

SZAKDOLGOZAT



MISKOLCI EGYETEM

A drónokkal történő csomagszállítás adatkezelési problémáinak vizsgálata

Készítette:

Bajusz Máté Bence

Programtervező informatikus

Témavezető:

Piller Imre

MISKOLC, 2021

SZAKDOLGOZAT FELADAT

Bajusz Máté Bence(BMA015) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Adatmodellek, szimuláció, Go

A szakdolgozat címe:

A drónokkal történő csomagszállítás adatkezelési problémáinak vizsgálata

A feladat részletezése:

A robotika fejlődésének és az online vásárlás egyre népszerűbbé válásának köszönhetően már kísérleti stádiumban vannak az olyan rendszerek, amelyek drónok segítségével oldják meg a csomagok kiszállítását. A dolgozat ezen problémakört vizsgálja az adatok továbbítása, kezelése és tárolása szempontjából.

Mivel ezen rendszerek még kialakulóban vannak, ezért a dolgozat szimulációk segítségével mutatja be, hogy az ilyen típusú logisztikai rendszerben hol keletkeznek adatok, azok milyen csatornákon kerülnek továbbításra és milyen mögöttes adatmodellekkel lehet ezek perzisztens tárolásához használni. Elemzésre kerül a hibamentes és a sorrendhelyes adatátvitel kérdésköre, illetve a dolgozat összehasonlítja az elterjedt adatátviteli formátumokat.

A szimulációk Go programozási nyelven készülnek. Az adatmodellek vizsgálata esetében egyaránt bemutatásra kerülnek a relációs és a NoSQL adatmodellek és adatbáziskezelő rendszer implementációik.

Témavezető: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje: 2020. szeptember 28.

.....
szakfelelős

EREDETISÉGI NYILATKOZAT

Alulírott **Bajusz Máté Bence**; Neptun-kód: BMA015 a Miskolci Egyetem Gépész-mérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *A drónokkal történő csomagszállítás adatkezelési problémáinak vizsgálata* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc,évhónap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

.....

dátum

témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás):

.....
.....
.....

konzulens (dátum, aláírás):

.....
.....
.....

3. A szakdolgozat beadható:

.....

dátum

témavezető(k)

4. A szakdolgozat szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....
..... egyéb mellékletet (részletezve)

.....

tartalmaz.

.....

dátum

témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíró neve:

.....

dátum

szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata:

a bíró javaslata:

a szakdolgozat végleges eredménye:

Miskolc,

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. Bevezetés	1
2. Koncepció	2
2.1. Hasonló rendszerek, megoldások áttekintése	2
3. Szállítási probléma	6
3.1. A szállítási probléma absztrakt modellje	6
3.1.1. Konkrét példa optimalizálásra	7
3.2. A szállítási probléma egy konkrét példája	8
3.3. A szállításhoz kapcsolódó számítási problémák	9
3.3.1. Descartes szerinti	9
3.3.2. Az Ortodroma számítás, a föld alakját figyelembe véve	10
4. Eszközketteszlet és tervezés	12
4.1. Rendszer követelmények	12
4.1.1. Általános leírás	12
4.1.2. Hatékonyság	12
4.1.3. Funkcionális követelmények	12
4.2. A Go nyelv áttekintése	13
4.3. Docker és a mikroszervízek	16
4.4. Felhasznált eszközök	18
4.4.1. Architektúra, rendszer felépítése	18
4.4.2. A Hexagonal architektúra és összehasonlítás a hagyma architektúrával	20
4.4.3. DDD tervezési elv	21
4.4.4. REST	23
4.4.5. HTTP/2 gRPC összehasonlítása HTTP/1.1 JSON üzenetváltással	24
4.4.6. RPC	25
4.4.7. gRPC	25
4.4.8. Adatbázisok	26
4.4.9. Protokollok és adatbázisok szerepe a programban	27
5. Szimuláció	28
5.1. Adatbázisok és modellek	28
5.1.1. Modellek az adatközpont programban	28
5.1.2. Relációs adatbázis, PostgreSQL	30
5.1.3. Relációs modell (5.1. ábra)	30
5.1.4. Dokumentum alapú adatbázis, MongoDB	32
5.2. Szimuláció működésének folyamata	34

5.3.	A rendszer felépítése a követelmények alapján	34
5.4.	Az rendszer működése	40
5.5.	Telepítés, tesztelés	41
5.6.	Szállítási probléma a programban	43
5.7.	Mérések	45
5.7.1.	JMeter	46
5.7.2.	ghz	47
6.	Összefoglalás	51
Irodalomjegyzék		52

1. fejezet

Bevezetés

Manapság a vásárlások egyre nagyobb része történik online, a termékeket pedig házhoz szállítják futárral. A robotika fejlődése miatt, mint ahogy sok más egyszerű munka, a csomagszállítás is automatizálódni fog, melyre drónok is bevethetők. Egy ilyen rendszer nagyon sok adatot generál, amit fel kell dolgozni valós időben és eltárolni. Az elmúlt évtizedben olyan ekommersz óriások, mint az Amazon már próbálkoztak több megoldással drón szállítás terén, így a közeljövőben elképzelhető, hogy egy nagyon valós probléma lesz a drónokat irányító, ellenőrző rendszer működtetése. Egy olyan rendszert szeretnék bemutatni, ami gyors, hatékony, képes konkurrens működésre elosztott rendszerként a felhőben, valós időben tudja szimulálni a drónok működését, telemetria adatainak feldolgozását. A rendszer több alkalmazásból épül fel. Az alkalmazások Go (Golang) nyelven készültek és Docker konténerekben futnak. A rendszer modulárisan épül fel, így különböző komponenseket kicserélhetjük, hogy összehasonlíthassuk a rendszer hatékonyságát a különböző konfigurációkkal. Összehasonlításra kerül relációs és dokumentum alapú adatbázisok teljesítménye, valamint olyan internetes kommunikációra használt protokollok és formátumok hatékonysága, mint a HTTP/2 gRPC, HTTP/1.1 JSON. Olyan érdekességekről is lesz szó, mint hogy a Föld felszínén 2 pont között nem az egyenes a legrövidebb út.

2. fejezet

Koncepció

2.1. Hasonló rendszerek, megoldások áttekintése

Az Amazonnak már működő szállítási rendszere van, mely drónnal 30 percen belül eljuttatja a 2,3 kg-nál kisebb tömegű csomagokat az ügyfelekhez. Ezek a drónok csak 24 km-es hatótávval rendelkeznek, így csak akkor biztonságos egy kézbesítés, ha a célállomás maximum 12 km-re van. A csomag maximum térfogatáról nincsenek pontos adatok. Ezt a rendszert *Prime Air*-nek hívják [17]. Az Amazon Prime Air egy drónja látható a 2.1 ábrán.



2.1. ábra. Amazon Prime Air drón

Először 2016-ban próbálták ki, azóta még nem sikerült bevezetni különböző jogi korlátozások miatt. Olyan jogi elvárásoknak [20] kell megfelelni, mint például az alábbiak.

- Muszáj egy drón pilótának ellenőrizni a repülést, szükség esetén kötelező közbeavatkoznia.
- Egy ilyen pilóta csak 1 drónt felügyelhet egyszerre, kivéve ha rendelkezik több felügyeletéhez megfelelő engedélyel.
- A drónokat nem lehet közvetlen egy személy vagy gépjármű felett működtetni.
- Problémát jelent továbbá, hogy nem minden légtér megfelelő a drónokkal való szállításra. A reptér mellett fekvő nagyvárosokban például biztos hogy nem lehet drónokkal szállítani.

- A csomagot szállító drónnak is kell rendelkeznie engedélyel, hogy alkalmas kereskedelmi szállításra.

Az FAA 2020-ban engedélyezte az Amazonnak a drónokkal való csomagszállítást, de egyelőre csak tesztelik a technológiát pár városban. A jövőben teljesen elköpzelhető, hogy a csomagok felét drónok szállítják ki. Önmagában az Amazon naponta átlagosan 7 millió csomagot szállít ki, és bevallásuk szerint a csomagok 86%-a drónnal szállítható lenne.

Az Amazon felhőszolgáltásai között is található olyan szolgáltatás, amellyel könnyedén kezelhetjük az IoT eszközöket, jelen esetben a drónokat. Ilyen az *IoT Core* vagy az *IoT Things Graph* amin grafikus felületen összeköthetjük és beállíthatjuk az eszközeink közötti kommunikációt. A DHL is próbálkozott hasonló csomagkézbesítő megoldásokkal. Ók egy úgynevezett *Parcelcopter*-el szállították a csomagokat (2.2. ábra).



2.2. ábra. DHL Parcelcopter V4.0 [10]

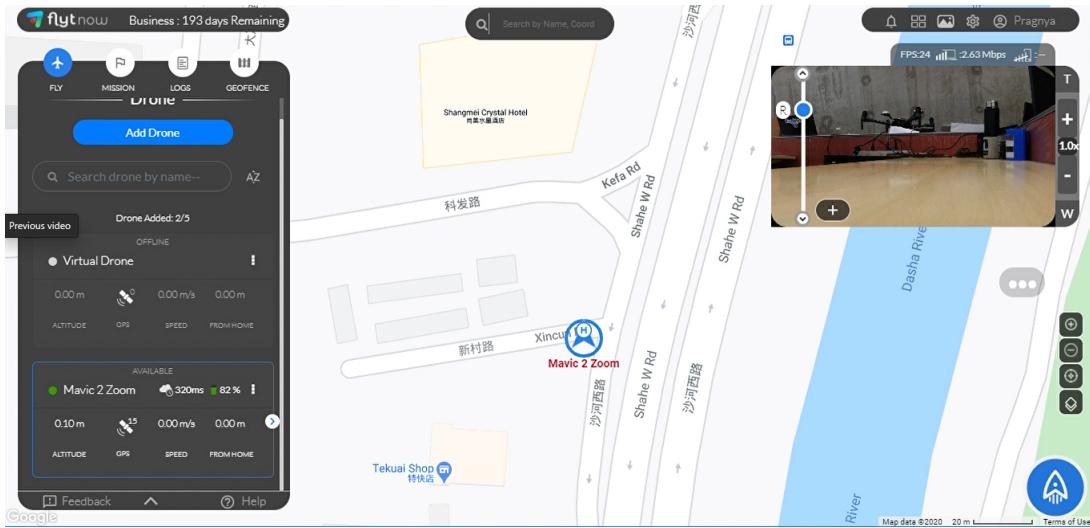
A DHL Percecopter alapvetően más problémát akar megoldani, a nagyon nehezen elérhető helyekre probálnak csomagokat szállítani.

A *FlytBase* [5] vállalat már kínál szoftveres megoldásokat drónok automatizációjára és valós idejű figyelésére és irányítására, útvonal tervezésre valamint flotta menedzselére. Már van kész megoldásuk a következőkre:

- biztonsági megfigyelések,
- automatikus dokkolás állomásokhoz,
- veszélyhelyzetre reagálás (ez lehet vér, szerv vagy gyógyszer szállítás),
- csomag szállítás.

Elterjedt termékük a *FlytNow*, ami úgy működik, hogy a drónunkat összekötjük egy alkalmazáson keresztül a felhőben lévő *FlytNow* rendszerrel, majd interneten keresztül elérünk egy verzérőlőpultot (2.3. ábra) amin irányíthatjuk a drónjainkat és láthatjuk az adatokat. Ez a rendszer arra összpontosít, hogy gyorsan, bonyolult telepítés és beállítás nélkül tudunk drónokat irányítani. Ezt a rendszer magánembereknek és cégeknek egyaránt elérhető, több csomagban. Van egy másik *FlytWare* nevű termékük raktár

menedzselésre és logisztikai feladatokra. Olyan feladatokra specializálódnak a drónok, mint a bárkód szkennelés, zárt környezetben repülés. Természetesen a raktár adatait elérjük a rendszer vezérlőpultján. Így például egy leltár sokkal egyszerűbben megoldható.



2.3. ábra. Flytnow livestream dashboard [2]

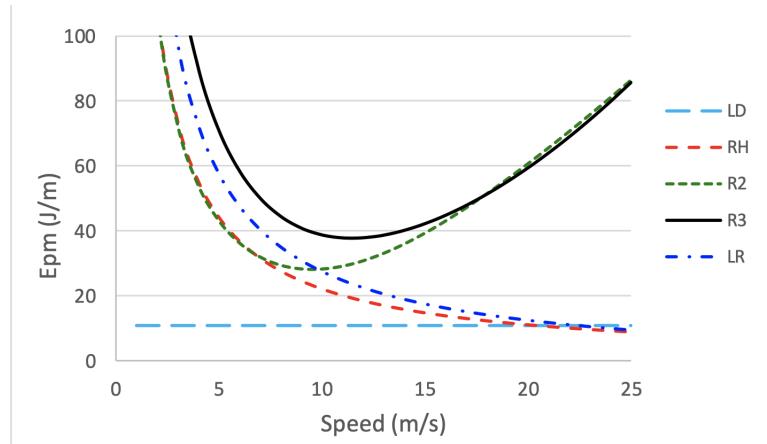
Az ANRA Techonlogies[4] cég is foglalkozik különféle drón irányítás, megfigyeléssel. A SmartSkies néven futó rendszere például drónokat használ többféle szenzorral, ahhoz hogy a pontosabb adatokat kapjunk megfigyeléses mérésekhez. Például egy olyan légtérben, ahol rengeteg drón repül, előfordulhatnak hibák, egy drón rossz adatokat küld a helyzetéről vagy teljesen megszűnik a kommunikáció, esetleg idegen tárgyak akadályozzák a légtérben való forgalmat, stb. Ebben a rendszerben minden drón el van látva radárral, kamerával és más szenzorokkal. Így az összegyűjtött adatokat kombinálva egy sokkal pontosabb képet kapunk, ami hasznos a navigációhoz, és akár fél percre előre meg tudjuk mondani mi lesz az adatok alapján, előre jelezve az esetleges hibákat.

A BeeCluster[3] egy nyílt forráskódú, Golang-ban írt drón orchesztrációs rendszer. Előnye, hogy prediktív optimalizációt használ, tehát a feladatok előtt egy előszimulációt futtat, ami alapján optimalizálja, azaz hatékonyabbá és gyorsabbá teszi a valós folyamatot. Olyan feladatokra használják, mint a földrajzi feltérképezés, Wi-Fi lefedettség térkép készítése, új utak feltérképezése.

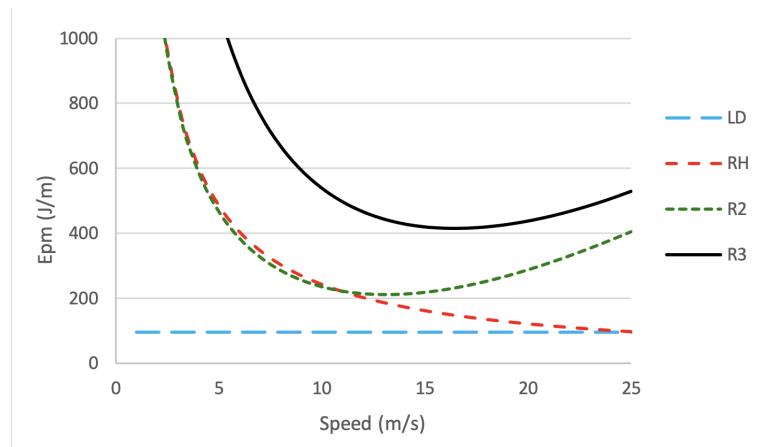
Jelenleg nincs a piacon olyan, bárki számára elérhető szállításra gyártott drón, ami hatékonyan szállít csomagot. Egyelőre csak a szállításban érdekelt cégek használnak ilyen drónokat, de sajnos nem osztják meg a pontos paramétereket. Annyi használható információ érhető el, hogy az Amazon teszt drónjai átlag 70 km/h sebességgel 12 kilométeres körzetben tudnak szállítani 2,3 kg-tól könnyebb csomagokat, de nem tudjuk a drón súlyát és azt se, hogy mennyi energiát fogyaszt.

A Missouri Egyetemen végeztek egy, a drónok energia fogyasztására irányuló kutatást[13], amiből több érdekes dolog is kiderül. Mind a kis (1-3 kg) és nagy (5-15 kg) drónok hatékony sebessége 12 m/s körül van (2.4. és 2.5. ábra).

A szállítást célszerű az energiafogyasztás, vagy a gyorsaság szempontjából optimalizálni. Ha az eddig meglévő adatokkal számolunk, tehát hogy egy drón 12 m/s átlagsebességgel fog haladni és maximum 12 km-re van a célállomás, teljesen életszerű hogy egy 5 km-re lévő célállomáshoz való csomagszállítás pár perc alatt megtörténik.



2.4. ábra. Kis drónok fogyasztása



2.5. ábra. Nagy drónok fogyasztása

Ha kétszáz milliszekundumonként küld adatot egy drón, az az adatbázisban többezér beszűrést eredményez.

3. fejezet

Szállítási probléma

3.1. A szállítási probléma absztrakt modellje

A szállítási modellünk alapvetően úgy nézne ki, hogy a raktár és a kiszállítási hely közti távolság függvényében a csomagokat a raktárból drónok, vagy drónokkal felszerelt teherautók viszik ki. A teherautók a kiszállítási hely közelében használnák a drónokat, hogy kézbesítsék a csomagokat.

Feltételezzük, hogy a drón és a csomag ugyanazon a raktárba vagy tehergépkocsiba található, de a szimuláció szemszögéből nem fontos, hogy raktár vagy tehergépkocsi, az a lényeges, hogy egy helyen található, és a drón automatikusan fel tudja venni az adott csomagot valamilyen rendszeren keresztül.

A drónok 3G, 4G vagy 5G hálózaton kommunikálnának az adatközpontokkal. Az adott hálózaton elérhető sávszélesség függvényében a drónok változtathatják az elküldött adatok mennyiségét és a küldés ütemezését, gyakoriságát.

A drónok repülése előre megtervezett. A szállítási feladat egy speciális esetét feltételezzük, a hozzárendelési feladatot. Tehát, hogy egy drónhoz pontosan egy csomagot rendelünk. A kis hatótáv és erőforrások hiánya miatt a drónok egyébként sem tudnának egynél több csomagot szállítani, csak nagyon ritka esetben. Azt, hogy melyik drónhoz melyik csomag tartozik a hozzárendelési feladat optimalizálásával kapjuk meg. A drónokból a szállítás során keletkező nagymennyiségű adatot az adatközpontoknak valós időben, hibamentesen és hatékonyan kell tudnia kezelní anomáliák nélkül. Ehhez nagyon fontos a terhelés megfelelő elosztása.

