálásához és működéséhez. A Compose használatával YAML fájlban definiálni lehet különböző konténereket más-más paraméterekkel szolgáltatásainak konfigurálására. Ezután egyetlen parancssal elindíthatóak a docker konténerek, az összes együtt működő szolgáltatás a konfigurációból. [4] A feladathoz biztosan kell használni, mivel a drónirányítás több különböző funkcióból valósul meg, így a konténer definíciója szerint érdemes minden egyes applikációt elkülöníteni és Compose-al összekötni a működésüket.

2.3. Docker Swarm

A Docker Swarm olyan fizikai vagy virtuális gépek (node-ok) csoportja, amelyek futtatják a Docker alkalmazást, és amelyeket együttesen egy swarm-t alkotnak, hogy összekapcsolódjanak egy clusterként. Miután egy csoport node-ot összekapcsoltak, továbbra is futtható rajtuk a Docker vezérlőparancsok, ezeket most a worker node-ok hajtják végre. A klaszter tevékenységét egy swarm manager irányítja, és a klaszterhez csatlakozó gépeken osztja ki a feladatokat. A docker swarm manager és worker node-ból áll és alkalmazható a swarm-ra a docker compose is (2.1. ábra).

2.4. Mesos

A Mesos az Apache szoftvere, mely egy absztrakt környezetet hoz létre CPU, RAM és háttértárral, akár 10.000 node-ig. A Mesos egy nyílt forrású cluster kezelő szoftver. Alkalmazásokat kínál API-kat az erőforrás-kezelésre és az ütemezésre a cluster-en keresztül. A Mesos rugalmasságot és az elhaló alkalmazások újraindítását biztosítja. Lehetővé teszi a framework-ök számára, hogy ütemezzék és végrehajtják a feladatokat API-n keresztül. A Mesos-architektúra egy master node-ból áll, amely az egyes node-okon futó worker-eket és a worker-eken futó framework-öket kezeli, melyeken a szolgáltatások futnak. Az összes alkalmazásdefiníció egy JSON-fájlban található, amelyet továbbítanak a Mesos / Marathon REST API-hoz.



2.1. ábra. Docker Swarm architektúra [16]

2.5. Kubernetes

Egy docker rendszerben a definiált konténer egy példánya futtatható a `docker run` utasítással. Kuberneteset használva a kube control CLI-on keresztül akár ezer példánya is futtatható ugyanannak az alkalmazásnak.

```
$ kubectl run --replicas=100 wordpress
```

2.4. kódrészlet. Példa 100 alkalmazás indítására

A futtatott alkalmazásokat egy másik paranccsal felskálázhatjuk.

```
$ kubectl scale --replicas=200 wordpress
```

2.5. kódrészlet. Példa alkalmazás skálázására

Tehát egy kihasználtsági emelkedő esetén egyszerűen lefoglalhatunk több erőforrást a felhasználás tekintetében. A Kubernetes a Docker host programot használja az egyes node-okon alkalmazások futtatásához. Alapvetően konténereket támogat, azonban nem csak a Docker-t, hanem pl. a Rocket-et és a Cryo-t is.

2.5.1. Architektúra

Egy Kubernetes cluster architektúrája fizikai vagy VM node-okból áll és minden node egy worker (2.2. ábra). Egy node meghibásodása esetén a rajta futó service-t átveszi a többi node. A node-ok között van egy kitüntetett master node, amelyik tárolja a cluster információkat és végzi a manager service-ek processzeit.

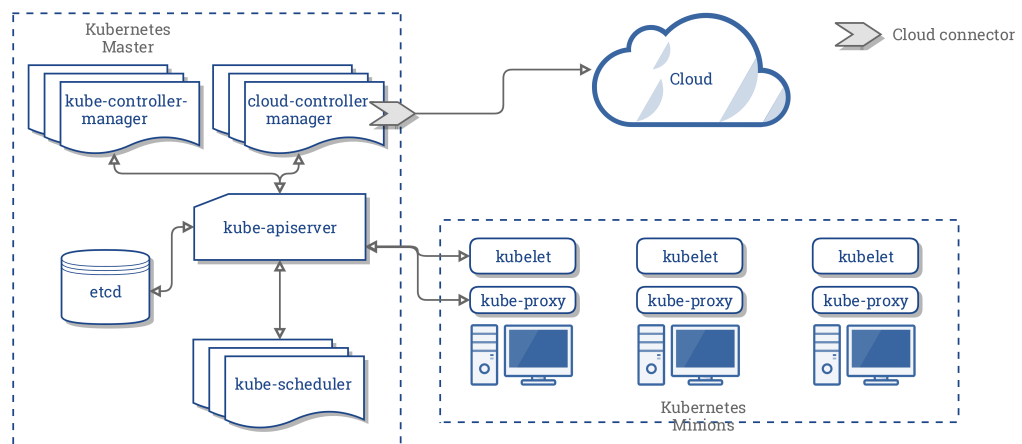
2.5.2. Komponensek

API server

A Kubernetes frontend-jét web UI-t és API-t szolgáltat. Ezzel kommunikál a felhasználó menedzser, CLI-t és a többi UI eszköz is.

etcd

Elosztott kulcsérték tároló, a cluster menedzseléséhez szolgáló információknak.



2.2. ábra. Kubernetes architektúrája [11]

Scheduler

Az elosztott működést valósítja meg a node-okon, minden új feladatot eloszt egy node-hoz.

Controller

Figyeli a végpontokat és beavatkozik ha valami tönkremegy.

Container runtime

A konténerek keretszoftvere, leggyakoribb esetben a Docker.

Kubelet

Ez a service minden node-on fut a cluster-en, a feladata, hogy a node-hoz kiszervezett konténerek fussanak az elvárt módon.

Kube control

CLI, amin keresztül az adminisztrátor tud szolgáltatásokat indítani és cluster menedzsment feladatokat ellátni.

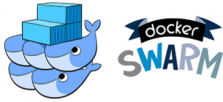


2.6. Keretrendszer meghatározása

A nagyszámú robotvezérlés feladatkörében hasonlítom össze a három legelterjedtebb orkesztrációs megoldást.

A vezérlési rendszerre legoptimálisabb választás a **Kubernetes**.

Indoklás

Egy QoS minőségbiztosított alkalmazás számára nagyon fontos a skálázhatóság és a perzisztencia. Ha kiesik egy node, akkor is fontos, hogy egy kis lassulással is, de stabil maradjon a szolgáltatás. Nagyon jól kezel multi-konténer alkalmazásokat, könnyű a szolgáltatás működése közben új konténereket indítani. A minőségbiztosítási feltételeket a Kubernetes biztosítja a legjobban.

Konténer orkesztráció rendszerek			
			 kubernetes
Konténerre optimalizált	✓	✗	✓
Egyszerű telepítés	✓	✗	✗
Szolgáltatás skálázhatóság	✗	✗	✓
Horizontális skálázhatóság	✓	✓	✓
Vertikális skálázhatóság	✗	✓	✓
Multi-konténer deploy	✓	✗	✓
Minimum node	1	3 masters + slaves	master + 2 slaves

2.1. táblázat. Konténer orkesztrációs rendszerek összehasonlítása

3. fejezet

Robotvezérlés környezete

3.1. Robotirányítás

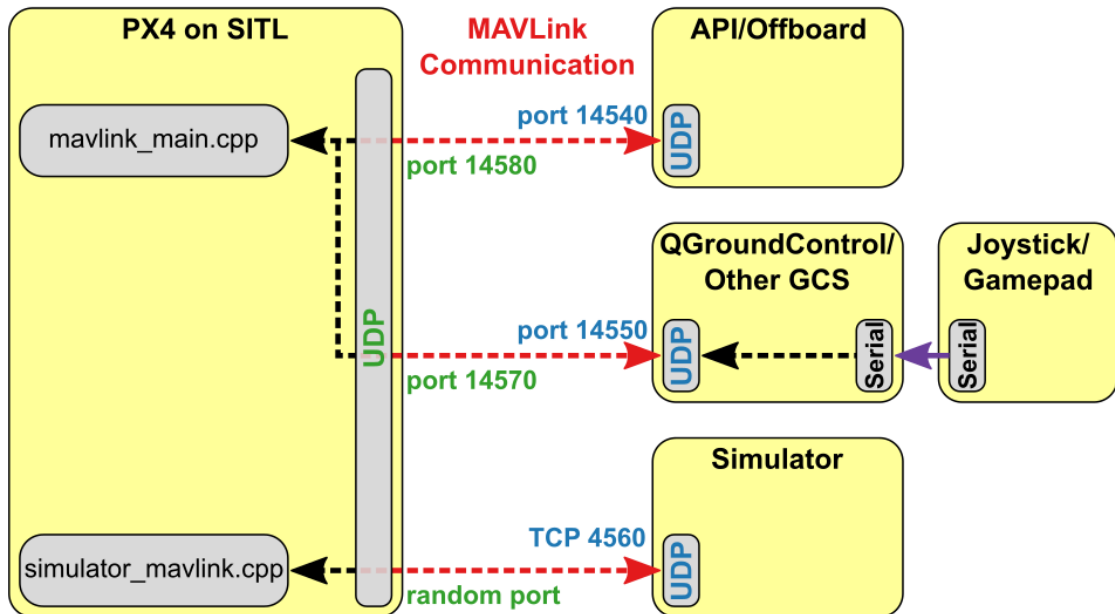
Robotirányításhoz szükséges, hogy legyen egy mechanikai rendszer, ez a robot azon részrendszere, amely az akciót valósítja meg. Az akció során szükség lehet a robot mozgatására a környezetben, ezt a helyváltoztató berendezés végzi. Motorok, különböző mechanikai tagok teszik lehetővé a helyváltoztatást. A szenzoros rendszer belső állapota maga a mechanikai rendszer állapota, míg a külső állapot a környezet állapotát jellemzi. Sokféle külső környezeti állapot létezik. Ahhoz, hogy a különböző környezeti tényezőket, például hőmérséklet, fényerősség, mágnesesség, láthatóvá és érzékelhetővé tegyük a robotunk számára, fel kell szerelni a megfelelő szenzorokkal. A dolgozat során egy Mantis Q drónt (3.1. ábra) használunk robotként, a többit szimuláljuk.

3.2. Robot operációs rendszer - ROS

Bármilyen robot rendelkezik különféle eszközökkel amivel érzékelik a világot és mozognak benne. A Robot operációs rendszer egy nyílt forráskódú könyvtárakat és eszközöket kínál a szoftverfejlesztés segítségére robotot, mint hardvert irányító alkalmazások létrehozásához. Hardver absztrakciót, driver-eket, könyvtárakat, megjelenítőket, üzenetek továbbítását, csomagkezelést és más szolgáltatást is nyújt. [22] A ROS node-ok kommunikálni tudnak egymással, a szolgáltatások, hogy kérést küldjenek és választ kapjanak a node-ok között. A *rosservice* szolgáltatással kapcsolódhatunk a ROS szerveréhez. A *roslaunch* utasítás egy *.launch* kiterjesztésű fájl alapján indít egy robotot, a megadott paraméterekkel.



3.1. ábra. Yuneec Mantis Q [25]



3.2. ábra. PX4 kommunikációja Mavlink protokollal [19]

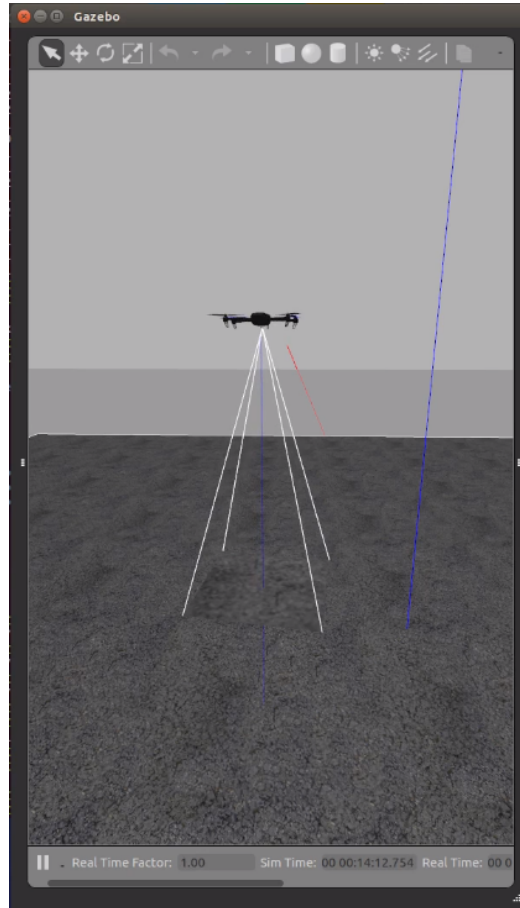
3.3. Kommunikáció - Mavros, Mavlink

A MAVLink egy egyszerű üzenetküldési protokoll a drónokkal (és a fedélzeti drónkomponensek között) történő kommunikációhoz. A MAVLink hibrid publish-subscribe és point-to-point tervezési mintát követi. Az adatfolyamok témákként kerülnek elküldésre / közzétételre, míg a konfigurációs alprotokollok, például a missziók vagy a paraméterek point-to-point közötti újraküldéssel. Az üzeneteket az XML fájlok határozzák meg. Minden XML fájl meghatározza az adott MAVLink rendszer által támogatott üzenetkészletet. [15] A MAVLink nagyon hatékony, extrém kicsi az overhead, így QoS célra ideális.

A mavros egy ROS kiegészítő, amely megvalósítja a MAVLink kommunikációját a ROS-t futtató számítógépek, a MAVLink-kompatibilis autopilotok és a MAVLink-kompatibilis Ground Control Station (GCS) között.

