Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Mesterséges Intelligencia Tanszék

Földi Balázs

IPARI KÖRNYEZETBELI SZOLGÁLTATÁS MEGVALÓSÍTÁSA KUBERNETES KÖRNYEZETBEN

KONZULENS

Dr. Simon Csaba

BUDAPEST, 2025

Tartalomjegyzék

Összefoglaló **6**

Abstract **7**

1 Bevezetés **8**

2 Ipari környezetű robotikai rendszerek és a távoli vezérlés szerepe **10**

2.1 Az ipari automatizálás és robotika fejlődése 10

2.1.1 A robotika fejlődése és történeti áttekintés 10

2.1.2 Gazdasági hatások és munkaerőpiaci előnyök 11

2.1.3 A fejlődés jövőbeli trendjei 11

2.2 Felhőalapú irányítás előnyei és kihívásai robotikai alkalmazásokban 12

2.3 A Yahboom Dogzilla S2 robotkutya mint ipari demonstrációs platform 14

2.3.1 A demonstrációs eszköz kiválasztásának indoklása 14

2.3.2 Számítási kapacitás és rendszerarchitektúra 14

2.3.3 Szoftveres ökoszisztéma: ROS 2 14

2.3.4 Szenzorok és beavatkozók a távoli felügyelethez 15

2.3.5 Összegzés 15

3 ROS 2 keretrendszer **16**

3.1 Rétegzett rendszerarchitektúra 16

3.2 A köztesréteg (Middleware) 18

3.2.1 A Data Distribution Service (DDS) szerepe és működése 18

3.3 Kommunikációs alapfogalmak: csomópontok, témák és szolgáltatások 19

3.3.1 Csomópontok (Nodes) 19

3.3.2 Témák (Topics) 19

3.3.3 Szolgáltatások (Services) 20

3.3.4 Műveletek (Actions) 22

3.4 Ipari alkalmazhatóság, valós idejű működés 24

4 Felhőalapú rendszerek és a Kubernetes **25**

4.1 Bevezetés a modern infrastruktúra-architektúrákba 25

4.2 A felhőalapú számítástechnika rendszertana és gazdasági modellje 25

4.2.1 A szolgáltatási modellek (SPI) részletes elemzése 26

4.2.2 Telepítési modellek (Deployment Models) 28

4.3 Konténertechnológiák 29

4.3.1 Virtuális gépek (VM) és konténerek 29

4.3.2 A Docker szerepe és technológiai újításai 30

4.4 A Kubernetes architektúrája és működési mechanizmusai ... 32
4.4.1 A vezérlősík (Control Plane) ... 32
4.4.2 A munkavégző csomópontok (Worker Nodes) ... 33
4.5 Kubernetes alapfogalmak és objektumok ... 34
4.5.1 Pod: A legkisebb egység ... 34
4.5.2 Deployment és ReplicaSet: Az alkalmazás életciklusa ... 34
4.5.3 Service és Ingress: Hálózati elérés ... 35
4.6 Hálózati kommunikáció és biztonság ... 35
4.6.1 Container Network Interface (CNI) ... 35
4.7 Skálázás, és monitorozás ... 36
4.7.1 Automatikus Skálázás (Autoscaling) ... 36
4.7.2 Monitorozás: Prometheus és Grafana ... 36
4.8 Konklúzió ... 36
5 Rendszertervezés ... 37
5.1 Bevezetés a rendszertervezésbe ... 37
5.2 Rendszer-architektúrális döntések és topológia ... 37
5.2.1 A kommunikációs híd: A DDS korlátai és az MQTT választása ... 37
5.2.2 A hibrid rendszer rétegei ... 39
5.3 Felhő infrastruktúra tervezése és validációja ... 41
5.3.1 Az üzenetközvetítő: Mosquitto Broker implementáció ... 41
5.3.2 A Robot API mikroszolgáltatás ... 42
5.4 A kommunikációs köztesréteg (Middleware) részletes elemzése ... 42
5.4.1 MQTT Topic struktúra és adatmodellek ... 43
5.4.2 QoS (Quality of Service) szintek ... 43
5.5 Robot-oldali szoftverarchitektúra ... 43
5.5.1 Az MQTT-ROS 2 híd ... 44
5.5.2 Hardver absztrakciós réteg (DogzillaActionNode) ... 44
5.6 Validált parancskészlet ... 46
5.7 Konklúzió ... 46
6 Mérések és értékelés ... 47
6.1 A felhőalapú vezérlési infrastruktúra hálózati teljesítményének elemzése ... 47
6.1.1 A mérés célja és módszertana ... 47
6.1.2 A késleltetés (Latency) értékelése ... 47
6.1.3 Stabilitás és Jitter elemzése ... 47

6.1.4 A diagramok értelmezése 48
6.1.5 Következtetés 49
6.2 A teljes körű, fizikai robottal végzett vezérlési lánc teljesítményelemzése 50
6.2.1 Bevezetés és mérési elrendezés 50
6.2.2 A késleltetés (Latency) részletes elemzése 50
6.2.3 Stabilitás és Jitter 51
6.2.4 Összehasonlító elemzés (Infrastruktúra vs. Fizikai Robot) 51
6.2.5 Vizualizáció és Diagramok 51
6.2.6 Konklúzió 52
6.3 A rendszer teljesítményelemzése privát 5G környezetben 52
6.3.1 A kísérleti környezet hálózati topológiája és specifikációi 52
6.3.2 A hálózati késleltetés és stabilitás elemzése 56
7 A munka értékelése és jövőbeli fejlesztési lehetőségek 58
7.1 Az elért eredmények átfogó értékelése 58
7.1.1 A hibrid architektúra teljesítményének összehasonlító elemzése 59
7.2 Hálózati infrastruktúra és az 5G szerepe 60
7.2.1 Az 5G technológia integrációjának tapasztalatai és a Local Breakout 61
7.2.2 Edge Computing: A hierarchikus feldolgozás modellje 61
7.3 Autonómia növelése: Felhőalapú SLAM 62
7.4 Közvetlen ROS 2 felhő-integráció és alagút-technológiák 63
7.4.1 Hálózati szintű alagút (Network Layer Tunneling) 63
7.4.2 Middleware szintű áthidalás és Offloading (Zenoh & FogROS2) 64
7.5 Biztonságtechnikai fejlesztések 65
7.5.1 Kölcsönös hitelesítés (mTLS) és teljesítményhatásai 65
8 Összefoglalás 66
9 Függelék 67
9.1 Függelék Nyilatkozat generatív mesterséges intelligencia alkalmazásáról 67

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2025. 12. 12.

...…….

Földi Balázs

Összefoglaló

A modern ipari automatizálás és az Ipar 4.0 egyik legdinamikusabban fejlődő területe a felhőalapú robotika (Cloud Robotics), amely a robotikai eszközök fedélzeti számítási kapacitásának korlátait a felhőinfrastruktúra skálázható erőforrásaival igyekszik áthidalni. Jelen szakdolgozat e technológiai konvergencia gyakorlati megvalósíthatóságát vizsgálja, fókuszba helyezve a konténerizált környezetek és a modern robotikai keretrendszerek integrációs lehetőségeit.

A dolgozat központi témája egy olyan hibrid, felhőalapú vezérlőrendszer tervezése és implementálása, amely képes egy ROS 2 (Robot Operating System 2) alapú mobil robot – konkrétan egy Yahboom Dogzilla S2 robotkutya – távoli, valós idejű irányítására.

A megvalósítás során a rendszer a Kubernetes orkesztrációs platformra támaszkodik, amely biztosítja a vezérlő szoftverkomponensek automatizált telepítését, skálázását és menedzselését. A dolgozat bemutatja a konténerizáció (Docker) szerepét a szoftveres környezet egységesítésében, valamint ismerteti a kifejlesztett hardver-absztrakciós réteget és a felhőoldali API-t, amelyek lehetővé teszik a parancsok és telemetriai adatok kétirányú áramlását.

A munka jelentős részét képezi a létrehozott architektúra teljesítményének validálása. A dolgozat ismerteti a mérési metodikát, amely külön vizsgálja a tisztán infrastrukturális (felhő-kliens) és a teljes körű, fizikai robottal végzett (End-to-End) kommunikációt. A vizsgálat kiterjed a hálózati késleltetés (latency), a késleltetés-ingadozás (jitter) és a csomagvesztés elemzésére, keresve a választ arra, hogy egy ilyen architektúra alkalmas-e a teleoperációs feladatok ellátására. Végezetül a dolgozat áttekinti a rendszer jövőbeli fejlesztési irányait.

Abstract

Cloud Robotics is one of the most dynamically developing areas of modern industrial automation and Industry 4.0, aiming to bridge the limitations of onboard computing capacity in robotic devices using the scalable resources of cloud infrastructure. This thesis investigates the practical feasibility of this technological convergence, focusing on the integration opportunities of containerized environments and modern robotics frameworks.

The central theme of the thesis is the design and implementation of a hybrid, cloud-based control system capable of remote, real-time control of a ROS 2 (Robot Operating System 2) based mobile robot, specifically a Yahboom Dogzilla S2 robot dog.

For implementation, the system relies on the Kubernetes orchestration platform, which ensures the automated deployment, scaling, and management of control software components. The thesis presents the role of containerization (Docker) in unifying the software environment and describes the developed hardware abstraction layer and cloud-side API, which enable the bidirectional flow of commands and telemetry data.

A significant portion of the work involves validating the performance of the created architecture. The document details the measurement methodology, which separately examines purely infrastructural (cloud-client) communication and full-scale communication involving the physical robot (End-to-End). The investigation covers the analysis of network latency, jitter, and packet loss, seeking to determine whether such an architecture is suitable for teleoperation tasks. Finally, the thesis reviews future development directions.

1 Bevezetés

A modern ipari környezetekben egyre nagyobb igény mutatkozik a rugalmas, távolról vezérelhető és dinamikusan skálázható rendszerek iránt. Az automatizálás, a robotika és a mesterséges intelligencia fejlődése új lehetőségeket nyitott meg az olyan szolgáltatásalapú architektúrák számára, amelyek képesek valós idejű kommunikációra és autonóm működésre. E fejlődési irány egyik legfontosabb technológiai alapját a felhőalapú számítástechnika és a konténerizáció képezi, amelyek lehetővé teszik az erőforrások megosztását, a szolgáltatások elkülönített futtatását és az automatikus skálázását.

A szakdolgozat központi témája egy ipari környezetre is adaptálható, felhőalapú vezérlőrendszer megvalósítása, amely képes egy Yahboom Dogzilla S2 robotkutya távoli irányítására. A rendszer célja, hogy demonstrálja, miként integrálható a ROS 2 (Robot Operating System 2) keretrendszer – amely az ipari és kutatási robotikai fejlesztések szabványává vált – egy Kubernetes-alapú felhőinfrastruktúrába. Ez az integráció egy hibrid architektúrát valósít meg: míg a robot hardverközeli, valós idejű beavatkozást igénylő funkciói helyben futnak, addig a felhasználói interakciót kiszolgáló webes réteg és a központi koordinációs logika konténerizált formában, a felhőbe kerül kiszervezésre. Ez a megközelítés biztosítja a rendszer horizontális skálázhatóságát és a szolgáltatások központosított menedzselését, hatékonyan tehermentesítve ezzel a robot fedélzeti számítógépét a hálózati kiszolgáló feladatok alól.

A téma indokoltsága több szempontból is jelentős. Egyrészt a felhőrobotika (cloud robotics) napjaink egyik leggyorsabban fejlődő kutatási és ipari területe, amely a számítási kapacitások központosításával és megosztásával egyszerűsíti a robotikai alkalmazások fejlesztését és karbantartását. Másrészt a Kubernetes, mint ipari szabvány a konténeralapú szolgáltatásmenedzsmentben, kiváló lehetőséget kínál az ilyen rendszerek skálázható és megbízható működtetésére. A robotika és a felhőtechnológiák összekapcsolása olyan új irányt jelöl ki, amely alapot teremthet autonóm, hálózatba kötött robotflották ipari szintű működtetéséhez.

A szakdolgozat nyolc fejezetben tárgyalja a robotikai rendszerfejlesztés lépéseit az elméleti alapozástól a megvalósításig.

A második fejezet az ipari környezetű robotikai rendszerek fejlődését és a távoli vezérlés, különösen a felhőalapú irányítás szerepét vizsgálja. Bemutatja a kutatás hardveres platformját, a Dogzilla S2 robotkutyát, részletezve annak hardverarchitektúráját, számítási kapacitását és a környezetérzékeléshez használt 4D LiDAR technológiát.

A harmadik fejezet a szoftveres alapokat, a ROS 2 keretrendszert ismerteti. Részletesen tárgyalja a rétegzett rendszerarchitektúrát, a DDS middleware szerepét, a kommunikációs alapfogalmakat (csomópontok, témák, szolgáltatások), valamint a rendszer ipari alkalmazhatóságát és valós idejű működését.

A negyedik fejezet a modern infrastruktúra-architektúrákra fókuszál. Áttekinti a felhőalapú számítástechnika szolgáltatási és telepítési modelljeit, valamint a konténertechnológiákat (Docker). A fejezet jelentős része a Kubernetes architektúrájának (vezérlősík, munkavégző csomópontok), alapvető objektumainak (Pod, Deployment, Service) és hálózati megoldásainak (CNI) bemutatásával foglalkozik, kiegészítve a skálázás és a monitorozás (Prometheus, Grafana) kérdéseivel.

Az ötödik fejezet a rendszertervezés lépéseit tartalmazza, míg a hatodik fejezet a megvalósított rendszer méréseire és értékelésére tér ki.

Végül a hetedik és nyolcadik fejezet összegzi a dolgozat felvázolja a jövőbeli fejlesztési lehetőségeket és összegzi az elért eredményeket.

2 Ipari környezetű robotikai rendszerek és a távoli vezérlés szerepe

2.1 Az ipari automatizálás és robotika fejlődése

Az ipari robotika és automatizálás a gépészeti és mechatronikai tudományok egyik leggyorsabban fejlődő területe, melynek lendületét az ipari robotok választékos alkalmazhatósága, a kiterjedt teherkategóriák, valamint a megnövekedő méretek és teljesítmények adják [1]. Az ipari automatizálás a számítógépeket, a vezérlőrendszereket és az információs technológiát használja az ipari folyamatok és gépek kezelésére, célja a manuális munka kiváltása a hatékonyság, a sebesség, a minőség és a teljesítmény javítása mellett. Bár az automatizálás és a robotika fogalmai eltérőek, a fizikai feladatok automatizálásában gyakran együtt járnak [2].

2.1.1 A robotika fejlődése és történeti áttekintés

Az ipari automatizálás korai formái már a második világháború alatt megjelentek az Egyesült Államokban a CNC (Computer Numerical Control) gépek alkalmazásával a nagy pontosságú repülőgépgyártásban [1]. Maguk az ipari robotok azonban az 1960-as évek végén jelentek meg az ipari környezetben, azokon a területeken, ahol az emberi fizikai teljesítőképesség elérte a határait. A korai évtizedekben a termelés hatékonyságának növelése volt a fő mozgatóerő [1] [2].

Az első robotok nehéz, lassú és költséges hidraulikus meghajtású rendszerek voltak. A nagy technológiai áttörés a szervomeghajtások továbbfejlesztésével jött el, ami lehetővé tette a hidraulikus rendszerekről az elektromos motorral hajtott robotokra való áttérést. Az 1980-as években az alkalmazások fő motivációja a termékek minőségének és reprodukálhatóságának javítása lett. Ekkoriban mintegy 150 robotgyártó cég működött világszerte. A robotok sorozatgyártása a 90-es évekre tehető, ekkor vált lehetővé a modern gyártási filozófiák, mint a just in time vagy just in sequence megvalósítása. Az ezredforduló óta a további fejlesztések, mint például a vízió rendszerek (kamera-rendszerek) megjelenése nyitott meg új alkalmazási területeket [1].

2.1.2 Gazdasági hatások és munkaerőpiaci előnyök

Az ipari automatizálás számos előnnyel jár. A robotok alkalmazásával a működési költségek csökkenthetők, mivel nincs szükség bér- és járulékai fizetésére, szociális-egészségügyi juttatásokra. Az automatizált rendszerek folyamatos munkavégzést tesznek lehetővé, ami jelentősen növeli a termelékenységet. Mivel a gépek fáradtság nélkül, nagy ismétlőképességgel dolgoznak, javul a minőség, és csökken a hibák száma. A robotizált rendszerek rugalmasabbak is, mivel könnyebben átprogramozhatók más feladatok elvégzésére. A veszélyes szerepek robotokra való átruházása növeli a munkahelyi biztonságot is [2].

A gazdasági elemzések szerint az ipari robotok intenzívebb használata hozzájárult a munka termelékenységének növekedéséhez az európai iparban 1993 és 2015 között. Ami a foglalkoztatást illeti, a rendelkezésre álló adatok arra utalnak, hogy az ipari robotok használata 1995–2015 között kismértékű, de jelentős növekedéssel járt a teljes foglalkoztatásban az EU-ban, különösen a feldolgozóiparban. Az adatok nem támasztják alá azt a narratívát, hogy a robotok nagyszabású munkahelymegszüntetést okoznának [3].

2.1.3 A fejlődés jövőbeli trendjei

A robotika jövőbeli képességei szorosan kapcsolódnak a mesterséges intelligencia (MI) fejlődéséhez, mivel a robotok a mesterséges intelligencia fizikai megnyilvánulásai. Az MI, különösen a gépi tanulás és a gépi látás révén, folyamatosan javítja a robotok alkalmazkodóképességét és tudatosságát.

A következő hat-hét évben várható fő technológiai trendek közt szerepel az MI-alapú autonóm robotika. Az MI fejlesztések lehetővé teszik a robotok számára, hogy minimális emberi beavatkozással is képesek legyenek bonyolult feladatokat ellátni, és elvezethetnek a humanoid, általános célú robotok megjelenéséhez is, amelyek feladat-specifikus programozás nélkül működhetnek. Később a fejlett MI és érzékelő technológiák révén a robotok valóban együttműködhetnek az emberekkel, sőt, tanulhatnak is tőlük. A természetes nyelvi feldolgozás és a Generatív MI leegyszerűsíti a programozási feladatokat, lehetővé téve a dolgozók számára a robotokkal való kommunikációt természetes nyelven [4].

2.2 Felhőalapú irányítás előnyei és kihívásai robotikai alkalmazásokban

A felhőalapú számítástechnika robotikába történő integrálása jelentős előnyökkel jár, amelyek segítenek áthidalni a hagyományos, önálló robotrendszerek veleszületett korlátait, mint amilyen a fedélzeti hardver és a számítási követelmények szabta kapacitás. A felhőalapú irányítás fontos előnye, hogy a robotok növelt számítási teljesítményhez és tárhelyhez jutnak hozzá. Ennek köszönhetően a robotoknak nem kell a bonyolult feladatokat a fedélzeten végrehajtaniuk, hanem kiszervezhetik a számításigényes műveleteket (például az objektumfelismerést, a mintafelismerést, a számítógépes látást vagy a beszédfelismerést) a felhőbe. Ezek a feladatok a felhőben sokkal gyorsabban, valós időben oldhatók meg, kihasználva a párhuzamos vagy a hálózati számítási képességeket. A felhőinfrastruktúra ezenkívül támogatja a számítási erőforrások skálázható és igény szerinti elérhetőségét [5] [6].

A felhőalapú robotika révén lehetőség nyílik könnyűsúlyú, alacsony költségű és okosabb robotok építésére, amelyek intelligens agya a felhőben található. Ezen felül a számítási munka felhőbe való delegálása csökkenti a robot fedélzeti terhelését, ami hozzájárul a hardver egyszerűbb karbantartásához és a hosszabb akkumulátor-élettartamhoz. Ami az adatok tárolását illeti, a felhővel támogatott robotok hozzáférnek a felhő által kínált nagy tárhelyekhez, ahol eltárolhatják az összes hasznos információt későbbi felhasználásra. Ez fontos például az olyan rendszerek esetében, mint a SLAM, ahol hatalmas mennyiségű érzékelő adat keletkezik. A felhőalapú irányítás lehetővé teszi a robotok számára, hogy információkat osszanak meg a bonyolult feladatok megoldásának módjáról, és hozzáférjenek az emberi tudáshoz is. A felhő egy olyan megosztott tudásbázist biztosít, amelyen keresztül a robotok új készségeket és ismereteket tanulhatnak egymástól. Az olyan projektek, mint a RoboEarth, ezt a globális tudásgyűjtést és -megosztást célozzák [5] [6]. Mivel a felhő képes a komplexitást kezelni, lehetővé teszi az automatizálás kiterjesztését minden területre, növelve a rugalmasságot az iparban, például a gyártóüzemek átszervezésekor.

A jelentős előnyök ellenére a felhőalapú robotika számos komoly kihívással is küzd. Az egyik legfőbb probléma a késleltetés (latency). A késleltetés a felhőrobotikai rendszerek központi kérdése, amely a robot és a felhő között fellépő számítási kommunikációs késleltetést jelenti. Ez az idő magában foglalja az adatok beszerzését, a számítási feladatok kiszervezését és az eredmények visszanyerését a felhőből. Mivel a felhőrobotikának folyamatosan gyűjtenie és elemeznie kell az adatokat, és gyors döntéseket kell hoznia, az ilyen rendszerek érzékenyek a késleltetésre [5] [7].