A drónok csomagszállítási folyamatáról a következőket feltételezzük:

- Egy drón véges akkumulátoridővel vagy üzemanyaggal rendelkezik, és minden vissza kell tudnia térnie egy töltőállomásra, amely lehet egy drónszállító teherautó vagy raktár.
- A csomag kézbesítésnek a csomag átvételére vonatkozó ideje elhanyagolható.
- minden drón fel van szerelve megfelelő kommunikácos és helymeghatározó eszközökkel, valamint kommunikál az adatközpontokkal.
- Feltételezzük, hogy a drónok el vannak látva szenzorokkal, hogy felismerjék az akadályokat, és erről értesíték az adatközpontokat, de maguktól is képesek beavatkozni.
- Egy drón csak 1 csomagot képes szállítani.

A mi esetünkben csak akkumlátorral üzemelő drónok lesznek, és a drónok fogyasztása előre ismert. minden drón küld telemetria adatokat. A drónok a telemetria adatok részeként küldenek helymeghatározásra alkalmas földrajzi szélesség, hosszúság koordinátákat és magasságot. Továbbá, kapunk még információkat a drón sebességéről, gyorsulásáról, tájoló iránytűjének állásáról, motorjainak hőmérsékletéről, akkumulátor(ok) töltöttségről és hőmérsékleről, mindezt egy időpeccettel megbélyegezve hogy az üzenetek elküldésének idejét is tudjuk. Hogy el tudjuk dönteneni melyik drón hova tud a leghatékonyabban szállítani, a következő mennyiségeket jelölni kell. Hány km-re van a felszállási ponttól a csomag leadási helye (távolság km-ben), mekkora a drón fogyasztása egy adott csomag súlyával együtt. Mivel minden drónhoz csak 1 csomagot rendelünk, ez a szállítási feladat egy speciális fajtája, úgynevezett hozzárendelési feladat amit a *magyar módszer algoritmussal*[21] meg lehet oldani. Tehát, ha ismerjük a drónokat és a csomagokat, minden drónhoz rendelhetünk egy képletet ami megmutatja a fogyasztási költséget az adott csomaggal. Mivel ismerjük a drónok felszállási helyét és a csomagok címzett helyét is, jelölni tudjuk a távolságot is. A távolság és fogyasztásból kijön egy adott szállítás költsége. Értelemszerűen az adott szállításhoz azt a drónt keressük, amelyik minimális költséggel szállítja el a csomagot. Azaz a fogyasztást szeretnénk minimalizálni. Persze az optimalizációt lehetne még bonyolítani, tiltási tényezőket használni, például hogy egy drón maximum milyen súlyú csomagot vihet.

3.1.1. Konkrét példa optimalizálásra

Egy optimalizálási feladat a következőképp nézhet ki. Az egyszerűség kedvéért tegyük fel, hogy 4 drón és 4 csomag kapcsolatát vizsgáljuk. Jelölje d_i a drónokat és c_j a csomagokat, $i, j = 1, 2, 3$. A feladat modellje a következő:

A drónok fogyasztása távolság egységenként rendre: d_1 fogyasztása = 800 d_2 fogyasztása = 700 d_3 fogyasztása = 650 d_4 fogyasztása = 390

A csomagok súlyával most az egyszerűség kedvéért nem számolunk, csak a távolságot vesszük figyelembe. c_1 távolsága = 0,749 c_2 távolsága = 1,255 c_3 távolsága = 2,269 c_4 távolsága = 1,844

A költségeket a következő képlettel számolhatjuk ki. $Költség_{ij} = d_i \cdot c_j$. A kapott költségek a következő táblázatban 3.1 figyelhetőek meg.

3.1. táblázat. Optimalizálási feladat táblázat. Az adott csomag szállításának költsége az adott drónnal.

Csomag \ Drón	d1	d2	d3	d4
c1	599.92	524.93	487.43	292.46
c2	1004.19	878.67	815.9	489.54
c3	1815.68	1588.72	1475.24	885.14
c4	1475.76	1291.29	1199.1	719.43

$$\begin{aligned}
 & 599.92x_{11} + 524.93x_{12} + 487.43x_{13} + 292.46x_{14} + 1004.19x_{21} + 878.67x_{22} + \\
 & 815.9x_{23} + 489.54x_{24} + 1815.68x_{31} + 1588.72x_{32} + 1475.24x_{33} + 885.14x_{34} + \\
 & 1475.76x_{41} + 1291.29x_{42} + 1199.1x_{43} + 719.43x_{44} \rightarrow \min .
 \end{aligned}$$

A feladat megoldását kezdjük a költségmátrix felírásával, ami lényegében megegyezik az előző táblázattal.

	<i>D1</i>	<i>D2</i>	<i>D3</i>	<i>D4</i>
<i>C1</i>	599.92	524.93	487.43	292.46
<i>C2</i>	1004.19	878.67	815.9	489.54
<i>C3</i>	1815.68	1588.72	1475.24	885.14
<i>C4</i>	1475.76	1291.29	1199.1	719.43

Mivel a feladatban a kereslet és kínálat egyensúlyban van, alkalmazhatjuk a magyar módszer algoritmus következő lépéseit. A kapott végeredmény az, hogy a minimális költség 3562.83. A következő mátrix már csak a megoldást tartalmazza az eredeti költségmátrixra vetítve:

	<i>D1</i>	<i>D2</i>	<i>D3</i>	<i>D4</i>
<i>C1</i>	599.92			
<i>C2</i>		878.67		
<i>C3</i>			885.14	
<i>C4</i>			1199.1	

3.2. A szállítási probléma egy konkrét példája

A szállítás úgy kezdődik, hogy az adatközpont lekérdezi a szállításra váró csomagokat. Ezután lekérdezi az összes szabad drónot. Elemzi a drónok és a csomagok paramétereit alapján, hogy hogyan lenne optimális a szállítás. Ez valamilyen eredményt szolgáltat, ami alapján az adatközpont a csomagokhoz drónokat rendel a megfelelő útvonallal. A drónok elhagyják a raktárat a csomaggal, és folyamatosan küldik az adatokat állapotukról, az adatközpont pedig elemzi és feldolgozza ezeket.

Pédakén, t egy drón felszáll egy csomaggal, a csomagban lévő tárgy kevesebb mint 1kg. A drón a kijelölt útvonalon halad, nem adódik semmilyen probléma a cél elérése közben. A drón leadja a csomagot a kijelölt helyen, és jelez az adatközpontnak hogy sikeresen kézbesítette a csomagot. Ekkor az adatközpont jelez a megrendelőnek hogy a csomagját átveheti. A drón felemelkedik és visszatér a raktárba, a kijelölt útvonalon. Itt az akkumulátorját szükség esetén cserélik/feltöltik, majd a drón felveheti a következő csomagot.

Egy másik példaként, egy drón felszáll egy csomaggal, a csomagban lévő tárgy kevesebb mint 1kg. A drón a szállítás közben különösen magas hálózat interferenciát érzékel, és hibajelet küld az adatközpontnak. Az adatközpont az elemzés szerint arra jut, hogy az interferencia nem természetes, túl nagy az esélye hogy a drónt szándékosan

zavarják. Az adatközpont jelez a drónnak, hogy a szállítást szakítsa meg, térjen vissza a raktárba. A drónnak a hiba miatt a legközelebbi repülés előtt több diagnosztikai teszten át kell mennie, hogy megbizonyosodjunk a drón üzemképes működéséről.

3.3. A szállításhoz kapcsolódó számítási problémák

A szállítás pontos szimulációjához a Föld alakját is figyelembe kell venni a repülésnél. Ugyanis itt a Descartes féle koordináta rendszer nem jól működik, mivel a Földnek gömb alakja van. Emiatt teljesen máshogyan kell számolni távolságot, mint a Descartes féle geometriában.

3.3.1. Descartes szerinti

$$P_1(5, 6, 3),$$

$$P_2(7, 4, 9),$$

$$V = 10 \frac{m}{s}.$$

Koordináták kiszámítása polárkoordináták segítségével:

$$x = r \cdot \cos(\phi) \cdot \sin(\theta),$$

$$y = r \cdot \sin(\phi) \cdot \sin(\theta),$$

$$z = r \cdot \cos(\theta).$$

Koordináták kiszámítása vektorokból:

$$x = X_2 - X_1 = 7 - 5 = 2,$$

$$y = Y_2 - Y_1 = 4 - 6 = -2,$$

$$z = Z_2 - Z_1 = 9 - 3 = 6,$$

Sugár kiszámítása pitagorasz tételel:

$$r = \sqrt{2^2 + (-2)^2 + 6^2} = \sqrt{44}.$$

$\cos(\phi)\sin(\phi)\cos(\theta)\sin(\theta)$ Szögek kiszámítása a koordinátákból:

$$\cos(\phi) = \frac{x}{\sqrt{x^2 + y^2}},$$

$$\sin(\phi) = \frac{y}{\sqrt{x^2 + y^2}},$$

$$\cos(\theta) = \frac{\sqrt{(x^2 + y^2)}}{\sqrt{x^2 + y^2 + z^2}},$$

$$\sin(\theta) = \frac{z}{\sqrt{x^2 + y^2 + z^2}}.$$

További számítások, segédvektorok kiszámítása:

$$V_x = \frac{x}{\sqrt{x^2 + y^2 + z^2}} \cdot V = \frac{2}{\sqrt{44}} \cdot 10 = \frac{10\sqrt{11}}{11} = 3.0151,$$

$$V_y = \frac{y}{\sqrt{x^2 + y^2 + z^2}} \cdot V = \frac{-2}{\sqrt{44}} \cdot 10 = -\frac{10\sqrt{11}}{11} = -3.0151,$$

$$V_z = \frac{z}{\sqrt{x^2 + y^2 + z^2}} \cdot V = \frac{6}{\sqrt{44}} \cdot 10 = \frac{30\sqrt{11}}{11} = 9.0453,$$

t idő múlva egy adott pillanatban koordináták meghatározása:

$$x = X_0 + V_x \cdot t = X_0 + \frac{X}{\sqrt{x^2 + y^2 + z^2}} \cdot t = 5 + \frac{2}{\sqrt{44}} \cdot 10 \cdot t,$$

$$Y = y_0 + V_y \cdot t = y_0 + \frac{y}{\sqrt{x^2 + y^2 + z^2}} \cdot t = 4 - \frac{2}{\sqrt{44}} \cdot 10 \cdot t,$$

$$z = z_0 + V_z \cdot t = z_0 + \frac{z}{\sqrt{x^2 + y^2 + z^2}} \cdot t = 6 + \frac{6}{\sqrt{44}} \cdot 10 \cdot t.$$

3.3.2. Az Ortodroma számítás, a föld alakját figyelembe véve

Mivel gömbi geometria lényegesen eltér az euklideszi geometriától, ezért a távolság-számításra használt matematikai képletek is eltérőek. Az euklideszi geometriában a legrövidebb távolságot a két pontot összekötő egyenes, a nem euklideszi geometriában a két pontot összekötő geodetikus vonal (gömb esetén fókör) mentén mérjük.

A legrövidebb út kiszámítása

$$d = \arccos(\sin\phi_1 \cdot \sin\phi_2 \cdot \cos\phi_1) \cdot \cos\phi_2 \cdot \cos\Delta\lambda) \cdot R$$

Irány kiszámítása

Ahogy a fókörön haladunk az irányunk folyamatosan változik. Tehát amikor megérkezünk a célt, nem ugyanabban az irányba nézünk mint amikor elindultunk. A drón szimulációban ez a telemetria adatokban is megfigyelhető.

$$\varphi = \text{atan2}(\sin\Delta\lambda \cdot \cos\phi_2, \cos\phi_1 \cdot \sin\phi_2 - \sin\phi_1 \cdot \cos\phi_2 \cdot \cos\Delta\lambda)$$

ahol $\phi_1\lambda_1$ a kezdeti pont, és $\phi_2\lambda_2$ a vég pont és $\Delta\lambda$ pedig a különbség a hosszúsági fokokban.

Cél koordináták kiszámítása, ha ismerjük a megtett távolságot, irányt, és kezdő koordinátákat

$$\begin{aligned}\phi_2 &= \arcsin(\sin\phi_1 \cdot \cos\delta + \cos\phi_1 \cdot \sin\delta \cdot \cos\theta) \\ \lambda_2 &= \lambda_1 + \arctan2(\sin\theta \cdot \sin\delta \cdot \cos\phi_1, \cos\delta - \sin\phi_1 \cdot \sin\phi_2)\end{aligned}$$

ahol ϕ a szélességi fok, λ a hosszúsági fok, θ az irány, δ a szögtávolság d/R és d a megtett távolság, R pedig a Föld sugara.

4. fejezet

Eszközkészlet és tervezés

4.1. Rendszer követelmények

4.1.1. Általános leírás

Egy olyan elosztott rendszer létrehozása a cél, ami képes a drónszállítás közben keletkező valóságnak megfelelő adatokat szimulálni és hatékonyan feldolgozni. Ezeknek az adatoknak a feldolgozását vizsgálom, több kontextusból. Egyrészt a hálózaton való kommunikació hatékonyságát vizsgálom, tehát hogy mennyi adatforgalommal jár a kommunikáció, mennyi erőforrással jár csak a kommunikáció a küldő és fogadó részéről (tehát a feldolgozás nélkül). Másrészt azt, hogy milyen hatékony az kommunikációval közölt adatok feldolgozása, azaz a hálózaton használt adatformátum átalakítása a programban használt formátumra. Továbbá összehasonlítjuk az adatbázisba való mentés hatékonyságát. Mivel elosztott rendszerekről van szó, nem egy darab program lesz. Mikro-szervízhez hasonlóan működnek majd a programok. Ez azt jelenti, hogy ha 100 adat feldolgozást végző program fut egy terhelést elosztó proxy mögött, akkor ugyanolyan viselkedés várható mint ha csak 1 program futna. A drón szenzorai által gyűjtött adatokat telemetria adatoknak nevezzük. A programok egy részének ezeket az adatokat kell tudnia generálni, egy másik részének feldolgozni.

4.1.2. Hatékonyság

A hatékonyság alatt egyszerre értjük a teljesítményt, tehát hogy mennyire gyors a feldolgozás, és azt is, hogy ez a feldolgozás mennyi erőforrást használ. Mindezt úgy kell elérnünk, hogy az adatok feldolgozása aszinkron módon történik, de az aszinkron feldolgozás nem járhat lényeges többletmunkával, és semmilyen módon nem akadályozhatja a rendszer megfelelő működését. Tehát fontos, hogy jól használjuk ki az erőforrásainkat, és a rendszer bármikor skálázható legyen horizontálisan is.

4.1.3. Funkcionális követelmények

A szimulációban többféle protokollt és adattovábbításra használt formátumot hasonlítunk össze, így a programot úgy kell felépíteni, hogy ki lehessen cserélni ezeket részeket anélkül hogy a program üzleti logikájának működését befolyásolnánk. Az adatközponkok és drónok közötti telemetria adat kommunikációt és annak feldolgozását összehasonlítjuk egy HTTP 1.1 -en működő JSON adatformátumot használó REST API-n, egy

HTTP 2 -n futó gRPC-t használó végponton is. Az adatok mentését is több adatbázison kell tesztelni, így az adatbázisnak is cserélhetőnek kell lennie.

4.2. A Go nyelv áttekintése

A Go program nyelv, (sokszor Golangként emlegetik) egy nyílt forráskódú modern programozási nyelv melyet a Google fejlesztett ki [24].

A Google-nél olyan problémák adódtak, hogy a konkurrenciát nem tudták megfelelően kezelní a még eredetileg 1 processzoros számítógépekre kifejlesztett nyelvek, mint a Java, C++.

Persze azóta sokat fejlődött ezen nyelvek konkurrenciakezelés, de nem tudnak versenyezni a Go-val, ha a gépi és fejlesztői hatékonyságot is figyelembe vesszük. Probléma volt az is, hogy ezeknek a nyelveknek nagyon nagy a fordítási idejük. A 2000-es évek végén ez azt jelentette hogy egy Java vagy C++ program a Google-nél 1 hét alatt fordult le, csak hogy ki tudják próbálni. A nyelvek bonyolultságával is baj volt, ahogy fejlődtek a nyelvek egyre nehezebb volt egy régi programot továbbfejleszteni, illetve új programozóknak megtanulni a nyelvet.

Hogy ezeket a problémákat orvosolják, a Google a legjobb tervezőket hívta össze, hogy együtt alkossanak egy új nyelvet. Így született meg a Golang. A fejlesztést olyan szakemberek vezették, akik korábban a Unix operációs rendszer, a Java HotSpot JVM vagy az UTF-8 karakterkódolás fejlesztésében is kulcsszerepet játszottak. Például Ken Thompson, Rob Pike és Robert Griesemer.

A cél az volt, hogy régebbi általános célú nyelvek hiányosságait kiküszöböljék, és csökkentsék a bonyolultságot, hiszen a megváltozott üzleti és technológiai körülmények között ezek a nyelvek nem bizonyultak elég hatékonyak. Nem várhatták el, hogy egy friss diplomás hatékonyan kezeljen egy olyan "felpuffadt" nyelvet mint a Java.

A felhőben való futtatásra olyan alkalmazások készítésére volt igény, amelyek nagy hatékonysággal futnak és kiválóan skálázódnak. A végeredmény egy olyan nyelv amely általános célú, könnyen tanulható, kifejező, tömör, letisztult és hatékony. Éppen ezért kiválóan alkalmazható ott, ahol a kódmezőkhöz nagyszámú, gyakran cserélődő, változó összetételű programozói gárda fér hozzá, akár időben és földrajzi elhelyezkedésben is megosztva.