3.4. Vezérlő - PX4

A PX4 tag körben elterjedt mint robot vezérlő eszköz, akár egyéni, akár ipari felhasználásra. A nyílt forráskódú PX4 használható drón, de akár tengeralattjáró vagy hajó vezérlésére is, sőt könnyedén testreszabható eszközöket lehet vele készíteni, illetve megosztani a közösségi platformjukon. [20] A ROS használható PX4-el és a Gazebo szimulátorral. A MAVROS MAVLink node-on keresztül használja a PX4-el való kommunikációhoz. A ROS és Gazebo integrált rendszer a 3.2 ábrán látható módon végzi a kommunikációt egy generikus PX4 szimulációs környezetben. A PX4 a szimulátorral (például Gazebo-val) kommunikál, ahonnan megkapja a szenzor adatot, esetünkben a kamera adatát a szimulált világból, illetve elküldi a motor és rotor vezérlési paramétereit. A PX4-el közvetlen fizikai eszközöket mozgatunk, ebben a környezetben a fejlesztő feladata, hogy megvalósítsa, hogy például egy méteres magasságba felszálláshoz a drónnak milyen eszközeivel mit kell csinálni. A PX4 továbbá kommunikál a GCS-el és a fedélzeti API-val, ami esetünkben a ROS, ahova parancsokat tud küldeni. [19]



3.3. ábra. Gazebo kamerás drón modell

3.5. Szimulációs környezet - Gazebo

Mivel a dolgozat egy nagy számú eszközkiszolgálást szeretne bemutatni és a tanszéknek egy Mantis Q drónja van, ezért a többinek a kiszolgálását szimulálni fogjuk, erre pedig kell egy szimulációs környezet, ami a Gazebo lesz. A Gazebo szimulátor lehetővé teszi az algoritmusok gyors tesztelését, a robotok tervezését, a regressziós tesztek elvégzését és akár AI rendszer kialakítását. Továbbá lehetőséget nyújt a robotok pontos és hatékony szimulálására komplex beltéri és kültéri környezetben. Kézhez kapunk egy 3D-s felületet is, amivel figyelni tudjuk a szimulációt, továbbá felhő alapú támogatást is biztosít, ami a témában még jól fog jönni. A Gazebo különböző modelleket támogat, mint például a SITL optical flow, ami pont egy Mantis Q jellegű kamerás drónt szimulál bejövő és feldolgozható optikai adatcsomaggal (3.3. ábra). Az ábrán látható drónból kijövő fénykúp által vetődő keret lesz a kamera képe. Sőt egy szimulált világba akármennyi modellt képesek vagyunk szimulálni a következőképpen.

```
cd src/Firmware
DONT_RUN=1 make px4_sitl_default gazebo
Tools/gazebo_multi_vehicle.sh -m sitl_optical_flow -n 10
```

3.1. kódrészlet. 10 optikai adatfolyamos drón szimulálása Gazebo-val

Az 3.1. számú listázásban futtatott script egyépként a szimulációs fájlrendszerben *xacro* fájlokat módosítva éri el, hogy létrejöjjenek a kívánt modellek. Ezen a scripten könnyen lehet módosítani saját tetszésünk szerint. A modellek elérésének az UDP portjai 14560-tól,

a TCP portjai 4560-tól inkrementálódnak, továbbá az összes a 14550 porton broadcast-el.
[9]

A Gazebo felbontható szerverre és kliensre és a következő számításokat végzik:

Szerver

- Fizika kiszámítása
- Szenzorok szimulálása
- Engine API

Kliens

- Renderelés
- GUI

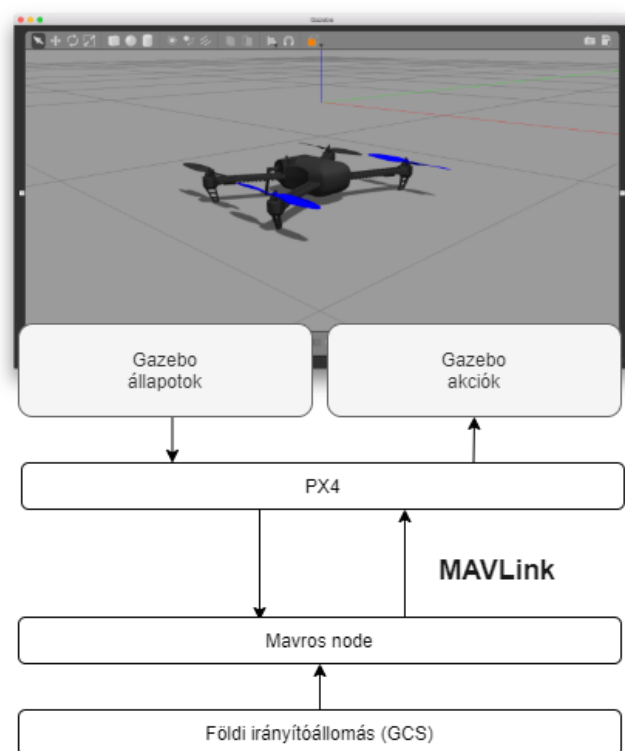
3.6. Együttes működés

A következő roslaunch utasítás egy lokális szimulációt indít a ROS-t összekötve MAVROS-al a 3.2. ábrán látható módon keresztül.

```
roslaunch mavros px4.launch fcu_url:="udp://:14540@127.0.0.1:14557"
```

3.2. kódrészlet. Lokális PX4 szimuláció ROS-on keresztül Mavlink-el összekötve

A szimulált világban való videót a QGroundControl alkalmazással lehet megfigyelni, továbbá akármennyi modell manuális irányítását is ezzel az eszközzel teszteltem. Persze a későbbi tömeges szimulációhoz, majd valamilyen automatizmusra lesz szükség. Az együttes működéshez szükséges egyfajta proxy, a mavlinken keresztül való kommunikációhoz, ami a Mavros (3.4. ábra).



3.4. ábra. Az együttes működés kommunikációja Mavros node-on keresztül

4. fejezet

Drónvezérlést tesztelő környezetek

4.1. Képfelismerésen alapuló drónvezérlő automatizált drónvezérlő scenario

A diplomatervezés második félévére továbbfejlődött a projekt, aminek integrálása a feladatom, így a 4. fejezetben taglaltaknál egy komplexebb rendszert is kialakíthatunk, de továbbra is a már tesztelt négy konténeren alapszik a megoldásom. Azonban ebben a fejezetben megnézzük, hogyan néz ki a felhő nélküli, egy gépen, konténereken megvalósított megoldás. A fejlesztéshez kaptam egy *drone-control-2* nevű könyvtárat, ami egy `docker-compose.yml` fájlt és a hozzá tartozó Docker konténerek alkönyvtárait tartalmazza. A féléves munkám során készített kódbázis feltétele, hogy a gyökörkönyvtár szülője tartalmazza a *drone-control-2* nevű projektet vagyis annak a 2020 november harmadikán módosított változatát.

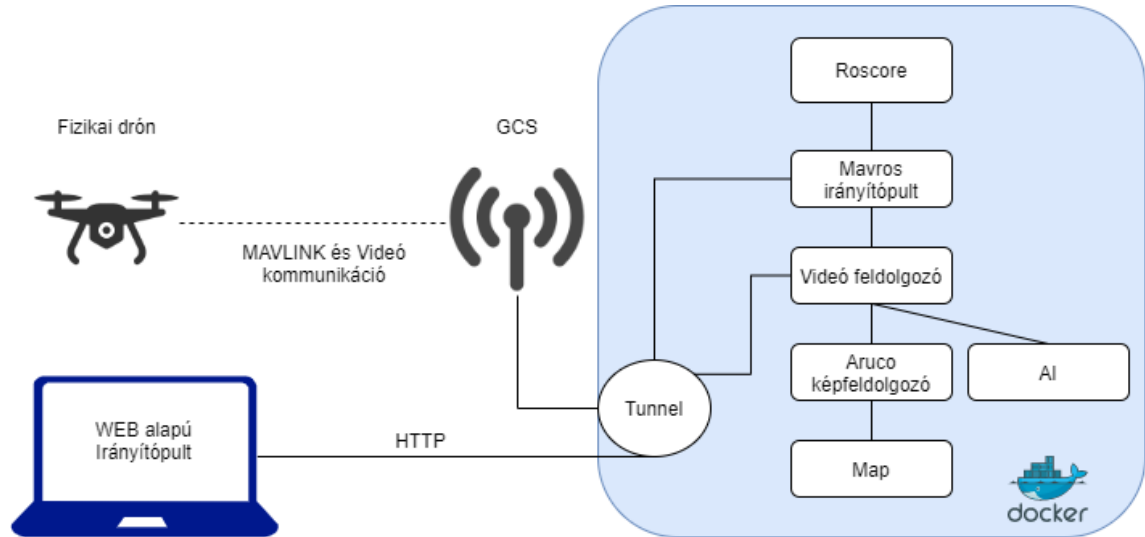
A földön használt rendszer a következő komponensekből áll:

- Fizikai drón
- GCS kommunikációs eszköz
- VKE irányítóközpont

A GCS eszköz valósítja meg a közeghozzáférést a fizikai drónnak, amin keresztül csatlakozik a hálózathoz. Ezt a közeghozzáférési technológiát nem elemezzük, azonban feljegyezzük, hogy ebben a kiépített rendszerben a vezérlőfelületen kiadott parancs a drónhoz $T_{GCS} = 300ms$ késleltetés idő alatt kezdi meg a végrehajtását.

A VKE irányítóközpont felel minden feldolgozandó és irányítandó tényezőről, továbbá a felhasználói interfészről is. A kész VKE rendszer blokkdiagramja a 4.1. ábrán látható. A kék háttér mögött minden komponens Docker konténerben fut és hostname-ek alapján kommunikál. A komponensek a következő feladatokat látják el:

- Tunnel: bevezeti kívülről a Mavlink és a HTTP kommunikációt
- Roscore: A ROS számításait végzi
- Mavros irányítópult: A felhasználói interfész és az irányítás megvalósítása
- Videó feldolgozó: A drón kamerájából érkező jelet fogadja
- Aruco képfeldolgozó: Az Aruco kódokat detektálja a kamerajelen



4.1. ábra. Kész VKE földi rendszer

- AI: A kamerakép mesterséges intelligencia alapú feldolgozása
- Map: Az aruco kódok alapján parancsokat ad ki

Ezen bemutatott rendszer az amit felhőbe integrálok, fizikai drón helyett szimulációval és leegyszerűsítve a funkciókat és a hálózati megvalósítást. A négy konténer amivel tovább dolgozom, az a Roscore, Mavros irányítópult, Aruco képfeldolgozó és ami a 4.1. számú ábrán nincs rajta, a PX4 alapú Gazebo szimuláció konténere. Az ábrán látható konténerek mindegyike Docker technológiával megvalósított és a *drone-control-2* gyűjtőkönyvtár tartalmaz egy *docker-compose.yml* fájlt, ami az adott konténerek más környezetben való összehangolásában van segítségünkre. Az egyes almappák mindegyike tartalmaz egy *Dockerfile*-t, ami megmutatja, hogy Docker környezetben az adott konténer miként van létrehozva, mik a szükséges elvégzendő műveletek az alapműködésükhöz.

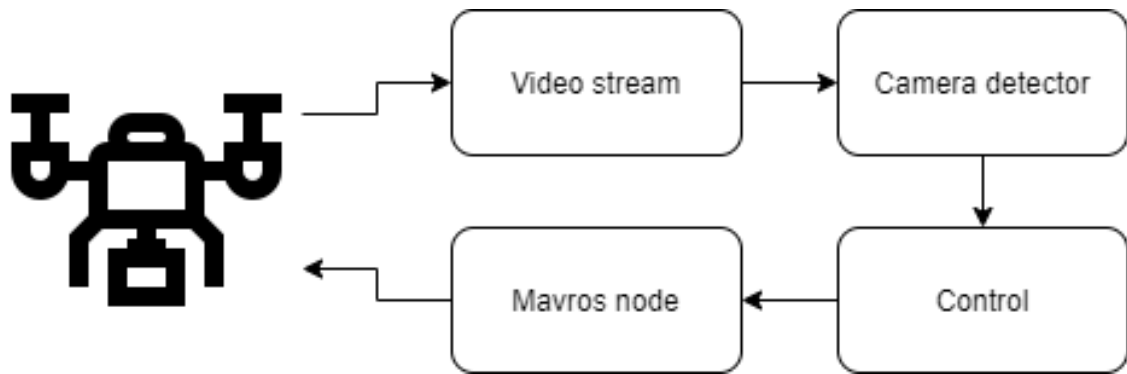
A felhőberendszerbe való integrálás előtt két tesztet végzek Ubuntu VM-eken, amik a működés szempontjából fontosak lesznek.

4.2. Azonos konténerrel egy drónvezérlés virtuális környezetben

Az első ilyen teszt egy a TMIT tanszék egyik előre készített konténerével lesz, ez a DockerHub-on megtalálható *bmehsnlab/aruco_detect_image_v2* konténer. Ebbe a konténerbe bele van építve a videó stream, a kép feldolgozása, ami aruco kódokat detektál, a Mavros node működése és az irányító program. Ezen konténerek indítása előtt ki kell adnunk az *xhost* parancsot, mivel az X szerveren keresztül kommunikálnak egymással. Az X szerver kommunikációjához a konténereknek felcsatoljuk a */tmp/.X11-unix* fájlt.

```
xhost +
docker run --net=host -v /tmp/.X11-unix:/tmp/.X11-unix --name stream bmehsnlab/aruco_detect_image_v2
docker run --net=host -v /tmp/.X11-unix:/tmp/.X11-unix --name dtctor bmehsnlab/aruco_detect_image_v2
docker run --net=host -v /tmp/.X11-unix:/tmp/.X11-unix --name mavros bmehsnlab/aruco_detect_image_v2
docker run --net=host -v /tmp/.X11-unix:/tmp/.X11-unix --name cntrol bmehsnlab/aruco_detect_image_v2
```

4.1. kódrészlet. Azonos konténerek indítása négy különböző feladattal és az X szerveren való kommunikációt megvalósítva

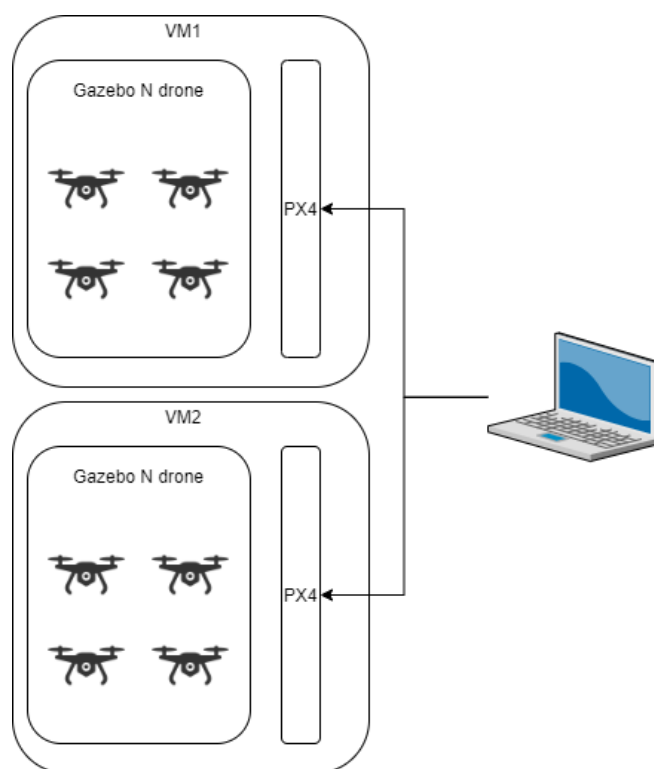


4.2. ábra. Négy azonos konténerrel drónirányítás

A teszt architektúrája a 4.2. képen látható. A teszt sikeresnek mondható, mivel a drónt sikerült irányításra bírni, illetve az optikai adatfolyamot fel tudta dolgozni az aruco kód-feldolgozó, habár aruco kódok nem voltak a szimulált világban.

4.3. Két VM-en több drón szimulációja és vezérlése

A következő teszten egy host OS-ből indított két VM-en teszteltem a több drón irányítását manuálisan. A teszthez telepítettem a VM-eken a fejezetben már felsorolt szoftvereket és környezeteket és a 3.1. listázásban bemutatott módon több drónt szimulációt is indítottam egy VM-en. Majd ezeket a földi irányítóállomás szimulátorával manuálisan vezéreltem. A teszt architektúrája a 4.3. ábrán látható.