A késleltetési problémát tovább bonyolítja, hogy a felhőalapú robotika gyors és folyamatos internetkapcsolatot igényel a robotok és a távoli felhő között. A vezeték nélküli hálózatok azonban a vezetékes hálózatokkal összehasonlítva gyakran szakaszosak, alacsony sávszélességűek és kevésbé megbízhatóak lehetnek. Ráadásul a nagy mennyiségű adat tovább növeli a kommunikációs késedelmeket. A folyamatos kapcsolat hiánya esetén a felhőalapú alkalmazások lelassulhatnak vagy elérhetetlenné válhatnak, ami a robotot „agyatlanná” teszi, ezért az ismétlődő és időkritikus feladatok valós idejű végrehajtása fedélzeti feldolgozást igényel [5] [6].

Másik kulcsfontosságú terület az erőforrások és a feladatok hatékony elosztása. A teljesítmény egyik meghatározó tényezője az a döntés, hogy egy adott feladatot kiszervezzünk-e a felhőbe, vagy helyi erőforrásokkal dolgozzuk fel. Szükség van olyan megoldásokra, amelyek képesek kezelni a hálózati topológia változásait, és biztosítják, hogy a feladatok elosztása rugalmasan történjen, minimalizálva a késleltetést [5].

Végül a felhőtechnológia használata miatt kiemelt szerepet kapnak az adatvédelmi és biztonsági aggályok. Mivel a robotikai adatok távoli felhőszervereken tárolódnak és dolgozódnak fel, ez sebezhetővé teheti az alkalmazásokat a rosszindulatú felhasználókkal és hackerekkel szemben. A távoli adatokhoz való illetéktelen hozzáférés, manipuláció és a kulcsfontosságú adatok elvesztése valós veszélyt jelentenek. Az ipari IoT-alkalmazások különösen szigorú követelményeket támasztanak az informatikai biztonsággal szemben, ami megköveteli a kontrollált és biztonságos adatmegosztást a beszállítói lánc minden szereplője között. Ugyanakkor az is fontos tényező, hogy a mobil robotok esetében optimalizálni kell az energiahatékonyságot is, megkeresve az ideális kompromisszumot a számítási feladatok kiszervezéséhez szükséges energiafogyasztás és a roboton belüli helyi számítás energiafelhasználása között [5] [7].

2.3 A Yahboom Dogzilla S2 robotkutya mint ipari demonstrációs platform

2.3.1 A demonstrációs eszköz kiválasztásának indoklása

A szakdolgozat célja egy felhőalapú Kubernetes klaszterből történő távoli robotvezérlés megvalósítása. Ehhez egy olyan hardverre van szükség, amely architektúrájában tükrözi a modern ipari robotok (pl. Boston Dynamics Spot, Unitree Go) felépítését, ugyanakkor költséghatékony és laboratóriumi környezetben biztonságosan üzemeltethető. A választott eszköz, a Yahboom Dogzilla S2. A négylábú kialakítás és a 12 szabadságfokú (12-DOF) mozgásrendszer lehetővé teszi a minden irányú mozgást, ami komplexebb vezérlési kihívást jelent, mint a hagyományos kerekes robotok irányítása, így alkalmasabb a felhőalapú vezérlés késleltetés-érzékenységének és megbízhatóságának tesztelésére.

2.3.2 Számítási kapacitás és rendszerarchitektúra

A Dogzilla S2 központi vezérlőegysége egy Raspberry Pi 5 számítógép, amely jelentős előrelépést jelent a korábbi modellekhez képest. A 4GB/8GB RAM elegendő a ROS 2 node-ok, a videó tömörítés és a hálózati kommunikációs modulok párhuzamos futtatásához anélkül, hogy a vezérlési ciklusidő (control loop) sérülne [8].

2.3.3 Szoftveres ökoszisztéma: ROS 2

A robot szoftveres alapja a ROS 2 keretrendszer. Ez közvetlen kompatibilitást biztosít az ipari szabványokkal.

A ROS 2 kommunikációs rétege támogatja a megbízható, valós idejű adatcserét, ami a távoli vezérlés alapfeltétele. A hardverfunkciók (kamera, Lidar, motorvezérlés) különálló ROS node-okként érhetők el (pl. /cmd_vel a mozgásvezérléshez, /scan a Lidar adatokhoz). Ez a modularitás lehetővé teszi, hogy a felhőben futó vezérlő alkalmazás szabványos üzeneteken keresztül kommunikáljon a robottal, elfedve a hardver-specifikus driver-eket [8].

2.3.4 Szenzorok és beavatkozók a távoli felügyelethez

A felhőalapú irányításhoz a robotnak folyamatosan adatokat kell szolgáltatnia környezetéről (upstream), és végre kell hajtania a kapott parancsokat (downstream).

A 360 fokos, 2D lézerszkenner pontfelhőt generál a környezetről. Ez az adatfolyam a sávszélesség-igényes, de kritikus fontosságú a SLAM (térképezés) és az akadálykerülés szempontjából. A távoli operátor ezen adatok alapján látja a robot elhelyezkedését a térképen. Az 1080p felbontású kamera biztosítja a vizuális visszacsatolást (First Person View - FPV). A Raspberry Pi 5 képes a videójel hardveres kódolására, minimalizálva a sávszélesség-igényt a felhő felé. A 12 db fémfogaskerekes busz-szervó pedig lehetővé teszi a járást, fordulást és a test dőlésszögének (roll, pitch, yaw) állítását. A robot képes inverz kinematikai számításokat végezni a fedélzeten, így a felhőből elegendő magas szintű parancsokat (pl. lineáris sebességvektor) küldeni, nem szükséges minden egyes motor pozícióját külön továbbítani, ami csökkenti a hálózati terhelést [8].

2.3.5 Összegzés

A Yahboom Dogzilla S2 technológiai paraméterei – a nagy teljesítményű Raspberry Pi 5, a natív ROS 2 támogatás és a gazdag szenzorkészlet – ideális platformmá teszik a "Cloud Robotics" koncepciók demonstrálására. A rendszer képes szimulálni egy valós ipari környezetet, ahol a robot autonóm funkcióit és felügyeletét egy központosított, felhő alapú Kubernetes infrastruktúra koordinálja, biztosítva a skálázhatóságot és a központi menedzsmentet.

3 ROS 2 keretrendszer

A robotika szoftveres infrastruktúrájának evolúciója az elmúlt két évtizedben nagy átalakuláson ment keresztül. A Robot Operating System (ROS) megjelenése előtt a robotikai fejlesztés jelentős része az alapvető kommunikációs protokollok, hardver-illesztőprogramok és szenzor-adatfeldolgozó algoritmusok újraalkotásával telt, ami lassította az innovációt és rontotta a kutatási eredményeket. A ROS 1 2007-es megjelenése, amelyet eredetileg a Stanford Artificial Intelligence Laboratory és a Willow Garage indított útjára, forradalmasította ezt a területet azáltal, hogy egy egységes, nyílt forráskódú keretrendszert biztosított [9]. Azonban ahogy a robotika kilépett a kutatólaboratóriumok steril környezetéből a valós ipari alkalmazások, az autonóm járművek és a kereskedelmi termékek világába, a ROS 1 architektúrális korlátai egyre nyilvánvalóbbá váltak. A ROS 2 egy teljesen újragondolt architektúra, amely a modern szoftverfejlesztési elvekre, a valós idejű követelményekre és az ipari szabványokra épít, miközben megőrzi a ROS ökoszisztéma alapvető értékeit.

3.1 Rétegzett rendszerarchitektúra

A ROS 2 architektúrája egy szigorúan definiált, rétegzett modellt követ, amely biztosítja a modularitást. Ez a struktúra lehetővé teszi a fejlesztők számára, hogy magas szintű robotikai alkalmazásokat írjanak anélkül, hogy ismerniük kellene az alacsony szintű kommunikációs mechanizmusok részleteit, miközben lehetőséget ad a rendszer finomhangolására is.

Réteg	Megnevezés	Leírás és Funkció
1. Felhasználói réteg	User Application	A fejlesztő által írt kód, amely a robot specifikus logikáját (pl. navigáció, manipuláció, érzékelés) tartalmazza.
2. Nyelvi kliens könyvtárak	Client Libraries (rclcpp, rclpy)	A felhasználói kód által közvetlenül hívott API-k. A ROS 1-gyel ellentétben ezek a könyvtárak vékony burkolók (wrappers) egy közös C implementáció felett, biztosítva a viselkedésbeli konzisztenciát.
3. ROS kliens könyvtár	ROS Client Library (rcl)	A rendszer "agya". C nyelven írt közös implementáció, amely tartalmazza a node-ok kezelésének, a végrehajtás ütemezésének és a paraméterkezelésnek a logikáját. Ez csökkenti a kódduplikációt a különböző nyelvi bindingok között.
4. Middleware interfész	ROS Middleware Interface (rmw)	Egy absztrakciós réteg, amely elválasztja a ROS 2 logikát az alatta futó konkrét kommunikációs technológiától. Ez definiálja a szabványos API-t a middleware gyártók számára.
5. Middleware implementáció	DDS Implementation	A konkrét adatátviteli réteg (pl. Fast DDS, Cyclone DDS, RTI Connext). Ez felelős az üzenetek szerializációjáért, a hálózati felderítésért és a QoS szabályok érvényesítéséért.
6. Operációs rendszer	OS / RTOS	A hardver felett futó réteg. A ROS 2 támogatja a Linuxot, Windowst, macOS-t, valamint valós idejű rendszereket (pl. QNX, VxWorks).

1. táblázat A ROS 2 rétegei

Az 1. táblázatban látható architektúra, különösen az rcl és rmw rétegek szétválasztása, hatalmas eltérést jelent a ROS 1 monolitikus megközelítésétől [10]. A ROS 1-ben a Python és C++ könyvtárak teljesen különálló implementációk voltak, ami gyakran vezetett inkonzisztens viselkedéshez (pl. eltérő paraméterkezelés vagy időzítés). A ROS 2-ben az rcl réteg biztosítja, hogy a logika központi helyen legyen kezelve, így a Python (rclpy) és C++ (rclcpp) interfészek viselkedése egységesebb és kiszámíthatóbb [11].

3.2 A köztesréteg (Middleware)

A ROS 2 technológiai ugrását leginkább a kommunikációs infrastruktúra, azaz a middleware megváltoztatása testesíti meg. A korábbi verziók saját fejlesztésű, sokszor korlátozott képességű megoldásai helyett a ROS 2 az Object Management Group (OMG) által szabványosított Data Distribution Service (DDS) technológiát alkalmazza alapértelmezettként [12]. Ez a fejezet részletezi a DDS működési elvét, előnyeit, valamint az RMW réteg szerepét a technológiai függetlenség biztosításában.

3.2.1 A Data Distribution Service (DDS) szerepe és működése

A DDS egy adatközpontú kommunikációs szabvány, amelyet kifejezetten elosztott, valós idejű rendszerek számára terveztek. A DDS számos olyan problémát old meg natívan, amelyekkel a ROS 1 közösség évekig küzdött [12] [13].

A DDS működésének alapja a Globális Adattér (Global Data Space) koncepciója. A hálózatban részt vevő node-ok (Domain Participants) ebbe a virtuális térbe publikálnak adatokat, illetve ebből olvassák ki azokat. Ez a modell tökéletesen illeszkedik a ROS Publish-Subscribe paradigmájához, de a DDS ennél többet nyújt.

A DDS egyik legfontosabb tulajdonsága a dinamikus felfedezés. A résztvevők multicast (vagy konfigurálható unicast) üzenetek segítségével automatikusan megtalálják egymást a hálózaton. Amikor egy új node csatlakozik, meghirdeti a jelenlétét, a publikált és feliratkozott témáit, valamint a minőségi követelményeit (QoS). Nincs szükség központi szerverre vagy Master node-ra; a rendszer önszerveződő és hibatűrő. Ha egy node kiesik, a többi kommunikációja zavartalanul folytatódik [12].

A Real-Time Publish-Subscribe (RTPS) protokoll biztosítja az egymás közti működést a különböző gyártók DDS implementációi között. Ez azt jelenti, hogy egy rendszeren belül vegyesen használhatók különböző DDS megvalósítások, és ezek képesek zökkenőmentesen kommunikálni egymással [12] [14].

A DDS nem csupán bájtokat küld, hanem strukturált adatokat. Ismeri az adatok típusát és szerkezetét, ami lehetővé teszi a tartalom alapú szűrést és a hatékony szerializációt.

3.3 Kommunikációs alapfogalmak: csomópontok, témák és szolgáltatások

A ROS 2 rendszer működésének megértéséhez szükséges az architektúra alapvető építőelemeinek ismerete. A rendszer nem egyetlen monolitikus programként fut, hanem számos, egymással kommunikáló, független entitás hálózataként. Ez a fejezet a négy legfontosabb alapfogalmat: a Csomópontokat (Nodes), a Témákat (Topics), a Szolgáltatásokat (Services) és a Műveleteket (Actions) vizsgálja meg részletesen.

3.3.1 Csomópontok (Nodes)

A Node a ROS 2 legkisebb önálló szoftveres egysége. A modularitás elve alapján minden node egyetlen, jól körülhatárolt feladatért felelős. Egy tipikus robotrendszerben külön node felel a lézeres távolságmérő adatainak olvasásáért, egy másik a térképezésért, egy harmadik az útvonaltervezésért, és egy negyedik a motorvezérlésért.

A node-ok egyik legfontosabb tulajdonsága, hogy dinamikusan fedezik fel egymást a DDS mechanizmusai révén. Minden node rendelkezhet paraméterekkel, amelyek futásidőben konfigurálhatók (pl. egy szenzor mintavételezési frekvenciája).

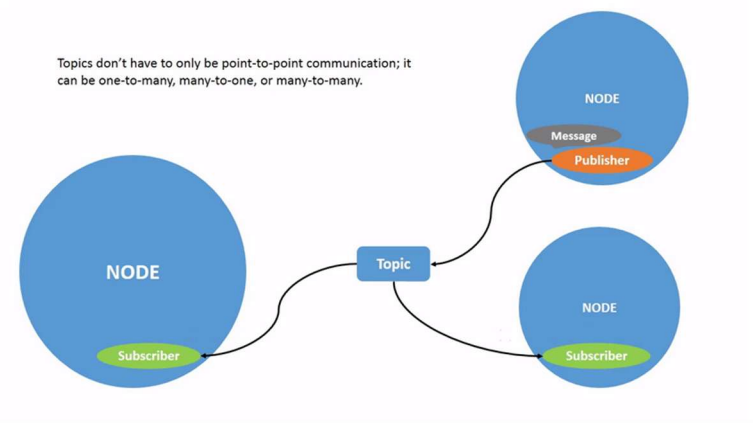
Fontos még megemlíteni, hogy a node-ok hierarchikus névterekbe szervezhetők (pl. /robot1/camera_driver, /robot2/camera_driver), ami lehetővé teszi több azonos típusú robot párhuzamos működtetését ütközések nélkül [15] [16].

3.3.2 Témák (Topics)

A Téma (Topic) a ROS 2 leggyakrabban használt kommunikációs csatornája, amely a Publish-Subscribe (Pub/Sub) mintát valósítja meg. Ez a mechanizmus a folyamatos adatfolyamok kezelésére szolgál.

A működési elve egyszerű, a Publisher (Közzétevő) node adatokat küld egy adott nevű témára. A Subscriber (Feliratkozó) node-ok, amelyek érdekeltek ebben az adatban, feliratkoznak a témára. A kommunikáció aszinkron: a küldő nem tudja, ki hallgatja (ha egyáltalán bárki), és nem vár választ. Ezt a működét szemlélteti az 1. ábra.

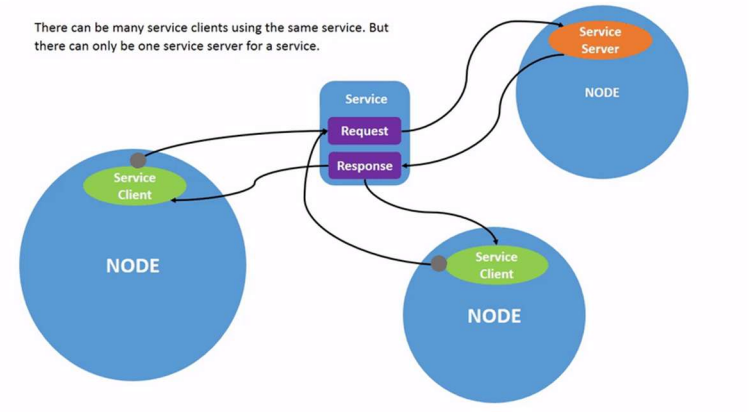
Minden téma szigorúan típusos. Az üzenettípusokat .msg fájlokban definiálják (pl. sensor_msgs/Image, geometry_msgs/Twist). A típusbiztonság garantálja, hogy a rendszer már a csatlakozáskor kiszűrje az inkompatibilis kommunikációt. A node-ok közötti kapcsolat "sok-a-sokhoz" (many-to-many) jellegű. Egy témára több node is publikálhat, és több node is feliratkozhat rá. Erre hívja fel a figyelmet az 1. ábra felirata. Például egy robot pozícióját (/odom téma) egyszerre használhatja a navigációs alrendszer, a vizualizációs eszköz (RViz) és egy adatgyűjtő node [16] [17].



1. ábra Témák publikálása és fogadása [18]

3.3.3 Szolgáltatások (Services)

Míg a témák folyamatos adatfolyamokra lettek kitalálva, bizonyos feladatok másféle interakciót igényelnek, ahol a kérésre adott válasz fontos. Erre szolgálnak a Szolgáltatások (Services), amelyek a Kliens-Szerver (Request-Response) modellt követik. Ezt a felépítést szemlélteti a 2. ábra. Az ábra felhívja a figyelmet arra, hogy több kliens is használhatja ugyan azt a szolgáltatást, de csak egy szolgáltatás szerver lehet szolgáltatásonként.



2. ábra Szolgáltatások működése [19]

A Service Server (Szolgáltató) node felkínál egy szolgáltatást egy adott néven. A Service Client (Kliens) kérést küld, és megvárja a választ. A szolgáltatások definíciója .srv fájlokban történik, amelyek két üzenetstruktúrát tartalmaznak: egyet a kérésnek (Request) és egyet a válasznak (Response), --- elválasztóval. A kliens oldali hívás lehet szinkron (blokkoló) vagy aszinkron. A ROS 2-ben az aszinkron hívások (Future/Promise mintával) preferáltak, mivel a blokkoló hívások megállíthatják a node teljes végrehajtását, ami deadlock helyzetekhez vezethet, ha a node egyszálú végrehajtót használ. Olyan műveleteknél érdemes használni, amelyek gyorsan lefutnak és nyugtázást igényelnek. Például szenzor kalibrálása, paraméter lekérdezése, matematikai számítás (pl. inverz kinematika) kérése, vagy egy kapcsoló átállítása. Nem alkalmas hosszú ideig tartó folyamatokra, mivel a kliens addig nem tud mással foglalkozni (vagy bonyolult aszinkron kezelést igényel), és nincs visszajelzés a folyamat állapotáról [16] [17].

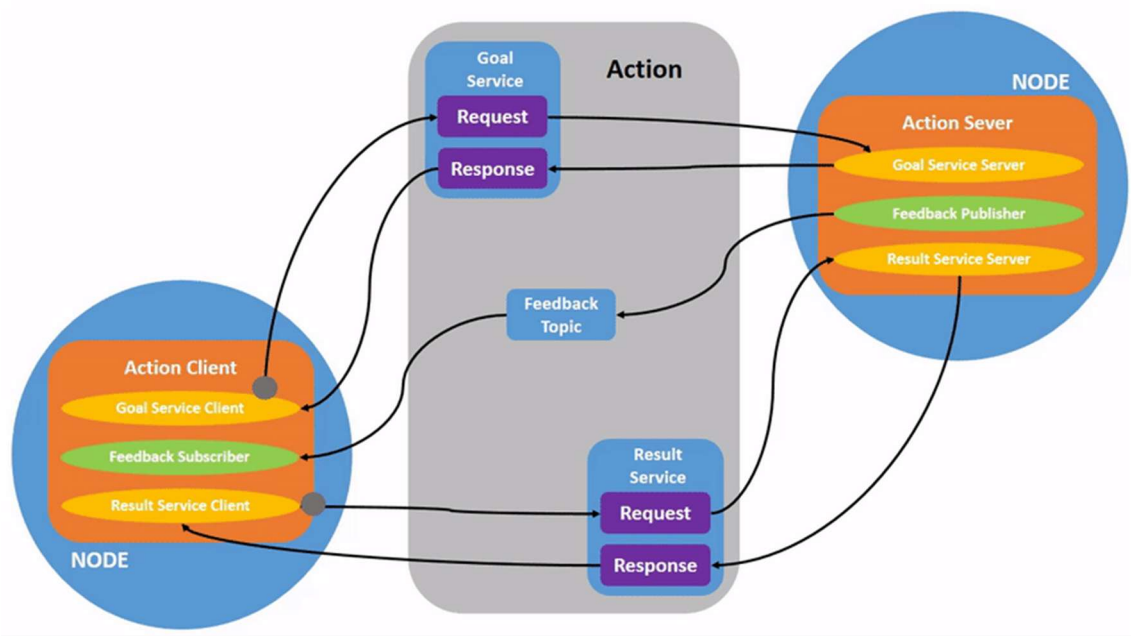
Tulajdonság	Topic (Téma)	Service (Szolgáltatás)
Kommunikációs Minta	Publish-Subscribe	Request-Response (Client-Server)
Kapcsolat típusa	Aszinkron, egyirányú stream	Szinkron/Aszinkron tranzakció
Kardinalitás	Many-to-Many (N:M)	One-to-One (1:1) - elvileg egy szerver
Ideális használat	Folyamatos adatok (szenzorok)	Gyors, diszkrét műveletek (lekérdezés)
Visszacsatolás	Nincs közvetlen visszacsatolás	Van (Response üzenet)

2. táblázat Topic és Service összehasonlítása

A 2. táblázat összefoglalja az eddig leírtakat a Topic-okról és Service-ekről. A Topic-ok alkotják a robot "idegrendszerét", ahol az információ folyamatosan áramlik, míg a Service-ek a specifikus, azonnali beavatkozások és lekérdezések eszközei. Azonban a robotikában gyakoriak a hosszú ideig tartó, komplex feladatok (pl. "navigálj a konyhába"), amelyekhez egyik fenti modell sem ideális. Erre a ROS 2 az Action mechanizmust kínálja.