Szintaktikailag a C-hez hasonlít, de nagyon könnyen tanulható nyelv. A legnagyobb újítás a konkurrencia mechanizmus (főleg a saját ütemező) és hibakezelés körül volt. A Go konkurrencia mechanizmusa megkönnyíti olyan programok írását amelyet a legtöbbet hozzák ki a többmagos és hálózaton összekötött gépekből. Gyorsan lefordul gépi kódra de rendelkezik garbage collection-el és runtime reflection is van beépítve. Gyors, statikus nyelv de úgy érezzük mintha egy dinamikus futás időben fordított nyelv lenne. Tartalmaz *race detector*-t is ami nagyon fontos egy ilyen nyelvben, mert fel tudja ismerni a konkurrens/párhuzamos programozásnál felmerülő gyakori problémákat. Mivel nagyon jól kihasználja a gép erőforrásait és a hálózatot manapság a felhőben futó kódok nagyrésze Go. "Beszédes, hogy a Linux Foundation által létrehozott Cloud Native Computing Foundation (CNCF) keretein belül támogatott projektek nagy része Go-ban íródott, többek között a Kubernetes, a Prometheus, az Istio, de ebben írták a Dockert is." [18]

A Go konkurrencia mechanizmusa

A konkurrencia a nyelv alapjának része. A 3 legfontosab doleg ebből a szempontból az hogy a Go-nak saját ütemezője van, a gorutinok ("goroutine") és csatornák ("channel"). A goroutin egy szójáték a hasonló coroutine-al. A lényege hogy ezek a folyamatok a Go runtime-on belül futnak, sokkal kisebbek, mint egy processz, megabájtok helyett csak pár kilobájt, így pillekönnyűnek számítanak az Operációs rendszer processzeihez képest. *Egy programban akár több millió gorutin is futhat egyszerre!* A saját ütemező (scheduler) sokkal hatékonyabb, mintha csak létrehoznánk egy processzt és hagynánk hogy az Operációs rendszer ütemezze, mivel így az ütemező meg tudja határozni hogy az adott gorution milyen állapotban van (várakozik, futtatható, vagy fut) és hatékonyabban tudja kiosztani az erőforrásokat. Értelemszerűen például egy blokkoló, épp I/O műveletet végző gorutint nem fog futtatni hanem helyette más gorutinokat részesít előnyben. De fontos, hogy az ütemező nem determinisztikus, így nem tudjuk előre megmondani melyik gorutin fog következni.

4 féle esemény fordulhat elő a Go programokban ami miatt az ütemező döntéseket hozhat:

- a *go* kulcsszó használata,
- garbage collection,
- rendszer hívások,
- szinkronizálás és orchesztráció, vezérelés.

A go kulcsszó használata: A *go* kulcsszó létrehoz egy új gorutint, így az ütemezőnek lehetősége van beavatkozni.

Szinkronizálás és vezérlés: Ha egy atomi, mutex, vagy csatorna művelet, függvényhívás miatt egy Gorutin blokkolni fog, az ütemező kontextus cserével egy másik gorutinnak adhat erőforrást. Ha a blokkolt Gorutin újra futtathatóvá válik, sorba állítja az ütemező és végül visszakapja az erőforrást.

A gorutinok ugyanabban a címtérben futnak, ezért a megosztott memóriához való hozzáférést szinkronizálni kell.

A csatornák a típusos vezetékek amelyeken adatokat küldhetünk és fogadhatunk a csatorna operátorral, `<-`. Az adatok a nyíl irányába folynak. Ahhoz, hogy jobban megértsük a csatornák szükségességét meg kell értenünk a következő, a Golang tervezői szerint információmegosztásra megfelelő konkurrencia modellt: *"Do not communicate by sharing memory; instead, share memory by communicating."* Tehát arról van szó, hogy ha direkt megosztott memórián keresztül kommunikálunk abból csak baj lehet, a megosztott memória ilyen használata veszélyes, helyette egy olyan módszert kell alkalmazni, ami védelmet biztosít a megosztott memória felett. A Golang csatornái pont ezt az elvet követik.

```

1   ch <- v    // Snd v to channel ch.
2   v := <-ch  // Receive from ch, and
3   // assign value to v.

```

Alapértelmezett beállítás szerint a csatornák blokkolnak, amíg az egyik oldal küld vagy fogad. Ez lehetővé teszi a szinkronizációt anélkül, hogy zárákat vagy feltétel változókat hoznánk létre.

Bufferelt csatornák

```
1 ch := make(chan int, 100)
```

Tehát a Go konkurrenciamodelljének sikere abban rejlik, hogy saját ütemezőt használ a gorutinok vezérléséhez, a nyelv magját képezi a gorutinok közötti szinkronizációs tevékenységek megoldása, továbbá az, hogy típusos csatornákon kommunikálhatnak a gorutinok egymással.

A Go hibakezelése

A hibakezelés nagyban megváltozott azzal, hogy a Go-ban nem try és catch blokkba zárjuk a hibára futható programkódot, hanem van egy beépített hiba típus, ami a függvények visszatérési értéke lehet. Fontos tudni, hogy a Go-ban egy függvény többféle visszatérési értékkel rendelkezhet. Ha nem egy nagyon egyszerű programot fejlesztünk, a legtöbb függvényünk utolsó visszatérési értéke egy hiba típus lesz. Csak az olyan atomi műveleteket végző függvényeknél nem szoktunk hiba típust használni, ami biztos, hogy nem fut hibára normális körülmények között. Például egy függvénynek ami összeadásokat, vagy egyéb egyszerű műveleteket végez értelemszerűen nincs értelme hogy hibával térjen vissza. De egy olyan függvénynek, ami egy fájlt nyit meg, már van értelme hiba típussal visszatérnie, mert több hiba is előfordulhat az instrukciók futása közben, lehet nem létezik a fájl, vagy nincs jogosultságunk megnyitni.

```
1 func Open(name string) (file *File, err error)
```

A fenti függvény például egy Fájl típus pointerét és egy hiba típust ad vissza. Ha nem futott hibába a program, az err változó nil értéket kap.

Tehát egy hiba kezelése például így nézhet ki:

```
1 file, err := Open("fajlnev")
2 if err != nil {
3     log.Println(err)
4     return nil, err
5 }
```

Ennek a hibakezelésnek az előnye, hogy a hibák, mint visszatérési értékek buborék-ként felpezsegnek a függvényhívásokon keresztül. De a programokban amiket fejlesztünk célszerű saját hiba típusokat létrehoznunk rétegenként, vagy legalább az üzleti logika rétegben. Ez jó a kód átláthatóságához, az esetleges hibakereséshez (debugging) elengedhetetlen. Ezen kívül fontos, hogy az alkalmazásunk nyitott részén ne látszódjanak olyan hibák, amik másra nem tartoznak. Például nem előnyös az, ha egy API hívás azzal tér vissza, hogy a relációs adatbázishoz használt driver milyen hibába ütközött. Így ugyanis eláruljuk az alkalmazásunkhoz használt infrastruktúrát vagy más akár bizalmas információt, megkönnnyítve a lehetséges támadásokat.

Pár vállalat aki használja:

- Google,

- Uber,
- Netflix,
- Youtube,
- Dropbox,
- BBC.

Az előnyökhöz persze hátrányok is tartoznak, az egyszerűség mellé nem férnek el a generikus függvények. Maga a nyelv nem tartalmaz osztályokat, illetve nincs öröklődés. Viszont az objektum orientált paradigmákat használja, és így is kell benne programozni. Tartalmaz interfészket, a típusokhoz így a struchhoz is tudunk hozzárendelni úgynevezett receiver függvényeket, amelyeket más nyelvekben metódusoknak nevezünk, mert a típus egy példányára tudjuk meghívni.

4.3. Docker és a mikroszervízek

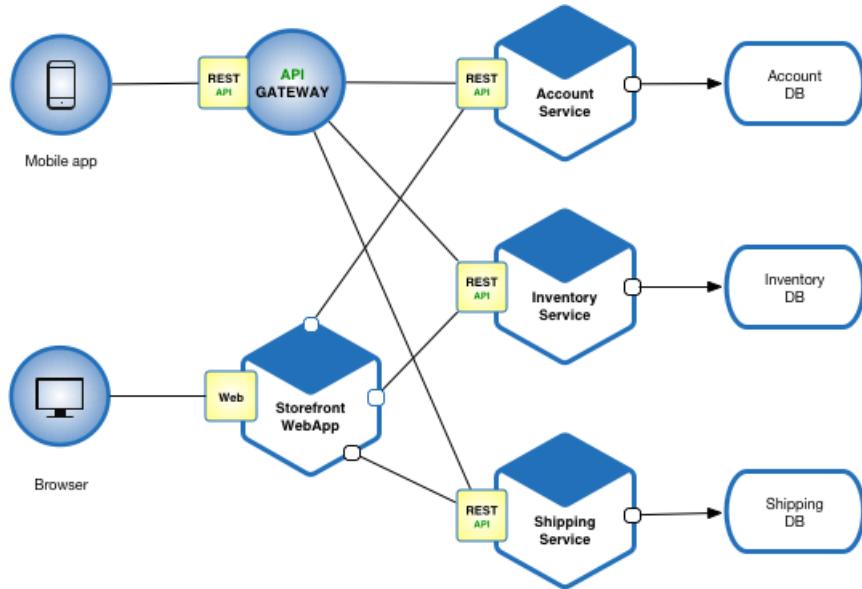
A mikroszervizes rendszer tervezési architektúra napjainkban nagyon elterjedt, azért mert triviális előnyei vannak a hagyományos monolitikus rendszer architektúrával szemben. Két legnagyobb előnye, hogy könnyű megváltoztatni a szoftver és jobban skálázható mint a monolitikus rendszerek.

A *monolitikus* rendszerek hátrányai:

- Ahogy nő az alkalmazás, vele együtt egyre nehezebb a fejlesztőknek továbbfejleszteni, és karbantartani azt.
- Megváltoztatni az applikáció technológiai összetételét szinte lehetetlen, a fejlesztőknek egy rémálom.
- minden egyes változtatás kihat az egész alkalmazásra, így azt minden szinten újra át kell nézni, karbantartani, fejleszteni. Ez felesleges munka a fejlesztők részéről.
- Nem skálázható horizontálisan.

A *mikroszervizes* rendszerek (4.1. ábra) előnyei:

- Csökkenti a komplexitást azzal, hogy a fejlesztők csapatokba oszlanak, minden csapat egy vagy viszonylag kevés szervízt fejleszt.
- Csökkenti a fejlesztés kockázatát, nem kell újraépíteni az egész alkalmazást egy fejlesztés után.
- Könnyű karbantartás, rugalmas fejlesztés mivel inkrementálisan tudunk fejleszteni egy vagy több szervízt, nem kell minden egyszerre.
- Egyszerű skálázhatóság.

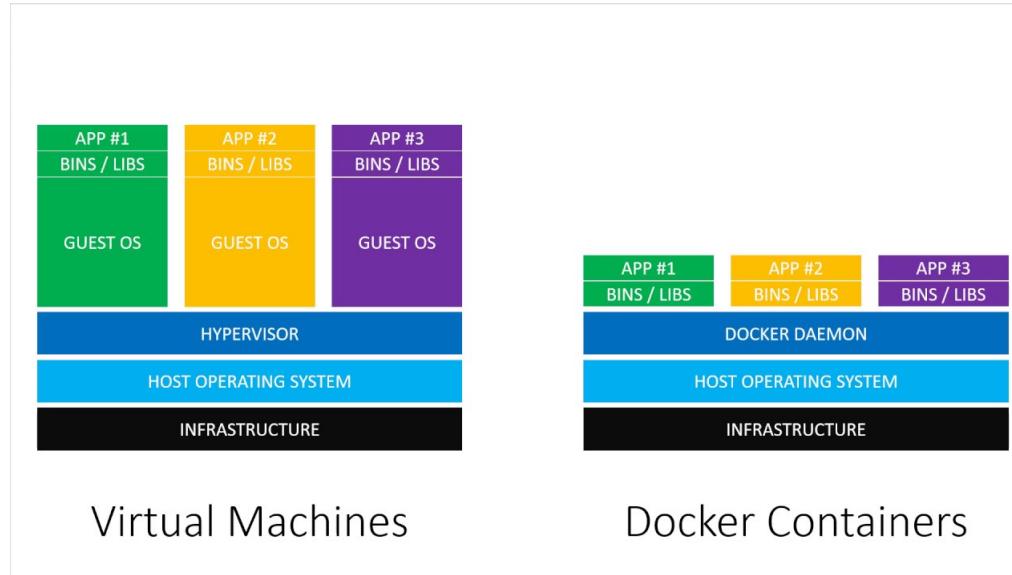


4.1. ábra. Mikroszervizek [19]

Tehát a mikroszervizek, a nevéről értetődően kis részemből állnak, és a monolitikus rendszerek problémáit hivatottak megoldani. Viszont ahhoz, hogy a kis, különálló szolgáltatásokat amiknek sokszor nagyon eltér a technológiai összetétele, megbízhatóan tudjuk működtetni ahhoz egy kiszámítható környezetbe kell zárni amely el van szigetelve más programuktól. Ugyanis csak így lehet tesztelni, hogy a szoftverfűggőségek teljesülnek, és eltérő infrastruktúra alatt is működni fog az adott szolgáltatás, alkalmazás. Például éles és teszt környezetben eltérő infrastruktúra lehet. De a fejlesztők dolgát is megkönnyíti ez a módszer. Ezeket a kiszámítható környezetekhez régen virtuális gépeket használtak, de ez mára elavult, helyette konténereket használnak amelyek "könnyebbek" és jobban kihasználják az erőforrásokat. A 4.2. ábrán látható a virtuális gépek és konténerek kapcsolata. Legegyeszerűbben a kettő kapcsolatát úgy lehetne leírni, hogy a virtuális gépek a rendszereket izolálják el egymástól, míg a konténerek az alkalmazásokat izolálja.

Docker

A Docker a világ vezető szoftver konténerizáló platformja [6]. A megadott alkalmazást, mikroszervízt beilleszt egy úgynevezett Docker konténerbe, amelyet aztán függetlenül lehet karbantartani és telepíteni. Hogy jobban megértsük a Docker működését, nézzünk egy konkrét példát. Egy alkalmazás fejlesztését három fejlesztő, a fejlesztők rendre Windows, Mac OS-t és Linuxot használnak. Konténerek nélkül minden fejlesztőnek órákon át tartó erőfeszítés lenne az alkalmazás telepítése a saját fejlesztői környezetükbe, és további erőfeszítéseket kell majd tenniük annak érdekében, hogy ugyanazt az alkalmazást később felhőbe telepítsék. A Docker konténerek tehát egy olyan környezetet biztosítanak minden számítógépen amin fut a Docker, ami ugyanúgy működik. Legyen szó fejlesztői, teszt vagy éles környezetről, minden kiszámítható a működés. A Docker úgy működik, hogy egy előre meghatározott lemezből, és egy bizonyos *Dockerfile*-ból épít egy konténeret egy úgynevezett build folyamatban kereszül. Ez a build folyamat rétegezett, azaz ha a fájlt módosítjuk, a módosítás csak az adott réteget, és az arra épülő rétegeket befolyásolja. Így nem kell minden újra építenünk a konténeret, azt egy cache-ből előszedjük addig a rétegig, amit még nem befolyásolt a fájl módosítása. A



4.2. ábra. Docker konténerek és VM [12]

Dockerfile-ban vannak megadva a parancsok a telepítéssel kapcsolatban.

4.1. példa. Egy konkrét példa egy egyszerű *Dockerfile*-ra:

```

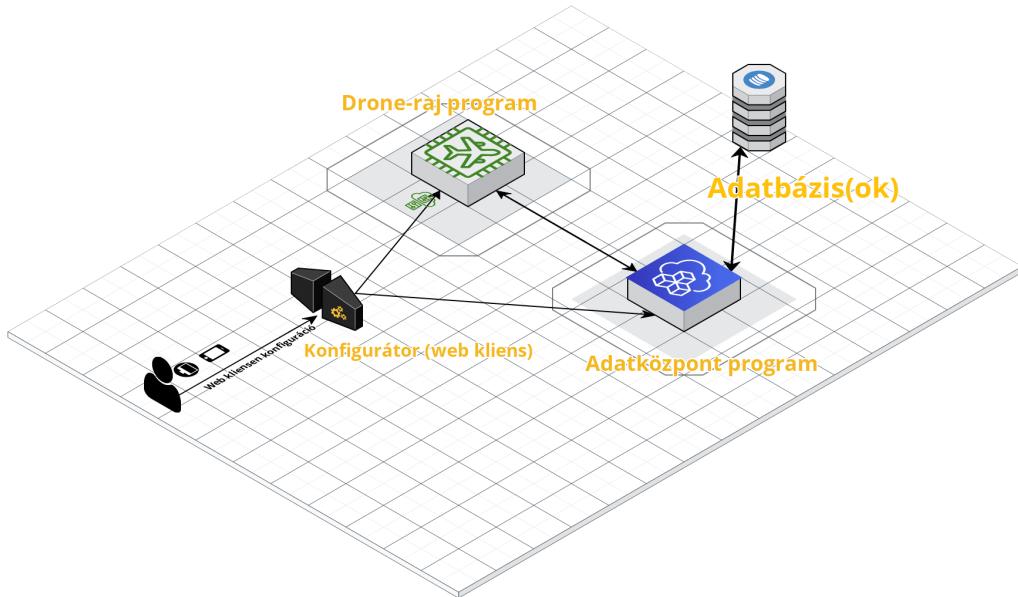
1  FROM golang:1.16 AS builder
2  RUN mkdir -p /go/src/drone-delivery/server
3  WORKDIR /go/src/drone-delivery/server
4
5  COPY go.mod /go/src/drone-delivery/server
6  COPY go.sum /go/src/drone-delivery/server
7  RUN go mod download
8  COPY ./ /go/src/drone-delivery/server
9  WORKDIR /go/src/drone-delivery/server/cmd/drone-delivery-server
10 RUN go install
11
12 ENTRYPOINT ["/go/bin/drone-delivery-server"]
13 EXPOSE 5000
14 EXPOSE 50051
15

```

4.4. Felhasznált eszközök

4.4.1. Architektúra, rendszer felépítése

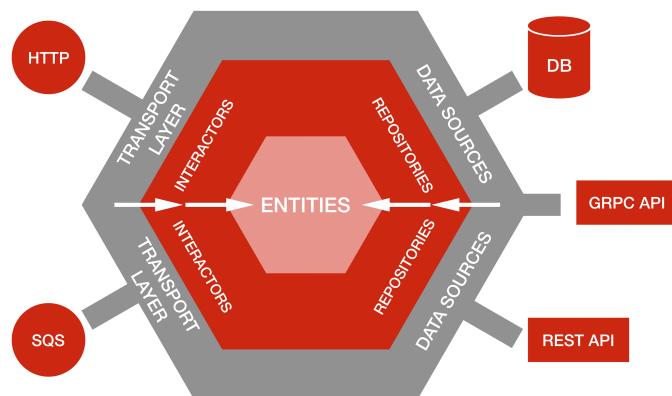
A rendszer működésének szimulációjához három programot használunk (4.3. ábra). Egy program az adatközpont, egy program a drónok szerepét veszi fel, és van még egy egyszerű böngészős web kliens amin paraméterek alapján konfigurálhatjuk a másik 2 program működését. Az adatközpont program, és a drón-raj program a magas fokú konkurrencia kihasználásához Go nyelven készül.