4.3. ábra. Két VM-en több drón irányítása

5. fejezet

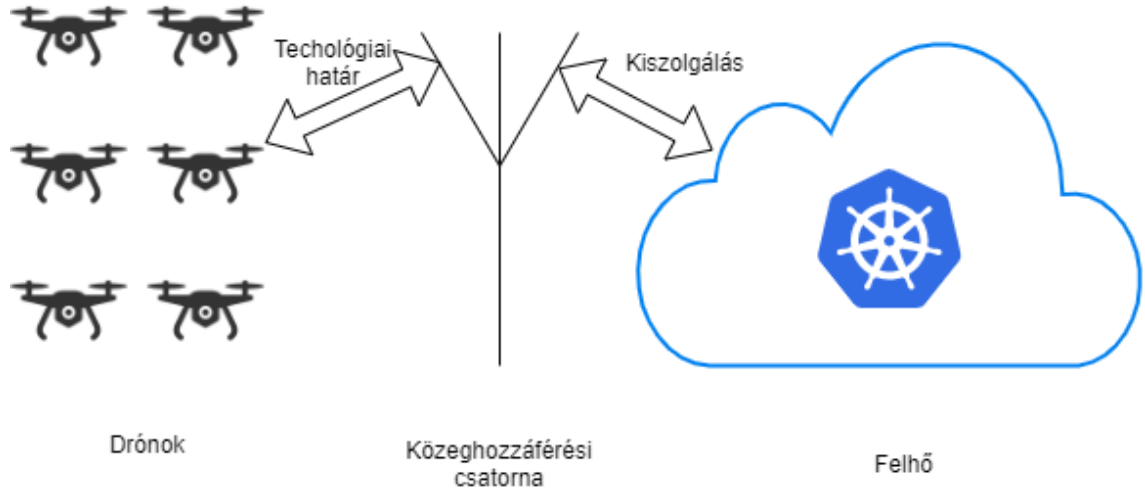
QoS sztohasztikus becslése

A diplomaterv feladatai közé tartozik a Quality of Service (QoS), azaz a szolgáltatás minőségének a biztosításának megvalósítása, továbbá az 5G feltételrendszerével a feladat áttekintése. Egy durva modellben (5.1. ábrán látható), a robotok felhővel való kiszolgálásának késleltetését két helyen vizsgálhatjuk:

1. Van egy technológiai határ, amit a nagy eszközszám kommunikációja határoz meg, ez annak a határa, hogy hány eszköz tud stabilan kommunikálni egy hálózaton, például az egyetem hálózatán. Tudjuk, hogy a Bluetooth, IEEE 802.11 különböző változatai és az LTE is más-más eszközmennyiséget tud kiszolgálni egyszerre egy antennán keresztül. Ebben a fejezetben ennek a limitációnak a kibővítéséről nem fog szó esni, de későbbiekben megnézzük, hogy mit tehet ennek a kibővítésnek az érdekében az 5G technológia.
2. A másik a felhő kihasználtsága és feladatvégzési ideje. Itt a felhő feladatokat kap sorban valamilyen intenzitással és ezeknek a feladatoknak van egy elvégzési ideje, nyilván a Kubernetes cluster kapacitásának függvényében. Nyilvánvaló, hogy a felhő kihasználtsága valamilyen függőségben lesz a kiszolgálás idejével, ami a QoS-t meghatározza. Ebben a fejezetben azt a határt nézzük meg, hogy hogyan tudjuk méretezni a felhőnket, hogy valamilyen t_0 felsőbecsléssel élhessünk egy drón kiszolgálásának idejére, természetesen miliszecundumban.

5.1. Tömegkiszolgálási modell

Valamennyi R drón valamilyen λ [kérés/s] intenzitással fordul a felhőhöz, hogy az számítását végezzen a kameraképen és megmondja mit kell csinálni. Erre a drón valamennyi idő múlva választ kap és ennek nem lehet a késleltetés értékét jelentősen megnövelni. A megbecsléséhez majd meg kell mérnünk, hogy átlagosan egy ilyen kérést a felhő mennyi idő alatt végez el, ez legyen μ . Feltételezhetjük, hogy nagy R drónszám esetén függetlenül érkeznek a kérések, így modellezhetünk Poission folyamattal a beérkező kéréseket nézve. Tehát ha egy drónnak a kérési intenzitása valamilyen feladatra $\frac{\lambda}{R}$, akkor az összes drónnak λ . Ezt tekinthetjük egy Markov folyamatnak, ahol az infinitezimális generátorból tudunk majd valamilyen sorhossz várhatóértékeket kiszámítani. Mivel nem vesznek el kérések, csupán torlódik a sor, ezért ez egy M/M/1 kiszolgálási modell. [14] Az infinitezimális



5.1. ábra. Több robot kiszolgálásának felhőből a durva modellje

generátor definíció szerint

$$G = \begin{bmatrix} -\lambda & \lambda & 0 & 0 & 0 & 0 & \dots \\ \mu & -(\lambda + \mu) & \lambda & 0 & 0 & 0 & \dots \\ 0 & \mu & -(\lambda + \mu) & \lambda & 0 & 0 & \dots \\ 0 & 0 & \mu & -(\lambda + \mu) & \lambda & 0 & \dots \\ 0 & 0 & 0 & \mu & -(\lambda + \mu) & \lambda & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Ami annyit mond, hogy a várakozó eszközök sorhossza λ intenzitással növekedik egyet, amikor érkezik egy kérés és μ intenzitással csökken egyet, amikor elvégződik egy kérés.

5.2. Várakozási idő várható értéke

Ha a tömegkiszolgálási modellünk jó és tényleg tekinthetjük a kérések eloszlását egy véletlennek, amit nem nagyon specifikus drónirányítás esetben megtehetünk, akkor egy M/M/1-es rendszer kiszolgálási késleltetését kell megnéznünk. A [14] jegyzet 4.4-es fejezet bebizonyítja, ennek a modellnek a késleltetését, ami a drón kérésére visszakapott válasz a felhő és az antenna között a D késleltetéssel arányos. A jegyzet bizonyítása alapján:

$$D = \frac{1}{\mu - \lambda}$$

6. fejezet

Kialakított Kubernetes alapú felhő

Ebben a fejezetben rátérünk a felhőrendszer tényleges megvalósítására. A 2. fejezetben megnéztük milyen lehetséges mai technológiák közül választhatunk, realizálhattuk, hogy a Kubernetes tűnik a legjobb választásnak ilyen célra, most pedig ezen a vonalon haladunk tovább. Megnézzük Kubernetesen belül milyen lehetőségek vannak, mik a probléma alapfeltételei és hogyan lehet integrálni a 4.1. fejezetben bemutatott konténerkollaborációt egy ilyen Kubernetes felhőbe.

6.1. Kubernetes technológiái

Több Kubernetes technológia közül választhatunk, bepillantunk némelyikbe, hogy mire jó és miért ezt választjuk vagy nem választjuk.

6.1.1. K8S

A K8s a Kubernetes rövidítése ("K", majd 8 "ubernete", majd "s" betű). Azonban általában, amikor az emberek Kubernetesről vagy K8-ról beszélnek, akkor az eredeti upstream projektről beszélnek, amelyet a Google valóban rendkívül elérhető és skálázható platformként tervezett.

Tehát a Kubernetes minden alapfunkcióval, mely összességének tulajdonságai:

- elválasztott Master és Worker node-ok, biztosítható az irányítás erőforrása
- etcd külön clusteren futtatható, biztosítható a terhelés kezelése
- ideális esetben külön bejáratú csomópontokkal rendelkezik, hogy azok könnyedén kezeljék a bejövő forgalmat, még akkor is, ha az alatta lévő csomópontok némelyike foglalt. [2]

6.1.2. K3S

A K3S egy egyszerűsített változata a K8S-nek, melynek forrása 40MB bináris fájl, amely teljesen implementálja a Kubernetes API-t. Rengeteg extra driver-t kihagytak belőle, melyre alap esetben nincs szükség tesztrendszer vagy egyszerű klaszter esetén. Ezeket a kihagyott funkciókat egyébként később hozzá lehet illeszteni a rendszerhez add-onokkal. [2]

6.1.3. Kind

A Kind egy Docker fölötti Kubernetes megvalósítás egy node-on. Egyszerű installálni, azonban nem a Kubernetes API-t használja.

6.1.4. MiniKube

A MiniKube az első Kubernetes technológia amely a fejlesztők ajánlása alapján a kezdőknek kipróbálásra a legalkalmasabb. Mivel egyszerű telepíteni, nincs nagy erőforrásigénye (2 vCPU/2GB RAM/20GB lemez). Egy gépre installálható, nem adható több node a klaszterhez. [3]

6.1.5. Miért K3S?

A tanszéki klaszter természetesen egy teljes kialakított K8S, melyen az eredeti API használható és teljesértékű szolgáltatásokat lehet tesztelni a Kubernetes összes optimalizálásával. A Kind más API-t használ, így a telepítést leegyszerűsíteni, azonban nem összeegyeztethető egy Kind-os applikáció K8S megvalósításával. MiniKube már egyel jobb, azonban csak egy node-ot használ, ebben a projektben pedig fontos a hálózati tesztelés több node között. Így marad a K3S, amellyel a legjobban szimulálhatjuk a tanszéki K8S rendszert és a megvalósított applikáció is könnyen portolható.

6.2. Földi scenario felkészítése a menedzselt felhőrendszerbe

A felhasznált Docker konténereket néhány esetben változtatni kellett, ez csak a *Dockerfile*-ra igaz, a forráskódok az eredeti esetben is működtek nem K3S rendszeren. Mindegyik konténerben volt egy kivétel, amely csak akkor engedte futtathatóvá tenni a konténert, ha az Docker rendszerben fut, ezt a *[.dockerenv]* fájl létezésére vonatkozó feltétel.

A PX4 szimuláció futtatószkripje (*vke_px4sim/docker-entrypoint.sh*) ugyan tartalmazza a beépített *ROS_IP* hirdetőcímet, amelyet a hálózati fejlesztés során többször átírtam, sikerült olyan végeredményre jutni, amelyben az eredeti konténer külső interfész IP-je maradhat.

Az előkészített Roscore konténert a végleges verzióban nem használom, hiszen egyszerűbb volt az eredeti *alpineros/alpine-ros:noetic-ros-core* publikus image-t megadni, amit a K8S API-ban tudtam testreszabni indított portszámmal és környezeti változókkal.

6.3. Kubernetes virtualizált telepítése Multipass VM-eken

Egy több node-os klasztert valósítok meg virtualizáció fölött. Virtuális gépek létrehozására rengeteg program létezik, én a Multipass-t választottam. A Multipass egy letisztult VM menedzser Linux-ra, Windows-ra és MacOS-re, amellyel egy parancs indítani és törölni különálló VM-eket bármely CloudInit-et tartalmazó image-ről. Azért választottam ezt, mert könnyedén szkriptelhető a Klaszter törlése és felhúzása, mivel a fejlesztés során sokszor szeretném az alapbeállításról indítani a klasztert.

Tehát írtam egy Bash scriptet, aminek segítségével létrehozok három VM-et és inicializálom rajtuk a K3S klasztert. Multipass VM-ek létrehozása Bash-ben egy-egy parancs (6.1. számú kódrészlet), melyek paramétereit természetesen egy config fájlból olvasok be, amiről még később szó lesz.

```
multipass launch --name master --cpus 2 --mem 2G --disk 2G
for w in "worker-1 worker-2"; do
  multipass launch --name $slave --cpus 2 --mem 2G --disk 30G
done
```

6.1. kódrészlet. Multipass VM-ek létrehozása

Létrehozás után a szkript kiadatja az Apt csomagkezelő program *update és upgrade* parancsait, hogy a legfrissebb Ubuntu 20.04 kompatibilis csomagok legyenek a VM-eken.

Frissítés után a master VM-en inicializálható a K3S master módban. A parancs eleje azt mutatja, hogy a master nevű VM-en hajtjuk végre a `”-”` utáni parancsokat (6.2. kódrészlet).

```
multipass exec master -- /bin/bash -c "curl -sL https://get.k3s.io | K3S_KUBECONFIG_MODE="644" sh  
-"
```

6.2. kódrészlet. K3S Master inicializálása

Ha ez sikeres, akkor a worker node-okat is inicializálhatjuk, amihez két dologra lesz szükség, a master külső IP-jére, amin keresztül a másik VM tudja elérni, illetve a K3S egyedi tokenre. A master külső IP-jét a multipass egyik parancsából olvassuk ki, a grep programmal rákeresve a master névre, majd az IP cím formátumára reguláris kifejezéssel. A tokenhez szimplán kiíratunk egy fájlt a cat programmal. Ezekkel pedig a master-hez hasonló módon a K3S dokumentációban megadott curl programhívással inicializáljuk a slave-eket. (6.3. kódrészlet)

```
K3S_TOKEN=$(multipass exec $master sudo cat /var/lib/rancher/k3s/server/node-token)  
MASTER_IP=$(multipass list | grep $master | grep -oE "\b([0-9]{1,3}\.){3}[0-9]{1,3}\b")  
  
for slave in $slaves; do  
    multipass exec $slave -- /bin/bash -c "curl -sL https://get.k3s.io | K3S_TOKEN=${K3S_TOKEN}  
        K3S_URL=https://${MASTER_IP}:6443 sh -"  
done
```

6.3. kódrészlet. K3S Slave-ek inicializálása

Sikerességet ellenőrizve a szkriptben még kiíratom a master-en a csatlakozott node-okat.

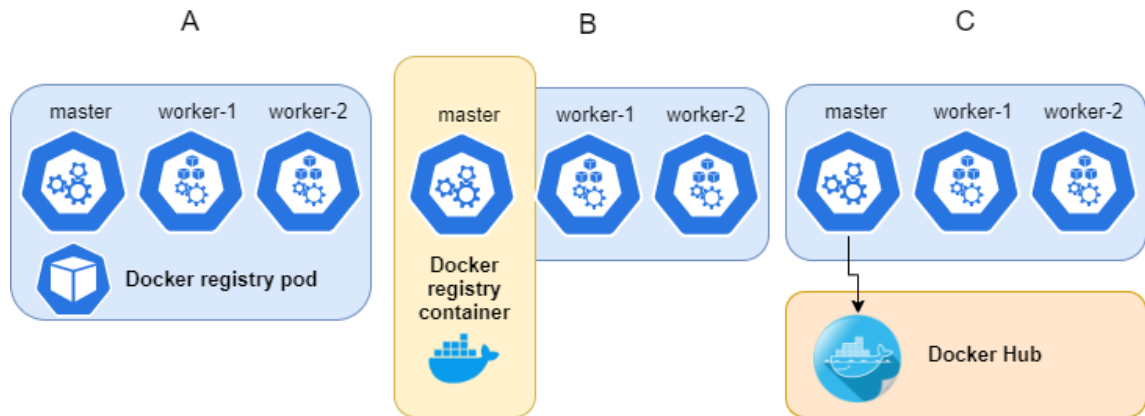
```
multipass exec $master kubectl get nodes
```

6.4. kódrészlet. Node-ok lekérdezése

6.4. Konténer registry, lokális és központi

Docker-Compose esetén a konténer kollaborációs megvalósításhoz elég volt definiálni a könyvtárt, mely forrása és konfigurációja alapján meg kell építeni az image-et és a lokális Docker környezet eltárolta ezen image-eket, amit fel is tudott használni az adott környezetben. Ez természetesnek tűnt, azonban a Kubernetes alapvetően nem tartalmaz ilyen fejlesztéseket, csakis lefordított képfájlokat tud publikus vagy privát registry-ből letölteni. A Docker Registry egy olyan verziókezelő, amely felépített konténer képfájlokat tárol és állít rendelkezésre aktívan használó rendszereknek, mint például a Kubernetes vagy Docker Swarm. Három megoldást próbáltam ki, hogy az egyedi lemezképeket használhassam a felhőmben (6.1. ábra).

A) Elsőnek úgy gondoltam, hogy a legegyszerűbb megoldás az ha a K3S vagy egy Docker környezetben, amely a K3S master-én futna, inicializálok egy előre készített Docker registry szerveret, arra feltöltöm a felépített konténerlemezeket és lokálisan elérni a Kubernetes master, deploy esetén. Tehát definiáltam egy pod-ot, amely az 5000-res porton elérhető a K3S rendszeren belül, a master-ről. A *registry* nevű image-et adtam meg, így a hivatalos Docker registry került letöltésre. Push-olni sikerült konténert, azonban a Kubernetes API-ra definiált Deployment-el már nem sikerült pull-olni SSL biztonsági okok miatt.