3.3.4 Műveletek (Actions)

Ahogy a bevezetőben írtam, a robotikában gyakran előfordulnak olyan összetett feladatok, amelyek végrehajtása hosszabb időt vesz igénybe (másodperceket vagy perceket), és nem blokkolhatják a rendszer működését. Erre a problémára a Témák túl lazák, nincs nyugtázás, a Szolgáltatások pedig túl szinkron jellegűek, blokkolhatnak, ezért ezek nem nyújtanak megfelelő megoldást. A Műveletek (Actions) ezt az űrt töltik ki: egy hibrid kommunikációs típust valósítanak meg, amely a Kliens-Szerver modellt ötvözi a visszajelzések lehetőségével.



3. ábra Műveletek működése [20]

Az Action mechanizmus szintén Kliens (Action Client) és Szerver (Action Server) alapú, hasonlóan a Szolgáltatásokhoz, de három alapvető különbséggel.

Az első a hosszú lefutási idő. A folyamatok aszinkron módon zajlanak anélkül, hogy blokkolnák a hívó felet. Második a visszajelzés (Feedback). A végrehajtás közben a szerver folyamatos státuszjelentést küldhet a kliensnek (pl. "a navigáció 50%-nál tart"). Harmadik pedig a megszakíthatóság (Cancel). A kliensnek lehetősége van a műveletet futás közben leállítani (pl. ha a robot akadályt észlel, vagy új parancsot kap).

A háttérben az Action valójában nem egy új, elemi kommunikációs protokoll, hanem a Topic-ok és Service-ek ügyes kombinációja. Amikor létrehozunk egy Action-t, a rendszer a háttérben automatikusan generál szolgáltatásokat a cél (Goal) kitűzésére és az eredmény (Result) lekérésére. Témákat (Topic) generál a visszajelzések (Feedback) és a státusz (Status) folyamatos publikálására. Ez a felépítést mutatja be a 3. ábra [20].

Hasonlóan a .msg és .srv fájlokhoz, az Action-öket .action kiterjesztésű fájlokban definiálják. A struktúra három részt tartalmaz [20]:

Goal (Cél): A kérés paraméterei (pl. Célkoordináta X, Y).

Result (Eredmény): A végső kimenet a művelet befejezésekor.

Feedback (Visszajelzés): Időszakos adatok a futás alatt.

Az Action mechanizmus minden olyan esetben ideális, ahol a robotnak fizikai mozgást kell végeznie, vagy komplex számítást futtatnia.

Összefoglalva míg a Topic a folyamatos adatfolyam (szenzorok), a Service a gyors "kérdés-válasz" (kapcsolók), addig az Action a "menedzselhető folyamatok" eszköze a ROS 2-ben.

3.4 Ipari alkalmazhatóság, valós idejű működés

A ROS 2 fejlesztésének legfőbb hajtóereje az volt, hogy a keretrendszert alkalmassá tegyék az ipari termelés és a biztonságkritikus alkalmazások szigorú követelményeinek teljesítésére.

A robotikában a valós idejű (real-time) kifejezés gyakran félreértett. Nem feltétlenül azt jelenti, hogy a rendszer gyors (low latency), hanem azt, hogy determinisztikus. A rendszernek garantálnia kell, hogy egy kritikus számítási feladat (pl. vészfékezés) egy előre meghatározott időablakon (deadline) belül minden körülmények között lefut. Ha a határidőt túllépjük (jitter), a rendszer instabillá válhat vagy balesetet okozhat [21].

A ROS 2 a következő technológiai megoldásokkal támogatja a valós idejű követelményeket. Az egyik az RTOS és kernel támogatás. A ROS 2 futtatható valós idejű operációs rendszereken (RTOS), mint például a QNX, VxWorks, vagy a PREEMPT_RT patch-csel ellátott Linux kernelen. Ezek az operációs rendszerek garantálják a szálütemezés pontosságát és a megszakítások (interrupts) kiszámítható kezelését. A másik ilyen technológiai megoldás a memóriaallokáció kontrollálása. A valós idejű rendszerek egyik legnagyobb ellensége a nem-determinisztikus memóriakezelés. A standard malloc és new hívások ideje nem garantált (keresni kell szabad blokkot), és memóriatöredezettséget (fragmentation) okozhatnak, ami idővel lassuláshoz vezet. A ROS 2 C++ könyvtárai (rclcpp) támogatják az egyedi memória-allokátorokat (custom allocators). A kritikus ciklusokban (real-time loop) a fejlesztőknek kerülniük kell a dinamikus foglalást. Ehelyett előre lefoglalt memóriapool-okat vagy O(1) komplexitású allokátorokat (pl. TLSF - Two-Level Segregate Fit) használnak, amelyek garantált idő alatt szolgálják ki a kéréseket. A virtuális memória laphibáinak elkerülése érdekében gyakran alkalmazzák az mlockall rendszerhívást a fizikai memóriába való rögzítéshez [21].

4 Felhőalapú rendszerek és a Kubernetes

4.1 Bevezetés a modern infrastruktúra-architektúrákba

Az elmúlt évtizedben jelentősen átalakult az informatikai infrastruktúrák tervezése és üzemeltetése. A fizikai hardverhez kötött, monolitikus alkalmazások korszaka lezárult, helyét átvette a szoftvervezérelt, dinamikusan skálázható és elosztott rendszerek világa. Ez az átalakulás szükséges volt. A digitális gazdaságban a gyorsaság, a rugalmasság és a költséghatékonyság váltak a versenyképesség elsődleges mérőszámaivá. A fejezet célja, hogy bemutassa a felhőalapú számítástechnika alapjait, a konténerizáció elterjedését, a Kubernetes orkesztrációs platform architektúráját, valamint a ROS 2 integrációjának speciális kihívásait ebben a környezetben.

Az fejezet során a Nemzeti Szabványügyi és Technológiai Intézet (NIST) definícióitól indulva haladok a gyakorlati megvalósítások, például a Docker és a Kubernetes belső mechanizmusai felé.

4.2 A felhőalapú számítástechnika rendszertana és gazdasági modellje

A felhőalapú számítástechnika (Cloud Computing) definíciója szerint egy olyan modell, amely lehetővé teszi a hálózati hozzáférést egy megosztott, konfigurálható számítástechnikai erőforráskészlethez (például hálózatokhoz, szerverekhez, tárolókhoz, alkalmazásokhoz és szolgáltatásokhoz), amely gyorsan, minimális menedzsment-erőfeszítéssel vagy szolgáltatói interakcióval rendelkezésre bocsátható. Ez a definíció csak a felszínt karcolja, a mélyebb megértéshez vizsgálnunk kell a szolgáltatási modellek (SPI modell: Software, Platform, Infrastructure) közötti finom különbségeket és azok operációs hatásait [22].

4.2.1 A szolgáltatási modellek (SPI) részletes elemzése

A felhőalapú szolgáltatások nem csupán technológiai rétegeket jelölnek, hanem felelősségmegosztási modelleket is. Ahogy haladunk az IaaS-tól a SaaS felé, úgy csökken a felhasználó kontrollja az infrastruktúra felett, és úgy nő a szolgáltató felelőssége a karbantartásért és biztonságért.

4.2.1.1 Infrastructure as a Service (IaaS)

Az Infrastruktúra mint Szolgáltatás (IaaS) a felhőalapú számítástechnika alapja. Ebben a modellben a szolgáltató biztosítja a fizikai adatközpontot, a hűtést, az áramellátást, a fizikai biztonságot, valamint a hálózati és tároló hardvereket. A felhasználó számára ezek az erőforrások virtualizált formában jelennek meg [23]. A NIST definíciója szerint az IaaS képességeket biztosít a fogyasztónak a feldolgozás, tárolás, hálózatok és egyéb alapvető számítástechnikai erőforrások igénybevételére, ahol a fogyasztó tetszőleges szoftvereket telepíthet és futtathat, beleértve az operációs rendszereket és alkalmazásokat [22] [24].

Az IaaS modell legfontosabb gazdasági előnye a CapEx (beruházási költség) átváltása OpEx-re (működési költség). Ahelyett, hogy szervereket vásárolnának, a vállalatok bérlik a kapacitást. Ez a Rapid Elasticity (gyors rugalmasság) elvével párosulva lehetővé teszi a horizontális és vertikális skálázást a terhelés függvényében. Ha egy webshop forgalma karácsonykor megtízszereződik, az infrastruktúra dinamikusan bővülhet, majd januárban visszaskálázódhat, elkerülve a kihasználatlan kapacitás fenntartását [22].

Az IaaS szintjén a felhasználó felelőssége a legkiterjedtebb. Bár a fizikai hardvert nem látja, felelős a virtuális gépek (VM) operációs rendszerének frissítéséért (patching), a tűzfalak konfigurálásáért és az alkalmazások telepítéséért. Ez a Managed by User terület magában foglalja az operációs rendszer, a köztesréteg (middleware), a futtatókörnyezet és az adatok kezelését [25]. Tipikus példák az Amazon EC2, a Google Compute Engine és a Microsoft Azure Virtual Machines.

4.2.1.2 Platform as a Service (PaaS)

A PaaS modell egy absztrakciós réteggel feljebb lép. Itt a szolgáltató nemcsak az infrastruktúrát, hanem a futtatókörnyezetet (runtime), az operációs rendszert és a köztesrétegeket is menedzseli. A NIST szerint a PaaS lehetővé teszi a fogyasztó számára, hogy a szolgáltató által támogatott programozási nyelvek, könyvtárak, szolgáltatások és eszközök segítségével létrehozott alkalmazásokat telepítsen a felhőinfrastruktúrára [22] [24].

Ez a modell csökkenti a Time-to-Market mutatót, mivel a fejlesztőknek nem kell az infrastruktúra konfigurálásával, vagy a szerverek felügyeletével foglalkozniuk. A fókusz kizárólag az üzleti logikára és az alkalmazáskódra helyeződik át. A PaaS környezetek gyakran tartalmaznak beépített eszközöket a CI/CD (Continuous Integration / Continuous Deployment) folyamatokhoz, automatikus skálázáshoz és monitorozáshoz [23].

A PaaS modellben a támadási felület megváltozik. Mivel az operációs rendszert a szolgáltató kezeli, a felhasználónak nincs lehetősége például kernel szintű módosításokra vagy egyedi OS konfigurációkra. Ugyanakkor az OS szintű sebezhetőségek javítása a szolgáltató feladata, ami gyakran gyorsabb és szakszerűbb, mint amit egy átlagos vállalat házon belül meg tudna valósítani [26]. A korlátot a rugalmasság elvesztése jelenti: ha az alkalmazás speciális könyvtárakat vagy OS beállításokat igényel, a PaaS korlátozó lehet.

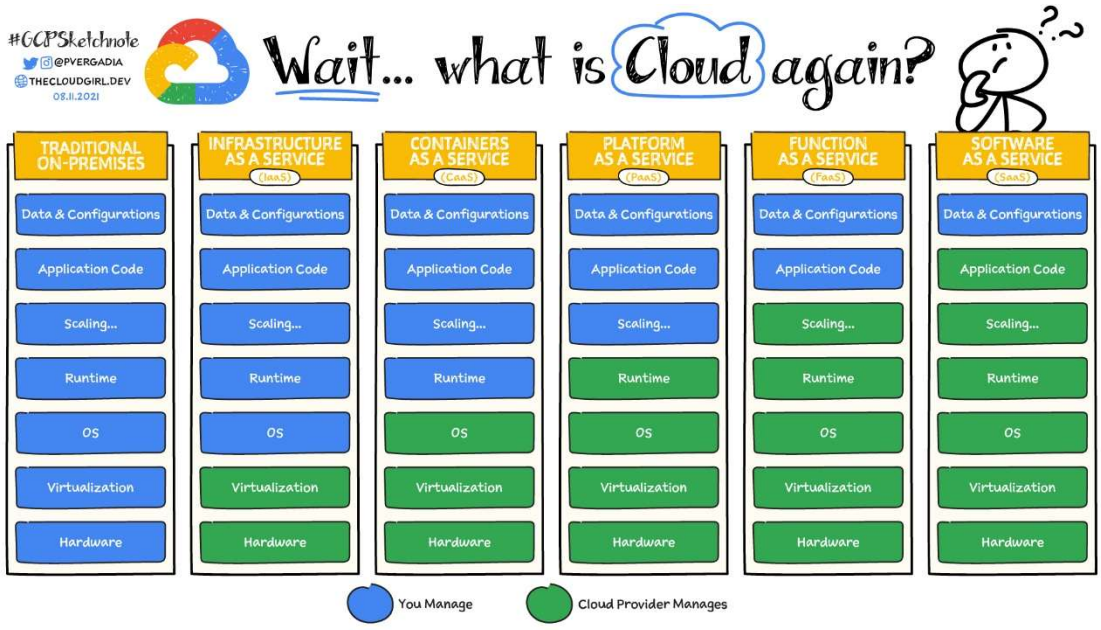
4.2.1.3 Software as a Service (SaaS)

A SaaS a legmagasabb szintű absztrakció, ahol a szoftver teljes egészében szolgáltatásként jelenik meg. A végfelhasználó számára az alkalmazás egy webböngészőn vagy vékonykliensen keresztül érhető el, anélkül, hogy bármit telepítenie kellene a saját eszközére. A szolgáltató felelős a teljes stackért: az adatközponttól kezdve a hálózaton és szervereken át egészen az alkalmazáskódig és az adatbiztonságig [23] [25].

A NIST definíciója hangsúlyozza, hogy a SaaS modellben a fogyasztónak nincs ellenőrzése az alapul szolgáló infrastruktúra felett, beleértve a hálózatot, szervereket, operációs rendszereket, tárhelyet, sőt, még az egyes alkalmazásképességeket sem, kivéve a korlátozott felhasználó-specifikus konfigurációs beállításokat [22].

Ez a modell a multi-tenancy (több bérlős) architektúrára épül, ahol egyetlen szoftverpéldány szolgál ki több ezer vagy millió felhasználót, logikailag elszeparálva az adataikat. Ez méretgazdaságossági szempontból rendkívül előnyös a szolgáltatónak, és alacsony belépési költséget jelent a felhasználónak.

Az előző alfejezetek leírásait foglalja össze az 4. ábra táblázata. A zöld mezők jelölik azokat az erőforrásokat, amiket a szolgáltató kezel, kékkel pedig azokat látjuk, amiket a felhasználónak kell kezelnie.



4. ábra Szolgáltatási modellek [25]

4.2.2 Telepítési modellek (Deployment Models)

A szolgáltatási modellek mellett a NIST négy telepítési modellt is megkülönböztet, amelyek az infrastruktúra tulajdonlását és hozzáférhetőségét írják le [24]:

Nyilvános felhő (Public Cloud): Az infrastruktúra nyitott a nagyközönség számára.

Magánfelhő (Private Cloud): Az infrastruktúra kizárólag egyetlen szervezet számára dedikált.

Közösségi felhő (Community Cloud): Az infrastruktúrát több szervezet osztja meg, amelyek közös érdekekkel rendelkeznek (pl. biztonsági követelmények).

Hibrid felhő (Hybrid Cloud): Két vagy több különböző felhőinfrastruktúra (magán, közösségi vagy nyilvános) kompozíciója, amelyek önálló entitások maradnak, de szabványosított vagy saját technológia köti össze őket, lehetővé téve az adatok és alkalmazások hordozhatóságát [24].

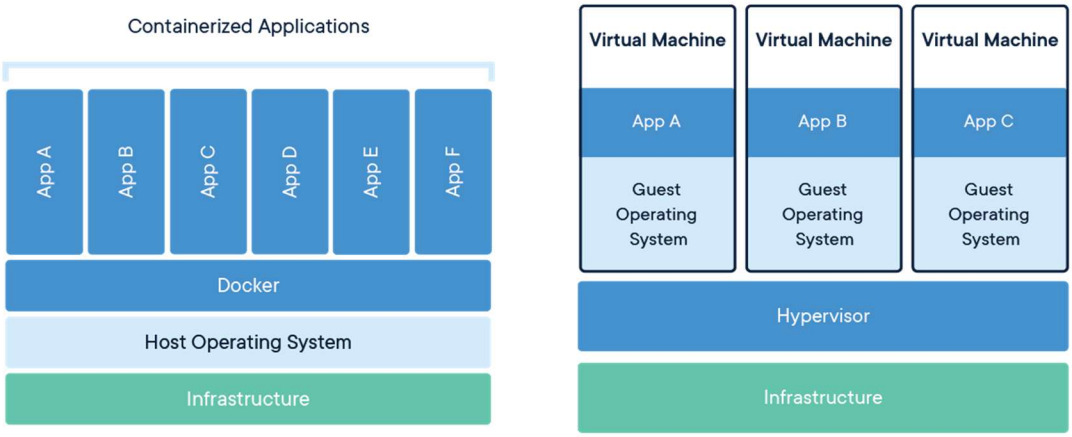
4.3 Konténertechnológiák

A felhőalapú rendszerek fejlődésének következő lépése a hardvervirtualizációtól (VM) az operációs rendszer szintű virtualizáció (konténerizáció) felé történő elmozdulás volt. Ez a technológiai váltás alapjaiban változtatta meg a szoftverfejlesztési és üzemeltetési (DevOps) gyakorlatokat.

4.3.1 Virtuális gépek (VM) és konténerek

A hagyományos virtuális gépek (VM) és a konténerek közötti alapvető különbség az izoláció szintjében és az erőforrás-megosztás módjában rejlik.

A Virtuális gépek működésének alapja a Hypervisor (vagy VMM - Virtual Machine Monitor). A Hypervisor feladata, hogy hardver-erőforrásokat (CPU, memória, I/O) emuláljon a vendég operációs rendszerek számára. Minden VM rendelkezik egy teljes saját operációs rendszerrel (Guest OS), saját kernellel, binárisokkal és könyvtárakkal. Ez látható a 5. ábra bal oldalán. Ez erős izolációt biztosít, mivel a VM-ek teljesen el vannak választva egymástól és a gazdagéptől is. Ugyanakkor ez jelentős overhead-del jár: minden egyes VM futtatása több gigabájt memóriát és lemezterületet foglal el, még akkor is, ha az alkalmazás maga minimális erőforrást igényelne. A rendszerindítás (boot) ideje percekben mérhető, mivel a teljes OS-nek be kell töltődnie [27] [28].



5. ábra Konténerek és VM-ek architektúrája [29]

Ezzel szemben a Konténerek az operációs rendszer szintjén virtualizálnak. A konténerek nem emulálnak hardvert, hanem közvetlenül a gazdagép (Host OS) kernelét használják. Az izolációt a Linux kernel két alapvető funkciója, a Namespaces (névterek) és a Cgroups (vezérlési csoportok) biztosítja.

A namespace mechanizmus biztosítja, hogy a konténer úgy érezze, mintha egy saját, dedikált operációs rendszeren futna. Saját folyamatlistát (PID), hálózati interfészeket (NET), fájlrendszert (MNT) és felhasználókat (USER) lát, elzárva a többi konténertől és a gazdagéptől.

A cgroups funkció felelős az erőforrások korlátozásáért és méréséért. Megakadályozza, hogy egy konténer felemássze a gazdagép összes CPU idejét vagy memóriáját, biztosítva a "szomszédok" békés együttélését [28].

Mivel a konténerek osztoznak a kernelen, és csak az alkalmazást, valamint annak függőségeit (binárisok, könyvtárak) tartalmazzák (ez a kialakítás látható a 5. ábra jobb oldalán), méretük rendkívül kicsi, csupán néhány megabájt és indításuk szinte azonnali, milliszekundumokban mérhető. Ez teszi őket ideálissá a mikroszolgáltatás-alapú architektúrákhoz, ahol a gyors skálázás kritikus [29].

4.3.2 A Docker szerepe és technológiai újításai

Bár a konténerizáció alapjai már a 2000-es években léteztek, a technológia széles körű elterjedését a Docker hozta el 2013-ban. A Docker egy olyan eszközkészletet és ökoszisztémát adott a fejlesztők kezébe, amely egyszerűvé és hordozhatóvá tette azok használatát.