4.3. ábra. A 3 program

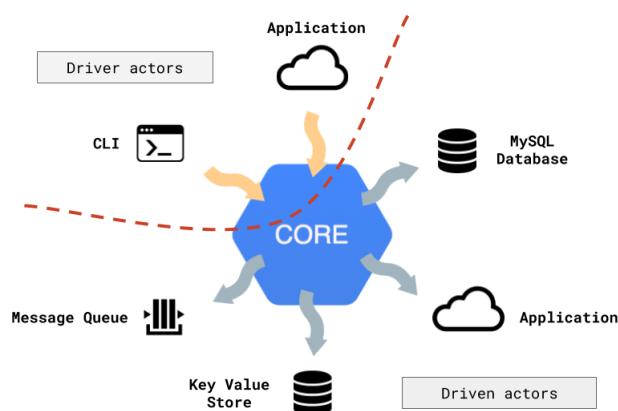
A programoknak egymástól elszigetelt környezetben kell működniük, hogy a valódi működést megfelelően tudjuk tesztelni. Egy számítógépen belül úgy tudjuk elérni, ha valamilyen módon konténerizáljuk a programokat, így minden program a saját elszigetelt környezetéhez fér hozzá, de a hálózaton tudnak egymással kommunikálni. Ilyen konténerizációs szoftver a Docker. Az adatközpontot szimuláló program lesz a legbonyolultabb. Ennek a programnak tudnia kell elvégezni a szállítás logisztikai feladatait, valamint hatékonyan feldolgozni és tárolnia az adatokat amiket a drónok küldenek.

Ahhoz, hogy a programot a követelményekben megfogalmazott módon építsük fel és cserélhető maradjon az adatbázis és a kommunikációs protokoll és adatformátum, egy olyan szoftver architektúrát és tervezést kell választani, ami megengedi ezt. Erre a problémára megoldásként a Hexagonal (másnéven ports and adapters) architektúrát (4.4. ábra) és DDD tervezési alapvet fogom alkalmazni.



4.4. ábra. Hexagonal architecture inward pointing dependencies [9]

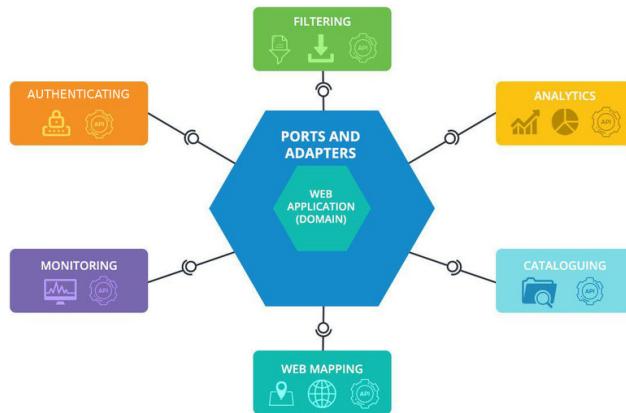
Így a program belülről kifelé épül fel, a program magja ahol az üzleti logika van interfaceket (portokat) alkalmazva kommunikál a program többi részével (adapterekkel), és nem támaszkodik semmilyen külső részre. Az interfacek miatt a külső rész a pontos implemetációja absztraktálva van, azaz csak az üzleti logikát ismerjük biztosan, minden ráépülő rész implementációja cserélhető marad. minden réteg csak befelé mutat, így az üzleti logikánk megmaradhat, miközben az alkalmazás teljes infrastruktúráját kicserélhetnénk. Így a végpontok ha teljesítik az interface elvárásait, csak dependency injection-el kicseréljük a végpontot és minden ugyanúgy működik, ám teljesen más az implementáció. Azt hogy több fajta input és output végpontot hogy támogatja az alkalmazás a 4.4. ábra) mutatja. Az alkalmazást fel lehet osztani verérlő és vezérelt részre 4.5. Ez nagyon jól igazodik az üzleti logikához, egy inputra szinte minden valamiféle vezérlést várunk el, tehát ahogy az alkalmazásunkat használjuk, vezéreljük, akkor az adatok hatására valamilyen folyamatot elvárunk. Ezután az alkalmazásunk kommunikálhat más, külső forrásokkal, amiket az alkalmazás használ, azaz vezéreli őket. Ilyen lehet egy adatbázis implementációja, vagy egy üzenet sor, amibe adatokat rakunk be, hogy valami más később kiolvassa. Azért jó ez a felépítés, mert így az alkalmazásunkat több helyről meghívhatjuk, de ugyanazt a működést produkálja az alkalmazás. Ezzel együtt jár, hogy felcserélhetőek ezen végpontok implementációi. A vezérelt egységeket is ki lehet cserélni, nem vagyunk röghöz kötve, mert egyszer így lett megírva az alkalmazásunk, csak a megfelelő adaptort kicseréljük, feltéve hogy megírtuk hozzá az implementációkat és az implementációt az interfészken keresztül teljesít minden feltételelt. Így például könnyen átállhatunk relációs adatbázisról egy dokumentum alapúra, vagy egy API-t is kicserélhetünk például JSON REST API-t gRPC-re ha valami miatt szükség van rá. A Netflix is ezt az architektúrát használja [9], ugyanis a gyors növekedésük közben több skálázhatósággal összefüggő problémába ütköztek, amit úgy oldottak meg, hogy ebben az architektúrában (4.6. ábra) szétosztották a feladatokat több, az adott kis feladatra megfelelő adatbázisra, mikroszervízre.



4.5. ábra. Hexagonal architecture driver and driven actors [22]

4.4.2. A Hexagonal architektúra és összehasonlítás a hagyma architektúrával

A hexagonal architektúra [11], vagy másnéven portok és adapterek architektúra a szoftver tervezés során használt minta. Célja lazán összekapcsolt alkalmazás komponensek



4.6. ábra. Hexagonal architecture ports and adapters [7]

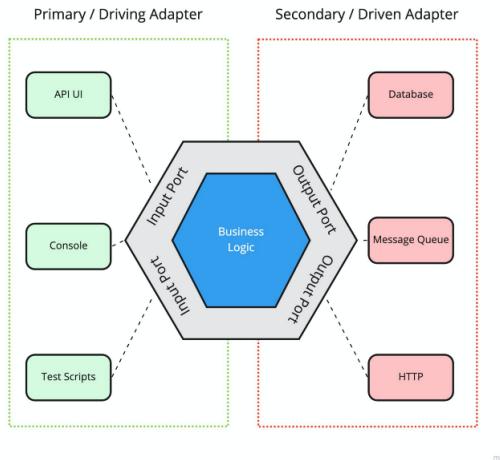
létrehozása, amelyek portok és adapterek segítségével könnyen összekapcsolhatóak. Ez cserélhetővé teszi a program komponenseket bármilyen szinten, és megkönnyíti a teszteket. A működése első látszatra hasonlíthat a hagyma (réteges) architektúrához, de két külön dologról van szó. Mindkét architektúra a Tiszta architektúra [14] elvet követi, ami arra törekszik hogy az alkalmazásunk üzleti logikája elhatárolódjon mindenféle külső környezettől, legyen az infrastruktúra, szoftver keretrendszer, user interface ami miatt ezek a tényezők cserélhetőek maradnak. Egy fontos különbség, a hexagonal (ports and adapters) és hagyma (réteges) architektúra között, hogy a hexagonal architektúban nincsenek pontosan megnevezett rétegek, van maga a program magja, ahol a domain modellek, és domain üzleti logika található, meg vannak a portok és adapterek. A portok interfészük írhatók le, amit az adaptereknek implementálni kell, így tudnak együtt működni. A programrészek között éles határok vannak, hiszen az adapterek nem tudnak egymásról, míg a hagyma architektúránál megengedett bizonyos "átfolyás" a rétegek között, hiszen ott a rétegek egymásra támaszkodnak és egy külső réteg használhatja bármelyik belső réteget. Általánosságban elmondható, hogy a hexagonal architektúra sokkal modulárisabb mint a hagyma architektúra, emiatt jobban tesztelhető. Hátránya az, hogy több kódot kell írni, így valamivel lassab nagy projekteknél.

Működése részletesen

A program magját, a domain komponenst írjuk meg először, ebbe definiáljuk az interféseket amik a portok a külső komponensekhez. Csak ezután kezdjük el fejleszteni a többi komponenst. minden komponens nyitott egy "porton". A komponensek közti kommunikáció ezeken a portokon történik (???. ábra). A portok egy absztrakt API-t definiálnak, aminek az implementációját adapterek valósítják meg, az objektum orientált nyelvekben interfész metódusokkal (port) vannak összekötve.

4.4.3. DDD tervezési elv

A domain vezérelt tervezés (Domain-driven design, DDD) [15] egy tervezési alapelv. A koncepciója az, hogy a szoftver struktúra és kód a domain üzleti logika alapján legyen felépítve. Tehát a jegyzék neveket, osztály neveket, metódus neveket, szervíz elnevezéseket az üzleti logikáról kell mintázni. Tehát a program kódja az üzleti logika



4.7. ábra. Hexagonal architecture driver and driven adapters Forrás: <https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0>

folyamatát követi, írja le. Például, ha egy szoftver söröket tart nyilván, söröket lehet hozzáadni és lekérdezni, akkor olyan osztályai lehetnek mint például *Beer*. Szervízt vagy valamilyen folyamatokat, műveleteket leíró absztraktot nevezhetünk el úgy mint *adding*, *fetching*. Valamint olyan metódusok is lehetnek mint *AddBeer*, *GetBeers*. A domain vezérelt tervezés a következő problémák megoldására használják:

- A projekt középpontjában a program magja, a domain üzleti logika van.
- Komplex folyamatok vannak, amik a domainre épülnek.
- Ha magas fokú együttműködés szükséges a programozók és a domain szakértői között.

A DDD-ben vannak a domain kifejezésére, létrehozására, lekérdezésére használt sajátos artifaktok.

- **Entity** Az entitás olyan objektum aminek van egyedi identitása van, és nem csak az attribútumai határoznak meg. Például, az Európai mozikban a jegy megadott székre szól, de Amerikában nincsenek megkülönböztetve a székek, egy jeggyel bárhol leülhetünk. Ebben az Amerikai mozi kontextusban a szék nem egy entitás, hanem egy érték objektum.
- **Value Object** Az érték objektumot attribútumai határozzák meg, nincs identitása.
- **Aggregate** Az aggregátum entitások és érték objektumok gyűjteménye lehet. minden aggregátumnak van egy gyökere amin keresztül más objektumok hivatkozhatnak az aggregátum elemeire. Az aggregátum gyökere felelős az aggregátum változásainak konzisztenciájának ellenőrzéséért.
- **Domain Event** Egy objektum, ami a domain szemponjából fontos eseményt ír le.

- **Service** Amikor egy művelet vagy folyamat nem tartozik egyetlen objektumhoz sem. A probléma természetét követve ezeket a műveleteket megvalósíthatjuk a szervizekben.
- **Repository** A domain objektumok lekérdezését egy speciális repository objektumra ruházzuk át, hogy alternatív tárolási megoldások esetén könnyen felcserélhessük.
- **Factory** A domain objektumok létrehozását egy speciális factory objektumra ruházzuk át, hogy alternatív megoldások esetén könnyen felcserélhessük.

Hátrányok

Nagyon magas fokú izoláció és enkapszuláció szükséges az üzleti logikában, hogy betartsuk az összes szabályt.

4.4.4. REST

A REST (Representational State Transfer) [8] egy szoftverarchitektúra típus internet alapú rendszerek számára. Egy REST architektúrának meg kell felelni a következő megszorításoknak:

- kliens-szerver architektúra,
- állapotmentesség,
- gyorsítótárazhatóság,
- réteges felépítés,
- egységes interfész.

Kliens-szerver architektúra

A kliens és szerver egymástól független, de egy egységes interfészen keresztül kommunálnak. A kommunikációt mindenkor a kliens kezdeményezi egy kéréssel, a szerver pedig egy válassal válaszol.

Állapotmentesség

A szerver nem tárolja a kliens állapotát a kérések között. minden kérés akármelyik klienstől tartalmazza az összes szükséges információt a kérés kiszolgálásához, az előző kérésektől függetlenül.

Gyorsítótárazhatóság

A válaszoknak ezért impliciten vagy expliciten tartalmazniuk kell, hogy gyorsítótárazhatóak-■ e vagy sem. Egy jó gyorsítótár lehetővé teszi, hogy teljesen megkerüljünk egyes kéréseket, továbbá megnöveli a rendszer skálázhatóságát és a teljesítményét.

Réteges felépítés

Egy kliens nem tudja megmondani hogy direkt csatlakozott-e a szerverhez, vagy közvetítők segítségével. A közvetítő szerverek megnövelhetik a rendszer skálázhatóságát terheléseloszlással és gyorsítótárak használatával.

Egységes interfész

Az egységes interfész kliens és szerver között szétválasztja az architektúrát, és lehetővé teszi, hogy egymástól függetlenül fejlődjenek az egyes részek.

Ha ezeket a megkötéseket teljesíti a webes szolgáltatásunk, azt mondhatjuk hogy "RESTful".

A REST működése

A kliensek kéréseket indítanak a szerverek felé, a szerverek pedig feldolgozzák a kéréseket és egy választ küldenek vissza. A kérések és a válaszok erőforrás reprezentációk köré épülnek. Ezek az erőforrás reprezentációk a mi esetünkben JSON dokumentumok. Az erőforrásokat az URL címével és a HTTP metódussal azonosítjuk. A következő példában láthatjuk ahogy egy PUT metódussal és az URL-ben megadott paraméterrel pontosan tudjuk azonosítani mit szeretnénk. A PUT requestet akkor használjuk ha egy már meglévő erőforrást szeretnénk felülírni, esetünkben az adatbázis konfigurációt kicseréljük. A :name paraméter pedig megnevezi az erőforrást amire a meglévőt ki szeretnénk cserálni. Válaszként egy JSON dokumentumot küldünk vissza válaszként a megfelelő státuszkóddal, hogy a kérés sikeres volt vagy sem.

```

1   router.PUT("/configure/database/:name", func(c echo.Context) error {
2       switch c.Param("name") {
3           case "mongodb":
4               t.ChangeService(m)
5               d.ChangeService(m)
6           case "postgres":
7               t.ChangeService(p)
8               d.ChangeService(p)
9           default:
10              return echo.NewHTTPError(
11                  http.StatusBadRequest,
12                  "no such database supported"
13              )
14      }
15      return c.JSON(http.StatusOK, "configuration complete")
16  })

```

4.4.5. HTTP/2 gRPC összehasonlítása HTTP/1.1 JSON üzenetváltással

REST és HTTP 1.1, JSON üzenetváltás A mikroszervízes infrastuktúra egy nagyon elterjedt módja az elosztott rendszerek tervezésének. Sok mikroszervíz a mai napig

REST API-n kommunikál, HTTP 1.1-es protokollon keresztül és JSON dokumentumokat küldenek és fogadnak. Ez a megoldás a fejlesztőknek kedvez, nagyon egyszerű így fejleszteni manapság, rengeteg eszköz van, hogy meggyorsítsa és megkönnyítse a munkákat. De ez a megoldás a teljesítmény rovására megy, ugyanis a következő problémák adódnak vele.

- A HTTP/1.1 szöveg alapú és nagyon "nehéz". A kommunikáció hatalmas adatmennyiséget igényel, ez egy felesleges teher.
- A HTTP/1.1 állapotmentes, ezért az állapotokat csak a fejlécben tudjuk jelezni, ami nem tömörített.
- A HTTP/1.1 unáris - azaz - egy kérésre egy választ kapsz. Nem lehet egyszerre több kérdést küldeni, minden kérésre pontosan egy válasz jön.
- minden egyes HTTP/1.1 kéréshez három irányú üzenet váltáshoz van szükség, csak hogy létrehozzuk a TCP kapcsolatot, mivel a TCP kapcsolat full duplex, és minden fél szinkronizálja és nyugtázza egymást.

Ebből arra következtetünk, hogy olyan szerver és szerver közötti kommunikációra ahol viszonylag sok apró kérés van, vagy a fenti problémák akadályozzák a programunk működését, akkor érdemes más megoldások után néznünk.

4.4.6. RPC

Az RPC [16] (Remote Procedure Call) egy protokoll, olyan távoli eljárás hívás amelynek segítségével, ugyanabban a hálózatban található távoli gépen futó procedúrákat futtatunk, anélkül, hogy foglalkoznánk a hálózati részletekkel. Az RPC a kliens-szerver modellt használja (a kérő a kliens, a programot futtató fél a szerver).

4.4.7. gRPC

A gRPC [23] egy Google által fejlesztett RPC keretrendszer, főleg mikroszervizek közötti kommunikációra. A gRPC HTTP/2-t használ és Protocol Buffert az üzenetváltáshoz JSON helyett. Ez szembemegy a megszokott mikroszervizes architektúra stílussal ami REST-re épül JSON üzenetváltásal HTTP/1.1-en. Triviális, hogy a legfontosabb előnye az, hogy HTTP/2-n fut és az üzenetváltás Protocol Bufferekkel történik így sokkal gyorsabb és hatékonyabb.

HTTP/1.1 és HTTP/2 összehasonlítása A HTTP/2 egy sokkal hatékonyabb protokoll, a streameléssel elérhetjük azt, hogy az egyik fél több üzenetet is küld, a másik fél viszont csak a kérések legvégén válaszol.

Protocol Buffer és JSON összehasonlítása Beláthatjuk, hogy a Protocol Buffer adatforgalom kontextusban és hardveres erőforrás kontextusban is egy sokkal hatékonyabb eszköz üzenetek továbbítására mint a JSON, XML, stb hagyományos formátumok. Mivel nem szöveges, lehet hogy nehezebb implementálni és debugolni a fejlesztőknek, viszont kevesebb adatforgalommal jár, és a számítógép erőforrásait is kíméli, nem úgy mint a JSON.