6.1. ábra. Registry megoldásaim egyedi konténerek esetén

B) Ezután hagytam a felhőbe integrálást, a master node-ra telepítettem Dockert és Docker-Compose-t, majd a deploy-oltam a hivatalos oldalról a registry-t (6.5. kódrészlet). A konténerek push-olása pedig szintén a master node-ról történt (6.6 kódrészlet). Ez a megoldás működött, azonban nem túl szép és optimális, mivel a master-en sokszor tárolódik egy konténer image (forrásként, Docker image-ként, Docker registry-ben, Kubernetes letöltött image-ként és példányosított konténerként).

```

multipass exec ${master} -- sudo docker run -d -p 5000:5000 --restart=always --name registry
registry:2

```

6.5. kódrészlet. Docker registry inicializálás a master docker környezetében

```

for container in "commander px4sim aruco"; do
  multipass exec ${master} -- sudo docker build ${drone_control_dir}/vke_${container}/. -t localhost
:5000/${container}
  multipass exec ${master} -- sudo docker push localhost:5000/${container}
done

```

6.6. kódrészlet. Build és push lokál konténer registry-be

C) Így inkább talán a legegyszerűbb megoldást választottam, igénybe vettem a hivatalos Docker Hub privát repository szolgáltatását, amely ugyan egy konténert biztosít ingyen, egy kis trükkkel, mi szerint ugyanazt a konténert töltöm fel 3 különböző taggel, ezt is meg lehet oldani (6.7. kódrészlet).

```

for container in "commander px4sim aruco"; do
  docker build ${drone_control_dir}/vke_${container}/. -t nanasidnl/drone_control:${container}
  docker push nanasidnl/drone_control:${container}
done

```

6.7. kódrészlet. Docker Hub build és push

A Docker Hub privát repository-hoz autentikálni kell az API-t, ahhoz hogy elérje az adott K8S deployment. Ehhez egy konfigurációs fájlba kiszerveztem a hitelesítő adatokat. A szolgáltatást létrehozó deploy szkriptbe pedig beleírtam, hogy használja fel ezt a fájlt és autentikáljon egy *regcred* titkon keresztül (6.8. kódrészlet). Ezt meg kell adni az API használatakor is (6.9. kódrészlet).

```

source ../config/docker-credentials.sh
multipass exec ${master} -- kubectl create secret docker-registry regcred --docker-server=${
  docker_server} --docker-username=${docker_username} --docker-password=${docker_password} --
  docker-email=${docker_email}

```

6.8. kódrészlet. Docker Hub autentikáció K3S-ről

```
imagePullSecrets:  
- name: regcred
```

6.9. kódrészlet. K8S API secret definiálása a konténerekhez

7. fejezet

Drónirányítás mint szolgáltatás a Kubernetes felhőben

Az előző (6.) fejezetben megmutattam, hogyan alakítottam ki a saját felhőrendszeremet. Ebben a fejezetben pedig megnézzük a diplomamunkának szánt feladat miként tud felkerülni ebbe a felhőbe. Végignézzük a Kubernetes API szolgáltatásait, mely technológiával mit lehet elérni a cél érdekében. Kezdjük a legegyszerűbb Pod-tól és betekintünk az API szolgáltatásainak jelentős részébe.

7.1. Szimpla 4 podos megvalósítás

Elsőként a legegyszerűbb összeállításban szerettem volna megbizonyosodni arról, hogy a szolgáltatás valóban tud működni a felhőben ezért elsőnek a négy konténert 4 különálló pod-ként próbáltam ki (7.1. kódrészlet).

A Pod a legkisebb telepíthető számítási egység, amely létrehozható és kezelhető a Kubernetesben. Egy Pod létrehozása esetén példányosodik az előre definiált konténer vagy konténer csoport, azonban a futás megkezdése után semmilyen felügyeleti szolgáltatásban nem részesül, nem indul újra leállás esetén, nem alkalmazódik rá semmilyen skálázás. Egyszerűen ha csak Pod szinten futtatunk alkalmazásokat, akkor ugyanott járunk mintha Docker Swarm környezetben dolgoznánk.

```
apiVersion: v1
kind: Pod
metadata:
  name: aruco
  labels:
    name: drone-hq
spec:
  hostname: aruco
  containers:
  - image: nanasidnl/drone_control:aruco
    name: aruco
  env:
  - name: ROS_MASTER_URI
    value: "http://roscore:11311"
  - name: ARUCO_LAUNCH
    value: "sim"
  imagePullSecrets:
  - name: regcred
```

7.1. kódrészlet. Példa egy Pod-ra a négyből

Jelen négy Podos megvalósításban a belső hálózati problémák nem merültek fel, hiszen a Pod neve hostname-ként is funkcionál, így egymás között könnyedén el tudták érni egymást

az alkalmazások. Ez egy működő verzió volt, azonban túl egyszerű. A drón szimulációt nem a felhőben képzeljük el, hiszen a fizikai drón sem a felhőben fog futni.

7.2. Konténerek hálózata, Service és Deployment

Ahhoz, hogy a Kubernetes szolgáltatásait érdemben igénybe tudjuk venni valamilyen magasabb szintű API-t kell használnunk. Ezért megnézzük a Deploymentet és a Service-t. Utóbbi arra szolgál, hogy a szolgáltatásunk megfelelő külső interface-el rendelkezessen, bármely mögöttes konténerkonstrukcióval, amely integritását a Deployment biztosítja. Mindegyik Pod megkapja a saját IP-címét, azonban egy Deployment során az egy pillanatban futó Podok halmaza eltérhet az adott alkalmazást egy időpillanattal később futtató Podok halmazától. A nem natív alkalmazások esetében a Kubernetes lehetőséget kínál hálózati port vagy LoadBalancer elhelyezésére az alkalmazás és a háttéralkalmazások között. [13]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: drone-hq
spec:
  replicas: 1
  selector:
    matchLabels:
      app: drone-hq
  template:
    metadata:
      labels:
        app: drone-hq
    spec:
      containers:
        - name: roscore
          image: alpineros/alpine-ros:noetic-ros-core
        - name: aruco
          image: nanasidnl/drone_control:aruco
        - name: commander
          image: nanasidnl/drone_control:commander
          ports:
            - containerPort: 80
          image: nanasidnl/drone_control:px4sim
          ports:
            - containerPort: 10000
      imagePullSecrets:
        - name: regcred
```

7.2. kódrészlet. Példa 4 konténeres deployment megoldásra

Elsőnek a 4 podos verzióból készítettem egy Deploymentet (7.2. kódrészlet), amely már egy Pod-ként definiálható, azonban érvényesülni fognak rá a Kubernetes integritás szabályai. A Roscore konténert éri el az összes, de Pod-on belüli kommunikáció a legkönnyebb, hiszen egy Pod egy lokális hálózatnak származtatható le, így a konténerek *localhost*-ra címezve tudnak kommunikálni egymással, mintha virtuális környezetben lennének. Külső interfész 2 van, a Gazebo szimuláció és az irányító dashboard.

```

apiVersion: apps/v1
kind: Service
metadata:
  name: drone-hq
spec:
  ports:
    - port: 80
      targetPort: 81
      name: dashboard
    - port: 10000
      targetPort: 10001
      name: gazebo

```

7.3. kódrészlet. Példa 4 konténeres megoldás service kivezetésére

A külső interface-ek kapcsán jön képbe a Service API, míg minden belső kommunikációt le tud kezelni a Deployment, ahhoz hogy ezeket a külső interface-eket elérjük kell definiálnunk egy Service-t a Deployment fölé (7.3. kódrészlet), amely a Deployment selectorán keresztül kapcsolódik hozzá és a definiált portokhoz rendel egy külső portot. Ez a külső port valjában csak a Kubernetes rendszeren definiált flannel hálózatról elérhető. Tehát más a rendszeren futó alkalmazás el fogja tudni érni, de kívülről nem. Márpedig a fizikai drón irányításra tervezünk, így következő lépésként a befelé való kommunikáció megvalósíthatósági technológiáit fogjuk megvizsgálni.

7.3. Szolgáltatás külső elérése

Több Kubernetes API szolgáltatás rendelkezésünkre áll, hogy elérhessük kívülről a konténerünket amin a felhasználói felület fut, illetve a drónnak kivezessük a roscore konténer Mavlink protját.

7.3.1. NodePort

A NodePort szolgáltatás a legprimitívebb módja a külső forgalom közvetlenül a szolgáltatásának elérésére. A NodePort, amint a neve is mutatja, megnyit egy adott portot az összes csomóponton (a virtuális gépek), és az erre a portra küldött forgalmat továbbítja a szolgáltatásnak. [23] A NodePortokat hasonlóan rendelhetjük hozzá a szolgáltatás portjaihoz, mint a sima proxy-n keresztüli külső portot, azonban a NodePort számozása 30000-től kezdődhet.

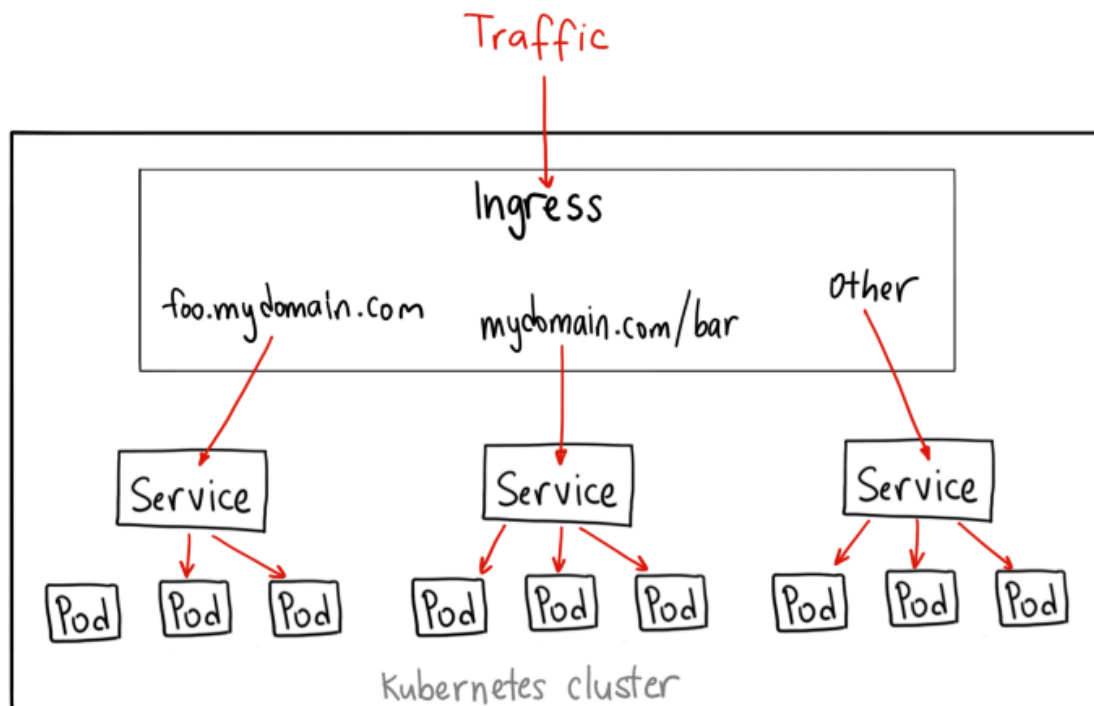
Ezzel a megoldással meghatározhatjuk, hogy a drón kívülről a Master külső IP-jén érjen el minden szükséges alkalmazást, illetve a drónt irányító személy is érje el a Dashboardot. A probléma a NodePort-tal, hogy más porton nem tud kommunikálni, így a Roscore véletlenszerűen generált portja nem lesz elérhető kívülről.

```

apiVersion: v1
kind: Service
metadata:
  name: drone-hq
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30080
      name: dashboard
    - port: 11311
      nodePort: 31311
      name: roscore
  selector:
    app: drone-hq

```

7.4. kódrészlet. Service kiegészítése NodePort-okkal



7.1. ábra. Ingress Service szétválasztása a klaszterben [23]

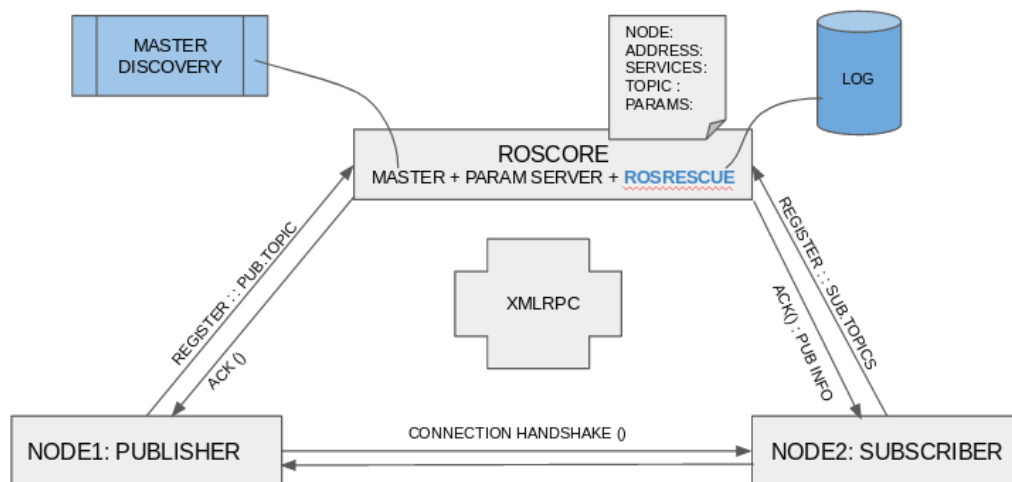
7.3.2. Ingress, szolgáltatások szétválasztása

Fontos megemlíteni az Ingress-t is, hiszen több alkalmazásban gondolkozunk, mivel lesz két konténerünk a felhőben amelyből egy példányt szeretnénk és a Roscore-t egészíti ki, ezt neveztem *drone-hq*-nak, ez tartalmazza az aruco kódolvasót és az irányítót. A másik pedig a Roscore, amiből szeretnénk backup futó szolgáltatást. Ha hosszútávú szolgáltatást szeretnénk alakítani vagy továbbfejleszteni, esetleg publikus felhőben, akkor mindenképpen érdemes az Ingress-t használni, mivel domain-eket, subdomain-eket és útvonalakat rendelhetünk a különböző szolgáltatásunk felé (7.1. ábra). Így ha publikus felhőben gondolkoznánk, akkor a roscore backup-okat ki lehetne vezetni különböző elérési címekre, illetve változtathatnánk a címek mögöttes elérését, ha szükséges.

7.3.3. ROS Port forwarding

Az Ingress megoldás biztosíthatná, hogy szétválasszuk a Roscore-t és a többi konténert a magasabb rendelkezésre állás érdekében, azonban a Roscore rendelkezik egy nem túl előnyös kommunikációs kapcsolódási stratégiával. Egy ROS rendszer működtethet számtalan eszközt, a mi példánkban egy Roscore-ra három ROS Node csatlakozik, egyedül a drón a felhőn kívülről (7.2. ábra). Mikor elsőnek csatlakozik egy Node, akkor megkap bizonyos paramétereket, mint például a portszám amin az adott node-nak kommunikálnia kell a Roscore-ral. Felmerül a kérdés, hogy a külső ROS Node hogyan fog bejutni a Kubernetes cluster szolgáltatásához véletlenszerűen generált porton.

Sokféle megoldáson gondolkoztam, azonban mivel egy egyszerű NAT-olt hálózat mögé is csak úgy lehet elhelyezni a kiszolgáló Roscore-t ha az elérhető portok intervallumát kinyitjuk a nagyvilág felé, mivel nem előre tudható, hogy milyen portot fog lefoglalni az adott Node-al való kommunikációra. Ezt részletesen bizonyítja a [7] számú hivatkozott cikk. Gondoltam olyan megoldásra is amely korlátozza a Linux alapú konténer által kiosztott



7.2. ábra. ROS Node csatlakozása Roscore-hoz [18]

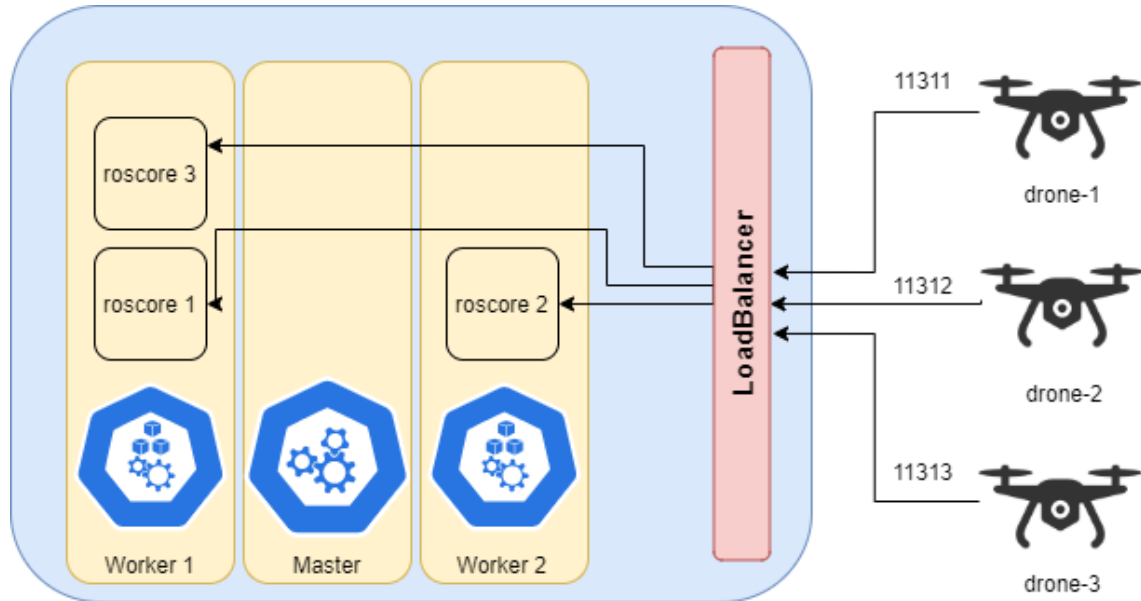
portszámot a Roscore alatt. Ezt úgy lehetne megoldani a Roscore konténer inicializálásánál lekorlátozom a kiosztható portokat a NodePort tartományra, ami 30000 és 32767 között van (7.5. kódrészlet). Majd amikor a külső eszköz felveszi a kommunikációt, akkor a logban kap egy hibát *"ERROR: Communication with node[http://10.42.2.13:32045/] failed!"*, amelyből ki tudjuk olvasni a portszámot és kinyithatjuk ezt a NodePortot. Ez azért visszás, mert súlyos másodpercek telnek el, amíg megkapjuk a hibaüzenetet, felolvassuk és eljuttatjuk a dróntól a klaszterig. Így ezt a megoldást QoS célú szolgáltatás alatt nem engedhetjük meg.