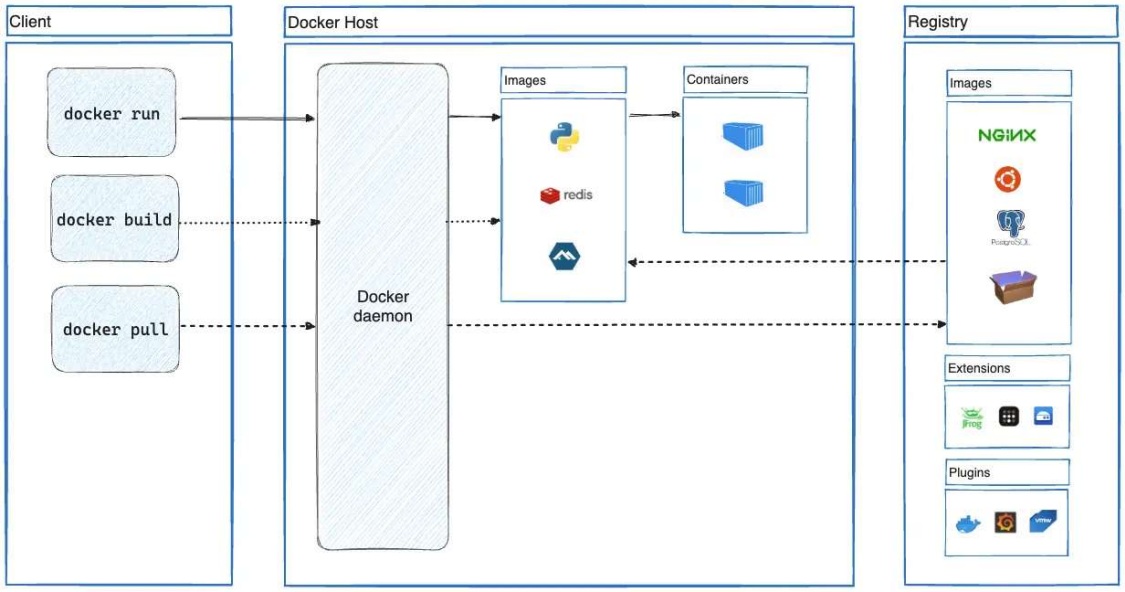
4.3.2.1 Docker Architektúra

A Docker egy kliens-szerver architektúrát alkalmaz, amely a 6. ábra diagramján látható.

Egy Docker Daemon **(**dockerd**)** háttérfolyamat fut a gazdagépen. Ez a komponens felelős a konténerobjektumok (képek, konténerek, hálózatok, kötetek) létrehozásáért és kezeléséért. Közvetlenül kommunikál a Linux kernellel.

A Docker Client (docker) a parancssori felület (CLI), amelyen keresztül a felhasználó utasításokat küld a démonnak (pl. docker build, docker run). A kliens és a démon kommunikálhat helyi socketen vagy REST API-n keresztül távolról is.

A Docker Registry konténerképfájlok (Images) tárolására és megosztására szolgáló központi tárház. A legismertebb a nyilvános Docker Hub, de vállalatok gyakran üzemeltetnek privát registry-ket is [30].



6. ábra Docker architektúra [30]

4.3.2.2 Docker Images és rétegek (Layers)

A Docker egyik legnagyobb újítása a rétegelt fájlrendszer (Union File System) használata. Egy Docker Image egy olvasható sablon, amely rétegekből épül fel. Minden utasítás a Dockerfile-ban (pl. RUN apt-get install python) egy új réteget hoz létre az előző tetején. Amikor egy konténer elindul, a Docker egy vékony, írható réteget (Container Layer) helyez a csak olvasható Image rétegek fölé. Ez lehetővé teszi a rendkívül hatékony tárolást és sávszélesség-kihasználást: ha több konténer ugyanazon az alap Image-en (pl. Ubuntu) alapul, a lemezen csak egyszer tárolódik az alapréteg, és minden konténer csak a saját módosításait tárolja [27].

A Docker legfőbb ígérete a "Build once, run anywhere" (Egyszer megépíted, bárhol futtatod). Mivel a konténer magában foglalja az alkalmazáskódot és az összes függőséget (könyvtárak, konfigurációk), garantált, hogy a fejlesztői laptopon futó szoftver ugyanúgy fog viselkedni a tesztkörnyezetben és az éles szerveren is. Ez kiküszöböli a klasszikus "matrix of hell" problémát, ahol a különböző környezetek inkompatibilitása okozott hibákat [27] [30].

4.4 A Kubernetes architektúrája és működési mechanizmusai

Ahogy a konténerek száma egyre nőtt, és a monolitikus alkalmazások helyét átvették a több száz mikroszolgáltatásból álló rendszerek. Szükségessé vált egy eszköz, amely automatizálja a konténerek telepítését, skálázását és menedzselését. A Google által fejlesztett, majd a Cloud Native Computing Foundation (CNCF) számára átadott Kubernetes (K8s) vált az iparági standarddá ezen a téren.

A Kubernetes egy deklaratív rendszer. A felhasználó leírja a rendszer kívánt állapotát (Desired State) YAML vagy JSON formátumban, a Kubernetes pedig folyamatosan dolgozik azon, hogy a tényleges állapot (Actual State) megegyezzen ezzel. Ez az öngyógyító mechanizmus a rendszer alapja [31].

4.4.1 A vezérlősík (Control Plane)

A Kubernetes architektúrája egy elosztott rendszer, amely egy központi vezérlőegységből (Control Plane) és munkavégző csomópontokból (Worker Nodes) áll. A Control Plane felelős a globális döntésekért és az események kezeléséért. Ennek a felépítését a 7. ábra szemlélteti.

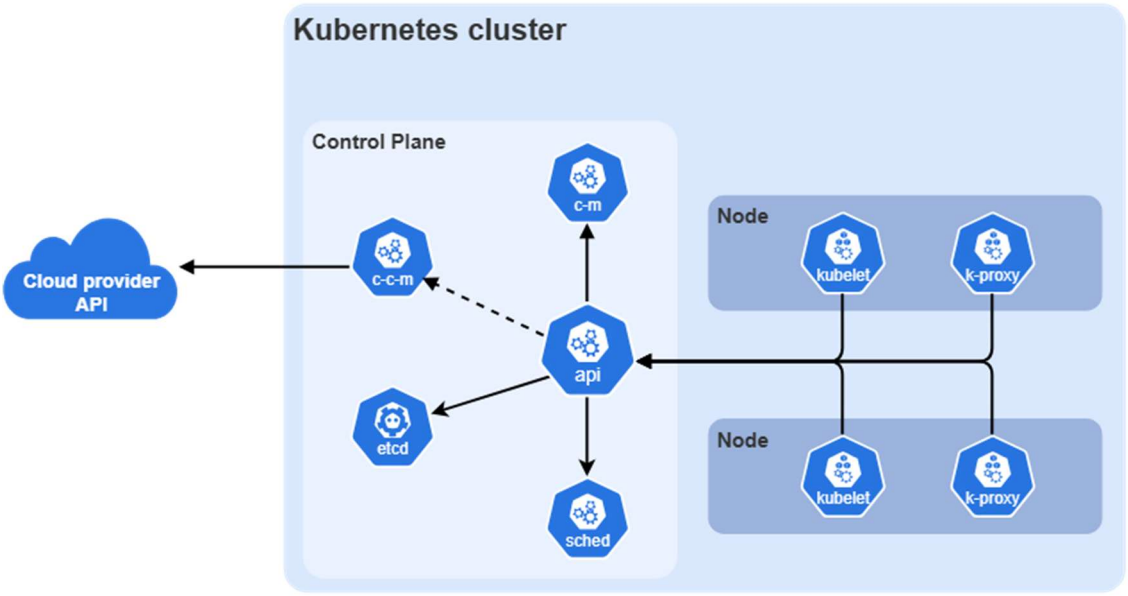
A kube-apiserver (api) komponens a Kubernetes bejárata. Minden kommunikáció – legyen az felhasználói parancs (kubectl), belső komponensek közötti kommunikáció vagy automatizált szkriptek – ezen a REST API felületen keresztül zajlik. Az API szerver végzi a kérések hitelesítését (Authentication), jogosultságkezelését (Authorization) és validálását, mielőtt azokat perzisztálná. Mivel állapottal nem rendelkezik (stateless), könnyen skálázható horizontálisan több példány futtatásával.

Az etcd egy elosztott, konzisztens kulcs-érték tár, amely a Kubernetes memóriája. Minden klaszteradatot (konfigurációk, állapotok, titkok) itt tárolnak. Ez az "egyetlen igazság forrása" (Single Source of Truth). Mivel a teljes rendszer működése ettől függ, az etcd-t szigorú konzisztenciát biztosító algoritmus vezérli, és termelési környezetben mindig biztosítják a magas rendelkezésre állását.

A kube-scheduler (sched) feladata eldönteni, hogy az újonnan létrehozott podok melyik Node-on fussanak. Ez egy komplex optimalizálási probléma: a döntés során figyelembe veszi az erőforrásigényeket (CPU, RAM), a hardveres/szoftveres megszorításokat (pl. GPU megléte), az affinitási szabályokat (mely podok szeretik vagy utálják egymást) és a "taint/toleration" beállításokat.

A kube-controller-manager (c-m) komponens futtatja a vezérlő hurkokat (Controller Loops). Ezek a háttérfolyamatok folyamatosan figyelik a rendszer állapotát az API szerveren keresztül, és beavatkoznak, ha eltérést tapasztalnak. Például a Node Controller figyeli a csomópontok elérhetőségét; ha egy Node leáll, elindítja a rajta lévő podok újraütemezését. A Replication Controller biztosítja, hogy mindig a megfelelő számú pod példány fusson.

A cloud-controller-manager (c-c-m) komponens választja el a Kubernetes belső logikáját a felhőszolgáltató-specifikus megvalósításoktól. Lehetővé teszi, hogy a Kubernetes kommunikáljon az AWS, Google Cloud vagy Azure API-jaival, például Load Balancerek létrehozásához vagy tárolókötetek (Volumes) felcsatolásához [32] [33].



7. ábra Kubernetes klaszter komponensei [34]

4.4.2 A munkavégző csomópontok (Worker Nodes)

A Node-ok (lehetnek fizikai szerverek vagy VM-ek) azok az egységek, ahol az alkalmazások ténylegesen futnak. A Control Plane irányítja őket. Ők a 7. ábra jobb oldalán láthatók.

Minden Node-on fut egy kubelet ügynök. Ez a "kapitány" a hajón: kommunikál a Control Plane-nel, fogadja a PodSpec definíciókat, és gondoskodik arról, hogy a konténerek elinduljanak és fussanak. Rendszeresen jelenti a Node és a podok állapotát az API szervernek [32].

A kube-proxy felelős a Kubernetes hálózati modelljének megvalósításáért a Node-on. Hálózati szabályokat tart karban (Linux iptables vagy IPVS segítségével), amelyek lehetővé teszik a hálózati kommunikációt a podok között a klaszteren belül és kívül [33].

A Container Runtime szoftver, amely a konténerek tényleges futtatását végzi. Bár a Docker volt a legelterjedtebb, a Kubernetes bevezette a Container Runtime Interface-t (CRI), amely lehetővé teszi más, könnyűsúlyú futtatókörnyezetek használatát is, mint például a containerd vagy a CRI-O. Ez a modularitás növeli a rendszer rugalmasságát és csökkenti a függőséget egyetlen gyártótól [33] [35].

4.5 Kubernetes alapfogalmak és objektumok

A Kubernetes működésének megértéséhez szükséges az alapvető objektumok ismerete. Ezek az absztrakciók teszik lehetővé a komplex alkalmazások leírását és menedzselését.

4.5.1 Pod: A legkisebb egység

A Kubernetes legkisebb egysége nem a konténer, hanem a Pod. Egy Pod egy vagy több konténert foglal magában, amelyek szorosan összetartoznak (pl. egy webkiszolgáló és egy naplógyűjtő sidecar konténer). A Podon belüli konténerek osztoznak a tárhelyen és a hálózati névtéren (Network Namespace), vagyis közös IP-címmel rendelkeznek, és localhost-on keresztül kommunikálhatnak egymással. A podok múlandó természetűek, tehát nem tervezik őket meggyógyítani. Ha egy Node meghal, a rajta lévő podok is elvesznek, és a rendszer újakat indít helyettük [36].

4.5.2 Deployment és ReplicaSet: Az alkalmazás életciklusa

Mivel a podok múlandóak, közvetlen menedzselésük nehézkes. A Deployment egy magasabb szintű absztrakció, amely deklaratív módon írja le az alkalmazás kívánt állapotát. A Deployment létrehoz egy ReplicaSet-et, amelynek egyetlen feladata biztosítani, hogy a megadott számú pod példány (replica) mindig fusson. A Deploymentek teszik lehetővé a verzióváltásokat (Rolling Updates) és a visszagörgetést (Rollback) állásidő nélkül [36].

4.5.3 Service és Ingress: Hálózati elérés

A podok IP-címei dinamikusan változnak. A Service egy stabil hálózati absztrakciót biztosít egy logikai pod-csoport felett. A Service egy fix IP-címet (ClusterIP) és DNS nevet rendel a podokhoz, és terheléselosztást végez közöttük.

Típusai:

- **ClusterIP:** Csak a klaszteren belüli elérést biztosít (alapértelmezett).
- **NodePort:** Minden Node egy adott portján teszi elérhetővé a szolgáltatást.
- **LoadBalancer:** A felhőszolgáltató külső terheléselosztóját integrálja.

Míg a Service a TCP/UDP rétegen működik, az Ingress a HTTP/HTTPS (Layer 7) forgalom menedzselésére szolgál. Lehetővé teszi az URL alapú routingot, SSL terminálást és virtuális hosztok kezelését, így egyetlen IP-címen több szolgáltatás is publikálható [36].

4.6 Hálózati kommunikáció és biztonság

A Kubernetes hálózati modellje egy flat (lapos) hálózatot feltételez, ahol minden pod közvetlenül elérhet minden más podot NAT nélkül. Ez a modell egyszerűsíti az alkalmazások tervezését, de komoly biztonsági kihívásokat is rejt.

4.6.1 Container Network Interface (CNI)

A Kubernetes nem implementálja magát a hálózatot, hanem egy interfészt (CNI) definiál, amelyet külső pluginok valósítanak meg. Ezek a pluginok felelősek az IP-címek kiosztásáért (IPAM) és a routingért [35] [37].

Két fő megközelítés létezik. Az egyik az overlay hálózatok (Encapsulation). Itt a podok forgalmát egy másik protokollba (pl. VXLAN) csomagolják, hogy átvigyék a fizikai hálózaton. Ez rugalmas, de extra csomagolási költséggel (overhead) jár (pl. Flannel) [37].

A másik pedig a routed hálózatok (Unencapsulated). Ennél a podok IP-címei közvetlenül routolhatók a hálózaton (pl. BGP használatával). Ez nagyobb teljesítményt nyújt és alacsonyabb késleltetést, de a fizikai hálózat támogatását igényli (pl. Calico) [37].

4.7 Skálázás, és monitorozás

A Kubernetes egyik legnagyobb értéke az automatizált üzemeltetés képessége.

4.7.1 Automatikus Skálázás (Autoscaling)

A rendszer három dimenzióban képes alkalmazkodni a terheléshez:

Horizontal Pod Autoscaler (HPA): A podok számát (replikák) növeli vagy csökkenti a CPU/RAM terhelés alapján. Ideális a változó forgalmú, állapotmentes (stateless) alkalmazásokhoz. Reakcióideje percekben mérhető.

Vertical Pod Autoscaler (VPA): A meglévő podok erőforráskereteit (requests/limits) módosítja. Olyan alkalmazásokhoz ajánlott, amelyek nehezen skálázhatók horizontálisan (pl. régi monolitok vagy adatbázisok).

Cluster Autoscaler (CA): Infrastruktúra szintű skálázás. Ha a podok nem férnek el a meglévő Node-okon, a CA új Node-okat indít a felhőszolgáltatónál. Ha a Node-ok kihasználatlanok, lekapcsolja őket költségmegtakarítás céljából [38].

4.7.2 Monitorozás: Prometheus és Grafana

Dinamikus környezetben a statikus monitorozás hatástalan. Az iparági szabvány a Prometheus, egy idősoros adatbázis (TSDB), amely "pull" modellt használ: rendszeresen lekérdezi (scrape) a metrikákat a célpontoktól. Lekérdező nyelve, a PromQL, rendkívül rugalmas [39]. A Grafana biztosítja a vizualizációt: a Prometheus adataiból látványos, valós idejű kimutatásokat (dashboards) épít, segítve az üzemeltetőket a trendek és anomáliák azonosításában [40].

4.8 Konklúzió

A felhőalapú rendszerek és a Kubernetes ökoszisztéma érettsége mára lehetővé teszi a legkomplexebb, elosztott alkalmazások futtatását is. Az infrastruktúra (IaaS) és a platform (PaaS) rétegek absztrakciója, párosulva a konténerizáció hatékonyságával, soha nem látott rugalmasságot biztosít.

5 Rendszertervezés

5.1 Bevezetés a rendszertervezésbe

A modern robotikai rendszerek fejlesztése során az egyik legnagyobb kihívást a számítási kapacitás és a valós idejű működés közötti egyensúly megteremtése jelenti. A szakdolgozat keretében egy olyan hibrid, felhőalapú architektúra tervezését és megvalósítását tűztem ki célul, amely képes áthidalni a fizikai robotok korlátozott erőforrásai és a felhő technológiák által kínált skálázhatóság közötti szakadékot. Ez a fejezet a rendszertervezés folyamatát, a meghozott architektúrális döntéseket, valamint az implementáció részletes megvalósítását tárgyalja.

A rendszertervezés során kiemelt figyelmet fordítottam az ipari szabványok alkalmazására. A ROS 2 (Robot Operating System 2) keretrendszer használata biztosítja a modularitást és a hardver-függetlenséget, míg a Docker konténerizáció és a Kubernetes orkesztráció a szoftverkomponensek hordozhatóságát és menedzselhetőségét garantálja. Az alábbiakban részletesen elemzem a rendszer topológiáját, a kommunikációs protokollok kiválasztásának indokait, valamint a szoftverkomponensek belső logikáját, folyamatosan összevetve a tervezési szándékot a tényleges implementációval, feltárva az esetleges eltéréseket és biztonsági kockázatokat.

5.2 Rendszer-architektúrális döntések és topológia

A rendszertervezés legkritikusabb fázisa a megfelelő topológia és kommunikációs minták kiválasztása volt. A feladat komplexitását a változatos környezet adta. A determinisztikus, alacsony késleltetésű robotikai hálózatot kell összekapcsolni a nem-determinisztikus, TCP/IP alapú felhő infrastruktúrával.

5.2.1 A kommunikációs híd: A DDS korlátai és az MQTT választása

A ROS 2 alapértelmezett köztesrétege (middleware) a DDS (Data Distribution Service), amely egy decentralizált, UDP multicast alapú kommunikációs szabvány. Bár a DDS kiválóan teljesít lokális hálózatokon (LAN), a felhőalapú kiterjesztése számos technikai akadályba ütközik, amelyeket a tervezés során figyelembe kellett vennem.

Az egyik ilyen probléma a multicast routing hiánya. A Kubernetes hálózati modelljei (CNI - Container Network Interface) és a nyilvános felhőszolgáltatók infrastruktúrája gyakran nem támogatják, vagy biztonsági okokból tiltják a multicast forgalmat. Mivel a DDS Discovery mechanizmusa (a résztvevők automatikus felfedezése) alapértelmezésben multicast csomagokra épül, a ROS 2 node-ok láthatatlanok maradnának egymás számára a felhő és a robot között, hacsak nem alkalmazunk komplex Discovery Server konfigurációt.

A másik kihívás a NAT (Network Address Translation) átjárhatósága. A robotok gyakran privát hálózatokon, NAT mögött helyezkednek el (pl. 4G/5G mobilhálózat vagy vállalati Wi-Fi). A DDS peer-to-peer adatátvitele megköveteli, hogy a végpontok közvetlenül címezhetők legyenek, ami NAT környezetben csak bonyolult VPN (Virtual Private Network) megoldásokkal hidalható át.

Végül az utolsó probléma a dinamikus portkezelés. A DDS szabvány dinamikusan rendel UDP portokat a különböző témákhoz (Topics). Ez a viselkedés összeegyeztethetetlen a szigorú tűzfal-szabályokkal és a Kubernetes Service objektumok statikus port-továbbítási logikájával.

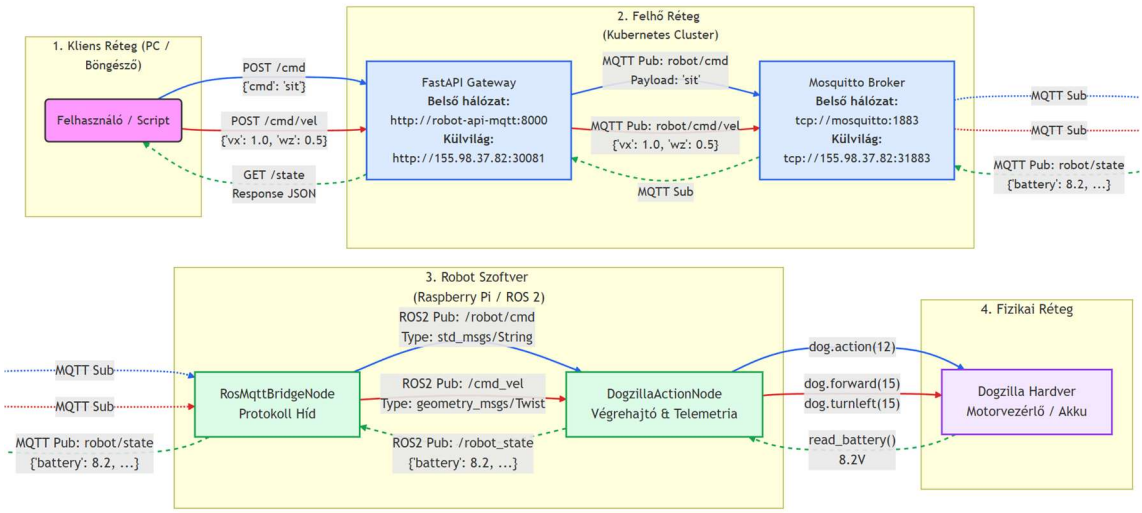
Ezen korlátok felismerése vezetett ahhoz a stratégiai döntéshez, hogy a rendszer a közvetlen ROS 2 bridging helyett egy alkalmazás szintű átjárót (Gateway) és egy dedikált üzenetközvetítő protokollt alkalmazzon. A választás az MQTT (Message Queuing Telemetry Transport) protokollra esett.