4.1. táblázat. HTTP/1.1 és HTTP/2

HTTP/1.1	HTTP/2
Szöveges formátum	Bináris formátum
Szöveges, nem tömörített fejléc	Tömörített fejléc
1 kérés, 1 válasz TCP kapcsolatonként	1 TCP kapcsolatot újra felhasználunk, Unáris kérések, Szerver streamelés, Kliens streamelés, Két-irányú streamelés

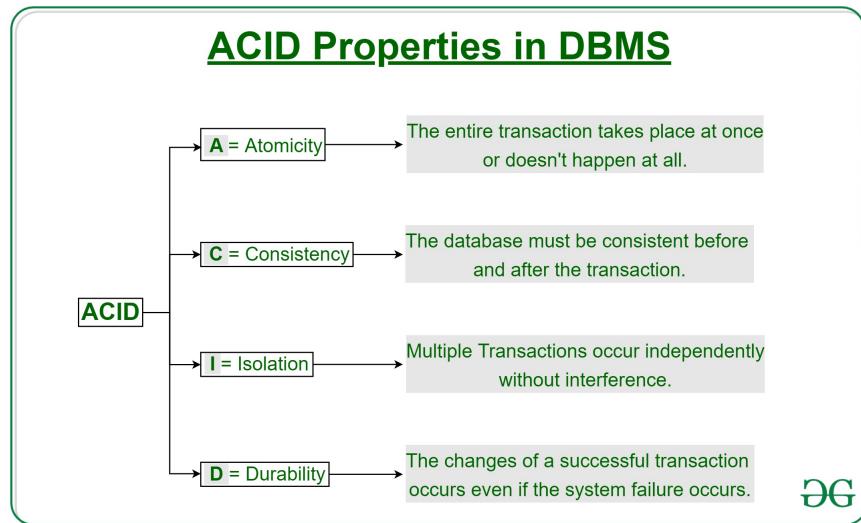
4.2. táblázat. JSON és Protocol Buffer

JSON	Protocol Buffer
Nincs szigorú séma definíció vagy típus	Szigorú séma formátum és típus biztonság
Szöveg alapú	Bináris
A szöveg formátum miatt lassú szerializáció és deszerializáció CPU és memória intenzív	Bináris formátum miatt gyors szerializáció és deszerializáció
Adatok manuális konvertálása	Generált kód a protokol buffer séma alapján

4.4.8. Adatbázisok

Az adatok adatbázisba mentése, adatbázisból olvasása közben több probléma merülhet fel, a rendszer konkurrens felépítéséből kiindulva. Például, amikor az épp szabad drónoknak adunk feladatot, lehet hogy 2 vagy több egyede az adatközpont programunknak kiolvassa azt az értéket, hogy a drón nem csinál semmit, az állapota szabad, adhat neki feladatot, ha van szállítsra váró csomag. De, az üzleti logika futtatása közben az egyik egyed módosíthatta az értéket arra, hogy már repül, vagy épp csomagot vesz fel. Az ilyen Lost Update problémákkal szemben védelmet ad, ha az adatbázis rendelkezik ACID tulajdonságokkal. Az ACID tulajdonságok (atomiság, konzisztencia, izoláció, tartósság) felelősek az adatbázis integritásáért (4.8. ábra).

Tehát csak olyan adatbázisok jöhettek szóba a program implementálásánál, aminek az integritása garantálható. Hogy két eltérő adatbázist hasonlítsunk össze, egy SQL és egy NoSQL adatbázis ami teljesíti az ACID integritást megfelelő lesz. A PostgreSQL adatbázis kiváló relációs adatbázis, nem véletlen, hogy az egyik legelterjedtebb. Teljesíti az ACID tulajdonságokat, és már 35 éve használják, így biztosan megfelelően tesztelt. A MongoDB dokumentum alapú adatbázis ACID, de csak dokumentum szinten. A mi szimulációinkhoz viszont így is megfelel a követelményeknek. Érdekességeként, a legtöbb dokumentum alapú adatbázis nem ACID, ezért nem annyira népszerűek. A dokumentum alapú adatbázisok általában jobban teljesítenek ha nagyon sok írás történik lekérdezés pedig kevés, vagy ha nincs bonyolult lekérdezés ami több JOIN-t igényel.



4.8. ábra. ACID tulajdonságok Forrás: <https://www.geeksforgeeks.org/acid-properties-in-dbms/>

4.4.9. Protokollok és adatbázisok szerepe a programban

A drón raj programot és az adatközpontot úgy tervezzük meg, hogy a telemetria adatok küldéséhez használt rész kicsérélhető legyen valamilyen módon (például dependency injekción keresztül). Így tudjuk összehasonlítani a több protokollon keresztüli kommunikációt. Az adatbázis végpontot is hasonlóan ki kell tudnunk cserélni, hogy ne kelljen módosítani a programon a különböző szimulációkhöz. A következő kód példában láthatjuk, hogy működhet a dependency injekció. Itt a PostgreSQL adatbázissal hoztuk létre a szervízt, de a MongoDB adatbázissal is létrehozhatjuk, ha teljesíti az interfész követelményeit.

```

1  postgresStorage, err := postgres.NewStorage(config.PostgresConfig)
2  if err != nil {
3      log.Println("Error connecting to database")
4      panic(err)
5  }
6  mongoStorage, err := mongodb.NewStorage(config.MongoConfig)
7  if err != nil {
8      log.Println("Error connecting to database")
9      panic(err)
10 }
11 var ts telemetry.Service
12 var ds drone.Service
13 var rs routing.Service
14 ts = telemetry.NewService(postgresStorage, logger)
15 rs = routing.NewService(logger)
16 ds = drone.NewService(postgresStorage, jsonAdapter, logger, rs)

```

5. fejezet

Szimuláció

5.1. Adatbázisok és modellek

A programban a drón, és telemetria adatmodell a legfontosabb. Ezeket az adatokat olvassuk és mentjük, illetve szükségesek az azonosításhoz vagy bármilyen értelmes következtetéshez a problémához kapcsolódóan.

5.1.1. Modellek az adatközpont programban

Telemetria modell

A programban a telemetria modell a következőképpen néz ki:

```
1 package models
2
3 import "time"
4
5 type Telemetry struct {
6     Speed           float64      'json:"speed" db:"speed"'
7     Location        GPS          'json:"location"'
8     Altitude        float64      'json:"altitude"'
9     Bearing         float64      'json:"bearing"'
10    Acceleration   float64      'json:"acceleration"'
11    BatteryLevel   int          'json:"battery_level"'
12    BatteryTemperature int        'json:"battery_temperature"'
13    MotorTemperatures []int      'json:"motor_temperatures"'
14    Errors          []TelemetryError 'json:"errors" db:"errors"'
15    TimeStamp       time.Time    'json:"time_stamp"'
16    DroneID         int          'json:"drone_id"'
17 }
18
19 type TelemetryError int
20
21 const (
22     MotorFailure TelemetryError = iota
23     BeaconSignalStrengthLow
24     BeaconSignalInterference
```

```

25     BeaconTemperatureTooHigh
26     GPSInt eference
27     GPSSignalLost
28     GPSTemperatureTooHigh
29     ProcessorTemperatureTooHigh
30     BatteryFailure
31     FailedToEjectPackage
32     PackageLost
33     DestinationDistanceTooFar
34 )
35
36 type GPS struct {
37     Latitude float64 `json:"latitude" bson:"latitude"`
38     Longitude float64 `json:"longitude" bson:"longitude"`
39 }
```

Drón modell

A DDD-ben leírt elvek alapján, a drón modell mindenféleképp egy Entity-nek felel meg.

```

1  type Drone struct {
2      ID          int           `json:"id" db:"drone_id" bson:"id"`
3      Telemetry   Telemetry    `json:"telemetry" bson:"telemetry"`
4      Parcel      Parcel       `json:"parcel"`
5      Destinations []Destination `json:"destinations"`
6      Consumption float64      `json:"consumption"`
7      Weight      float64      `json:"weight"`
8      State       DroneState   `db:"state" bson:"state"`
9  }
10
11 type DroneState string
12
13 const (
14     DroneFree    DroneState = "free"
15     DroneInFlight DroneState = "in-flight"
16 )
```

Parcel modell (szállítandó csomag)

A drónok által szállított csomag így néz ki:

```

1  type Parcel struct {
2      ID          int           `json:"id" db:"id" bson:"id"`
3      TrackingID string        `json:"tracking_id" `
4      Name        string        `json:"name" db:"name" bson:"name" `
```

```

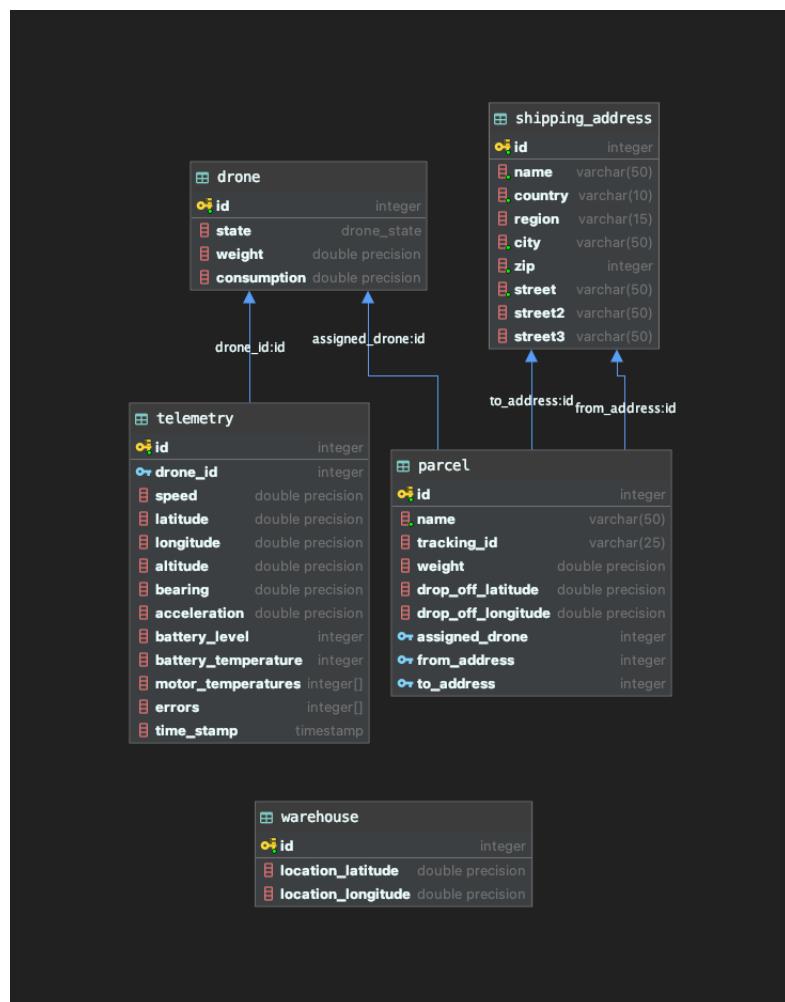
5     Weight      float64           'json:"weight" db:"weight"'
6     Location    GPS               'json:"location" bson:"location"'
7     FromAddress ShippingAddress 'json:"from_address"'
8     ToAddress    ShippingAddress 'json:"to_address"'
9     DropOffSite GPS              'bson:"drop_off_site" '
10    AssignedDrone int             'json:"assigned_drone" '
11
}

```

5.1.2. Relációs adatbázis, PostgreSQL

A relációs adatbázissal is működik a szimuláció.

5.1.3. Relációs modell (5.1. ábra)



5.1. ábra. PostgreSQL adatbázis modell

Lost Update probléma

A legtöbb relációs adatbázis támogatja a tranzakciókat. A tranzakciók ACID tulajdon-ságokkal bírnak. Adatbázisok esetén az ACID az Atomicity (atomiság), Consistency (konziszenciá), Isolation (izoláció), és Durability (tartósság) rövidítése. Ezek nélkül az adatbázis integritása nem garantálható. A PostgreSQL többféle izolációs szintet biztosít a tranzakciókhöz [1]. A Lost Update problémát garantáltan megoldja a PostgreSQL *Serializable* izolációs szintje (5.1.3. ábra).

5.1. táblázat. Standard SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not Possible

PostgreSQL telepítése az alkalmazáshoz

Ha nem található a fejlesztői környezetünkben PostgreSQL adatbázis, akkor telepíteni kell azt. Az adatbáziskezelőhöz csatlakozva, le kell futtatni a következő SQL scriptet.

```
1 CREATE TYPE drone_state AS ENUM ('free', 'in-flight');
2 CREATE TABLE drone
3 (
4     id      SERIAL PRIMARY KEY,
5     state    drone_state      DEFAULT 'free',
6     weight   DOUBLE PRECISION default 4,
7     consumption DOUBLE PRECISION DEFAULT 500
8 );
9
10 CREATE TABLE warehouse
11 (
12     id      SERIAL PRIMARY KEY,
13     location_latitude DOUBLE PRECISION DEFAULT 48.080922,
14     location_longitude DOUBLE PRECISION DEFAULT 20.766208
15 );
16
17 CREATE TABLE shipping_address
18 (
19     id      SERIAL PRIMARY KEY,
20     name    VARCHAR(50) NOT NULL,
21     country VARCHAR(10) NOT NULL,
22     region   VARCHAR(15) DEFAULT NULL,
23     city     VARCHAR(50) NOT NULL,
24     zip      INT      NOT NULL,
25     street   VARCHAR(50) NOT NULL,
26     street2  VARCHAR(50) DEFAULT NULL,
27     street3  VARCHAR(50) DEFAULT NULL
28 );
29
30 CREATE TABLE parcel
31 (
```

```

32     id          SERIAL PRIMARY KEY,
33     name        VARCHAR(50) NOT NULL,
34     tracking_id VARCHAR(25)           default '',
35     weight       DOUBLE PRECISION      default 1,
36     drop_off_latitude DOUBLE PRECISION    DEFAULT 0,
37     drop_off_longitude DOUBLE PRECISION   DEFAULT 0,
38     assigned_drone  INT REFERENCES drone (id) DEFAULT NULL,
39     from_address   INT REFERENCES shipping_address (id),
40     to_address     INT REFERENCES shipping_address (id)
41 );
42
43 CREATE TABLE telemetry
44 (
45     id          SERIAL PRIMARY KEY,
46     drone_id    INT REFERENCES drone (id),
47     speed        DOUBLE PRECISION DEFAULT 0,
48     latitude     DOUBLE PRECISION DEFAULT 0,
49     longitude    DOUBLE PRECISION DEFAULT 0,
50     altitude     DOUBLE PRECISION default 1,
51     bearing      DOUBLE PRECISION DEFAULT 0,
52     acceleration DOUBLE PRECISION DEFAULT 0,
53     battery_level INT                  DEFAULT NULL,
54     battery_temperature INT             DEFAULT NULL,
55     motor_temperatures INTEGER[] ,
56     errors       INTEGER[] ,
57     time_stamp    timestamp           DEFAULT NULL
58 );
59 INSERT INTO warehouse (id) VALUES (1);

```

5.1.4. Dokumentum alapú adatbázis, MongoDB

MongoDB telepítése az alkalmazáshoz

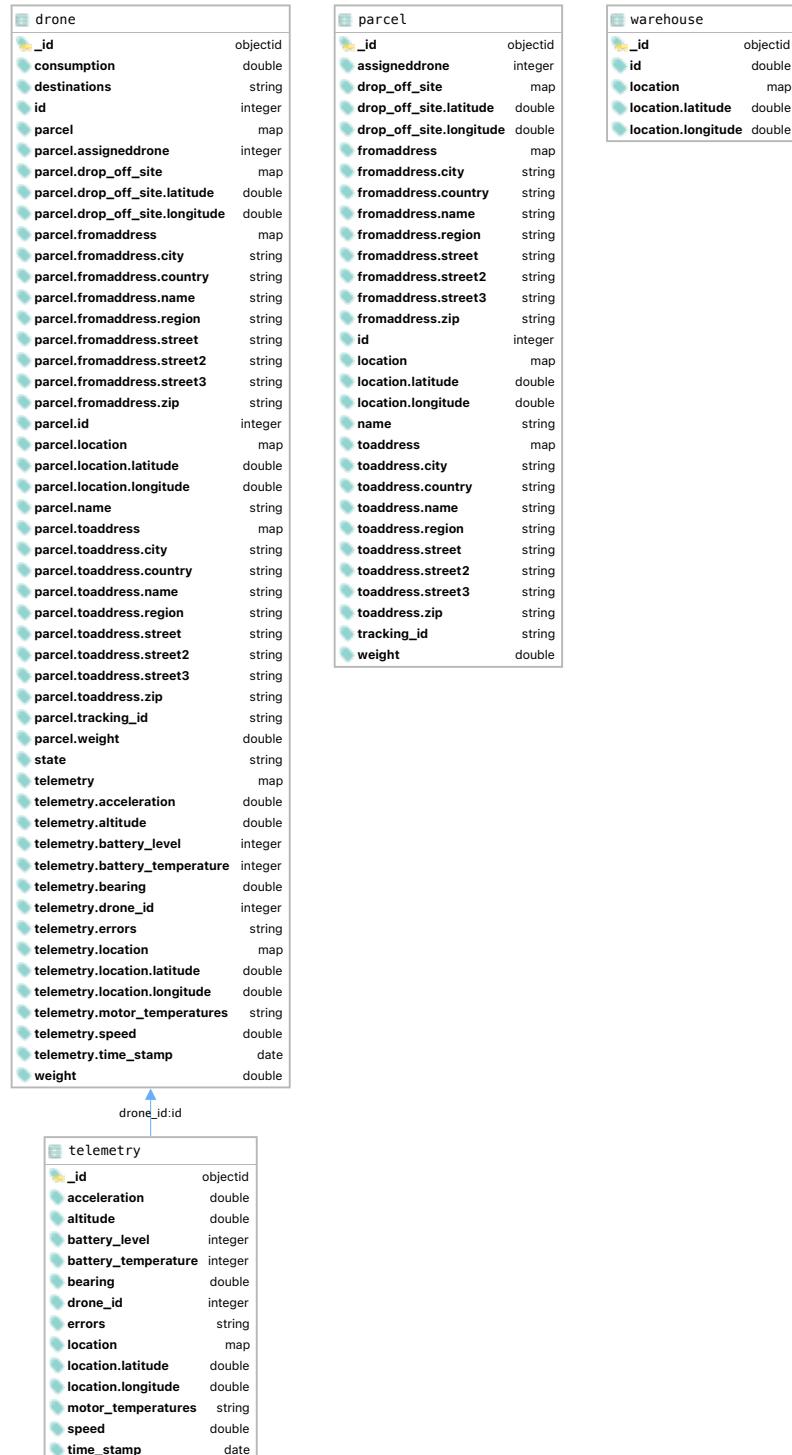
Ha nem található a fejlesztői környezetünkben MongoDB adatbázis, akkor telepíteni kell azt. Az adatbáziskezelő konzolján keresztül le kell futtatni a következő két parancsot, egy megfelelő jogosultságú felhasználóval. Ez egyik parancs létrehozza a *drone_delivery* adatbázist, a másik parancs létrehoz nekünk egy felhasználót az előbb létrehozott adatbázishoz. A későbbiekben ezzel a felhasználóval csatlakozunk az adatbázishoz.

```
1 use drone_delivery
```

```

1
2 db.createUser(
3   {
4     user: "drone-user",
5     pwd: "drone-pwd",
6     roles: [
7       {
8         role: "readWrite",

```



5.2. ábra. MongoDB adatbázis vizualizáció

```

9   db: "drone_delivery"
10  }
11  ]
12 }
```

13

)

5.2. Szimuláció működésének folyamata

A szimuláció működése azzal kezdődik, hogy elindítjuk az adatközpont programot. Felcsatlakozik az adatbázisokra, majd létrehozza a szervizeket, és a REST API-n várja a bejövő kéréseket. Ez lehet például egy konfigurációs kérés, a drónok lekérdezése, vagy a szállítás indítása.