```
echo 30000-32767 > /proc/sys/net/ipv4/ip_local_reserved_ports
```

7.5. kódrészlet. Linuxon kiosztható port korlátozása

7.3.4. LoadBalancer

A valódi megoldást a LoadBalancer fogja nyújtani, ami egy standard megoldás arra, hogy egy teljes szolgáltatást kiirányítsunk a klaszterből, a teljes port intervallumon, amelyet a Roscore generálhat. Ez a port az egyik Node-on lesz elérhető, amelyiken az adott Pod fut. A LoadBalancert használhatjuk NodePort-tal is, ami a mögöttes alkalmazásunknak megfelel, hiszen onnan csak egy konstans portra van szükségünk, hogy a felhasználó elérhesse az irányító felületet. Tehát a LoadBalancer segít abban, hogy az elválasztott Roscore szolgáltatás egyedi Node elérésével legyen elérhető, továbbá lehetővé teszi, hogy különböző drónok Roscore-ja ugyanazon a Node-on fussanak és legyenek elérhetőek egymás mellett. A magok elsőkénti elérése inkrementális sorrendben fog történni az alapértelmezett porttól, a további kommunikáció pedig a kisorsolt egyedi portokon (7.3. ábra).



7.3. ábra. LoadBalancer megoldás Kubernetes felhőben

```
spec:
  hostname: roscore-$(DRONE_IDENTIFIER)
  containers:
  - name: roscore-$(DRONE_IDENTIFIER)
    image: alpineros/alpine-ros:noetic-ros-core
    env:
    - name: ROS_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: ROS_MASTER_URI
      value: "http://$(ROS_IP):$(MAVLINK_PORT)"
    args: ["roscore", "-p", "$(MAVLINK_PORT)"]
  hostNetwork: true
```

7.6. kódrészlet. Roscore Service specifikációja inkrementális változókkal

A K8S megvalósításhoz a szolgáltatás yaml fájlt kell módosítanunk az adott Service-hez (7.6. kódrészlet). Automatikusan deployoló szkriptekkel dolgoztam, így a yaml fájlokban megtalálható $\$(VARIABLE)$ formátumú változókat a klaszterhez való hozzáadás előtt adom meg. A *DRONE_IDENTIFIER* a drón azonosítója, egytől indulunk, ez alapján számoljuk ki a pontos portokat. A *ROS_IP* a konténer belső változója, amely a ROS Node-ok számára hirdetendő kapcsolattartó IP címet adja meg, a fájlban látható módon kivezettem, hogy a Kubernetes status.podIP változója legyen, tehát a Pod külső IP címe. A *ROS_MASTER_URI* pedig az a HTTP cím melyen kapcsolódni tud a ROS Node a core-hoz. A konténert nem hagyományos módon indítjuk, hanem a kiválasztott *MAVLINK_PORT* hirdető porton, melyet deploy során fog megkapni.

Ennél a pontnál vettem észre először a K3S hiányosságát, ugyanis a K8S alaphoz tartalmazza a Traefik modult, amely egy Ingress Controller-rel egybeépített LoadBalancer megvalósítás. A LoadBalancer használatához pedig szükséges hozzáadnunk utólag ezt a modult klaszterhez. Így a K3S-t installáló szkriptet kiegészítettem a Traefik pod-ként való hozzáadásával a klaszterhez (7.7. kódrészlet).

```
multipass exec $master -- sudo kubectl apply -f /var/lib/rancher/k3s/server/manifests/traefik.yaml
```

7.7. kódrészlet. Traefik hozzáadása a klaszterhez

7.4. Teljes klaszter létrehozása N drónnal

7.4.1. Kauzalitási probléma

Szimulációnk létrehozásához több különálló lépést kell megtennünk, miután üzemképes a K3S/K8S klaszter. Nézzük meg milyen információ függőségei vannak az egyes komponenseknek.

- A drónnak tudnia kell a Roscore-t futtató Node külső IP-jét
- Az irányító konténernek ismernie kell a drón IP címét

Ezen feltételekből következik a kialakított sorrendem, hogy hogyan inicializáljuk a virtuális tesztkörnyezetet. A diplomatervhez csatolt forráskód *deploy* könyvtárában sorszámozással megtalálhatóak ezek Bash szkriptek formájában.

1. K3S klaszter létrehozása, kimenet: master és worker-ek IP címei
2. ROSCore-ok indítása a klaszteren, kimenet: melyik drón-hoz tartozó core melyik Node-ra került
3. Drónok indítása, külső környezetben (Docker), kimenet: drónok IP címei
4. Drone-HQ indítása a klaszteren

A lépések végrehajtása után feláll a teljes rendszer és a drónok irányíthatóak a master külső IP-jén keresztül.

7.5. Deploy szkriptek

Az előző felsorolásból kimaradt egy nulladik szkript, amely arra szolgál, hogy a legfrissebb verziójú konténereket felépíti és feltölti a DockerHub-ra, privát repository-ba. A szkriptek egy közös konfigfájlt használnak (*config*), amely-ben a klaszter paraméterei, kezdő portszámok és a drónok száma szerepel. Ezt a konfigfájlt használja Bash és Python program is, ezért a Bash felkonfigurálására létrehoztam egy kiegészítő szkriptet (*configUp.sh*), amely lemenedzseli a mapparendszerből következő változókat, hogy mi hol van, erre csak a Bash szkripteknek van szüksége. Utána minden deploy szkript ezt a külső konfigurációt hívja meg és konzisztens változókat fognak használni.

A megoldásom két sablonfájlt tartalmaz a forrás *kubernetes* könyvtárában, *kuber-roscore.yml* és *kube-drone-hq.yml*. Mindkét fájl tartalmaz környezeti változókat, melyek értékét az adott drón azonosítószámában találhatunk ki. Ugyan a környezeti változókat tudná kezelni a Kubernetes API, a Multipass rendszeren keresztül viszont ha létrehozzuk az adott Service-t, akkor a master környezeti változóit használná fel, amit a Multipass rendszerén keresztül nem tudunk beállítani. Így ezekből a yaml fájlokból létrehoz a szkript annyit amennyi drón szerepel a konfigurációban. Ezután a Linuxos sed program segítségével cseréljük ki a környezeti változókat Bash változó értékekre. Ahhoz, hogy a gazdagép fájlrendszeréből fel tudja használni a virtuális master a yaml fájlt, a Multipass segítségével mount-olom a könyvtárat a végrehajtás előtt (példa: 7.8. kódrészlet). A példában az látszik, hogy a mount parancs után információkat gyűjtünk, a konfigurációból származik a Roscore node neve, ami kap egy címkét, majd ez alapján megkeressük a Multipass listázásával a külső IP-jét, hasonlóan a master-nek. Ez után minden drónra elvégezvén (for cikusbán), kiszámítjuk az adott drón Mavlink csatlakozási portját, lemásoljuk a sablonfájlt és a már említett környezeti változókat kicseréljük benne, végül

a *kubect*l *apply* paranccsal elindítjuk a klaszteren.

```
multipass mount ${work_dir} ${master}
multipass exec $master -- kubectl label nodes ${ROSCORE_NODE} dedicated=roscore
MASTER_IP=$(multipass list | grep $master | grep -oE "\b([0-9]{1,3}\.){3}[0-9]{1,3}\b")
ROSCORE_IP=$(multipass list | grep $ROSCORE_NODE | grep -oE "\b([0-9]{1,3}\.){3}[0-9]{1,3}\b")
for i in $(seq 1 ${NUMBER_OF_DRONES}); do
    MAVLINK_PORT=$((i-1+MAVLINK_START_PORT))
    cp ${service_dir}/kube-roscore.yml ${service_dir}/kube-roscore-$i.yml
    sed -i 's/${MAVLINK_PORT}/${MAVLINK_PORT}/g' ${service_dir}/kube-roscore-$i.yml
    sed -i 's/${DRONE_IDENTIFIER}/${i}/g' ${service_dir}/kube-roscore-$i.yml
    multipass exec $master -- sudo kubectl apply -f ${service_dir}/kube-roscore-$i.yml
done
```

7.8. kódrészlet. Roscore információk összegyűjtése és deploy

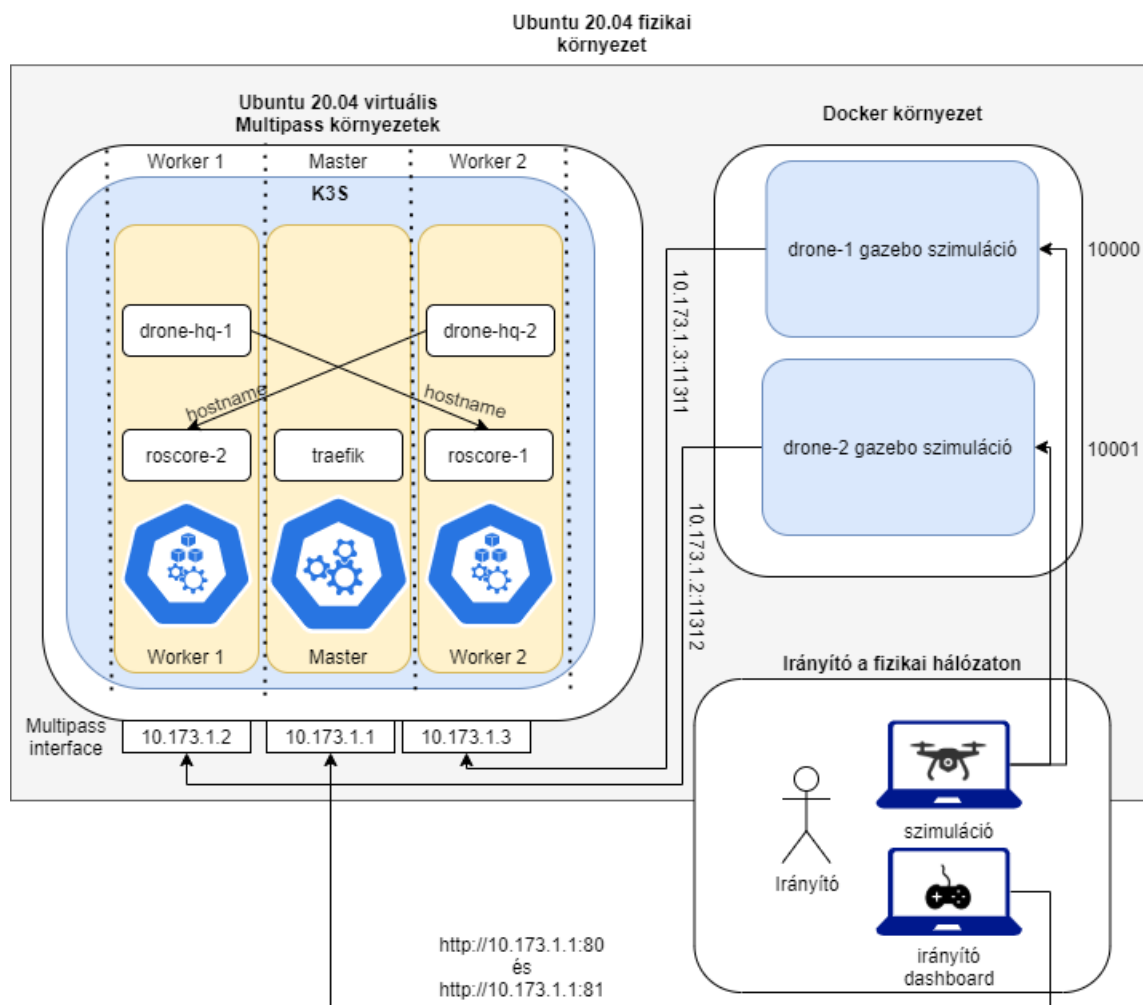
A Roscore deploy szkriptjéhez nagyon hasonló a drónokat indító szkript és a Drone-HQ-t indíti szkript. A drónok indításánál Docker megoldást alkalmazok a külső környezetben, ahol hasonlóan származnak le az adott információk, a szimuláció külső portja is hasonlóan inkrementális a Mavlink porthoz (7.9. kódrészlet). A Roscore címét a master-en keresztül kapjuk meg, a *kubect*l *get pods - wide* parancs standard kimenetéből kiolvassuk grep programmal, először az elnevezett Service nevére szűrve, utána az IP formátumra. A drón indítása Docker paranccsal történik, ahol megadjuk a környezeti változókat, melyeket alapján el fogja érni az adott Roscore-t, illetve a dashboard-ot átirányítjuk a 10000-res portról inkrementálisan, hogy akár mennyi drónt lehessen irányítani.