Az MQTT egy nyílt szabványú, rendkívül könnyűsúlyú hálózati protokoll, amelyet kifejezetten alacsony sávszélességű, nagy késleltetésű vagy megbízhatatlan hálózatokon üzemelő eszközök – például IoT szenzorok és mobil robotok – számára terveztek [41]. Működésének alapja az eseményvezérelt „közzététel-feliratkozás” (Publish-Subscribe) minta, amely alapvetően különbözik a hagyományos HTTP kérés-válasz modelljétől. Ez a mechanizmus lehetővé teszi a kommunikáló felek teljes logikai szétválasztását (decoupling): az adatot küldő félnek és az adatot fogadó félnek nem kell ismerniük egymás hálózati identitását, sőt, még egy időben sem kell feltétlenül elérhetőnek lenniük a sikeres adatcseréhez [42].

A kliensek (robot és felhő alkalmazás) nem közvetlenül egymással, hanem egy központi szerverrel (Broker) kommunikálnak. Ez feloldja a NAT problémákat, mivel minden kapcsolat kimenő irányú TCP kapcsolatként indul a bróker felé. A teljes kommunikáció egyetlen szabványos porton (1883) zajlik, ami egyszerűsíti a tűzfal és a Kubernetes Ingress/Service konfigurációt. A protokoll fejléc-overheadje minimális, ami ideális a mobil robotok korlátozott sávszélességű vezeték nélküli kapcsolataihoz.

5.2.2 A hibrid rendszer rétegei

A megtervezett architektúra a funkcionális szétválasztás elvét követve négy, egymástól jól elkülöníthető logikai rétegre tagolódik. Ez a hierarchikus felépítés lehetővé teszi a komponensek független fejlesztését, tesztelését és skálázását. A rétegek elrendezését és a köztük lévő adatkapcsolatokat az alábbi ábra (lásd 8. ábra) szemlélteti.



8. ábra A hibrid felhő-robotikai rendszer architektúrája és kommunikációs csatornái

5.2.2.1 Kliens Réteg (Client Layer)

A rendszer legfelső szintje, a felhasználói interakció pontja. Ez a réteg felelős a vezérlési szándék megfogalmazásáért és a visszacsatolás megjelenítéséért. Ez lehet egy Python alapú parancssori eszköz (CLI), egy webes dashboard vagy egy mobilalkalmazás.

A kliens nem kommunikál közvetlenül a robottal vagy az MQTT brókerrel. Kizárólag szabványos HTTP kéréseket küld a Felhő réteg felé (REST API), ami platformfüggetlenné teszi a vezérlést.

5.2.2.2 Felhő réteg (Kubernetes Cluster)

Ez a réteg biztosítja a globális elérhetőséget és a számítási erőforrásokat. A Kubernetes klaszterben futó komponensek konténerizált formában, egymástól izolálva, de hálózatilag összekötve működnek.

Itt két komponens fut. Az egyik a Mosquitto Broker. Ez az üzenetelosztó központ. A másik pedig a Robot API, ami a felhasználói parancsokat fogadó és validáló REST interfész. Ezekre jellemző a magas rendelkezésre állás, horizontális skálázhatóság, perzisztencia nélküli (stateless) működés.

5.2.2.3 Robot Szoftver Réteg (Robot Software Layer)

Ez a réteg a fizikai robot fedélzeti számítógépén (Raspberry Pi 5) fut. Itt történik a hálózati protokollok és a robotikai vezérlés illesztése. A réteg két fő komponensre bontható.

Az egyik komponens a protokoll híd (RosMqttBridgeNode). Ez a ROS 2 csomópont csatlakozik a felhőhöz. Feladata a JSON formátumú MQTT üzenetek átkonvertálása a robot belső, ROS 2 üzeneteire (pl. geometry_msgs/Twist), illetve a robot állapotának visszaküldése.

A másik komponens a végrehajtó logika (DogzillaActionNode). Ez a komponens tartalmazza a robot-specifikus viselkedési szabályokat. Értelmezi a magas szintű parancsokat (pl. „guggolj le”), és vezérlőjeleket generál a hardver számára.

5.2.2.4 Fizikai Réteg (Hardware Layer)

A rendszer legalsó szintje, a valós világra ható fizikai eszközök összessége. Itt találhatóak például az aktuátorok. A Dogzilla S2 12 darab busz-szervó motorja, amelyek a mozgást végzik. Itt kapnak szerepet a szenzorok is. Az akkumulátor-feszültségmérő, az IMU (gyorsulásmérő/giroszkóp), amelyek adatait a szoftverréteg olvassa ki.

A Szoftver Réteg és a Hardver Réteg között alacsony szintű soros (Serial/UART) vagy I2C kommunikáció zajlik a DOGZILLALib meghajtón keresztül.

5.3 Felhő infrastruktúra tervezése és validációja

A felhő oldali infrastruktúra definíciója deklaratív módon, Kubernetes manifest fájlok (YAML) segítségével történt. A következőkben részletesen elemzem és validálom a tervezett erőforrásokat.

5.3.1 Az üzenetközvetítő: Mosquitto Broker implementáció

A rendszer központi idegrendszerét az Eclipse Mosquitto MQTT bróker alkotja. A mosquitto.yaml fájl a következő implementációs részleteket tartalmazza.

5.3.1.1 Deployment stratégia

A bróker egy Kubernetes Deployment erőforrásként van definiálva eclipse-mosquitto:2 image használatával. A replikák száma szigorúan 1 (replicas: 1). Ez a döntés technikai szükségszerűség az adott konfigurációban: mivel az MQTT bróker memóriában tárolja a feliratkozási fákat és a visszatartott (retained) üzeneteket, több példány párhuzamos futtatása esetén az üzenetek nem jutnának el minden klienshez, hacsak nem alkalmazunk bróker-klaszterezést (bridging), ami azonban jelentősen növelné a komplexitást.

5.3.1.2 Konfiguráció menedzsment (ConfigMap)

A bróker beállításait a mosquitto-config nevű ConfigMap tárolja, amelyet fájlként csatolnak a konténerbe (/mosquitto/config/mosquitto.conf).

A listener 1883 beállítás a szabványos MQTT portot nyitja meg. Az allow_anonymous true engedélyezi a hitelesítés nélküli csatlakozást. Ez fejlesztési környezetben elfogadható a gyors tesztelés érdekében, de éles, ipari környezetben kritikus biztonsági kockázatot jelent. Bárki, aki ismeri a bróker IP-címét és portját, képes lehet vezérlő parancsokat küldeni a robotnak vagy lehallgatni a telemetriai adatokat.

5.3.1.3 Hálózati kitettség (Service):

A szolgáltatás elérését egy NodePort típusú Kubernetes Service biztosítja. A 31883-as NodePort kiválasztása lehetővé teszi, hogy a robot a Kubernetes klaszter bármely node-jának publikus IP-címét és ezt a portot használva csatlakozzon.

5.3.2 A Robot API mikroszolgáltatás

A felhasználói interakciót és a rendszer vezérlését a robot-api-mqtt nevű mikroszolgáltatás végzi. Ennek tervezése a modern, konténerizált alkalmazásfejlesztés elveit követi.

A szolgáltatás Python környezetben fut, a FastAPI keretrendszert használva. A választás indokolt a keretrendszer nagy teljesítménye és az aszinkron (async/await) kérések natív támogatása miatt, ami elengedhetetlen a hálózati I/O műveletek hatékony kezeléséhez. Az ASGI szerver (uvicorn) felelős a HTTP kérések fogadásáért és a Python kód futtatásáért.

A Dockerfile egy többlépcsős build folyamatot ír le. A python:3.12-slim image használata minimalizálja a konténer méretét és a támadási felületet (kevesebb telepített csomag miatt kevesebb a sebezhetőség). Az uvicorn indítási parancsa (CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]) biztosítja, hogy a szerver minden hálózati interfészen figyeljen.

A robot-api-mqtt.yaml fájl definiálja a futtatókörnyezetet. A szolgáltatás szintén NodePort típust használ, a 30081-es porton keresztül téve elérhetővé a REST API-t a külvilág számára. A belső 8000-es portot (ahol a Uvicorn figyel) képezi le a külső 30081-es portra.

Az előző fejezetekben említett port-összerendeléseket a 3. táblázat foglalja össze.

Szolgáltatás	Belső Port	NodePort (Külső)	Protokoll	Funkció
Mosquitto	1883	31883	TCP/MQTT	Robot kommunikáció
Robot API	8000	30081	TCP/HTTP	Felhasználói vezérlés

3. táblázat Összefoglaló táblázat a felhő portokról

5.4 A kommunikációs köztesréteg (Middleware) részletes elemzése

Az MQTT protokoll nem csupán egy szállító réteg, hanem a rendszer logikai működésének meghatározó része is. A tervezés során definiált topic struktúra és üzenetformátumok alkotják a rendszer API-ját.

5.4.1 MQTT Topic struktúra és adatmodellek

A rendszertervezés során kialakított topic hierarchia a funkcionális szétválasztás elvét követi. A következőkben bemutatom a tervezett struktúrát.

5.4.1.1 Parancs csatorna (Command Channel - Downlink):

robot/cmd: Ezen a csatornán érkeznek a diszkrét, magas szintű parancsok. A payload egyszerű szöveges string (pl. "sit", "stand_up"). Ez a csatorna felelős a robot állapotátmeneteinek vezérléséért.

robot/cmd/vel: A folyamatos mozgásvezérlés csatornája. A payload itt strukturált JSON objektum, amely a lineáris és szögsebességeket tartalmazza ({vx, vy, wz}). A JSON formátum választása biztosítja a bővíthetőséget és a könnyű olvashatóságot, szemben a bináris szerializációval.

5.4.1.2 Állapot csatorna (State Channel - Uplink):

robot/state: A robot ezen a csatornán küld visszajelzést a belső állapotáról, illetve a diagnosztikai üzenetekről. Ide érkeznek a "pong" válaszok is a hálózati mérések során.

5.4.2 QoS (Quality of Service) szintek

Az MQTT protokoll három QoS szintet definiál, amelyek a kézbesítési garanciát szabályozzák. A mqtt_bridge_node.py kódjában a mqtt_client.connect hívásnál nincs explicit QoS beállítás, ami az alapértelmezett QoS 0 (At most once) szintet jelenti.

A QoS 0 ("Fire and forget") a leggyorsabb és legkisebb hálózati terheléssel járó mód, de nem garantálja az üzenet megérkezését. Ez a folyamatos sebességvezérlésnél (robot/cmd/vel) elfogadható, hiszen, ha egy sebességparancs elveszik, a következő (jellemzően 10-100 ms múlva) pótolja. Azonban a diszkrét parancsoknál (pl. "sit" vagy "stop") ez kockázatos lehet, mivel a parancs elvesztése a robot nem kívánt állapotban maradását eredményezheti.

5.5 Robot-oldali szoftverarchitektúra

A robot fedélzeti szoftvere a ROS 2 (Robot Operating System 2) keretrendszerre épül. Az architektúra modularitását két fő node biztosítja: az MqttBridgeNode és a DogzillaActionNode. Ez a szétválasztás lehetővé teszi a kommunikációs logika és a hardvervezérlés független fejlesztését és tesztelését.

5.5.1 Az MQTT-ROS 2 híd

Ez a komponens a rendszer kapuőre, amely a külső MQTT világot köti össze a belső ROS 2 DDS hálózattal.

A node inicializálásakor elindul egy háttérszál a self.mqtt_client.loop_start() hívással. Ez a paho-mqtt könyvtár által kezelt szál felelős a hálózati forgalom lebonyolításáért, a keep-alive csomagok küldéséért és a callback függvények (on_message) hívásáért. A főszál eközben a rclpy.spin(node) segítségével a ROS 2 eseményhurkot futtatja.

A híd node intelligens konverziót végez:

JSON → Twist: A robot/cmd/vel topicra érkező JSON adatokat (vx, vy, wz) kiolvassa, és egy geometry_msgs/Twist objektum lineáris (x, y) és anguláris (z) komponenseibe tölti. Ha egy mező hiányzik a JSON-ból, alapértelmezett 0.0 értéket használ, ami robusztussá teszi a rendszert a hibás adatokkal szemben.

String → String: A robot/cmd üzeneteit közvetlenül továbbítja.

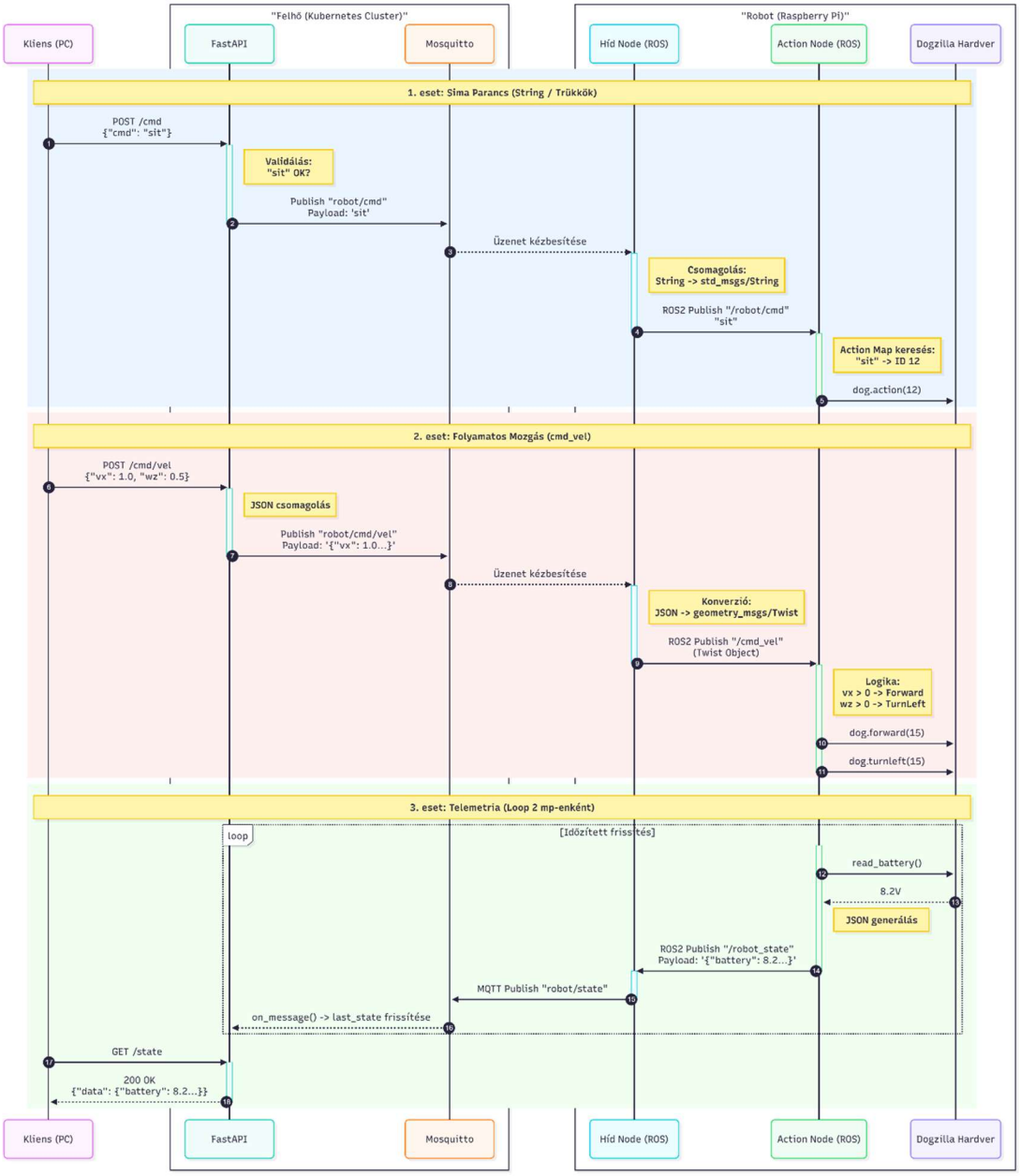
A kódban található egy speciális rövidzár logika. Ha a robot/cmd üzenet tartalmazza a "ping" szöveget, akkor megállítja a továbbítást a ROS felé. Ez a megoldás lehetővé teszi a késleltetés (RTT) mérését anélkül, hogy a ROS 2 késleltetése befolyásolná az eredményt. Így különválasztható a hálózati infrastruktúra és a robot belső szoftverének teljesítménye.

5.5.2 Hardver absztrakciós réteg (DogzillaActionNode)

A fizikai világra való hatást a DogzillaActionNode végzi (dogzilla_action_node.py). Ez a node valósítja meg a Hardver Absztrakciós Réteget (HAL).

A node a /robot/cmd ROS topikon hallgatózik. A beérkező szöveges parancsokat egy normalizálási lépés után (kisbetűsítés, strip) dolgozza fel. A tervezés alapja egy "Command Pattern" implementáció, ahol a parancsokat egy leképezési táblázat (action_map) segítségével fordítja le vezérlő kódokra.

Ez a szótár alapú megközelítés O(1) komplexitású keresést biztosít, szemben a hosszú if-else láncokkal (amelyek O(n) komplexitásúak lennének). Ez a robotika valós idejű környezetében fontos optimalizáció. A DOGZILLA könyvtár action(code) metódusa végzi a tényleges szervóvezérlést. A mozgási parancsok ("forward", "left", stb.) külön ágon, a move(x, y, r) metódussal hajtódnak végre, fix sebességgel (speed=15). Ez a fix sebesség egy egyszerűsítés; a jövőben érdemes lehet a sebességet is paraméterként átadni. A 9. ábra tökéletesen illusztrálja a protokoll-konverzió és a hardver absztrakció folyamatát, amiről az 5.5 fejezet is szól.



9. ábra Vezérlési és telemetriai üzenetváltások szekvencia-diagramja

5.6 Validált parancskészlet

A robot-api-mqtt szolgáltatás szigorú bemeneti validációt végez. A main.py definiálja a VALID_COMMANDS halmazt. Csak azok a parancsok kerülnek továbbításra az MQTT bróker felé, amelyek szerepelnek ebben a listában.

Mozgás: forward, backward, left, right, turn_left, turn_right, stop.

Pozíció: lie_down, stand_up, sit, squat, push_up.

Szociális/Trükk: handshake, wave_hand, pray, seek, pee.

Ez az "Allowlist" (fehérlista) megközelítés alapvető biztonsági funkció, amely megakadályozza a "Command Injection" típusú támadásokat és a robot érvénytelen állapotba vezérlését.

5.7 Konklúzió

Összességében a rendszertervezés sikeresen valósította meg a kitűzött célokat: létrehozott egy működőképes, moduláris hidat a felhő alapú Kubernetes infrastruktúra és a ROS 2 alapú robotika között, validálva a modern IoT technológiák alkalmazhatóságát ebben a domainben.

A fejezetben említett fájlok az alábbi nyílvános GitHub repozitóriumban érhetőek el: https://github.com/balazsfoldi/dogzilla-s2-basic-cloud-control.

6 Mérések és értékelés

6.1 A felhőalapú vezérlési infrastruktúra hálózati teljesítményének elemzése

6.1.1 A mérés célja és módszertana

A vizsgálat célja a robotvezérlési rendszer felhőoldali infrastruktúrájának teljesítményelemzése volt, különös tekintettel a hálózati késleltetésre (latency) és a kapcsolat stabilitására (jitter). A mérés egy hibrid kommunikációs hurkon keresztül történt, amely magában foglalta a kliens oldali parancsküldést HTTP protokollon keresztül a Kubernetes klaszterben futó FastAPI mikroszolgáltatás felé, majd onnan az üzenet továbbítását az MQTT brókeren keresztül vissza a klienshez. A teszt során 200 darab szekvenciális mérési csomag került kiküldésre, szimulálva a valós idejű vezérlés hálózati terhelését, a fizikai robot hardverének bevonása nélkül. Ez a módszer lehetővé tette a „tiszta” infrastrukturális késleltetés izolálását.

6.1.2 A késleltetés (Latency) értékelése

A mérési eredmények alapján a rendszer átlagos válaszideje (Round-Trip Time, RTT) 314,2 ms volt. Ez az érték magában foglalja a HTTP kérés felépítését, a Kubernetes Service és a belső hálózati routing idejét, a FastAPI alkalmazás feldolgozási idejét, valamint az MQTT üzenetszórás késleltetését.

A mért érték nagyságrendje (~300 ms) egy tipikus, földrajzilag távolabb elhelyezkedő felhőszolgáltatás (USA) és egy európai végpont közötti kommunikációra utal, ami egyezik a valósággal. Bár ez az érték meghaladja a "hard real-time" rendszerek (pl. ipari robotkarok) által megkövetelt 10-20 ms-os szintet, a jelenlegi alkalmazási területen – egy négylábú robot (Dogzilla S2) teleoperációs vezérlése – elfogadhatónak tekinthető. A ~300 ms-os késleltetés az emberi operátor számára érzékelhető, de a robot mozgásának jellegéből adódóan (séta, pozícióváltás) nem akadályozza a biztonságos irányítást.