Közben a drón-raj program is elindul, ez a program is egy REST API-n várja a kéréseket. A drón raj program az API-n megkapja a drónokat, és a hozzájuk rendelt csomagokat, majd megkezdődik a szimuláció, a drón-raj program telemetria adatokat küld az adatközpont programnak. A drón-raj program olyan telemetria adatokat fog generálni, amelyek minél jobban a valóságot tükrözik és ezeket az adatokat tovább fogja küldeni az adatközpontnak.

```

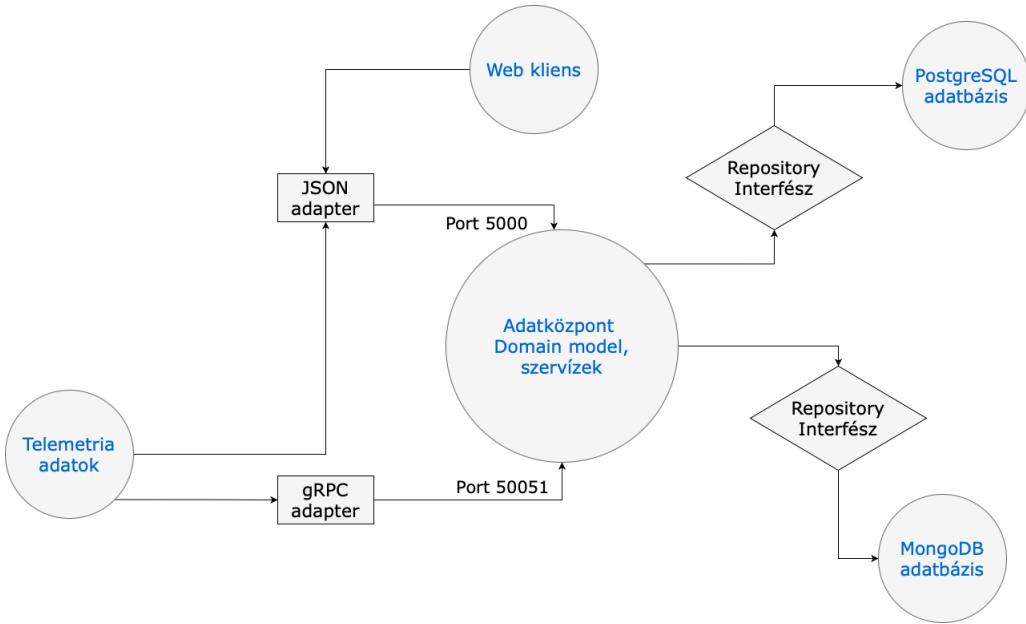
1  {
2      "telemetry": {
3          "speed": 2,
4          "location": {
5              "latitude": 48.08092020178216,
6              "longitude": 20.766208061642047
7          },
8          "altitude": 50,
9          "bearing": 178.68412647009515,
10         "acceleration": 0,
11         "battery_level": 98,
12         "motor_temperatures": [
13             41,
14             43
15         ],
16         "time_stamp": "2021-04-01T15:04:05.630Z",
17         "drone_id": 2
18     }
19 }
```

Ezeket a telemetria adatokat az adatközpont program fogadja a megadott porton és lementi az adott adatbázisba (5.3. ábra).

5.3. A rendszer felépítése a követelmények alapján

Szerkezeti felépítés

Az alkalmazás jegyzék struktúráját a 5.4. ábra mutatja. A *drone-delivery* jegyzék a program gyökér jegyzéke, itt található a *docker-compose.yml* fájl, ami az indításhoz



5.3. ábra. Adatközpont program működésének folyamata

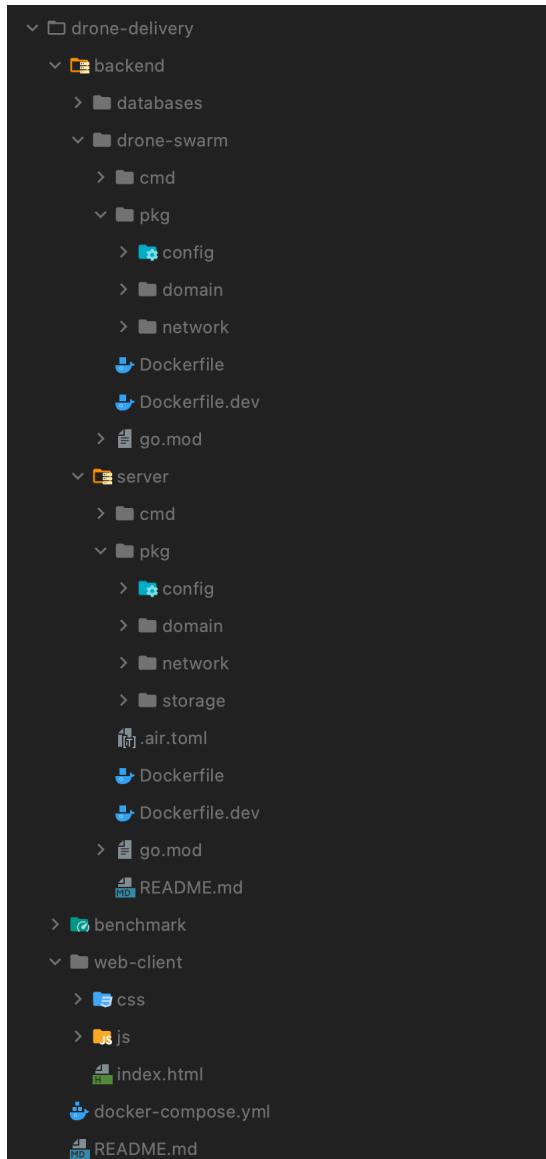
szükséges. A *backend/server* tartalmazza az adatközpont programot és a *backend/drone-swarm* a drón-raj programot. Mindkét jegyzéknek hasonló a felépítése. A *cmd* jegyzékbe van az indító *main* függvény. A *pkg* jegyzék tartalmazza a kódbázis nagy részét, a domain és infrastuktúrához kapcsolódó részek itt helyezkednek el. A *pkg/domain*-ben a domain kódbázis található, az infrastruktúrát és a hozzá tartozó adaptereket megvalósító kód a *storage*, *network* jegyzékekben van. A *drone-delivery/web-client*-ben található a web kliens, egy egyszerű HTML oldal ahonnan beállítjuk a szimulációt.

Portok és Adapterek, Domain Driven Design

A hexagonal tervezési architektúra minden programban úgy került implementálásra, hogy a program magja, a domain üzleti logika a *pkg/domain* jegyzékben helyezkedik el. Ez a domain jegyzék modellezte le az üzleti logika működését. Itt találhatóak az interfések, amik a hexagonal architektúra portjaiként szolgálnak, ezt kell implementálniuk az adaptereknek, így van összekötve a domain a különböző adapterekkel. A DDD elemeit megfigyelhetjük az elnevezésekben, valamint vannak Entity-k és Value Objectek.

```

1  # Entity
2  type Drone struct {
3      ID           int          `json:"id" db:"drone_id" bson:"id"`
4      Telemetry    Telemetry    `json:"telemetry" bson:"telemetry"`
5      Parcel       Parcel       `json:"parcel"`
6      Destinations []Destination `json:"destinations"`
7      Consumption   float64     `json:"consumption" db:"consumption"`
8      bson:"consumption"`
9      Weight       float64     `json:"weight" db:"weight" bson:"weight"`
10     State        DroneState  `db:"state" bson:"state"`
11 }
  
```



5.4. ábra. Jegyzék szerkezet

```
12 # Value Object
13 type Destination struct {
14     Coordinates           GPS
15     ParcelDestination    bool
16     WarehouseDestination bool
17 }
18
19 # Value Object
20 type GPS struct {
21     Latitude float64 `json:"latitude" bson:"latitude"`
22     Longitude float64 `json:"longitude" bson:"longitude"`
23 }
```

Portok

Minden szervízhez tartozik interface (port) amin az adapterek vagy más szervízek meg-hívhatják a szervíz metódusait. Például az adatközpont drón szervízében olyan metódusok találhatóak amiket a REST API hív meg. Itt a REST API az adapter, pontosabban vezérlő adapter.

```

1  type Service interface {
2      DeliverParcels() error
3      ProvisionDrone(wh models.Warehouse, d models.Drone) error
4      GetFreeDrones() ([]models.Drone, error)
5      GetDronesDelivering() ([]models.Drone, error)
6      ChangeService(r Repository)
7      ReinitializeDatabase(repos ...Repository) error
8  }
```

Vezérelt adapterek az adatbázist és kimenő kommunikációt megvalósító adapter. Az adatközpont programban az adatbázisoknak a Repository interfészét kell implementálni, de a gRPC és JSON kommunikációhoz is tartozik interfész.

```

1  type Repository interface {
2      GetFreeDrones() ([]models.Drone, error)
3      GetParcelsInWarehouse() ([]models.Parcel, error)
4      GetWarehouse() (models.Warehouse, error)
5      GetDronesDelivering() ([]models.Drone, error)
6      SetDroneState(droneID int, state string) error
7      ReInitializeDeliveryData(drones []models.Drone, parcels []models.
8          Parcel) error
9  }
10
11 type OutboundAdapter interface {
12     FetchProvisionDroneEndpoint(wh models.Warehouse, d models.Drone)
13     (success bool, err error)
14 }
```

Ugyanígy a drón-raj programban.

```

1  type OutboundAdapter interface {
2      SendTelemetryDataToServer(t models.Telemetry) error
3  }
```

Adapterek

Az adaptereket a portokon keresztül érjük el. Alább a drón-raj program Outbound-Adapter interfészét megvalósító adapter implementációját látjuk, gRPC-vel. Csak azokat a részeket tartalmazza a példa ami szükséges a megértéshez. Mivel a

```
1     SendTelemetryDataToServer(t models.Telemetry) error
```

metódus megtalálható az adapterben, az interfész teljesül és az adapter használható a porton keresztül.

```
1  package grpc
2
3  ...
4
5  type Adapter struct {
6      cc *grpc.ClientConn
7      tsc protobuf.TelemetryServiceClient
8      streams map[int]StreamClient
9  }
10
11 }
12
13 func (a *Adapter) SendTelemetryDataToServer(t models.Telemetry) error {
14
15     var err error
16
17     var streamer protobuf.TelemetryService_TelemetryStreamClient
18
19     ...
20
21     telemetryDataRequest := protobuf.TelemetryDataRequest{
22         Telemetry: &protobuf.Telemetry{
23             Speed: t.Speed,
24             Location: &protobuf.GPS{
25                 Latitude: t.Location.Latitude,
26                 Longitude: t.Location.Longitude,
27             },
28             Altitude: t.Altitude,
29             Bearing: t.Bearing,
30             Acceleration: t.Acceleration,
31             BatteryLevel: int32(t.BatteryLevel),
32             BatteryTemperature: int32(t.BatteryTemperature),
33             MotorTemperatures: temperatures,
34             Errors: telemetryErrors,
35             TimeStamp: timestamppb.New(t.TimeStamp),
36             DroneId: int32(t.DroneID),
37         },
38     }
39     err = streamer.Send(&telemetryDataRequest)
40 }
```

Az adatközpont program MongoDB és PostgreSQL adatbázisához kapcsolódó, a Repository interfész megvalósító adapterek a *pkg/storage* jegyzékben helyezkednek el.

Dependency Injection

A Hexagonal architektúra elvárja a kicsérélhetőséget, laza kapcsolatokat és hogy az alkalmazás komponensek portokon kapcsolódjanak össze. Ennek más előnyei is vannak, így lényegesen könnyebb a tesztelhetőség és karbantartás. Arról már volt szó, hogy mik az adapterek és hogyan használjuk őket a portokon keresztül. De arról még nem volt szó, hogy pontosan hogyan adjuk át az adaptereket, hogyan hivatkozunk az adapterekre. Ehhez dependency injekciót használunk. Azaz létrehozunk egy adaptert, szolgáltatást (szervíz) és azt átadjuk egy másik szolgáltatásnak, adapternek. Olyan is előfordulhat, hogy a az alkalmazás domain részében két szolgáltatás egymás között portokon kommunikál. Például a drón szolgáltatás létrehozásához szükséges paramétereket az alábbi kód részletben láthatjuk. Ezekből a paraméterekből a *Repository*, *OutboundAdapter* típusúak portok más adapterekhez. Később a szolgáltatásban ezekre a portokra hivatkozunk, ezeken keresztül érjük el az adapterek metódusait.

```

1 func NewService(r Repository, ea OutboundAdapter, l gokitlog.Logger,
2 rs routing.Service) *service {
3     return &service{r, ea, l, rs}
4 }
```

Az alábbi példán láthatjuk, hogy a PostgreSQL, mongoDB adatbázisokat implementáló, és a külső kommunikációért felelős JSON adaptert átadjuk különböző szolgáltatásoknak. Ezek rendre *postgresStorage*, *mongoStorage*, *jsonAdapter*.

```

1 postgresStorage, err := postgres.NewStorage(config.PostgresConfig)
2 ...
3 mongoStorage, err := mongodb.NewStorage(config.MongoConfig)
4 ...
5 jsonAdapter := outbound.NewJSONAdapter()
6 geo := ellipsoid.Init("WGS84", ellipsoid.Degrees, ellipsoid.Meter,
7 ellipsoid.LongitudeIsSymmetric, ellipsoid.BearingIsSymmetric)
8 ...
9 ts = telemetry.NewService(postgresStorage, logger, geo)
10 rs = routing.NewService(logger)
11 ds = drone.NewService(postgresStorage, jsonAdapter, logger, rs)
```

Majd a szolgáltásokat átadjuk egy vezérlő adapternek, egy REST API-nak ami kéréseket fogad és ez alapján a szolgáltatások portjain a megfelelő metódust meghívja.

```
1 router := rest.Handler(ds, ts, postgresStorage, mongoStorage)
```

Konfiguráció

A konfiguráció beállításáért a *config* csomag felel. Az adatközpont, és drón-raj programban is található egy *SetConfig* függvény a config csomagban. Ez környezeti változókat olvas ki, és az alapján beállítja a megfelelő értékeket. A környezeti változókat a *docker-compose.yml* fájlban tudjuk módosítani.

```

1 func SetConfig() {
2     flag.StringVar(&DroneSwarmURL, "drone swarm domain url", os.
3     Getenv("DRONE_SWARM_URL"),
4         "An url for the drone swarm application, with protocol")
5
6     PostgresConfig.UserName = os.Getenv("PGUSER")
7     PostgresConfig.Database = os.Getenv("PGDATABASE")
8     PostgresConfig.Host = os.Getenv("PGHOST")
9     PostgresConfig.Port = os.Getenv("PGPORT")
10    PostgresConfig.SSLMode = os.Getenv("PGSSL")
11    PostgresConfig.PW = os.Getenv("PGPASSWORD")
12
13    MongoConfig.UserName = os.Getenv("MONGO_USER")
14    MongoConfig.Database = os.Getenv("MONGO_DB")
15    MongoConfig.Host = os.Getenv("MONGO_HOST")
16    MongoConfig.Port = os.Getenv("MONGO_PORT")
17    MongoConfig.PW = os.Getenv("MONGO_PWD")
18 }
```

```

1 func SetConfig() {
2     flag.StringVar(&ServerHTTPDomain, "server domain for http", os.
3     Getenv("SERVER_DOMAIN"), "domain of server, with protocol")
4     flag.StringVar(&ServerHTTPPort, "server port for http", os.Getenv(
5         "SERVER_PORT"), "the port the server is listening on")
6     flag.StringVar(&ServerGRPCDomain, "server domain for grpc", os.
7     Getenv("SERVER_GRPC_DOMAIN"), "domain of server, with protocol")
8     flag.StringVar(&ServerGRPCPort, "server port for grpc", os.Getenv(
9         "SERVER_GRPC_PORT"), "the port the server is listening on")
10
11     PostgresConfig.UserName = os.Getenv("PGUSER")
12     PostgresConfig.Database = os.Getenv("PGDATABASE")
13     PostgresConfig.Host = os.Getenv("PGHOST")
14     PostgresConfig.Port = os.Getenv("PGPORT")
15     PostgresConfig.SSLMode = os.Getenv("PGSSL")
16     PostgresConfig.PW = os.Getenv("PGPASSWORD")
17 }
```

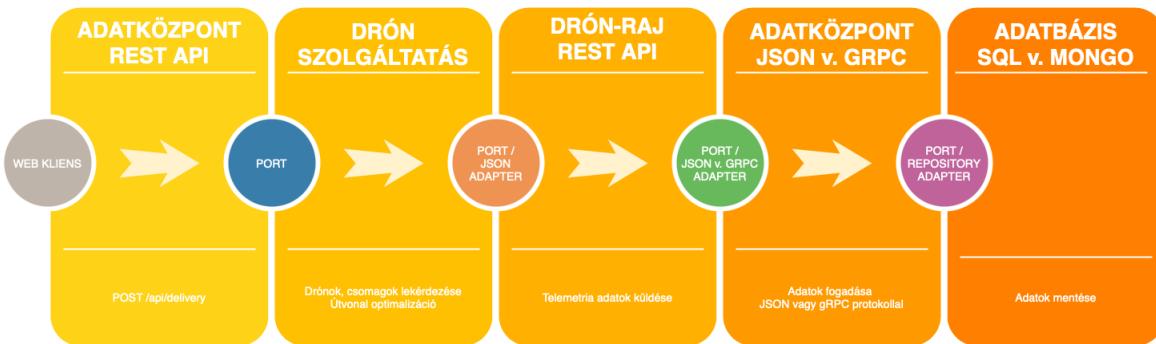
5.4. Az rendszer működése

A rendszer úgy működik, hogy először elindul az adatközpont program, felcsatlakozik az adatbázisokra, létrejönnek a vezérelt adapterek. Ezeket az adaptereket átadjuk a szolgáltatásoknak. Létrehozzuk a vezérlő adaptereket, ezeknek az adaptereknek átadjuk a szolgáltatásokat. A két vezérlő adapter az adatközpontban a REST API ami 5000-es

porton hallgat, illetve a gRPC végpont ami 50051-es porton hallgat. A drón raj program elindul, hasonló módon mint az adatközpont programban, először a vezérelt adapterek, majd a vezérlő adapterek jönnek létre. Létrejön a gRPC kapcsolat az adatközpont és drón-raj program között. A drón raj program REST API-ja a 2000-es porton hallgat. A web klienssel indíthatjuk a szimulációt, miután megnyitottuk a böngészővel, egy egyszerű kezelő felület fogad.