```
for i in $(seq 1 ${NUMBER_OF_DRONES}); do
    ROSCORE_IP=$(multipass exec $master -- kubectl get pods -o wide | grep roscore-$i | tail -n 1 |
        grep -oE "\b([0-9]{1,3}\.){3}[0-9]{1,3}\b")
    MAVLINK_PORT=$((i-1+MAVLINK_START_PORT))
    SIMULATION_PORT=$((i-1+SIMULATION_START_PORT))

    docker run --name drone-$i -e ROS_IP=${ROSCORE_IP} -e ROS_MASTER_URI=http://${ROSCORE_IP}:${MAVLINK_PORT} -p ${SIMULATION_PORT}:10000 -d nanasidnl/drone_control:px4sim
done
```

7.9. kódrészlet. Drónok indítása külső Docker környezetben

Egy teljesen felépített két drónos rendszer a 7.4. ábrán látszik. Az egy fizikai gépes környezetben a három Multipass VM-en a K3S rendszer látható, mellette ugyanabban a környezetben a Docker, amin keresztül a két szimuláció fut. A K3S klaszteren belül a három Node-on öt Pod-ot láthatunk, a Traefik alkalmazást, ami a Roscore LoadBalancer-e miatt szükséges, illetve a két-két drón kiszolgáló Pod-ot. A szimuláció konténerei a Multipass külső IP címein a hozzájuk tartozó porton érik el a Roscore-t. Az irányító lehet az adott környezetben, de akár a hálózaton más fizikai környezetben is, csak el kell érnie az alkalmazásokat az adott portokon.



7.4. ábra. Teljes környezet két szimulációval

8. fejezet

Drónkapcsoló állomás Kubernetes felhőben

Ebben a fejezetben, a kialakított K3S alapú drón kiszolgáló központot fejlesztem tovább, úgy hogy valamilyen QoS feltételeket valósítson meg. Megnézzük, hogyan választottam technológiát a szoftver architektúráját, részleteit. Tárgyaljuk, hogy milyen architekturális változást kell végeznünk ahhoz, hogy javítsuk a válaszidőt. Megmutatom a Node váltási lehetőségeket és becsléseket teszek a hatékonyságukra.

A fejlesztendő szoftver a drón közelébe tervezett, tehát nem a felhőben fog futni, hanem a drónok központi irányításának a kiegészítése. E szerint futtatását azon a gépen célszerű végezni, mely a drónnal is egy hálózaton/környezetben van.

8.1. Felhasznált technológia

Olyan technológiában kell megkezdeni a fejlesztést, amely lehetőséget biztosít egyszerű kezelésére az eddig felállított rendszernek és környezetnek. Szükséges belevinni modularitást és skálázhatóságot, továbbfejlesztési lehetőség érdekében. Tehát a felhasználó számíthat rá, hogy nagyobb számú drón vagy robotvezérlést is támogat a rendszer.

Mivel egy QoS megvalósításáról beszélünk, amelynek paraméterei változhatnak a szükségletek függvényében, ezért az implementáció részeként csak a logikai működést szükséges megvalósítani, az egyes kritériumok változtathatóak. A bevezetőben szó esett különböző jövőbeli automatikus drónfelhasználásnak, melyben volt példa élő szórakoztatóipari fellépésre és mezőgazdasági munkára is. Ezen iparágakban teljesen más QoS feltételeket szabnak meg. Míg egy élő fellépésben nagyon hamar belerondíthat a show-ba pár késleltetésből félreszinkronizált drón, addig a mezőgazdaságban inkább hosszú idejű stabilitásnak kellhet megfelelni. Ezért az algoritmus megalkotásánál paraméterekben fogunk dolgozni, melyeket a felhasználás definiál.

8.1.1. Python

A szakmai előítéleteim alapján már sejtettem, hogy a Python lesz az a nyelv, amelynek a legtöbb támogatása van a natív felhőtechnológiákhoz. A Python egy értelmezett, magas szintű és általános célú programozási nyelv. A Python tervezési filozófiája hangsúlyozza a kód olvashatóságát a jelentős szóköz jelentős felhasználásával. Nyelvi konstrukcióinak és objektum-orientált megközelítésének célja, hogy segítsen a programozóknak világos, logikus kódot írni kis és nagy projektekhez. A Python dinamikusan be van írva és

automatikus garbage-collector-ral működik. Támogatja a több programozási paradigmát, beleértve a strukturált (különösen az eljárási), az objektum-orientált és a funkcionális programozást. A Python-t átfogó szabványos könyvtárának köszönhetően gyakran "csomagokkal együtt" használt nyelvként írják le. [21]

Amely lehetőségek miatt kifejezetten a Python technológiát választottam azok az alábbiak:

- Kubernetes API támogatás
- Docker API támogatás
- Könnyű naplózás, időalapú feljegyzés
- Gyenge objektumorientált támogatás, a modularitás és a tiszta kód miatt
- Egyszerűség, gyengén típusosság

A dolgozathoz mellékelt könyvtár *switchingCenter* nevű könyvtárban található a Python projekt implementáció. A könyvtárban található még egy *requirements* felsorolás, mely a használt Python könyvtárakat sorolja fel azzal a verzióval, mellyel teszteltem is. Ezen kiegészítések telepítése a Pip programmal lehetséges az adott környezetben (8.1. kódrészlet).

```
pip install -r requirements.txt
```

8.1. kódrészlet. Python könyvtárak telepítése

A programhoz tartozó beállításokat egy globális fájlba helyeztem, melyet a Bash és a Python program egyaránt fel tud használni, utóbbi a *ConfigObj* konfiguráció illesztő modullal.

8.1.2. Kubernetes könyvtár alkalmazása

Nyílt forráskódú, stabil verziójú Kubernetes Python kliens létezik, amelyet felhasználtam a fejlesztés során. [17] Mint a kliens főoldalán is látható, Kubernetes API-t széleskörűen támogatja, így az általam felhasznált Kubernetes API technológiákat is.

A jelenlegi környezetben a program felhasználja a Multipass program parancsait a Bash-en keresztül, ami nem túl kifinomult mérnöki kivitelezés, azonban a Multipass-nak még nincs Python kliense, illetve későbbi rendszerek esetén könnyen átfejleszthető másra. A külső program meghívására a *subprocess* modult használtam, mely segítségével kértem le a master VM-ről a K3S tokenet és a master IP-jét. Utána a kapott információkból fel tudtam építeni a Kubernetes autentikációt, megadva a Multipass által generált privát kulcsot is (8.2. kódrészlet).

```
aToken= subprocess.check_output(["multipass", "exec", self.config_parser.get('master'), "--", "kubect\nl describe secret $SECRET_NAME | grep -E '^token' | cut -f2 -d ':' | tr -d ' '\n").decode('utf-8')\nmaster_ip= subprocess.check_output(["multipass", "list", "|", "grep", self.config_parser.get('master\n'), 'grep -oE \\b([0-9]{1,3}\\.){3}[0-9]{1,3}\\b')\n").decode('utf-8')\n\naConfiguration = kubernetes.client.Configuration()\naConfiguration.verify_ssl = False\naConfiguration.host = "https://" + master_ip + ":443"\naConfiguration.api_key = {"authorization": "Bearer " + aToken}\naApiClient = kubernetes.client.ApiClient(aConfiguration)\n\nself.kApi = client.CoreV1Api(aApiClient)
```

8.2. kódrészlet. Kubernetes kliens felépítése

8.1.3. Docker könyvtár alkalmazása

A *Docker SDK for Python* segítségemre volt, hogy a Drónokat elérjem és parancsokat adhassak ki a konténeren belül. Az SDK nagyon egyszerű lehetőséget nyújtott kapcsolódni a környezethez, kifejezetten ha a szoftver is ugyanabban a fut.

Definiáltam egy Drone osztályt, amely példányosításával egy drónhoz kapcsolódik a program, melyet tárol és fenntartja a kapcsolatot. Az inicializálás a drón száma alapján történik, hasonlóan a korábbiakhoz, az azonosítóból kiszámoljuk a portokat, mely kezdőportokat a konfigurációból olvasunk ki (8.3. kódrészlet). A konténert a konvencionális nevezéktanon keresztül ismerjük fel (drone-1, drone-2, ...), mely konténer lekérése után bármilyen Docker parancsot végre tudunk hajtani a *self.drone* elnevezésű kliens segítségével.

```
class Drone():
    def __init__(self, drone_id, node_ip):
        self.config_parser = ConfigObj('../config/config')
        self.drone_id=int(drone_id)
        self.node_ip=node_ip
        self.docker_client = docker.from_env()
        self.mavlink_port = int(self.config_parser.get('MAVLINK_START_PORT'))+drone_id-1

        self.drone= self.docker_client.containers.get("drone-"+str(self.drone_id))
```

8.3. kódrészlet. Drón konténeréhez csatlakozás

8.1.4. Naplózás

Szoftveremben nagyon fontos a naplózás, melyet a *logger* modullal valósítottam meg. A QoS miatt minden egyes tevékenységet, amit a program elvégez nyers időbélyeg formátumban írom ki, hogy méréseket lehessen végezni a tevékenységek között (8.4. kódrészlet).

```
logging.basicConfig(filename='switchCenter.log', level=logging.INFO, format='%(asctime)s %(levelname)s %(message)s')
```

8.4. kódrészlet. Naplózás beállítása

8.2. Kivitelezési lehetőségek és várható kapcsolási idők

Tehát van egy automatizáltan felépített rendszerünk Kubernetes-szel és változtatható mennyiségű szimulált drónnal. És már a szoftver is készen áll, hogy bármely automatizált szabályokat figyelhessünk és bármilyen módosítást végrehajtsunk valós időben a rendszeren. De mik is azok a feltételek, amelyek egy QoS biztosítanak egy ilyen drónokat irányító felhőrendszerben? Definiálom ezeket egy drónra:

- A drón és a Kubernetes közötti válaszidő nem haladhat meg egy előre definiált maximális időt,
- A drón és a Kubernetes közötti sávszélesség nem lehet kisebb egy előre definiált minimális sávszélességnél, ez a sávszélesség szétbontható egy irányításra meghatározott biztonsági és egy videófolyam sávszélességre,
- Ha ezen feltételeknek nem felel meg az adott konstrukció, végezzen változtatást,
- Ha nem létezik olyan változtatást elvégezni, amellyel megfelelnek a feltételek, akkor érjen véget a program és jelezze, hogy véget ért a QoS feltételek melletti üzemelés, mely esetben létrejöhet valamilyen biztonsági procedura, például biztonságos leszállás.

A következőkben megnézzük, hogy mi az a változtatás amivel javítani lehet a kapcsolaton. Jelen rendszerben egy 3 Node-os felhőrendszerről beszélünk, amely lehet sokkal szélesebb is, a K8S határain belül. Az implementációnál figyelembe vettem a változtatható Node mennyiséget. Ha valamilyen kritériumnak nem felel meg a kapcsolat, akkor egyféleképp változtathatjuk a kapcsolatot, áthelyezzük a háttérszolgáltatást másik Node-ra. Ebben a lokális virtuális rendszerben ez nem tűnik megoldásnak, de egy nagyobb ipari klaszter esetén a távolságokból adódóan Node váltással érhetünk el jobb eredményt, biztosíthatjuk rendszerünket túlterheltség és véletlen hiba létrejötte esetén. Tehát akkor azt kell megnéznünk, mely módokon lehet kivitelezni.

8.2.1. Node váltás és új címhirdetés

Első ilyen megközelítés, hogy ha már automatizáltan fel tudjuk építeni a Roscore-t, akkor legyen a változtatás az, hogy új Node-ra deploy-oljuk a Roscore-t és közöljük a drónnal az új elérési címet. Nézzük meg mi ennek a pontos menete:

- Észrevesszük a hibát és reagálunk (t_r)
- Kiválasztunk egy megfelelő Node-ot (t_s)
- Deploy-oljuk a Roscore-t az új Node-on (t_d)
- Új címet adunk meg a drónnak (t_a)

Összesen $t = t_r + t_s + t_d + t_a$ idő. Azt fontos megjegyezni, hogy a t_r reagálási idő konfigurálható vagyis az, hogy milyen időközönként nézzük, amiből tudjuk, hogy a reagálási idő várható értéke az ellenőrzési periódusidő fele lesz.

8.2.2. Node váltás proxy mögött

Nézzünk meg egy másik megközelítést, amikor meg szeretnénk spórolni t_a címadási időt és kihelyezünk egy proxy-t amely megváltoztatja a mögöttes Node-ot miután a Roscore-t deploy-oltuk az új Node-ra. A probléma továbbra is fennáll, hogy ha egy Node-on fut a Roscore, akkor azt kívülről az elérhető portok nagy részén el kell tudnunk érni. A Kubernetes Service API működhetne ilyen proxy-ként, azonban nem fogja biztosítani a többi portot LoadBalancer nélkül. Sima Kubernetes proxy-val azonban elméletben meg lehet oldani (lásd dokumentáció [10]). Azonban ezzel a megoldással létrejönne a proxy kreálás/változtatás ideje, illetve az új proxy-n keresztül hitelesítést is kell végeznünk minden alkalommal.

8.2.3. Széleskörű rendelkezésre állás

A legtöbb időt jelentő t_d deploy időt, ami méréseim szerint 2 és 5 másodperc között van, akkor iktathatjuk ki, hogyha nem csak az aktív Node-on áll rendelkezésre az adott drónnak a Roscore, hanem legalább egy backup Node-on. Ezzel a verzióval ugyan átlagosan több erőforrást foglalunk, azonban optimalizálhatjuk a váltási időt, ami javít a QoS-en.

8.3. Megvalósítás: DaemonSet

Kiderült, hogy a legoptimálisabb QoS megoldása az lehet, ha minden worker Node-on rendelkezésre áll a Roscore minden drón számára. Ezt a DaemonSet Kubernetes API-val lehet megvalósítani.

A DaemonSet biztosítja, hogy az összes (vagy néhány) Node futtassa a Pod másolatát. Amint Node-ok hozzáadódnak a klaszterhez, a Podok automatikusan létrejönnek az új Node-okon. Amint a Node-okat eltávolítják, ezeket a Podokat a Kubernetes Garbage-Collector gyűjti össze. A DaemonSet törlése megtisztítja az általa létrehozott Podokat. Egyszerű esetben minden démontípushoz egy, minden csomópontot lefedő DaemonSet-et használnak. A bonyolultabb beállítás több DaemonSet-et is használhat egyetlen démon-típushoz, de különböző jelzőkkel és / vagy különböző memória- és CPU-kérésekkel a különböző hardvertípusokhoz. [12]

A Kubernetes Service API helyett használom a megvalósítandó ötletben a DaemonSet-et a következőképpen (8.5. kódrészlet). A konténer indítás és a változók ugyanazok maradtak, mint a Service-es megvalósításban, az *affinity* résszel egészítettem ki a yaml-t. Itt *hostname* alapján szűröm ki a Node-okat, azokon a Node-okon fog elindulni a Roscore, melyek nem tartalmazzák a master nevet.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: roscore-$(DRONE_IDENTIFIER)
spec:
  selector:
    matchLabels:
      app: roscore-$(DRONE_IDENTIFIER)
  template:
    metadata:
      labels:
        app: roscore-$(DRONE_IDENTIFIER)
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchFields:
                  - key: metadata.name
                    operator: NotIn
                    values:
                      - master
      hostname: roscore-$(DRONE_IDENTIFIER)
      containers:
        - name: roscore-$(DRONE_IDENTIFIER)
          image: alpineros/alpine-ros:noetic-ros-core
          env:
            - name: ROS_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
            - name: ROS_MASTER_URI
              value: "http://$(ROS_IP):$(MAVLINK_PORT)"
          args: ["roscore", "-p", "$(MAVLINK_PORT)"]
      hostNetwork: true
```

8.5. kódrészlet. DaemonSet megvalósítás, Roscore minden worker-en

8.4. Mérési adatok kinyerése

A rendszer automatikus működéséhez szükséges megtudnunk az aktuális sávszélességet és válaszidőt a drón és a node között. Mivel a szoftver a drón közelében fut, ezért a jelenlegi szimulált rendszerhez implementálom, azonban ezek a függvények változtathatóak lesznek. A program a konténerből tud kiadni alap Linuxos parancsokat, így a sávszélességet iperf programmal (8.6. kódrészlet), a válaszidőt ping programmal (8.7. kódrészlet) nyertem ki. Ha nem sikerülne elvégezni a program futását, akkor visszatér egy magas értékkel, melyet

lemér mégegyszer a váltás előtt. Ping esetén eldönthetjük, a függvény paraméterében, hogy hány ICMP packet átlagát szeretnénk venni.

```
def getBW(self, node_ip):
    iperf="iperf3 -f m -c "+node_ip
    (exit_code,output)= self.drone.exec_run(iperf)
    outputStr = output.decode('utf-8')
    try:
        return float(outputStr.split('\n')[-1].split(' ')[4])
    except:
        return 10000
```

8.6. kódrészlet. Sáv szélesség megállapítása a konténer és a node között

```
def getConnectionDelay(self, node_ip, n=1):
    pingCommand="ping "+node_ip+" -c "+str(n)
    (exit_code,output)= self.drone.exec_run(pingCommand)
    outputStr = output.decode('utf-8')
    for line in outputStr.split('\n'):
        if 'min/avg/max/mdev' in line:
            avg=line.split('/')[5]
            return float(avg)
    else:
        return 10000
```

8.7. kódrészlet. Késleltetés megállapítása a konténer és a node között

9. fejezet

Távoli Robotvezérlés Optimalizálása

Ebben a fejezetben a távoli robotvezérléshez használt algoritmust ismertetem. Az algoritmus célja, hogy a drónok vezérléséhez biztosítsa a QoS feltételeit, amit jelen esetben két paraméterben határozunk meg: sávszélesség és válaszidő. Ennek a két paraméternek a kontrollálásával biztosítható a videó folyam minősége a drón és a Kubernetes Node-ok között. A Kubernetes klaszter worker node-jai (rendelkezésre álló szerverek) erőforrás készletének függvényében a videófeldolgozás és a drón irányítása áthelyezhető (váltható, migrálható) az optimális teljesítmény érdekében. Az algoritmus rendszeresen méri az elérhető erőforrásokat, proaktívan vált a szolgáltatás zökkenőmentes fenntartása érdekében.