6.1.3 Stabilitás és Jitter elemzése

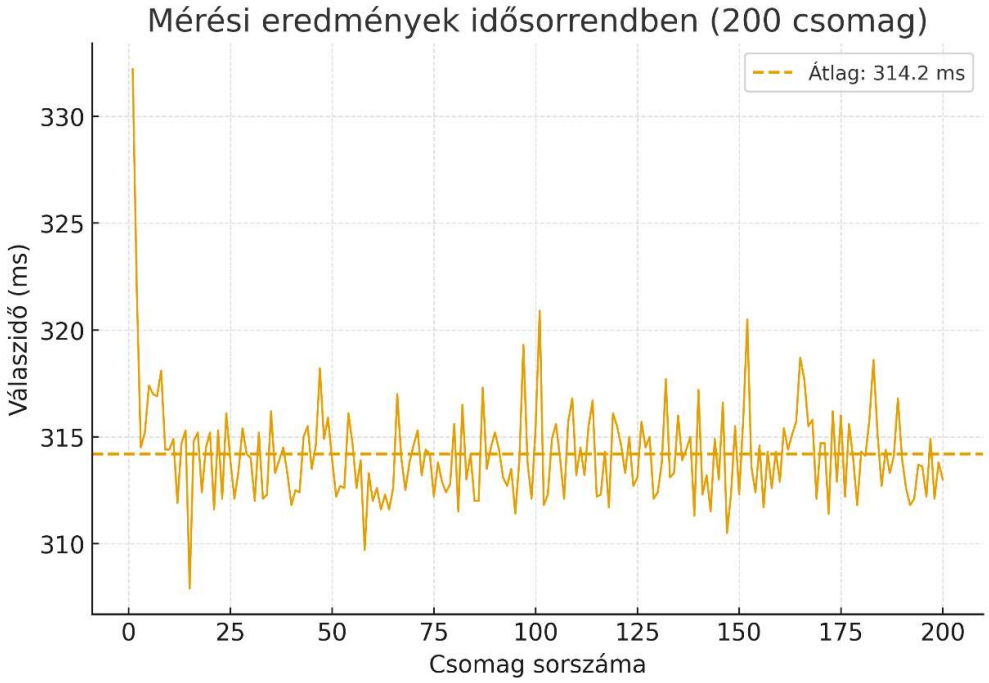
A hálózati kapcsolat minőségének legfontosabb mutatója jelen esetben nem az abszolút késleltetés, hanem annak ingadozása, azaz a jitter. A mérés során a válaszidők

szórása rendkívül alacsony, mindössze 2,4 ms volt. A leggyorsabb (307,9 ms) és a leglassabb (332,2 ms) válasz közötti különbség elhanyagolható.

Ez a determinisztikus viselkedés kulcsfontosságú a robotika szempontjából. Azt jelenti, hogy bár a parancsok késve érkeznek meg, ez a késés állandó és kiszámítható. A vezérlő algoritmusok és az operátor sokkal könnyebben tudnak alkalmazkodni egy stabilan 300 ms-os késleltetéshez, mint egy olyan kapcsolathoz, amely 50 ms és 500 ms között ugrál. A 0%-os csomagvesztés tovább erősíti a rendszer megbízhatóságát; a TCP alapú HTTP és MQTT protokollok sikeresen biztosították az adatcsomagok sértetlenséget.

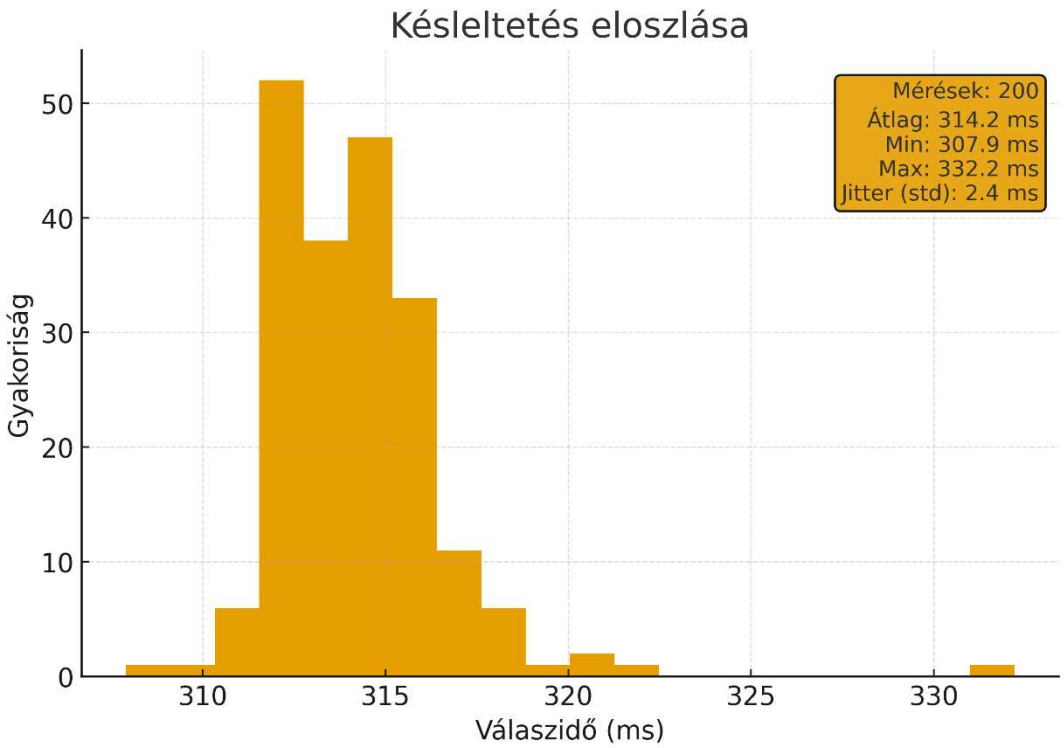
6.1.4 A diagramok értelmezése

A mérési adatokból készített vizualizációk alátámasztják a fenti statisztikai következtetéseket.



10. ábra infrastruktúra hálózati teljesítményének idősoros diagramja [ChatGPT]

Az adatok időbeli lefolyását ábrázoló vonaldiagram (10. ábra) egyenletes, közel lineáris eloszlást mutat az átlagérték (piros referenciavonal) körül. Nem figyelhetők meg szignifikáns kiugrások vagy periodikus lassulások, ami a Kubernetes klaszter és a hálózat terheléselosztásának megfelelő működésére utal. A rendszer teljesítménye a 200. üzenetnél is ugyanolyan stabil volt, mint az elsőnél, nem lépett fel puffer-telítődés vagy sávszélesség-korlátozás.



11. ábra infrastruktúra hálózati teljesítményét ábrázoló hisztogram [ChatGPT]

A válaszidők gyakorisági eloszlását mutató hisztogram (11. ábra) egy nagyon keskeny, haranggörbe-szerű alakzatot vesz fel. Az adatok döntő többsége egy szűk, 312 ms és 316 ms közötti intervallumba esik. Ez a csúcsos eloszlás vizuálisan is igazolja az alacsony jittert és a hálózat kiszámíthatóságát.

6.1.5 Következtetés

Összességében a mérés igazolta, hogy a kialakított hibrid felhő-architektúra stabil és megbízható kommunikációs csatornát biztosít. Bár a ~314 ms-os átlagos késleltetés bevezet egy fix holtidőt az irányításba, a rendszer alacsony ingadozása (jitter) és hibamentes működése alkalmassá teszi a robot távoli felügyeletére és magas szintű parancsok (pl. "menj előre", "guggolj le") kiadására. A késleltetés csökkentése érdekében a jövőben érdemes lehet a szervert földrajzilag közelebb költöztetni.

6.2 A teljes körű, fizikai robottal végzett vezérlési lánc teljesítményelemzése

6.2.1 Bevezetés és mérési elrendezés

A mérés második fázisában a teljes vezérlési lánc ("End-to-End") teljesítménye került vizsgálatra, immár a fizikai hardver, a Dogzilla S2 robot bevonásával. A mérés célja a valós felhasználói élmény kvantitatív meghatározása volt, ahol a parancskiadás és a fizikai visszajelzés közötti teljes időkülönbséget (Round-Trip Time, RTT) mértük. A kommunikációs útvonal a kliens PC-ről indult, áthaladt a felhőalapú FastAPI mikroszolgáltatáson és MQTT brókeren, majd Wi-Fi kapcsolaton keresztül elérte a robotot, amely a parancs feldolgozása után ugyanazon az útvonalon visszajelzést küldött.

6.2.2 A késleltetés (Latency) részletes elemzése

A 200 mintás méréssorozat eredményei alapján a rendszer átlagos teljes köridеje 471,5 ms volt. Ez az érték két fő komponensre bontható fel a korábbi, tisztán infrastrukturális mérések (~314 ms) ismeretében. Az egyik az infrastrukturális késleltetés (~314 ms), ami a vezetékes internet, a felhő szerverek és a belső routing ideje. A másik pedig a robot-kör késleltetése (~157,5 ms): Ez a többletidő a robot Wi-Fi interfészén történő adatátvitelből, a vezeték nélküli hálózat esetleges csomagütközéseiből, valamint a robot fedélzeti számítógépének (Raspberry Pi) Python kódjának futási idejéből tevődik össze.

A mért ~470 ms-os (majdnem fél másodperces) késleltetés teleoperációs környezetben egy látható, de kezelhető kategóriába esik. Bár a valós idejű, reflexszerű irányítást (pl. gyors akadálykerülés nagy sebességnél) megnehezítheti, de a robot tervezett felhasználási módjához (séta, előre programozott mozdulatok) megfelelő.

6.2.3 Stabilitás és Jitter

A mérés legkiemelkedőbb eredménye a rendszer stabilitása. A válaszidők szórása (jitter) mindössze 2,2 ms, ami gyakorlatilag megegyezik a tisztán kábeles/felhős mérés stabilitásával. Rendkívül fontos megállapítás, hogy a Wi-Fi kapcsolat és a robot hardvere nem vezetett be véletlenszerű ingadozást. A leggyorsabb (466,1 ms) és a leglassabb (477,5 ms) válasz között alig 11 ms a különbség. Ez a determinisztikus viselkedés lehetővé teszi prediktív vezérlési algoritmusok alkalmazását, mivel a késleltetés állandó, nem pedig változó zaj.

6.2.4 Összehasonlító elemzés (Infrastruktúra vs. Fizikai Robot)

Az alábbiakban összevetem a két mérési fázist:

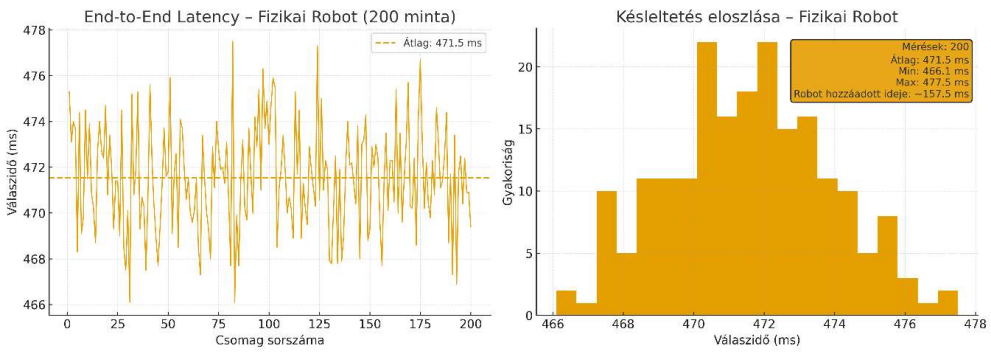
Mérési paraméter	Infrastruktúra (Robot nélkül)	End-to-End (Fizikai Robottal)	Különbség (Robot hatása)
Átlagos Latency	314,2 ms	471,5 ms	+157,3 ms
Minimum	307,9 ms	466,1 ms	+158,2 ms
Jitter (Stdev)	2,4 ms	2,2 ms	-0,2 ms

4. táblázat Infrastuktúra és End-to-End mérések összehasonlítása

A 1. táblázat alapján látszik, hogy a robot beiktatása egy tisztán additív késleltetést adott a rendszerhez (~157 ms), de nem rontotta a kapcsolat minőségét (jitter). Ez azt jelzi, hogy a helyi Wi-Fi hálózat és a robot szoftvere is optimálisan működik, nincs csomagvesztés vagy pufferelési probléma.

6.2.5 Vizualizáció és Diagramok

A mellékelt ábrák vizuálisan is megerősítik az eredményeket.



12. ábra End-to-End hálózati teljesítményt ábrázoló diagrammok

Balra egy vonaldiagram, jobbra pedig egy hisztogram látható [ChatGPT]

A 12. ábra vonaldiagramján jól látszik, hogy az adatok egy rendkívül lapos, vízszintes sávban helyezkednek el 470 ms körül. Nincsenek tüskék, ami a vezeték nélküli átvitel megbízhatóságát dicséri. A hisztogramon pedig látható, hogy az eloszlás szimmetrikus és csúcsos, a mérések 95%-a a 468-474 ms-os tartományba esik.

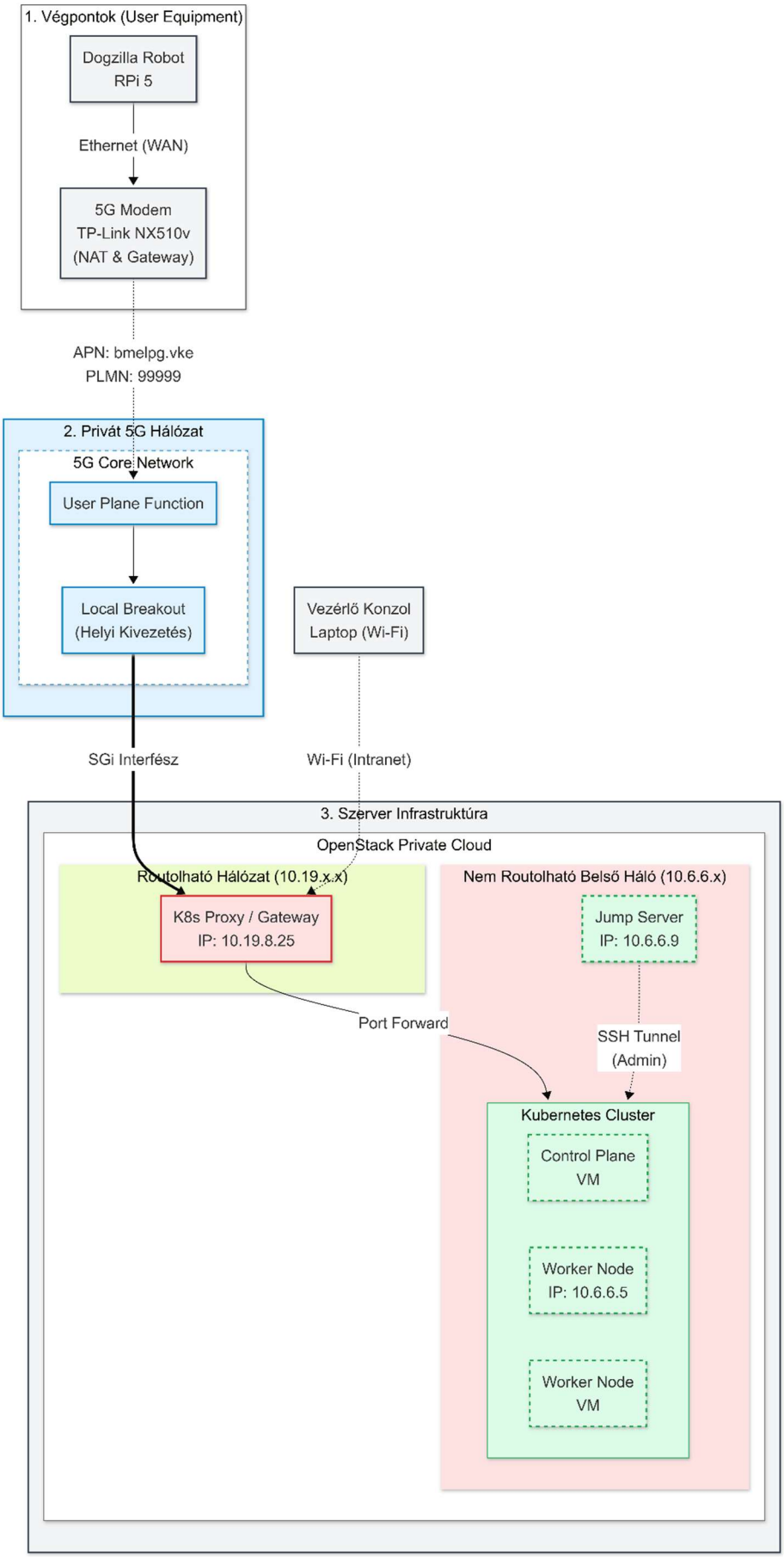
6.2.6 Konklúzió

A fejlesztett IoT robotvezérlési rendszer sikeresen vizsgázott. Bár a ~470 ms-os késleltetés jelentős, annak konstans jellege és a 0%-os csomagvesztés garantálja a megbízható működést. A rendszer szűk keresztmetszete jelenleg nem a robot vagy a Wi-Fi, hanem a felhő szerver földrajzi távolsága vagy hálózati topológiája (a ~314 ms-os alap késleltetés miatt). A felhasználói élmény javítása érdekében elsősorban a szerveroldali optimalizáció, közelebbi adatközpont javasolt.

6.3 A rendszer teljesítményelemzése privát 5G környezetben

6.3.1 A kísérleti környezet hálózati topológiája és specifikációi

A rendszer validálása és a mérések elvégzése egy dedikált, laboratóriumi környezetben kialakított Privát 5G Campus Hálózaton (Private 5G Network) történt. Az ötödik generációs (5G) mobil hálózatok napjaink legfejlettebb nyilvános rádiós kapcsolatot biztosító technológiája, amely sok iparág (pl. egészségügy, okos város, jármű, robotika, stb.) digitalizációját képes támogatni. Az 5G hálózatok architektúrája eltér a nyilvános felhőszolgáltatók (Public Cloud) elérési modelljétől, kezeli a rádiós kapcsolat teremtés és felhasználói azonosítás folyamatait, ezáltal biztosítja az ipari környezetek szigorú biztonsági, szegmentálási és késleltetési követelményeit. Középtávon az 5G hálózatok várhatóan fontos elemei lesznek az ipari környezet digitális infrastruktúrájának, ezért fontos volt ebben a környezetben is tesztelnem a megvalósításomat. Az általam használt hálózat felépítése három fő rétegre bontható: a rádiós hozzáférésre, maghálózatra és a szerveroldali virtualizációs infrastruktúrára [43]. Ennek a rendszernek a felépítését mutatja be a 13. ábra.



13. ábra Kísérleti környezet hálózati topológiája

6.3.1.1 Fizikai architektúra és Rádiós Hozzáférés (RAN)

A rendszer fizikai rétegének legfelső szintjén a kapcsolatot az 5G rádiós interfésze (5G New Radio – 5G NR) interfész biztosítja. A Dogzilla S2 robot kommunikációs modulja (UE - User Equipment) nem a nyilvános mobilhálózatra, hanem egy saját, kísérleti bázisállomására csatlakozik.

A robot hálózati csatolófelülete egy TP-Link 5G modem, amelyhez a robot fedélzeti számítógépe Ethernet kábelen keresztül csatlakozik (a modem WAN portját használva). A csatlakozás paraméterei a szigorúan izolált működést garantálják.

A hálózati infrastruktúra adatátviteli képességeit a korábbi TDK kutatás [44] mérési eredményei támasztják alá. A dedikált 5G környezetben végzett mérések alapján a kapcsolat átlagos késleltetése (RTT) mindössze 10,021 ms volt, rendkívül alacsony, 0,0735 ms-os jitter mellett. A rendszer adatátviteli sebessége letöltési (DownLink) irányban elérte a 157,5 Mbit/s, míg feltöltési (UpLink) irányban a 8,165 Mbit/s értéket. Ezek a paraméterek igazolják, hogy a hálózat képes biztosítani a stabil, valós idejű vezérléshez szükséges feltételeket.

6.3.1.2 Az 5G maghálózat

Az 5G hálózat vezérlését, beleértve a rádiós hálózatból érkező adatok irányítását az 5G maghálózat (5G Core – 5GC) biztosítja. Ezen belül az 5G hálózat részéről az adatforgalmat az (User Plane Function - UPF) irányítja. Az általam használt 5G környezetben a felhasználói síkban a forgalom lokalizációt (Local Breakout – lásd később) egy Kernel szintű IPv4 továbbítást (IPv4 Forwarding) alkalmazó szoftvermodulként valósítja meg.

Tehát az 5G rádiós interfészéről érkező adatcsomagok a rádiós antennához közeli szerver továbbítja az 5G hálózati környezetbe, onnan pedig a külső adathálózattól (jellemzően az Internettől) elválasztott helyi adathálózatba. Ez a helyi adathálózatot peremszámítási hálózatnak is szokták nevezni (edge network).