A folyamat (5.5. ábra) röviden úgy néz ki, hogy:

1. A web kliensben kiválasztjuk, hogy milyen protokollt és formátumot, valamint adatbázist szeretnénk használni. Alapértelmezettként PostgreSQL adatbázis és HTTP/1.1 protokoll JSON formátummal van kiválasztva.
2. A szimuláció indítása gomb megnyomásával az adatközpont program REST API `/api/delivery` végpontjára küld egy POST requestet a web kliens.
3. Az adatközpont lekérdezi a drónokat és csomagokat, majd ezek alapján optimalizálja az útvonalakat. A drónoknak most már megvan a hozzájuk tartozó csomagjuk és az útvonal amin haladniuk kell.
4. Ezt az információt az adatközpont elküldi egy POST requestben a drón raj program REST API `/provision` végpontjára.
5. A drón-raj program szimulálja a drónok működését, és telemetria adatokat küld az adatközpontnak JSON vagy gRPC formátumban, attól függ mit választottunk.
6. Az adatközpont elmenti az adatokat PostgreSQL vagy MongoDB adatbázisba, attól függ melyikbe, hogy mit választottunk.



5.5. ábra. Az szimuláció folyamata

5.5. Telepítés, tesztelés

Adatbázisok elindítása

A rendszer indítása úgy kezdődik, hogy megbizonyosodunk arról, hogy a MongoDB adatbázis és a PostgreSQL adatbázis fut és lehet rá csatlakozni. Ha ez a 2 adatbáziskezelő nem található meg gépünkön, akkor először telepíteni kell őket. Nálam a következőképp néz ki az indítás. A terminálban elindítom a MongoDB adatbázist.

```
1 $ brew services stop mongodb-community@4.4
```

Ezután a mongo parancsot kiadva kapcsolóm az adatbáziskezelő konzolos felületére. Megbizonyosodom, hogy létezik a drone_delivery adatbázis. Ha nem létezik, lefuttatom a telepítéshez szükséges parancsokat (5.1.4 szakasz).

```
1 $ mongo
```

Ezután elindítok egy PostgreSQL 11 vagy újabb verziójú PostgreSQL adatbázist, a fejlesztői környezetben ez egy Postgres nevű grafikus kezelőfelületű program által történik. Megbizonyosodok arról, hogy létezik a dbdrone_delivery adatbázis, és létezenek a megfelelő táblák. Ha nem léteznek, lefuttatom telepítéshez szükséges scriptet (5.1.3 szakasz).

Alkalmazás indítása Dockerrel

Az alkalmazást Docker konténerben indítom el a teszteléshez, így minden fejleszői környezetben elindul, ahol van telepítve a Docker. Az indításhoz elengedhetetlen 2 feltétel teljesülése: az adatbázisok futnak, és a *docker-compose.yml* fájlban a megfelelő konfiguráció van beállítva. A *drone-delivery* jegyzékben, a terminálban kiadom a következő parancsot.

```
1 $ docker-compose up --build
```

Ez megépíti a konténereket, letölti a szoftver függőségeket illetve csomagokat, majd a konténerekben elindulnak a programok (5.6. ábra). Ezután nincs más dolgunk, mint a *drone-delivery/web-client* jegyzékben található *index.html* fájlt megnyitni a böngészőben, az egyértelmű kezelőfelülettel tudjuk konfigurálni és indítani a szimulációt. Egy 3D diagramon nézhetjük ahogy a drónok mozognak (5.7. és 5.8. ábra).

5.6. Szimuláció működésének folyamata

```

docker-compose (do... 11) .erezek-domain (zsh) 12 ..cts/probby.hu (zsh) 13 ...plugin-master (z... 14 ..bexec/lib/ext (zsh) 15 ~ (zsh) 16 ..jmeter/mongo (zsh) 17 + 0.0s
=> => naming to docker.io/library/drone-delivery_server
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Successfully built 7f18a79deb3298bfec86903493100585b3264fa09877dea92fbffffcb7e5a8
Building drone-swarm
[+] Building 52.0s (15/15) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 37B
=> [internal] load context: 2B
=> [internal] load metadata for docker.io/library/golang:1.16
=> CACHED [ 1/10] FROM docker.io/library/golang:1.16@sha256:07a471a939cd942ed5c23714185dac476df214f94cf073ef2fbfa748d4926a48
=> [internal] load build context
=> => transferring context: 5.55kB
=> [ 2/10] RUN mkdir -p /go/src/drone-delivery/drone-swarm
=> [ 3/10] WORKDIR /go/src/drone-delivery/drone-swarm
=> [ 4/10] COPY drone-swarm/go.mod /go/src/drone-delivery/drone-swarm
=> [ 5/10] COPY drone-swarm/go.sum /go/src/drone-delivery/drone-swarm
=> [ 6/10] COPY ./server/ /go/src/drone-delivery/server
=> [ 7/10] COPY ./ /go/src/drone-delivery/
=> [ 8/10] WORKDIR /go/src/drone-delivery/drone-swarm/cmd/drone-delivery-swarm
=> [ 9/10] RUN go mod download
=> [10/10] RUN go install
=> exporting to image
=> => exporting layers
=> => writing image sha256:8f4c1f38e96b6826778beb7ff75feabde725e2ae09c325d61e3d16f8f8fd602
=> => naming to docker.io/library/drone-delivery_drone-swarm

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Successfully built 8f4c1f38e96b6826778beb7ff75feabde725e2ae09c325d61e3d16f8f8fd602
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Recreating drone-delivery_server_1 ... done
Recreating drone-delivery_drone-swarm_1 ... done
Attaching to drone-delivery_server_1, drone-delivery_drone-swarm_1
drone-swarm_1    ez a drón-raj a szimulacióban, a drónok példányait szimulálja.
server_1         ez lesz a szerver, az adatközpont a szimulacióban
server_1         2021/05/07 18:28:03 You are connected to your database
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] parsed scheme: ""
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] scheme "" not registered, fallback to default scheme
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] ccResolverWrapper: sending update to cc: [[{server:50051 <nil>} <nil> <nil>]]
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] ccResolverWrapper: sending update to cc: [[{server:50051 <nil>} <nil> <nil>]]
server_1        2021/05/07 18:28:03 You are connected to your database
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] ClientConn switching balancer to "pick_first"
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] Channel switches to new LB policy "pick_first"
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] Subchannel Connectivity change to CONNECTING
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] Subchannel picks a new address "server:50051" to connect
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] pickfirstBalancer: UpdateSubConnState: 0xc0001e58c0, {CONNECTING <nil>}
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] Channel Connectivity change to CONNECTING
server_1        2021/05/07 18:28:03 serving http on port: 5000...
server_1        2021/05/07 18:28:03 serving grpc on port 50051
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] Subchannel Connectivity change to READY
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] pickfirstBalancer: UpdateSubConnState: 0xc0001e58c0, {READY <nil>}
drone-swarm_1   INFO: 2021/05/07 18:28:06 [core] Channel Connectivity change to READY

```

5.6. ábra. Docker build, elindulnak a programok

5.6. Szállítási probléma a programban

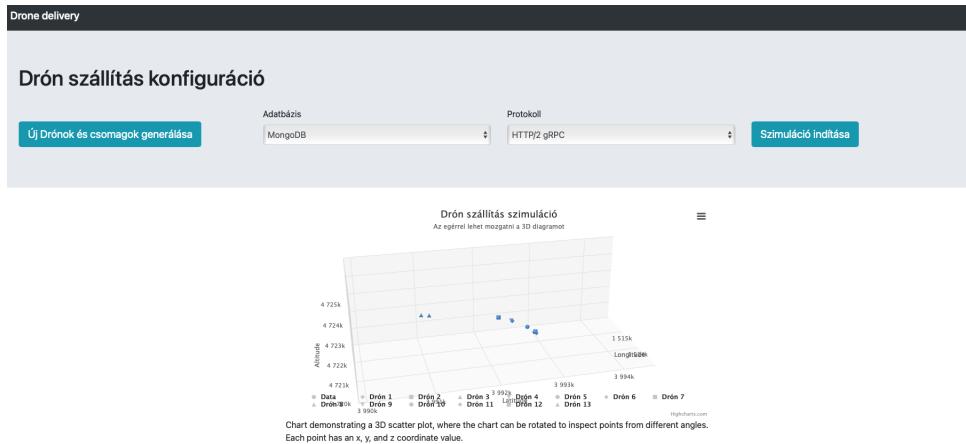
Az adatközpont távolság számítása

Az adatközpont program az Ortodroma számítást használja, hogy kiszámolja a távolságot az optimális útvonal keresése közben.

```

1      func (s *service) CalculateDistance(lat1, lng1, lat2,
2      lng2 float64, unit ...string) float64 {
3          const PI = float64(math.Pi) //3.141592653589793
4
5          radlat1 := PI * lat1 / 180
6          radlat2 := PI * lat2 / 180
7
8          theta := lng1 - lng2
9          radtheta := PI * theta / 180
10
11         dist := math.Sin(radlat1)*math.Sin(radlat2) +
12             math.Cos(radlat1)*math.Cos(radlat2)*math.Cos(radtheta)
13
14         if dist > 1 {
15             dist = 1
16         }
17
18         dist = math.Acos(dist)

```



5.7. ábra. Web kliens, szimuláció 3D diagramon

```

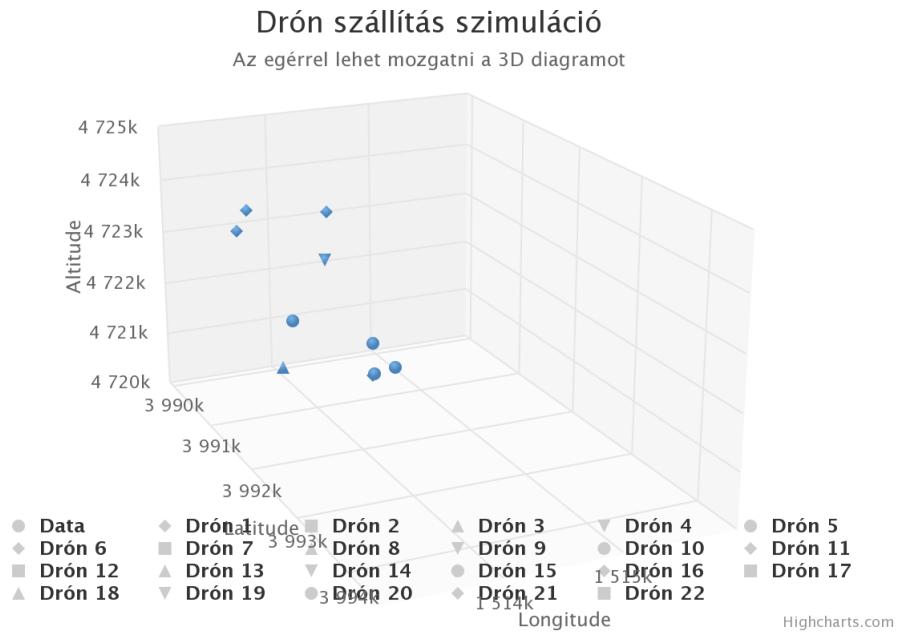
19         dist = dist * 180 / PI
20         dist = dist * 60 * 1.1515
21
22     if len(unit) > 0 {
23         if unit[0] == "K" {
24             dist = dist * 1.609344
25         } else if unit[0] == "N" {
26             dist = dist * 0.8684
27         } else if unit[0] == "METER" {
28             dist = dist * 1.609344 * 1000
29         }
30     }
31
32     return dist
33 }
```

A drón-raj program számításai

A pontos szimulációs értékekhez egy könyvtárat használunk, ami megfelel a WGS84 GPS rendszernek. Ezzel a könyvtárral pontos adatokat tudunk generálni a drón helyzetéről, irányáról.

```

1 geo := ellipsoid.Init("WGS84", ellipsoid.Degrees, ellipsoid.Meter,
2                           ellipsoid.LongitudeIsSymmetric, ellipsoid.BearingIsSymmetric)
3 routingService := routing.NewService(geo)
```



5.8. ábra. Web kliens, szimuláció 3D diagramon

Ezután a útvonalakért felelős szervízben két metódussal számoljuk ki a szükséges értékeket. A *CalculateDroneDistanceAndDirectionFromDestination* metódussal adott induló koordináták alapján megkapjuk a legrövidebb utat a célkoordinátákhoz, és az irányt is. A *CalculateDroneNextCoordinates* metódus megmondja adott induló koordináták és távolság valamint irány alapján, hogy hova érkezünk, azaz a célkoordinátákat.

```

1 func (s *service) CalculateDroneDistanceAndDirectionFromDestination(
2   currentLat, currentLon, destinationLat, destinationLon float64)
3   (distance, bearing float64) {
4     distance, bearing = s.geometry.To(currentLat,
5       currentLon, destinationLat, destinationLon)
6     return distance, bearing
7   }
8
9   func (s *service) CalculateDroneNextCoordinates(lat, lon,
10    dist, bearing float64) (nextLat, nextLon float64) {
11     nextLat, nextLon = s.geometry.At(lat, lon, dist, bearing)
12     return nextLat, nextLon
13 }
```

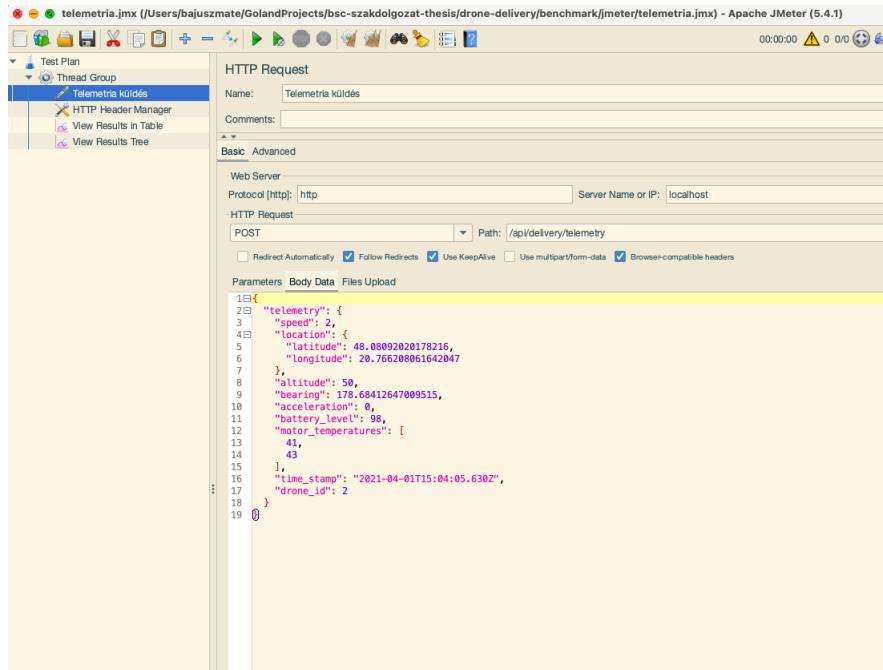
5.7. Mérések

Az alkalmazással kapcsolatban méréseket is végeztem, JMeter és ghz programokat használtam. Ezeket a méréseket fogom bemutatni és összehasonlítani. Összehasonlításra kerülnek az alkalmazásban használt különböző protokollok, formátumok és adatbázisok. minden mérés a telemetria adatok küldését és mentését ábrázolja, ugyanolyan körülmé-

nyek között. Üres adatbázissal, összesen 10000 telemetria adatot küldünk, 100 szálon. minden egyes méréshez tartozik egy rövid összegzés, hisztogramok a teljesítményről.

5.7.1. JMeter

A Jmeteres méréshez készítettem egy *telemetry.jmx* (5.9. ábra) konfigurációs fájlt. A mérést úgy állítottam be, hogy 100 szálon küldünk összesen 10000 db telemetria adatot. Ez a *drone-delivery/benchmark/jmeter* jegyzékben található. A mérést úgy végeztem,



5.9. ábra. Jmeter telemetry.jmx

hogy két jegyzékben lefuttattam a következő parancsot.