Tehát van valamennyi Node-unk amik között a drónok vezérlését áthelyezhetjük, az algoritmus úgy vált időközönként, hogy a késleltetést optimalizálja és biztosítja a sávszélességet is. Ha a sávszélességet nem lehet biztosítani, akkor egy ALARM állapotba fut, ekkor a drón biztonságosan leszáll és azonnal értesíti a felhasználót.

Az algoritmus bemenete tehát a Node-ok IP címének halmaza, amin keresztül kapcsolódik a drón a központhoz, a minimum sávszélesség a videó folyamra felé (vbw), a minimum sávszélesség a kontroll kommunikációra (cbw), ellenőrzések között eltelt időintervallum (T) és a jobb válaszidő esetén az az arány ami esetén vált (R). Az algoritmus a teljes üzemidő alatt fut T időközönként, a kimenete pedig a videó folyam vezérlése és a virtuális funkciók lokációjának a meghatározása, hálózati erőforrás vezérlése.

```
ALARM := false
while(!ALARM):
    migrationTrigger := false
    actLat = checkActualLat()
    (minLat, minNode) = MIN{checkAllNodeLat()}
    actBW = checkActualBW()
    moreBandwith = isThereNodeWithBetterBW()
    IF (minLat / actLat > R AND morBandWith):
        migrationTrigger = true
        nextNode = minNode
    criticalState = IF (cbw + vbw > actBW)
    IF (migrationTrigger OR criticalState):
        bestNode = findBestNode()
        bBW = checkActualBW(bestNode)
        IF (bBW < cbw + vbw):
            ALARM = true
        ELSE:
            swithTo(bestNode)
    sleep(T)
```

9.1. kódrészlet. Az optimalizáló algoritmus

Az algoritmusban használt migrationTrigger boolean változó-t igazra áll, hogyha a az algoritmus váltana egy jobb node-ra, az ALARM változó pedig igazra áll, ha baj van és le kell állítani biztonságosan a drónt. A minLat változóba tárolom a késleltetés minimumát, az actLat-ba pedig az aktuális node késleltetését. Hasonlóan az actBW az aktuális sávszélesség, a bestNode pedig az a node amelyikre vált az algoritmus.

Az algoritmust a Python alapú kapcsolóközpontba implementáltam, amely az ALARM módot a program befejeztével visszatérési értékként jelzi. Tehát a programot olyan integrált környezetben kell használni, amely kezeli az ALARM állapotot és jelez valamit a drónnak, például a biztonságos leállást.

10. fejezet

Mérések

Ebben a fejezetben az elvégzett mérésekről lesz szó, illetve azokból levont konklúziókról. Megnézzük, hogy a dolgozatban elkészített szoftverrendszer milyen hatékonysággal működik, milyen feltételekkel és minek milyen hatása van a QoS paraméterek elmozdulására. Kitekintünk arra, hogy egy 5G-s közeghozzáférési technológia esetén milyen változás jelenne meg a kommunikációban, elgondolkodunk rajta, hogy ez milyen előnyöket jelenthet. A méréseket az elkészített virtuális Multipass alapú K3S rendszeren végeztem, a mérési architektúra a korábbi 7.4. ábrán látható. A méréseket úgy végeztem, ahogyan a kapcsolóállomásba is implementáltam a valós idejű méréseket, annyi különbséggel, hogy nagyobb adatot átlagoltam a pontosság érdekében.

Két fontos dologra vagyunk kíváncsiak:

- A kapcsolóközpont viselkedése, meghibásodott Node esetén
- A kapcsolóközpont viselkedése nagy késleltetés esetén
- A drónok számának növekedése esetén hogyan alakul a válaszidő és a sávszélesség
- Flotta irányítása esetén a válaszidő és a flotta méretének kapcsolata

10.1. Node váltás ideje meghibásodott Node esetén

A drónkapcsoló állomás hiba esetén átvált egy backup node-ra, ahol ugyanaz a munka folytatódik, amit a drón megkezdett. A kapcsolóállomásba implementált *timestamp* alapú naplózórendszer segített abban, hogy pontos idők kapjak a reakcióról és a helyreállásról. A mérés menete egy drón normál üzemben működése közben:

1. Hibát idézünk elő, kikapcsoljuk azt a Node-ot amelyen fut a Roscore és rögzítjük az időt (10.1. kódrészlet) (t_0)
2. Megvárjuk amíg a kapcsolóállomás fel nem ismeri a hibát. (t_1)
3. Megvárjuk amíg talál egy új Node-ot és helyreáll az üzem. (t_2)

```
date +%s && multipass stop worker-1
```

10.1. kódrészlet. Node kikapcsolása

$t_2 - t_1$ helyreállási időre minimálisan 0.8, maximálisan 2.18, átlagosan 1.15 másodpercet mértem 10 mintavételezésből. $t_1 - t_0$ észrevételi időt több periódusidővel is kipróbáltam, mindegyikkel öt-öt mérést átlagolva.

- 5s ellenőrzési periódusidő esetén: 2.3s
- 10s ellenőrzési periódusidő esetén: 6.1s
- 15s ellenőrzési periódusidő esetén: 8.5s

Ez azt jelenti, hogy a jelentős késleltetést az ellenőrzések periódusából adódó késleltetés adja, amelynek periódusa alatt egyenesen oszlik el a várható keletkező hiba. Így annak beállítását a felhasználó mérlegelheti, hogy mennyire kritikus a drón munkája és mennyire lehet terhelni a hálózatot. Észrevétel után a kapcsolási idő már csak pár másodperc. Ha például 5s-os hibajavítást szeretnénk, akkor 5s-ra kell állítani az ellenőrzési periódusidőt. Fontos megjegyezni, hogy ezen kiesési idő alatt a drónnak kell valamilyen belső mechanizmussal rendelkeznie, amely tovább viszi a feladatot vagy hagyja biztonságosan lebegni a következő utasításig, amennyiben nem érkezik válasz egy bizonyos időkereten belül. A piacon található középkeletkező drónok rendelkeznek ilyen viselkedéssel, ipari használatban pedig fontos, hogy ilyen kritériummal ruházzanak be drónra.

10.2. Válaszidő különbség esetén a jobb Node kiválasztása

A drónkapcsoló állomás nem csak akkor vált át másik Node-ra ha az aktuálisan felhasznált worker Node kiesik, hanem akkor is ha egy bizonyos p paraméterrel alacsonyabb a válaszidő az adott Node felől, a legalacsonyabb elérhető válaszidőhöz képest. Ennek váltási algoritmusát részletesebben a 9. fejezetben részleteztem. A mérést hasonlóan végeztem, mint az előzőt, azonban mással implikáltam a változást, nem megszüntettem működés közben a Node-ot, hanem a Node belső interfészéhez hozzáadtam egy 0.5s-os késleltetést, hogy kívülről elérve a virtuális rendszerben ennyi késleltetés adódjon hozzá mindenféle kommunikációhoz (10.2. kódrészlet). Ennek rögzítettem az idejét és vártam az algoritmus ellenőrző mechanizmusát és a Node váltás bekövetkeztét.

```
tc qdisc add dev ens4 root netem delay 500ms
```

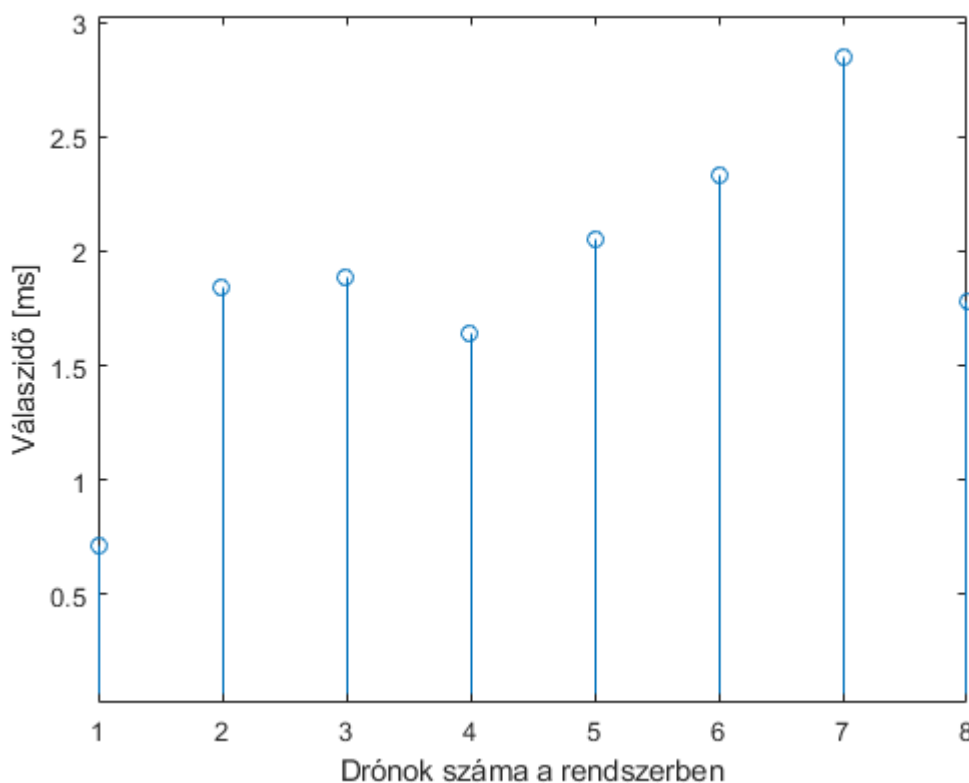
10.2. kódrészlet. Késleltetés hozzáadása a worker Node külső interfészéhez alapértelmezett Linux programmal

Megmértem az algoritmus szerinti váltást, amikor a 8.4 alfejezetben leírt QoS becslési eljárással 5 másodperces időközönként ellenőriztem a késleltetési időket. A méréseim alapján tíz mintavételből a teljes interfész lassítás után minimum 3.2, maximum 6.91, átlagosan pedig 4.2 másodperc alatt történt meg a Node váltás. Megítélésem szerint egy átlagos drónvezérlési feladatnál (ami meghatározza a kapcsolódási pontok változását) és átlagos hálózati dinamika esetén (egy jól üzemeltetett hálózatban az elérhető szerverek felé a késleltetés nem változik jelentős mértékben) ez megfelelően gyors váltás. Amennyiben valamilyen egyéb ok miatt gyorsabb váltás kell, azt a QoS megfigyelés gyakoriságának növelésével lehet elérni.

10.3. Terheltségi válaszidő viszonya

Következőnek azt néztem meg, hogy a virtuális rendszerben hogyan terhelődik a klaszter az irányítandó drónok számának növekedvén. Egy drónhoz tartozik három konténer a klaszterben, abból a Roscore-ból kettő van, mindkét worker Node-on, így összesen négy konténer tartozik egy drónhoz. A méréseket egy kitüntetett drón és hozzá tartozó Roscore között végeztem, akkor is amikor több drón volt a rendszerben.

A válaszidőt 25 ICMP ping visszaérkezési idők átlagaként számoltam ki, úgy hogy a drón virtuális konténeréből küldtem a Roscore konténer interfészére. Az eredményen az látszik,



10.1. ábra. Válaszidő a drónok számának függvényében

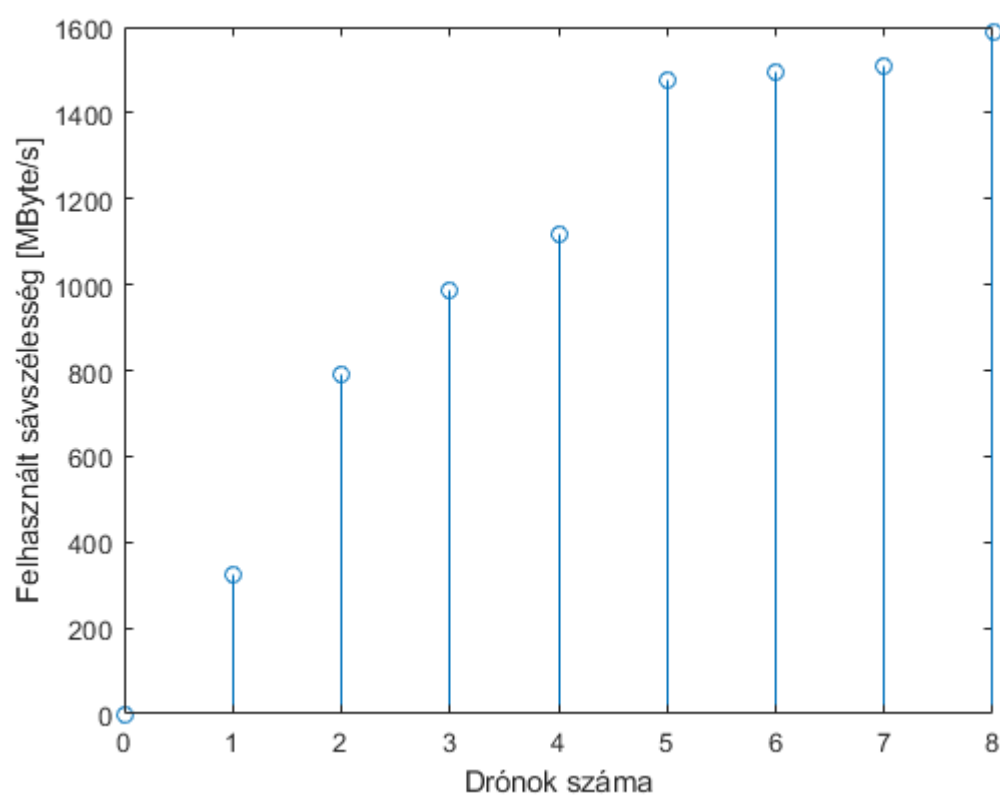
hogyan nő az üzemeltetett konténerek függvényében a válaszidő, nem drasztikusan, azonban valamilyen lineáris görbére illeszthetően (10.1. ábra).