Az architektúra egyik legfontosabb teljesítmény-optimalizációs eleme a korábban említett Local Breakout alkalmazása. A hagyományos mobilhálózati topológiákkal ellentétben – ahol az adatforgalom távoli központi maghálózatokon utazik keresztül – a kísérleti rendszerben az adatforgalom a rádiós hálózat elhagyása után azonnal, helyben kerül átirányításra a laboratórium belső IP-hálózatába (az én esetemben a 10.6.6.x/32 IP címtartomány). Ez a megoldás minimalizálja a hálózati ugrások (hops) számát és az átviteli késleltetést, valamint biztosítja, hogy az ipari adatok ne hagyják el a privát infrastruktúrát. A hálózatban a Local Breakoutot biztosító adatkezelési azonosító (APN) használatával érem el ezt a működést.

6.3.1.3 Szerveroldali infrastruktúra és virtualizáció

A rendszer számítási kapacitását (ahol a Kubernetes klaszter fut) egy többszintű, hierarchikus virtualizációs környezet biztosítja. Az infrastruktúra alapját nagy teljesítményű szerverek adják, amelyeken az egyes típusú hipervizor (hypervisor) fut. Ez a réteg felelős a hardveres erőforrások (CPU, Memória, I/O) absztrakciójáért. Ezek a fizikai szerverek (bear metal) vannak bekötve a korábban említett edge környezetbe, tehát a fizikai szerverek is a 10.6.6.0/32 alhálózatba tartoznak. Ez az alhálózat biztonsági okokból a külvilág (így a robot) számára közvetlenül nem elérhető ("láthatatlan").

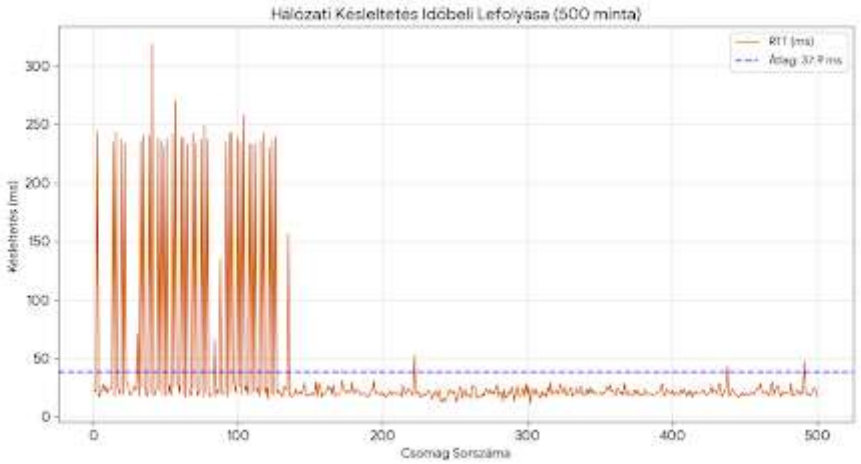
A hipervizor felett egy OpenStack alapú privát felhő rendszer üzemel [45]. Ez a réteg menedzseli a virtuális gépek életciklusát és a szoftveresen definiált hálózatot (SDN). A méréshez használt Kubernetes klaszter a magas rendelkezésre állás (High Availability) környezetet modellezve három OpenStack-ben indított ubuntu VM-en lett szétosztva. A Kubernetes klaszteren vezérlési funkciókat megvalósító csomópontja (az én esetemben OpenStack VM) lesz felelős a bejövő forgalom fogadásáért és a belső szolgáltatások (NodePort) felé történő továbbításáért, azaz ezen a csomóponton futó Kubernetes Proxy rendszerelemen (kube proxy) keresztül érik el a csomagok a virtualizált szerevereket. Tehát a robot (felhasználói eszköz – User Equipment) számára a belépési pont a fenti vezérlési funkciót biztosító OpenStack VM nyilvános IP címe (Floating IP Address), a 10.19.8.25.

6.3.2 A hálózati késleltetés és stabilitás elemzése

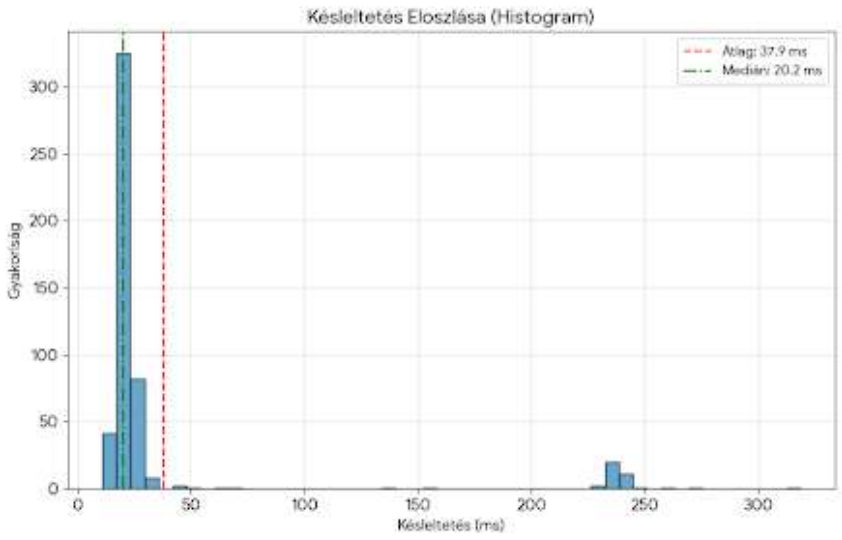
A rendszer teljesítőképességének legpontosabb meghatározása érdekében egy közvetlen, „End-to-End” mérést végeztem. A mérés során a vezérlő kliens (PC) közvetlenül a Kubernetes Proxy-n keresztül csatlakozott a Mosquitto brókerhez, a robot pedig a privát 5G hálózaton keresztül válaszolt.

6.3.2.1 Mérési eredmények kiértékelése

A vizsgálat során 500 darab szekvenciális adatcsomagot (Ping) küldtem ki, és mértem a teljes köridőt (Round-Trip Time - RTT), amely magában foglalta az uplink és a downlink irányú késleltetést, valamint a robot belső feldolgozási idejét.



14. ábra A hálózati késleltetés időbeli lefolyása [ChatGPT]



15. ábra Késleltetési értékek gyakorisági eloszlása [ChatGPT]

6.3.2.2 Az eredmények értelmezése

A mérési eredmények drasztikus javulást mutatnak a korábbi, publikus interneten keresztül mért értékekhez (~471 ms) képest. A 20 ms-os érték kiválónak tekinthető a távvezérlési feladatokhoz, lehetővé téve a robot azonnali reakcióját a felhasználói parancsokra.

Az adatok eloszlása azonban kettősséget mutat. A mérések több mint 75%-a 24 ms alatt maradt, ami bőven megfelel a valós idejű („Hard Real-Time" közeli) vezérlési követelményeknek. Ez az érték magában foglalja a teljes láncot.

Ezzel ellentéteben a hisztogramon (lásd 15. ábra) és az idősoros ábrán (lásd 14. ábra) is látható, hogy a mérések kb. 7-8%-ánál (39 eset) a késleltetés kiugrott 100 ms fölé (max. 318 ms).

6.3.2.3 Konklúzió

A mért 37,9 ms-os átlagos válaszidő és a csomagvesztés hiánya azt bizonyítja, hogy a kiépített rendszer alkalmas a robot dinamikus távvezérlésére. Bár jelennek meg késleltetési tüskék, az alacsony alapérték (Base Latency ~17-20 ms) kiváló felhasználói élményt tesz lehetővé. A rendszer a parancsok döntő többségét azonnal végrehajtja, a ritka lassulások pedig a nem-kritikus (séta, pozícióváltás) feladatoknál tolerálhatók.

7 A munka értékelése és jövőbeli fejlesztési lehetőségek

A szakdolgozat kutatási és fejlesztési tevékenysége során egy olyan hibrid, felhőalapú robotvezérlési architektúra került megtervezésre, implementálásra és validálásra, amely kísérletet tett a robotikai eszközök fedélzeti erőforrásainak korlátainak áthidalására a modern felhőtechnológiák skálázhatóságával. A megvalósított rendszer, amely a Kubernetes konténer-orkesztrációs platform, a ROS 2 (Robot Operating System 2) keretrendszer és az MQTT (Message Queuing Telemetry Transport) aszinkron kommunikációs protokoll szinergiájára épül, sikeresen demonstrálta a „Cloud Robotics" koncepció megvalósíthatóságát, ugyanakkor rávilágított a technológia jelenlegi korlátaira és az ipari bevezetés előtt álló kihívásokra.

Jelen fejezet célja a kutatás során elért eredmények összefoglalása, a mérési adatokból levonható mélyebb következtetések bemutatása, valamint azon technológiai fejlesztési irányok részletes kijelölése, amelyek a létrehozott prototípust egy robusztus, ipari környezetben is alkalmazható rendszerré emelhetik. A fejezet különös mélységgel tárgyalja a hálózati infrastruktúra evolúcióját (Privát 5G és Edge Computing), a számítási feladatok megosztásának fejlett módszereit (Cloud SLAM), valamint a kiberbiztonsági kockázatok kezelésének stratégiáit.

7.1 Az elért eredmények átfogó értékelése

A dolgozat központi hipotézise azon a feltevésen alapult, hogy a modern, felhő-natív technológiák (Kubernetes) és a könnyűsúlyú IoT protokollok (MQTT) képesek egy olyan absztrakciós réteget képezni, amely leválasztja a robotikai vezérlést a fizikai hardverről, lehetővé téve a globális elérhetőséget és a központosított menedzsmentet. A megvalósított rendszer központi eleme, a Yahboom Dogzilla S2 robotkutya, sikeresen integrálódott ebbe a hibrid környezetbe, validálva a tervezett hardver-absztrakciós réteg (HAL) és a kétirányú protokoll-híd (MQTT-ROS2 Bridge) működőképességét.

A fejlesztés során elért legfontosabb eredmények három fő kategóriába sorolhatók:

Architektúrális integritás és skálázhatóság: A Kubernetes klaszterben futó Mosquitto bróker és a FastAPI alapú vezérlő interfész stabilan üzemelt a tesztek során. A konténerizáció (Docker) nem csupán a telepítést egyszerűsítette le, hanem biztosította a szoftverkomponensek izolációját is. A rendszer architektúrája bizonyította, hogy a mikroszolgáltatás-alapú megközelítés életképes alternatívája a monolitikus robotvezérlő

szoftvereknek, lehetővé téve a komponensek (pl. a vizuális feldolgozás vagy az útvonaltervezés) független skálázását a felhőben.

Protokoll-konverziós hatékonyság: A ROS 2 DDS (Data Distribution Service) és az MQTT közötti átjáró hatékonyan kezelte a JSON alapú parancsok és a bináris ROS üzenetek közötti fordítást. A mérések igazolták, hogy ez a konverziós réteg minimális számítási többletterhelést jelentett a Raspberry Pi 5 processzorán, így a fedélzeti erőforrások döntő többsége megmaradhatott a kritikus, valós idejű feladatok (pl. inverz kinematika, szenzoradatok olvasása) számára.

Hibrid Vezérlési Modell: A rendszer sikeresen valósított meg egy hierarchikus vezérlési modellt. A "high-level" döntések a felhőből érkeztek, míg a "low-level" végrehajtás (szervomotorok koordinációja, egyensúlyozás) helyben, a robot fedélzetén történt. Ez a megosztás biztosította, hogy a hálózati késleltetés ingadozása ne okozzon azonnali stabilitásvesztést a robot mozgásában.

7.1.1 A hibrid architektúra teljesítményének összehasonlító elemzése

A 6. fejezetben bemutatott mérések alapján részletes képet kaphatunk a különböző hálózati topológiák teljesítményjellemzőiről. Az adatok mélyreható elemzése azonban túlmutat az egyszerű átlagértékek összehasonlításán; rávilágít a felhőalapú robotika legfontosabb mérnöki kompromisszumára: a determinisztikus működés és a hálózati késleltetés (latency) közötti feszültségre.

A mérések egyik legfontosabb, nem intuitív eredménye a rendszer rendkívüli stabilitása volt a nyilvános felhő és a Wi-Fi használata ellenére is. A ~471 ms-os teljes köridő (End-to-End) abszolút értékben magasnak tekinthető, hiszen közel fél másodperc telik el a parancs kiadása és a visszajelzés között. Egy 1 m/s sebességgel haladó robot esetén ez azt jelenti, hogy a robot fél métert tesz meg vakon, mielőtt a parancs megérkezne.

Ugyanakkor a mindössze 2,2 ms-os jitter (ingadozás) azt jelenti, hogy a késleltetés kiszámítható. A modern vezérléselmélet szempontjából a konstans, nagy holtidő sokkal könnyebben kezelhető, mint a változó, sztochasztikus késleltetés. Ez lehetővé teszi olyan modellezési technikák alkalmazását, ahol a robot mozgását a kliens oldalon előre szimuláljuk, és a késleltetett valós adatokat csak a modell korrekciójára használjuk. Ez a megállapítás validálja az MQTT protokoll TCP alapú megbízhatóságát ebben a kontextusban.

A mérési adatok dekompozíciója (314 ms infrastruktúra vs. 157 ms robot-kör) rámutatott, hogy a teljes késleltetés kétharmadáért nem a robot hardvere vagy a helyi vezeték nélküli hálózat, hanem a felhőszolgáltató és a nagy kiterjedésű hálózat (WAN) routingja felelős. Ez alapvetően megváltoztatja az optimalizációs stratégiát: a robotkód további optimalizálása (pl. C++ újraírás Python helyett) csak marginális nyereséget hozna. A valódi áttörést kizárólag a topológiai változtatás – azaz a szerverek fizikai közelebb hozása a robothoz – eredményezheti. Ez a felismerés vezeti át a gondolatmenetet a 7.2 fejezet témájára, az Edge Computing és az 5G szükségességére.

7.2 Hálózati infrastruktúra és az 5G szerepe

A szakdolgozat kutatásának egyik legfontosabb konklúziója, hogy a felhőrobotikai rendszerek teljesítményét a jövőben nem a processzorok órajele, hanem a hálózati infrastruktúra minősége és topológiája fogja meghatározni. A mért adatok alapján a rendszer szűk keresztmetszete a "Last Mile" (utolsó kilométer) és a felhő elérése. Ezen korlátok áttörésére a Privát 5G hálózatok (Private 5G Campus Networks) és a Multi-Access Edge Computing (MEC) technológiák mélyebb integrációja kínál megoldást.

Az 5G technológia (5G NR - New Radio) nem csupán a sávszélesség növelését jelenti, hanem egy teljesen új hálózati architektúrát vezet be, amelyet kifejezetten az ipari igények (Industry 4.0) kiszolgálására terveztek.

7.2.1 Az 5G technológia integrációjának tapasztalatai és a Local Breakout

A 6.3-as fejezetben bemutatott mérések során a rendszer egy dedikált, kísérleti 5G hálózaton üzemelt. A mért 10-30 ms-os késleltetés nagyságrendi ugrást jelent a Wi-Fi alapú ~470 ms-hoz képest. Ennek a drasztikus javulásnak a technikai hátterében az 5G architektúra egyik kulcseleme, a Local Breakout (LBO) áll, amelynek működését és jelentőségét érdemes részletesebben elemezni.

A hagyományos mobilhálózatokban (4G/LTE) és a publikus 5G hálózatok többségében az adatforgalom a bázisállomástól a szolgáltató központi maghálózatán (Core Network) keresztül utazik, gyakran több száz kilométert megtéve az országos gerinchálózaton, mire kilép az internetre (PGW/UPF), majd onnan visszautazik a felhasználó telephelyén lévő szerverhez. Ez felesleges terhelést és jelentős késleltetést ad a rendszerhez.

Ezzel szemben a Privát 5G környezetben alkalmazott Local Breakout technológia lehetővé teszi az adatforgalom optimalizálását. A rendszerben a User Plane Function (UPF) – amely az adatcsomagok továbbításáért felelős – decentralizált módon, fizikailag a hálózat peremén (Edge), közvetlenül a bázisállomások közelében vagy a vállalati telephelyen (On-Premise) kerül elhelyezésre. Amikor a Dogzilla robot (mint User Equipment - UE) adatot küld, az 5G rendszer a konfigurált szabályok alapján felismeri, hogy az adat a helyi vezérlőszervernek szól. Ahelyett, hogy a csomagot a központi maghálózatra továbbítaná, közvetlenül a helyi Ethernet hálózatra (LAN) irányítja.

A szakirodalom és a méréseim is igazolják, hogy a Local Breakout alkalmazásával a hálózati késleltetés minimalizálható, mivel megszűnik a Core Network és az Internet routing bizonytalansága.

7.2.2 Edge Computing: A hierarchikus feldolgozás modellje

A tisztán felhőalapú (Public Cloud) modell, amelyet a szakdolgozat jelenlegi verziója is alkalmaz, hosszú távon nem fenntartható az ipari robotikában a sávszélesség-igényes szenzorok (Lidar, mélységkamerák) terjedése miatt. A megoldás a Multi-Access Edge Computing (MEC) bevezetése, amely a számítási kapacitást a fizikai keletkezés helyéhez (Edge) hozza közel.

Az architektúra egy háromszintű, hierarchikus feldolgozási modellt követne:

Robot-oldali (On-Device) réteg: A Raspberry Pi 5 továbbra is végzi a hardver-közeli, szigorúan valós idejű feladatokat (motorvezérlés, IMU alapú stabilizálás, vészmegállás). Ez a réteg felelős az azonnali reflexekért.

Edge (Perem) réteg: A helyi 5G bázisállomás mellett vagy a gyárcsarnokban elhelyezett szerverek (MEC node-ok) futtatják a késleltetésre érzékeny, de nagy számítási igényű feladatokat. Ideális esetben a Mosquitto bróker, a lokális útvonaltervezés és a SLAM algoritmusok (Simultaneous Localization and Mapping) ide költöznének át a Public Cloud-ból. Ez a réteg biztosítja a 10-20 ms-os válaszidőt, és tehermentesíti a robotot a nehéz számítások alól. A Kubernetes könnyűsúlyú disztribúciói, mint a K3s vagy a MicroK8s, lehetővé teszik a konténer-orkesztráció kiterjesztését erre a rétegre is [46].

Cloud (Felhő) réteg: A távoli nyilvános felhő (AWS, Azure) szerepe átalakul, a valós idejű vezérlés helyett a hűvös adatok (cold data) tárolására, a gépi tanulási modellek tanítására (Global Model Training) és a teljes flotta globális optimalizálására (Fleet Management) fókuszál.

Ez a topológia lehetővé teszi a számítások legmegfelelőbb elosztását.

7.3 Autonómia növelése: Felhőalapú SLAM

A jelenlegi implementációban a Dogzilla S2 robot alapvetően teleoperációs módban működik: a mozgási döntéseket az emberi operátor hozza meg. Az ipari alkalmazhatóság kulcsa azonban az autonómia, amelynek alapfeltétele a környezet pontos feltérképezése és a robot saját pozíciójának meghatározása (SLAM).

A Raspberry Pi 5, bár jelentős előrelépés a korábbi modellekhez képest, korlátozott kapacitással rendelkezik komplex, 3D-s SLAM algoritmusok (pl. RTAB-Map, Cartographer, LIO-SAM) futtatására, különösen, ha párhuzamosan a videótömörítés és a ROS 2 kommunikáció is zajlik. A Cloud SLAM koncepciója megoldást kínál erre a problémára a számításigényes feladatok kiszervezésével.

7.4 Közvetlen ROS 2 felhő-integráció és alagút-technológiák

A szakdolgozatban megvalósított MQTT alapú protokoll-híd egy robusztus, de „alkalmazás-szintű” (Application Layer) megoldás. Ez a megközelítés izolálja a robotot a hálózattól, azonban minden egyes új adatfolyamhoz (pl. egy új szenzor) explicit módon fejleszteni kell a továbbító logikát, ami növeli a karbantartási igényt. A „Cloud Robotics” kutatások legújabb iránya a rendszer transzparenciájának növelésére törekszik, azaz a ROS 2 natív kommunikációjának (DDS) közvetlen kiterjesztésére a felhőbe. Ez lehetővé teszi a számítási feladatok kiszervezését (Computation Offloading), ahol a ROS node-ok (pl. SLAM, útvonaltervezés) a felhőben futnak, miközben úgy kommunikálnak a robottal, mintha egy lokális hálózaton lennének.

Ennek megvalósítására két fő technológiai irányvonal létezik, a hálózati szintű alagutazás (VPN) és a köztesréteg szintű áthidalás (Zenoh).

7.4.1 Hálózati szintű alagút (Network Layer Tunneling)

Ebben a modellben egy virtuális magánhálózat (VPN) jön létre a robot (Raspberry Pi) és a felhőben futó Kubernetes Podok között, létrehozva egy virtuális LAN-t (Overlay Network). A robotikai alkalmazásokra optimalizált VPN megoldások, mint például a Husarnet [47] vagy a WireGuard [48], képesek kezelni a NAT-áttörést (NAT Traversal) és a Peer-to-Peer kapcsolatok kiépítését.

A Husarnet kifejezetten robotikai rendszerekhez tervezett, P2P alapú VPN, amely minimalizálja a központi szerverek okozta késleltetést. Mérések szerint a Husarnet használata LAN környezetben mindössze kb. 1 ms többlet késleltetést ad a kapcsolathoz, és 5-10%-os adat-overheadet jelent, ami elfogadható kompromisszum a biztonságért és az egyszerű konfigurálhatóságért cserébe [49]. Ugyanakkor nagy sávszélességű adatfolyamok (pl. tömörítetlen videó) esetén a VPN titkosítása CPU-limitációba ütközhet a Raspberry Pi-n.