- *drone-delivery/benchmark/jmeter/mongo*
- *drone-delivery/benchmark/jmeter/postgres*

```
1 $ jmeter -n -t ./.. telemetria.jmx -l ./benchmark.csv -e -o ./output
```

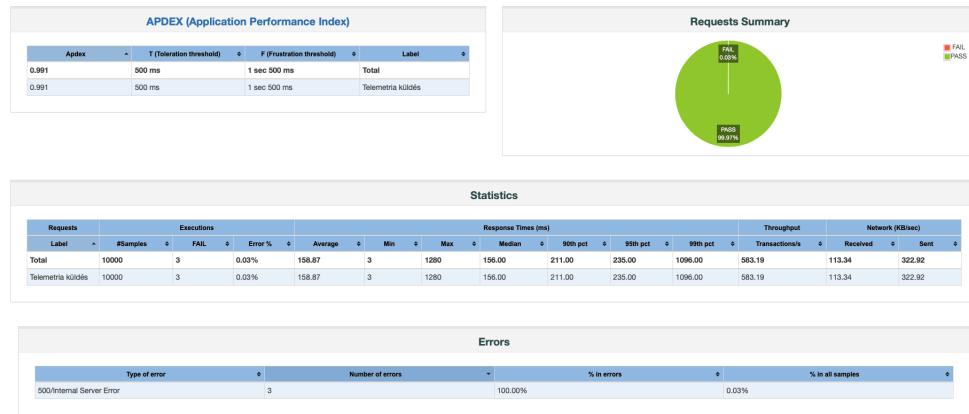
HTTP/1.1 JSON, PostgreSQL adatbázissal

A mérés közben a PostgreSQL adatbázis nem tudta kezelni a túl sok kapcsolatot (5.10. ábra), így 3-szor hibával tért vissza a mentés.

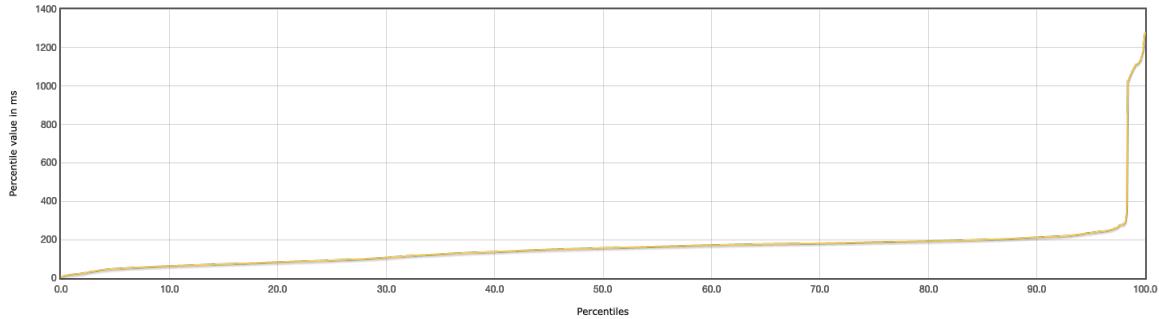
HTTP/1.1 JSON, MongoDB adatbázissal

5.1. megjegyzés. A MongoDB adatbázis jobban teljesített, mint a PostgreSQL. Az erőforrásokat is jobban használta ki, a mérések alapán azt a következtetést vonhatjuk le, hogy a PostgreSQL-ben sok konkurrens írás nem hatékony, a teljesítmény rovására megy.

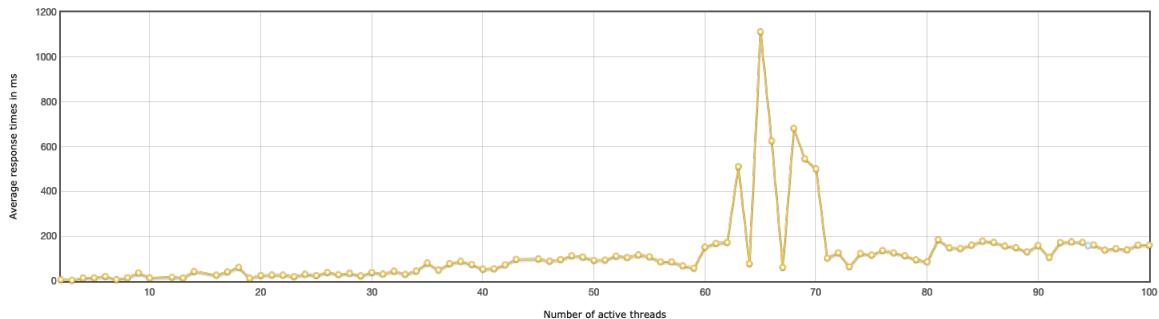
5.7. Mérések



5.10. ábra. Jmeter mérései PostgreSQL adatbázissal HTTP/1.1 protokoll JSON formá-tummal



5.11. ábra. Válaszidők PostgreSQL / JSON konfiguráció



5.12. ábra. Idő és szálak kapcsolata PostgreSQL / JSON konfiguráció

5.7.2. ghz

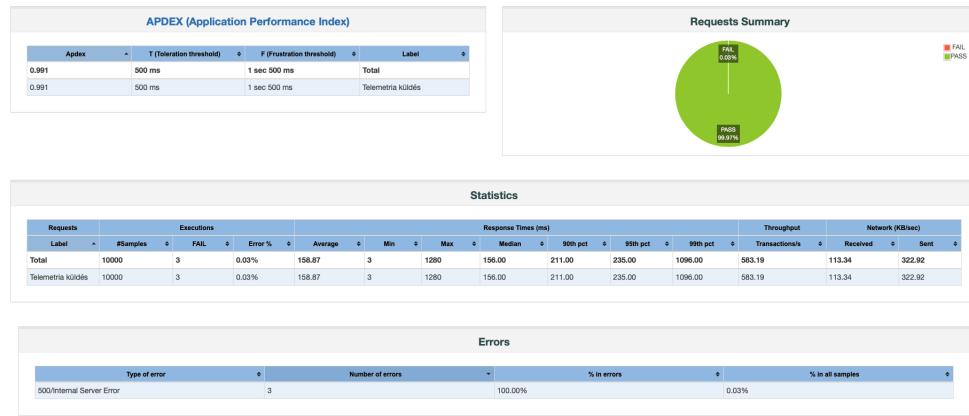
A ghz egy gRPC terhelés tesztelő program. Nincs hozzá grafikus felület. Először feltelepítjük a ghz programot, majd a következő parancssal tudjuk tesztelni, a kliens oldali Telemetria streamelést az alkalmazásunkhoz. Azaz itt a drón raj helyett a ghz program küldi a telemetria adatokat. Az output formátumnak HTML-t adtam meg, a `-c` paraméter a konkurrencia (szál) értéket jelöli, 100-at adtam meg. A `-n` paraméter requestek számát jelöli, ezt 10000-re állítottam.

```

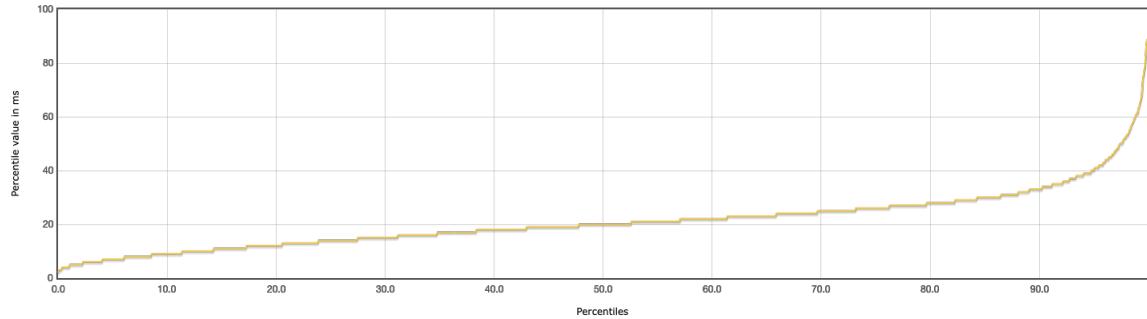
1 $ ~ ghz -c 100 -n 10000 --format html --output report-gRPC-postgres .
2   html --insecure \
  --proto /Users/bajuszmate/go/src/github.com/bajusz15/ghz-benchmark/ghz/

```

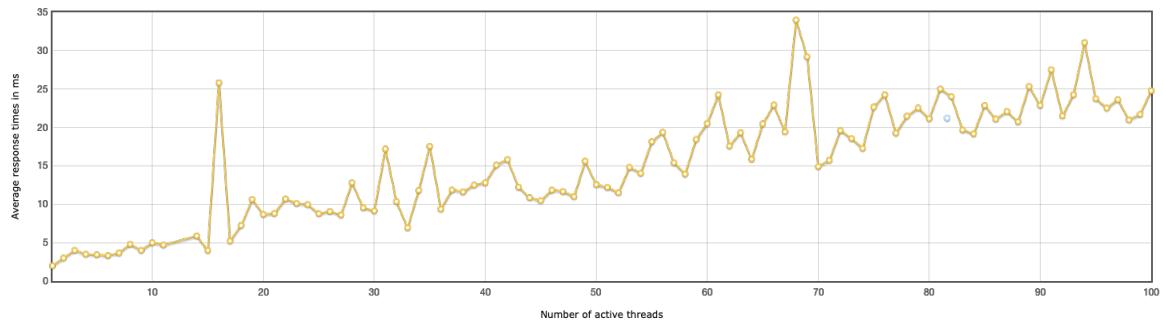
5.7. Mérések



5.13. ábra. Jmeter mérései MongoDB adatbázissal HTTP/1.1 protokoll JSON formá-tummal



5.14. ábra. Válaszidők MongoDB / JSON konfiguráció



5.15. ábra. Idő és szálak kapcsolata MongoDB / JSON konfiguráció

```

3   cmd/benchmark/protobuf/telemetry.proto \
4   --call telemetry_grpc.TelemetryService.TelemetryStream \
5   -d '['
6   {
7     "telemetry": {
8       "speed": 2,
9       "location": {
10         "latitude": 48.08092020178216,
11         "longitude": 20.766208061642047
12       },
13       "altitude": 50,
14       "bearing": 178.68412647009515,
15       "acceleration": 0,
16     }
17   }
18 ]
19 
```

```

15     "battery_level": 98,
16     "motor_temperatures": [] ,
17     "time_stamp": "2021-04-01T15:04:05.630Z",
18     "drone_id": 2
19   }
20 },
21 {
22   "telemetry": {
23     "speed": 2,
24     "location": {
25       "latitude": 48.08092020178216,
26       "longitude": 20.766208061642047
27     },
28     "altitude": 50,
29     "bearing": 178.68412647009515,
30     "acceleration": 0,
31     "battery_level": 98,
32     "motor_temperatures": [] ,
33     "time_stamp": "2021-04-01T15:04:05.630Z",
34     "drone_id": 2
35   }
36 }
37 ],
38 \      localhost:50051

```

HTTP/2 gRPC, PostgreSQL adatbázissal (5.16. ábra).

5.2. megjegyzés. A gRPC sokkal jobban teljesített mint a JSON. Ezt a diagramon is láthatjuk. Tehát a következtetések jók voltak, a HTTP/2 gRPC sokkal hatékonyabb mint a HTTP/1.1 JSON.

HTTP/2 gRPC, MongoDB adatbázissal (5.17. ábra).

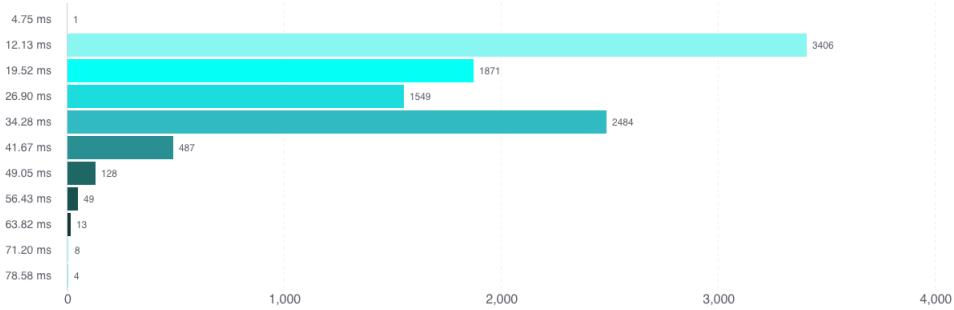
Összességében elmondható, hogy a gRPC a gyorsabb protokoll, és a MongoDB hatékonyabb erre a feladatra mint a PostgreSQL.

5.7. Mérések

Summary

Count	10000
Total	20.29 s
Slowest	78.58 ms
Fastest	4.75 ms
Average	20.17 ms
Requests / sec	492.75

Histogram



Latency distribution

10 %	25 %	50 %	75 %	90 %	95 %	99 %
9.66 ms	11.17 ms	15.68 ms	28.26 ms	32.58 ms	35.83 ms	47.17 ms

Status distribution

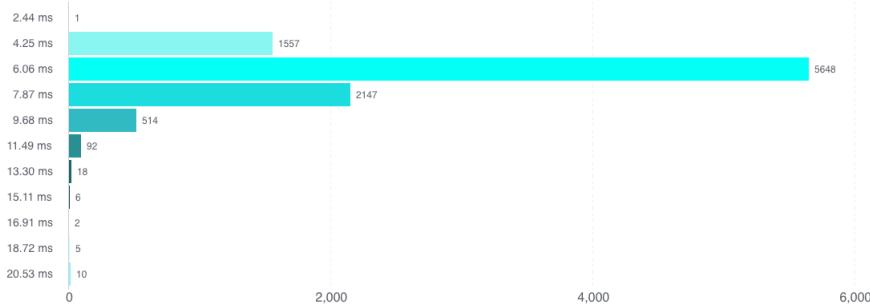
Status	Count	% of Total
OK	10000	100.00 %

5.16. ábra. Ghz mérései PostgreSQL adatbázison, gRPC

Summary

Count	10000
Total	5.62 s
Slowest	20.53 ms
Fastest	2.44 ms
Average	5.53 ms
Requests / sec	1778.17

Histogram



Latency distribution

10 %	25 %	50 %	75 %	90 %	95 %	99 %
4.04 ms	4.54 ms	5.26 ms	6.20 ms	7.33 ms	8.20 ms	10.07 ms

Status distribution

Status	Count	% of Total
OK	10000	100.00 %

5.17. ábra. Ghz mérései MongoDB adatbázison, gRPC

6. fejezet

Összefoglalás

Összességében meg vagyok elégedve az eredményekkel. A rendszer moduláris felépítése jól sikerült, az összehasonlítások tökéletesen működtek és érdekes eredményeket produkáltak, de persze semmi meglepő nincs az eredményekben.

Számomra nagyon érdekesek a különböző architektúrák, új kommunikációs protokollok és adatbázisok által nyújtott előnyök. Amit hiányolok a programból, de sajnos nem maradt idő az implementálásra az egy valós idejű Apache Kafka adatcsatorna. Az Apache Kafka egyfajta üzenet bróker, amely képes nagy forgalmú hiba toleráns működésre. Mivel a telemetria adatokra tekinthetünk úgy, mint kritikus adatok, egy ilyen modullal ki lehetne egészíteni a programot. A drónok szimulációja lehetett volna érdekesebb, ha tudunk valós térképek alapján útvonalakat optimalizálni. A rendszer jövőjét mindenkorban látom, hogy több protokollt és adatbázist lehet összehasonlítani. Érdekes eredmények születhetnének MQTT protokollal, és más dokumentum alapú vagy nagyon sok írásra tervezett adatbázisokkal.

Irodalomjegyzék

- [1] Transaction isolation, Oct 2016.
- [2] Application for on-demand drone delivery system, Oct 2020.
- [3] Build cross-platform and optimized drone applications., Oct 2020.
- [4] Smartskies™ fusion, Oct 2020.
- [5] Software solution for drone delivery system, Nov 2020.
- [6] Why docker?, Mar 2021.
- [7] Luiz Fernando Assis, Karine Ferreira, Lubia Vinhas, Luis Maurano, Cláudio Almeida, André Carvalho, Jether Rodrigues, Adeline Maciel, and Claudinei de Camargo. Terrabrasilis: A spatial data analytics infrastructure for large-scale thematic mapping. *International Journal of Geo-Information*, 8:513, 11 2019.
- [8] Shif Ben Avraham. What is rest - a simple explanation for beginners, part 2: Rest constraints, Sep 2017.
- [9] Netflix Technology Blog. Ready for changes with hexagonal architecture, Mar 2020.
- [10] Ben Coxworth Dhl. Dhl parcelcopter takes to tanzanian skies, Oct 2018.
- [11] <https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0>. Hexagonal architecture ports and adapters. *medium.com*, 2021.10.12.
- [12] Nick Janetakis. Docker offers a better way to build and distribute your applications, Feb 2021.
- [13] Donald C Sweeney Andrea C. Hupman Juan Zhang, James F. Campbell. Energy consumption models for delivery drones: A comparison and assessment - may 2020.
- [14] Robert C. Martin. The clean code blog, Aug 2012.
- [15] Pablo Martinez. Domain-driven design: Everything you always wanted to know about it, but were afraid to ask, May 2020.
- [16] Brein Matturro. What is remote procedure call (rpc)?, May 2020.
- [17] Mary Meisenzahl. Amazon's drone program was cleared by the faa - take a look at the machines it wants to use to deliver prime packages to your doorstep, Sep 2020.

-
- [18] Szerző: HWSW 2020. október 21. 08:33. Miért a go lett 2020 legjobban vágyott programozási nyelve?
 - [19] Chris Richardson. Microservices pattern: Microservice architecture pattern, Mar 2019.
 - [20] Jonathan Rupprecht. The truth about drone delivery (legal problems no one talks about), Feb 2021.
 - [21] Nagy Tamás, Feb 2021.
 - [22] Matías Varela. Hexagonal architecture in go, May 2020.
 - [23] vIns. grpc - an introduction guide. *Vinsguru*, Jun 2020.
 - [24] Serdar Yegulalp. What's the google go language (golang) really good for?, Oct 2019.

CD Használati útmutató

A CD tartalma:

- dolgozat jegyzék, Ez tartalmazza magáta szakdolgozatot *.tex* formátumban, és a hozzá tartozó képeket, stílusokat.
- drone-delivery jegyzék Ebben a jegyzékbenn a programokat és a hozzá tartozó rendszert, méréseket találjuk. A dolgozatban van részletezve a programok telepítése és használata. A mérések adatait tartalmazó HTML dokumentumokat meg lehet tekinteni önállóan, a *benchmark* jegyzéken belül.
- drone-delivery jegyzék A szakdolgozat *.pdf* formátumban.