A sávszélességet *iperf3* programmal néztem, szintén egy drón konténer és a hozzá tartozó Roscore között. A sávszélesség csökkenése is észrevehető, ugyan nincs mögötte az a linearitás, mint a válaszidő növekedésében, egy drónhoz képest megfelelő a rendelkezésre álló kiszolgáltató sávszélesség a klaszter felől. Az *iperf3* program a rendelkezésre álló szabad sávszélességet mutatja meg, melyet teljes terhelés alatt mér meg egy rövid időablakban. Alaphelyzetben, egyetlen drón kapcsolatba lépése nélkül a két konténer között $f_0 = 2430 MB/s$ -ot mértem a hálózat terhelésével. Ez azt jelenti, hogy drón kommunikáció nélkül ennyivel terhelhető a hálózat a két konténer között, amelyet a hardver és a virtualizáció határoz meg. Ez f_0 kezdeti értékre és kiszámított x függvény pontjaira illeszthető görbe a GeoAlgebra függvényillesztő segítségével

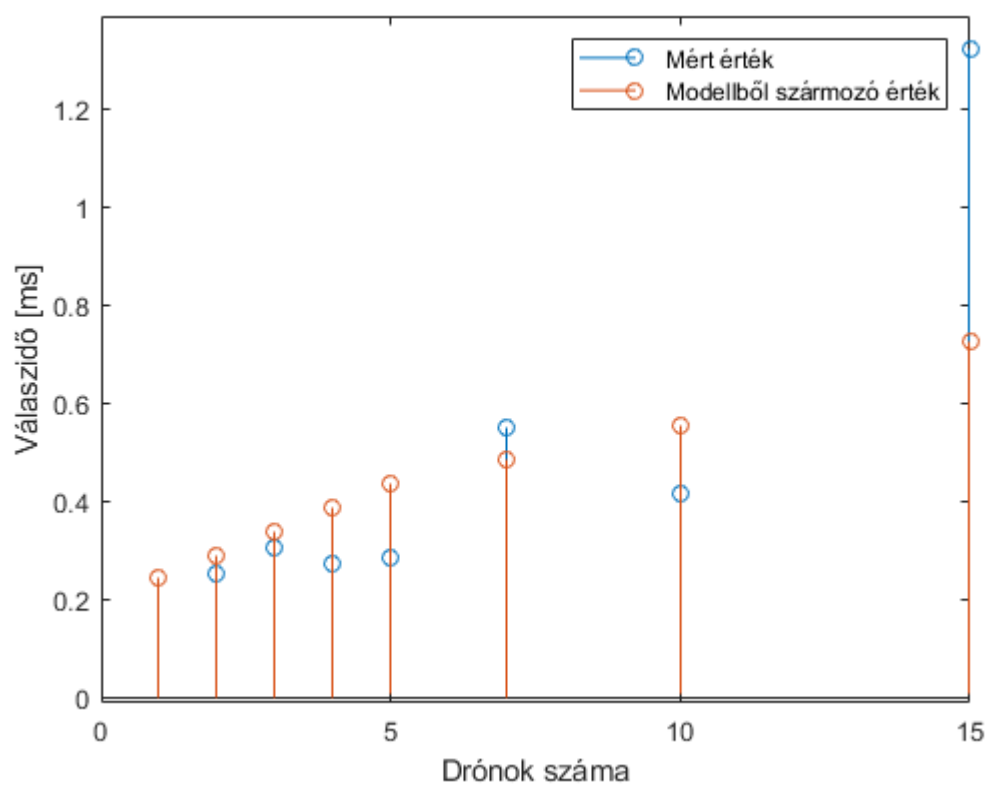
$$x(n) = 625 \cdot \ln(7.28n) [MB/s]$$

eredmény jött ki, ahol n a rendszerben lévő drónok száma. Egy felhasználási ábrát rajzoltattam ki, amely azt mutatja, hogy a kommunikáció nélküli elérhető sávszélességhez képest mennyi sávszélességet használnak a drónok (10.2. ábra), tehát:

$$x_n = |f_n - f_0|$$



10.2. ábra. Felhasznált sávszélesség a drónok számának függvényében



10.3. ábra. Kiszolgálási válaszidő a drónflotta méretének függvényében

Az 5 fejezetben vizsgáltam a matematikai modelljét egy M/M/1 kiszolgálási Erlang rendszernek. A diplomatervezés folyamatában a ROS rendszernek a megismerése közben kiderült, hogy a jelenlegi irányító szoftver arra nincs felkészítve, hogy egy Roscore-on keresztül irányítson több drónt, illetve az általam megalkotott végső Kubernetes rendszerben minden drónnak külön Roscore-ja van. Azért megnézzük, hogy mi lenne akkor ha lenne egy ROS-on keresztül N résztvevőjű flottát irányító szoftverünk, amely egy Roscore-on keresztül vezérelné a flottát. A mérést egy drón konténere és az egyetlen Roscore között végeztem a felhőben. A tiszta mérés kedvéért nem deploy-oltam az irányításra és kamerafeldolgozásra szolgáló konténereket. Egy és tizenöt drón között mértem a válaszidőt a ping programmal, 25 visszapattanást átlagolva. E mellé kimértem tcpdump-al egy félperces időablakban a drón és üzenetváltását a Roscore-al, amiből kijött, hogy $n=1$ -re $\mu = 46.73 krs/s$. Amiből kijön a kiszolgálási idő:

$$\lambda = \mu - \frac{1}{D_1} = 46.73 - \frac{1}{0.241} = 42.58 \frac{1}{s}$$

Kiszámítottam a kiszolgálási időből és a kérési intenzitásból a késleltetés többi értékét a modell alapján a D_1 értékre vetítve és a mért értékek mellé vetettem (10.3. ábra). Jól látszik, hogy növekszik a drónok növelésével az egy drón felé lehetséges kiszolgálási idő. Tehát növekszik az 5. fejezetben megmutatott μ érték, amely fordítottan arányos a késleltetéssel.

10.4. Késleltetés hozadéka 5G közeghozzáférés esetén

A mérésekben megnéztük, hogy milyen késleltetések és sáv szélességek vannak egy teljesen virtualizált rendszerben. Egy valós hálózaton és fizikai drónnal működő rendszerrel még hozzájön két rendszerből származó érték ezekhez a késleltetésekhez és sáv szélességekhez. Értelemszerűen a késleltetéshez hozzáadódik, a sáv szélességnél pedig az értékek minimumára csökken.

A hálózat késleltetése és a közeghozzáférési technológia késleltetése adódik hozzá a teljes válaszidőhöz. A tanszéken kipróbált Wifi-s közeghozzáférés körülbelül $300ms$ -ot növel a szimulációhoz képest. Az 5G $5-15ms$ körüli értéket is biztosíthat, így ezzel a technológiával minimalizálni lehet a teljes késleltetést az előző pontban kiszámoltakhoz konvergálva.

Összegzés

A dolgozatban megnéztem mire hasznának ma tömeges robotos irányítást, specifikusabban kitekintettünk, hogy mely iparágakban használhatóak a drónok, mire és hogyan használják. Megnéztem mire jó a konténerizáció, elmerültünk a konténer alapú felhőrendszerek világában, összehasonlítottuk a Docker Swarm-ot, a Mesos-t és a Kuberneteset. Átnéztük a robot- és drónirányítással kapcsolatos szoftvereket, hogy mik a lehetőségek, hogyan és minek szükséges jól együttműködni egy fizikai vagy egy szimulált drón irányításához. Megnéztem, hogyan lehet nagymennyiségű drónt szimulálni. Megnézhettünk két tesztet, konténerekkel való drónirányítás és jelfeldolgozás tesztjét és különböző VM-ekről drónszimulálás tesztet. Továbbá megnéztem, hogyan tudjuk megbecsülni nagyszámú drónkiszolgálásnak a késleltetését. Elkészítettem egy automatikusan felépülő virtuális Kubernetes rendszert. Ezen rendszeren teszteltem a Kubernetes API széles palettáján több drónvezérlési architektúrát. Definiáltam egy QoS feltételrendszert, majd kiválasztottam és módosítottam azt az architektúrát amely ennek a legjobban megfelel. Megterveztem és implementáltam egy kiegészítőszoftvert, amely folyamatosan figyeli a kapcsolatot a drón és a felhőrendszer között, amennyiben tud javítani a kapcsolaton ezt megteszi, ha kritikus állapotba lép és nem tud javítani, akkor pedig figyelmezteti a felhasználót. Implementáltam egy algoritmust, amely a távoli drónvezérlés során szükséges videó folyamatok és vezérlési adatok együttes minőségbiztosítását végzi. Méréseket végeztem a különböző részfeladatokról és terheltségről.

Ábrák jegyzéke

1.1.	Kritikus mérföldkövek folyamata [6]	8
1.2.	A drónok és felhő kapcsolatának durva modellje	10
1.3.	Ipar 4.0 komponensei [1]	11
1.4.	Amazon házhozszállítás problémája drónnal [24]	11
1.5.	Lady Gaga 300 drónnal a háttérben a 2017-es Super Bowl döntőjének fél- ideje alatti koncerten [8]	12
2.1.	Docker Swarm architektúra [16]	17
2.2.	Kubernetes architektúrája [11]	18
3.1.	Yuneec Mantis Q [25]	20
3.2.	PX4 kommunikációja Mavlink protokollal [19]	21
3.3.	Gazebo kamerás drón modell	22
3.4.	Az együttes működés kommunikációja Mavros node-on keresztül	24
4.1.	Kész VKE földi rendszer	26
4.2.	Négy azonos konténerrel drónirányítás	27
4.3.	Két VM-en több drón irányítása	28
5.1.	Több robot kiszolgálásának felhőből a durva modellje	30
6.1.	Registry megoldásaim egyedi konténerek esetén	34
7.1.	Ingress Service szétválasztása a klaszterben [23]	39
7.2.	ROS Node csatlakozása Roscore-hoz [18]	40
7.3.	LoadBalancer megoldás Kubernetes felhőben	41
7.4.	Teljes környezet két szimulációval	44
10.1.	Válaszidő a drónok számának függvényében	55
10.2.	Felhasznált sávszélesség a drónok számának függvényében	56
10.3.	Kiszolgálási válaszidő a drónflotta méretének függvényében	57

Kódrészletek jegyzéke

2.1. Példa két WordPress szolgáltatás párhuzamos indítására a 80 és 81-es portokon	15
2.2. Példa volume csatolásához	15
2.3. Példa alap robotoperációsrendszer konténerizációjához	16
2.4. Példa 100 alkalmazás indítására	17
2.5. Példa alkalmazás skálázására	17
3.1. 10 optikai adatfolyamos drón szimulálása Gazebo-val	22
3.2. Lokális PX4 szimuláció ROS-on keresztül Mavlink-el összekötve	23
4.1. Azonos konténerek indítása négy különböző feladattal és az X szerveren való kommunikációt megvalósítva	26
6.1. Multipass VM-ek létrehozása	32
6.2. K3S Master inicializálása	33
6.3. K3S Slave-ek inicializálása	33
6.4. Node-ok lekérdezése	33
6.5. Docker registry inicializálás a master docker környezetben	34
6.6. Build és push lokál konténer registry-be	34
6.7. Docker Hub build és push	34
6.8. Docker Hub autentikáció K3S-ről	34
6.9. K8S API secret definiálása a konténerekhez	35
7.1. Példa egy Pod-ra a négyből	36
7.2. Példa 4 konténeres deployment megoldásra	37
7.3. Példa 4 konténeres megoldás service kivezetésére	38
7.4. Service kiegészítése NodePort-okkal	38
7.5. Linuxon kiosztható port korlátozása	40
7.6. Roscore Service specifikációja inkrementális változókkal	41
7.7. Traefik hozzáadása a klaszterhez	41
7.8. Roscore információk összegyűjtése és deploy	43
7.9. Drónok indítása külső Docker környezetben	43
8.1. Python könyvtárak telepítése	46
8.2. Kubernetes kliens felépítése	46
8.3. Drón konténeréhez csatlakozás	47
8.4. Naplózás beállítása	47
8.5. DaemonSet megvalósítás, Roscore minden worker-en	49
8.6. Sávzsélesség megállapítása a konténer és a node között	50
8.7. Késleltetés megállapítása a konténer és a node között	50
9.1. Az optimalizáló algoritmus	51
10.1. Node kikapcsolása	53
10.2. Késleltetés hozzáadása a worker Node külső interfészhöz alapértelmezett Linux programmal	54

Irodalomjegyzék

- [1] Aethon: Industry 4.0 components (2020. május 23.). <https://aethon.com/mobile-robots-and-industry4-0/>.
- [2] Andy Jeffries: What's the difference between k3s vs k8s? (2020. december 6.). <https://www.civo.com/blog/k8s-vs-k3s>.
- [3] Cyprien Lecallier: Minikube, kubeadm, kind, k3s, how to get started on kubernetes? (2020. december 6.). <https://www.padok.fr/en/blog/minikube-kubeadm-kind-k3s>.
- [4] Docker docs: Overview of docker compose (2020. május 24.). <https://docs.docker.com/compose/>.
- [5] Frank Tobe: Lady gaga, 300 intel drones, and the super bowl (2020. május 23.). <https://www.therobotreport.com/lady-gaga-300-intel-drones-and-the-super-bowl/>.
- [6] Free Management Books: Project execution process (2020. május 23.). <http://www.free-management-ebooks.com/faqpm/processes-04.htm>.
- [7] Sami Salama Hussen Hajjaj – Khairul Saleh Mohamed Sahari: Establishing remote networks for ros applications via port forwarding: A detailed tutorial. 2017. 05. 1-13. sz.
- [8] Jacob Brogen: How intel lit up the super bowl with drones—and why (2020. május 23.). <https://slate.com/technology/2017/02/how-the-intel-drones-at-the-lady-gaga-super-bowl-halftime-show-worked.html>.
- [9] Jaeyoung Lim: Multi-vehicle drone simulation in gazebo (2020. május 27.). <https://auterion.com/multi-vehicle-drone-simulation-in-gazebo/>.
- [10] Kubernetes: Accessing clusters (2020. december 13.). <https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/>.
- [11] Kubernetes: Concepts underlying the cloud controller manager (2020. május 25.). <https://v1-17.docs.kubernetes.io/docs/concepts/architecture/cloud-controller/>.
- [12] Kubernetes: Daemonset (2020. december 13.). <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>.
- [13] Kubernetes Documentation: Service (2020. november 24.). <https://kubernetes.io/docs/concepts/services-networking/service/>.

- [14] Dr. Györfi László – Györi Sándor – Dr. Pintér Márta: *Tömegkiszolgálás*. 2002, Műegyetemi Kiadó.
- [15] Mavlink: Mavlink developer guide (2020. május 25.). <https://mavlink.io/en/>.
- [16] Mohamed Fawzy: Create cluster using docker swarm (2020. május 24.). <https://medium.com/tech-tajawal/create-cluster-using-docker-swarm-94d7b2a10c43>.
- [17] Opensource community: Kubernetes python client github (2020. december 13.). <https://github.com/kubernetes-client/python>.
- [18] Hanumant Singh Pushyami Kaveti: Ros rescue : Fault tolerance system for robot operating system. 2019. 10. 5. sz.
- [19] PX4: Ros with gazebo simulation (2020. május 26.). <https://dev.px4.io/v1.9.0/en/simulation/>.
- [20] PX4 developers: Open source autopilot (2020. május 26.). <https://px4.io/>.
- [21] Python: Python about (2020. december 13.). <https://www.python.org/about/>.
- [22] ROS: Wiki (2020. május 25.). <http://wiki.ros.org/>.
- [23] Sandeep Dinesh: Kubernetes nodeport vs loadbalancer vs ingress? when should i use what? (2020. november 24.). <https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0>
- [24] Verband für unbemannte Luftfahrt: Amazon's new delivery drone has 'fail-safe logic (2020. május 23.). <https://www.uavdach.org/?p=1294813>.
- [25] Yuneec: Mantis q (2020. május 25.). https://www.yuneec.com/en_GB/camera-drones/mantis-q/overview.html.