A VPN alapú megoldások fő kihívása, hogy a ROS 2 alapértelmezett DDS (Data Distribution Service) felderítési mechanizmusa multicast csomagokat használ, amelyeket a legtöbb VPN alapértelmezésben nem továbbít, így további konfigurációt (pl. Discovery Server) igényelnek.

7.4.2 Middleware szintű áthidalás és Offloading (Zenoh & FogROS2)

A jelenlegi kutatások élvonala a hálózati réteg helyett magát a kommunikációs köztesréteget reformálja meg. Az Eclipse Zenoh [50] protokoll egy olyan „Zero Overhead” megoldást kínál, amely kifejezetten a WAN hálózatokon keresztüli ROS 2 kommunikációra lett optimalizálva.

Egy összehasonlító mérés [51] szerint a Zenoh protokollal vezérelt robotok mutatták a legkisebb pályakövetési hibát (drift error) instabil hálózatokon, felülmúlva mind az MQTT, mind a natív DDS teljesítményét. A Zenoh képes a sávszélesség-igényes DDS felderítési forgalmat (discovery traffic) akár 99,97%-kal csökkenteni a WAN kapcsolaton, mivel csak a változásokat továbbítja a hálózaton [52].

A közvetlen integráció lehetővé teszi a FogROS2 keretrendszer alkalmazását, amely transzparens módon képes ROS 2 node-okat (pl. navigáció, SLAM) a felhőbe mozgatni. A FogROS2 mérései szerint a vizuális SLAM (Simultaneous Localization and Mapping) felhőbe szervezése akár 50%-kal csökkentheti a feldolgozási késleltetést a GPU-gyorsításnak köszönhetően, és jelentősen tehermentesíti a robot fedélzeti számítógépét [53].

Összefoglalva: a jövőbeli fejlesztés során a manuális MQTT híd helyett a Zenoh-plugin-ros2dds alkalmazása javasolt, amely biztosítja a nagy teljesítményű, transzparens adatátvitelt, megnyitva az utat a számításigényes feladatok (Cloud SLAM) dinamikus kiszervezése előtt.

7.5 Biztonságtechnikai fejlesztések

A szakdolgozat keretében megvalósított prototípus egyik tudatosan vállalt korlátja a biztonsági funkciók egyszerűsítése volt a fejlesztési sebesség maximalizálása érdekében. Amint azt az 5.3.1 fejezet is említi, a Mosquitto bróker jelenlegi konfigurációja „allow_anonymous true” és a titkosítatlan TCP kapcsolat (port 1883) éles, ipari környezetben elfogadhatatlan kockázatot jelent.

A robotika világában a biztonság (Safety) és a kiberbiztonság (Security) elválaszthatatlan. Egy kibertámadás – például egy "Man-in-the-Middle" (MitM) támadó, aki hamis parancsokat injektál a hálózatba – fizikai kárt okozhat a berendezésekben vagy veszélyeztetheti az emberi dolgozókat. Ezért a jövőbeli fejlesztés elengedhetetlen lépése a Zero Trust biztonsági modell bevezetése.

7.5.1 Kölcsönös hitelesítés (mTLS) és teljesítményhatásai

Az MQTT protokoll biztonságossá tételének egy szabványa a TLS (Transport Layer Security) felett megvalósított kölcsönös hitelesítés (Mutual TLS - mTLS). A hagyományos, szerveroldali TLS (mint a HTTPS a webböngészőben) csak a szerver (bróker) azonosságát igazolja a kliens felé. Ezzel szemben az mTLS során mindkét félnek – a robotnak és a felhőnek is – hitelesítenie kell magát kriptográfiai tanúsítványokkal [54].

8 Összefoglalás

A szakdolgozat részletesen bemutatta egy modern, ipari igényekre szabott robotvezérlési rendszer tervezését, megvalósítását és validálását. A munka során sikeresen ötvöztem a robotika (ROS 2), a felhőtechnológia (Kubernetes, Docker) és az IoT kommunikáció (MQTT) világát egy koherens, működőképes egésszé.

Létrejött egy moduláris, skálázható architektúra, amely sikeresen választja el a robot hardveres vezérlését a magas szintű logikától, bizonyítva a mikroszolgáltatás-alapú megközelítés életképességét a robotikában.

A mérések igazolták, hogy bár a rendszer stabilan működik nyilvános interneten keresztül is (alacsony jitter mellett), a valódi, reflex-szerű irányításhoz és a determinisztikus működéshez elengedhetetlen a minőségi hálózati infrastruktúra. A Privát 5G és a Local Breakout technológia integrációja bizonyította, hogy a vezeték nélküli technológia képes versenyre kelni a vezetékes megoldásokkal a késleltetés terén (~10-30 ms), megnyitva az utat a valós idejű felhőrobotika előtt.

Az Edge Computing alapú hierarchikus feldolgozás, a sávszélesség-optimalizált Cloud SLAM és az mTLS alapú "Zero Trust" biztonság nem csupán opciók, hanem előfeltételek a rendszer ipari termékké válásához.

A dolgozatban lefektetett alapok és a feltárt fejlesztési lehetőségek szilárd kiindulópontot biztosítanak a jövőbeli K+F projektek számára. A "Cloud Robotics" víziója – ahol a robotok a felhő végtelen erőforrásait használják "agyként", míg a hálózat biztosítja az "idegrendszert".

9 Függelék

9.1 Függelék Nyilatkozat generatív mesterséges intelligencia alkalmazásáról

☐ **Nem használtam** semmilyen generatív MI segédeszközt.

✓ **<u>Használtam</u>** generatív MI segédeszközt. Az MI-vel generált tartalmakat ellenőriztem, a generált kimenetek valóságtartalmáról meggyőződtem, az alábbi táblázatban megfelelően jelöltem minden használatot.

Felhasználási módok	**Generatív MI eszköz(ök) neve**	**Érintett részek** (fejezet, oldalszám, hivatkozás)	**Használat becsült aránya** (felhasználási módonként)
Irodalomkutatás			
Prompt lényegi része			
Programkód generálása	Gemini 3 Pro	6. Mérések és értékelés	Algoritmus tervezése: 30% Kód szintaxis: 100%
Prompt lényegi része	Írj két Python szkriptet egy szakdolgozati méréshez, amelyben a hálózati köridőt (RTT) vizsgálom egy vezérlő PC és egy robot között. Az első szkript (PC oldal) küldjön ciklikusan 'ping' üzeneteket egy megadott topikra, mérje precízen a válasz beérkezéséig eltelt időt, kezelje a csomagvesztést timeout figyeléssel, a folyamat végén készítsen statisztikát (min, max, átlag), és minden egyes mérés adatait (időbélyeg, státusz, késleltetés ms-ban) exportálja CSV fájlba a későbbi elemzéshez. A második szkript (Robot oldal) működjön egyszerű válaszadóként, amely feliratkozik a bejövő topikra, és az üzenet érkezésekor azonnal, feldolgozási késleltetés nélkül visszaküldi a választ. A kód legyen robusztus, hibakezeléssel ellátott, a hálózati paramétereket (IP, Port, Topikok) pedig változókban tárolja a könnyű konfigurálhatóság érdekében.		

Új ötletek, megoldási javaslatok generálása			
Prompt lényegi része			
Vázlat létrehozása (szövegstruktúra, vázlatpontok)	Gemini 3 Pro	2., 3., 4., 5., 7. fejezetek	3%
Prompt lényegi része	Szakdolgozatom témája [Téma]. Jelenleg a [X.] fejezeten dolgozom, aminek címe [FEJEZET CÍME]. Javasolj logikus alfejezeteket ehhez a részhez. Mindegyik alfejezethez írj egy 1 mondatos leírást arról, hogy minek kellene benne szerepelnie.		
Szövegblokkok létrehozása	Gemini 3 Pro	Teljes dokumentum	Tartalmi generálás 0% Nyelvi formázás: 80%
Prompt lényegi része	A szakdolgozatom **[X.]** fejezetéhez írtam egy nyers szöveget, amely tartalmazza az összes szükséges információt és forrást, de a fogalmaz nem jó. Írd át az alábbi szöveget koherens, magas színvonalú, tudományos magyar nyelvezettel. Kizárólag az általam megadott tényekből és adatokból dolgozz. Ne hallucinálj eredményeket. A szöveget tagold logikusan bekezdésekre, javítsd a gondolatmenet ívét. A szövegben jelölt hivatkozásokat ([1], [2]) pontosan tartsd meg az eredeti helyükön. A nyers szöveg: **[].**		
Képek generálása illusztrációs célból			
Prompt lényegi része			
Adatvizualizáció, grafikonok generálása adatpontok alapján	ChatGPT	10. ábra, 11. ábra, 12. ábra, 14. ábra, 15. ábra	4%
Prompt lényegi része	Itt vannak mérési adatok: [Adatok]. Generálj belőlük egy vonaldiagramot és egy hisztogramot. Emeld ki az átlagot, és mediánt.		

Prezentáció készítése			
Prompt lényegi része			
Egyéb (nevezze meg)			
Prompt lényegi része			
Összesített százalékos érték (a feladat érdemi részére nézve)			**5-10%**

Összesített érték rövid, szöveges indoklása:

A mesterséges intelligenciát kizárólag támogató eszközként alkalmaztam az implementáció gyorsítására (kódszintaxis generálása), az adatvizualizáció automatizálására és a dokumentáció nyelvi lektorálására. A szakdolgozat érdemi tudományos érték, a hibrid felhő architektúra tervezése, a mérési metodika kidolgozása, valamint az eredmények kiértékelése teljes mértékben saját, önálló mérnöki munka, amelyet az AI nem befolyásolt.

Irodalomjegyzék

[1] G. Mies, „ROBOTICS 2010,” DEBRECENI MŰSZAKI KÖZLEMÉNYEK 2010/2, 2010.

[2] The Welding Institute, „What is Industrial Automation and Robotics?,” [Online]. Available: https://www.twi-global.com/technical-knowledge/faqs/what-is-industrial-automation-and-robotics. [Hozzáférés dátuma: 01 november 2025].

[3] S. Torrejón Pérez, D. Klenert, R. Marschinski, E. Fernández-Macías and I. González Vázquez, “The impact of industrial robots on the EU economy,” 2020. [Online]. Available: https://joint-research-centre.ec.europa.eu/system/files/2020-10/jrc121953_policy_brief_the_impact_of_industrial_robots_on_the_eu_economy.pdf. [Accessed 01 november 2025].

[4] Siemens AG, „The next era of industrial robotics,” augusztus 2024. [Online]. Available: https://assets.new.siemens.com/siemens/assets/api/uuid:6d2a4049-9644-4bca-9d19-49bda3fed1e8/Tech-Trends-2030-The-Next-Era-of-Robotics-August-2024.pdf. [Hozzáférés dátuma: 01 november 2025].

[5] K. Shah, D. Shah és P. Rathod, „All about Cloud Robotics,” január 2022. [Online]. Available: https://irjiet.com/common_src/article_file/1643697732_d10e9264ca_6_irjiet.pdf. [Hozzáférés dátuma: 01 november 2025].

[6] A. Mukherjee, „Cloud Robotics Architectures: Challenges and Applications,” [Online]. Available: https://journals.indexcopernicus.com/api/file/viewByFileId/955186. [Hozzáférés dátuma: 1 november 2025].

[7] M. Puleri, R. Sabella és A. Osseiran, „Cloud robotics: 5G paves the way for mass-market automation,” 28 június 2016. [Online]. Available: https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/cloud-robotics-5g-paves-the-way-for-mass-market-automation. [Hozzáférés dátuma: 1 november 2025].

[8] Yahboom, „12DOF AI Large Model Robot Dog DOGZILLA S1/S2 for Raspberry Pi 5(ROS2-HUMBLE),” [Online]. Available: https://category.yahboom.net/products/dogzilla-s1. [Hozzáférés dátuma: 05 12 2025].

[9] Wikipedia, „Robot Operating System,” 05 11 2025. [Online]. Available: https://en.wikipedia.org/wiki/Robot_Operating_System. [Hozzáférés dátuma: 03 12 2025].

[10] Open Robotics, „Internal ROS 2 interfaces," 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Concepts/Advanced/About-Internal-Interfaces.html. [Hozzáférés dátuma: 03 12 2025].

[11] Open Robotics, „Creating an rmw implementation," 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Advanced/Creating-An-RMW-Implementation.html. [Hozzáférés dátuma: 03 12 2025].

[12] Open Source Robotics Foundation, Inc., „ROS on DDS," 07 2019. [Online]. Available: https://design.ros2.org/articles/ros_on_dds.html. [Hozzáférés dátuma: 03 12 2025].

[13] Data Distribution Service (DDS) Community, „ROS 2: What is DDS," [Online]. Available: https://community.rti.com/page/ros-2-what-dds. [Hozzáférés dátuma: 03 12 2025].

[14] H. Kutluca, „Robot Operating System 2 (ROS 2) Architecture," 06 12 2020. [Online]. Available: https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776. [Hozzáférés dátuma: 03 12 2025].

[15] Open Robotics, „Understanding nodes," 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html. [Hozzáférés dátuma: 03 12 2025].

[16] Luqman, „ROS2 Topics, Services, and Actions Explained: A Beginner's Guide," 28 04 2025. [Online]. Available: https://robotisim.com/ros2-topics-services-actions-explained/. [Hozzáférés dátuma: 03 12 2025].

[17] Open Robotics, „Topics vs Services vs Actions," 2025. [Online]. Available: https://docs.ros.org/en/jazzy/How-To-Guides/Topics-Services-Actions.html. [Hozzáférés dátuma: 03 12 2025].

[18] Open Robotics, „Understanding topics," 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html. [Hozzáférés dátuma: 07 12 2025].

[19] Open Robotics, „Understanding services," 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html. [Hozzáférés dátuma: 07 12 2025].

[20] Open Robotics, „Understanding actions,” 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html. [Hozzáférés dátuma: 10 12 2025].

[21] Open Robotics, „Understanding real-time programming,” 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Demos/Real-Time-Programming.html. [Hozzáférés dátuma: 03 12 2025].

[22] National Institute of Standards and Technology, „The NIST Definition of Cloud,” [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf. [Hozzáférés dátuma: 03 12 2025].

[23] C. R. China és M. Goodwin, „IaaS, PaaS, SaaS: What's the difference?,” [Online]. Available: https://www.ibm.com/think/topics/iaas-paas-saas. [Hozzáférés dátuma: 03 12 2025].

[24] E. Simmon, „Evaluation of Cloud Computing Services Based on NIST SP 800-145,” 02 2018. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-322.pdf. [Hozzáférés dátuma: 03 12 2025].

[25] Google Cloud, „PaaS vs. IaaS vs. SaaS vs. CaaS: How are they different?,” [Online]. Available: https://cloud.google.com/learn/paas-vs-iaas-vs-saas. [Hozzáférés dátuma: 03 12 2025].

[26] National Institute of Standards and Technology, „NIST Cloud Computing Reference Architecture,” 09 2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-292.pdf. [Hozzáférés dátuma: 03 12 2025].

[27] Amazon Web Services, „What’s the Difference Between Docker and a VM?,” [Online]. Available: https://aws.amazon.com/compare/the-difference-between-docker-vm/. [Hozzáférés dátuma: 03 12 2025].

[28] Microsoft, „Containers vs. virtual machines,” [Online]. Available: https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm. [Hozzáférés dátuma: 03 12 2025].

[29] Docker Inc., „Use containers to Build, Share and Run your applications,” [Online]. Available: https://www.docker.com/resources/what-container/. [Hozzáférés dátuma: 03 12 2025].

[30] Docker Inc., „What is Docker?," [Online]. Available: https://docs.docker.com/get-started/docker-overview/. [Hozzáférés dátuma: 03 12 2025].

[31] The Kubernetes Authors, „Overview," 11 09 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/. [Hozzáférés dátuma: 03 12 2025].

[32] H. Finch, „Explain the Architecture of Kubernetes," 02 09 2025. [Online]. Available: https://medium.com/@haroldfinch01/explain-the-architecture-of-kubernetes-719339ca7a3c. [Hozzáférés dátuma: 03 12 2025].

[33] S. Khisty, „Kubernetes Architecture: The Ultimate Guide," 11 08 2025. [Online]. Available: https://devtron.ai/blog/kubernetes-architecture-the-ultimate-guide/. [Hozzáférés dátuma: 03 12 2025].

[34] The Kubernetes Authors, „Kubernetes Components," 31 04 2025. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Hozzáférés dátuma: 03 12 2025].

[35] The Kubernetes Authors, „Network Plugins," 30 07 2024. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/. [Hozzáférés dátuma: 03 12 2025].

[36] The Kubernetes Authors, „Concepts," 22 06 2020. [Online]. Available: https://kubernetes.io/docs/concepts/. [Hozzáférés dátuma: 03 12 2025].

[37] Tigera, „Kubernetes CNI Explained," [Online]. Available: https://www.tigera.io/learn/guides/kubernetes-networking/kubernetes-cni/. [Hozzáférés dátuma: 03 12 2025].

[38] Apptio, „Kubernetes Autoscaling," [Online]. Available: https://www.apptio.com/topics/kubernetes/autoscaling/. [Hozzáférés dátuma: 03 12 2025].

[39] Prometheus Authors, „Overview," [Online]. Available: https://prometheus.io/docs/introduction/overview/. [Hozzáférés dátuma: 03 12 2025].

[40] Grafana Labs, „Explore your infrastructure with Kubernetes Monitoring," [Online]. Available: https://grafana.com/docs/grafana-cloud/monitor-infrastructure/kubernetes-monitoring/navigate-k8s-monitoring/. [Hozzáférés dátuma: 03 12 2025].

[41] A. Bank, E. Briggs, K. Borgendale és R. Gupta, „MQTT Version 5.0," 07 04 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. [Hozzáférés dátuma: 10 12 2025].

[42] U. Hunkeler és H. L. Truong, „MQTT-S – A Publish/Subscribe Protocol For Wireless Sensor Networks," 2008. [Online]. Available: https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/mqtt-s.pdf. [Hozzáférés dátuma: 10 12 2025].

[43] HSNLab, „5G/6G Research at the Ericsson-BME 5G Campus Network," 2025. [Online]. Available: https://www.hsnlab.hu/research/6g. [Hozzáférés dátuma: 10 12 2025].

[44] B. I. Eck és D. Z. Wirker, „TÁVOLI XR MEGJELENÍTÉST BIZTOSÍTÓ SZOLGÁLTATÁS MINŐSÉGÉNEK VIZSGÁLATA PEREMSZÁMÍTÁSI HÁLÓZATOKBAN," 2025. [Online]. Available: https://tdk.bme.hu/ConferenceFiles/VIK/2025/Paper/Tavoli-XR-megjelenitest-biztosito-szolgaltatas-20251103-001837.pdf?paperId=16951. [Hozzáférés dátuma: 10 12 2025].

[45] „OpenStack," 2025. [Online]. Available: https://www.openstack.org/. [Hozzáférés dátuma: 10 12 2025].

[46] Celona & RCR Wireless News (sponsored), „Making private 5G real for the AI-powered enterprise – the Celona way," 08 12 2025. [Online]. Available: https://www.ericsson.com/en/blog/2025/12/how-private-5g-and-edge-compute-drives-manufacturings-real-time-insights. [Hozzáférés dátuma: 10 12 2025].

[47] husarnet, „Operate At The Edge Of Latency," 2025. [Online]. Available: https://husarnet.com/. [Hozzáférés dátuma: 11 12 2025].

[48] J. A. Donenfeld, „WireGuard: fast, moder, secure VPN tunnel," 2025. [Online]. Available: https://www.wireguard.com/. [Hozzáférés dátuma: 11 12 2025].

[49] husarnet, „A VPN Designed For ROS & ROS 2," 2025. [Online]. Available: https://husarnet.com/robotics. [Hozzáférés dátuma: 11 12 2025].

[50] eclipse-zenoh, „zenoh-plugin-ros2dds," 2025. [Online]. Available: https://github.com/eclipse-zenoh/zenoh-plugin-ros2dds. [Hozzáférés dátuma: 11 12 2025].

[51] J. Zhang, X. Yu, S. Ha, J. P. Queralta és T. Westerlund, „Comparison of Middlewares in Edge-to-Edge and Edge-to-Cloud Communication for Distributed ROS2 Systems," 16 11 2024. [Online]. Available: https://arxiv.org/abs/2309.07496. [Hozzáférés dátuma: 11 12 2025].

[52] Eclipse Zenoh, „Integrating ROS2 with Eclipse zenoh," 28 04 2021. [Online]. Available: https://zenoh.io/blog/2021-04-28-ros2-integration/. [Hozzáférés dátuma: 11 12 2025].

[53] J. Ichnowski, K. Chen, K. Dharmarajan, S. Adebola, M. Danielczuk, V. Mayoral-Vilches, N. Jha, H. Zhan, E. LLontop, D. Xu, C. Buscaron, J. Kubiatowicz, I. Stoica, J. Gonzalez és K. Goldberg, „FogROS2: An Adaptive Platform for Cloud and Fog Robotics Using ROS 2," 24 04 2023. [Online]. Available: https://arxiv.org/abs/2205.09778. [Hozzáférés dátuma: 11 12 2025].

[54] HiveMQ Team, „HiveMQ - TLS and MQTT: How is the performance affected?," 02 05 2024. [Online]. Available: https://www.hivemq.com/blog/how-does-tls-affect-mqtt-performance/. [Hozzáférés dátuma: 10 12 2025].