MŰEGYETEM 1782

DIPLOMATERVEZÉSI FELADAT

Veress Gábor
Villamosmérnök hallgató részére

Keretrendszer energetikai felügyelethez

A villamosenergetikai rendszerekben egyre inkább elvárássá válik, hogy a felhasználóhoz közeli hálózatrészekben is távolról felügyelhető, és vezérelhető elemeket, például távolról is vezérelhető kismegszakítókat telepítsenek. A hatékonyság és a biztonság növelése egyaránt célja lehet az ilyen fejlesztéseknek.

Az ebben rejlő lehetőségek megvizsgálására érdemes olyan keretrendszert kidolgozni, melyben mérésre és beavatkozásra képes kis bonyolultságú végponti elemeket egy adatbázissal támogatott monitorozó komponenssel kötünk össze. Az adatgyűjtés eredményét egy felügyeleti logika dolgozhatja fel, és ennek döntéseit a végpontokat vezérlő információként használhatjuk fel. A teszteléshez és a hatások elemzéséhez a végponti elemek működésének és bemeneteinek szimulációjára is szükség van.

A rendszer stabilitásának növeléséhez célszerű a komponenseket konténer-környezetben futtatni, és a redundanciájukat, illetve skálázhatóságukat biztosítani.

A hallgató feladatai a következők:

- Tekintse át az egyszerű energetikai eszközök felügyeletét ellátó megoldásokat!
- Azonosítsa a szükséges komponenseket, és tervezze meg a keretrendszert!
- Valósítsa meg a monitorozás, az adattárolás, és a felügyeleti logika komponenseit és azok
- kommunikációját, figyelembe véve az alapvető biztonsági elvárásokat is!
- Készítsen skálázható, konténeralapú komponenseket, és hangolja össze az elemek működését Kubernetes segítségével!
- Alkalmazzon redundanciát a hálózatban is, és javasoljon megoldást a végponti elemek megbízható kezelésére!
- Dolgozzon ki a működés tesztelésére alkalmas szkenáriókat, melyek pillanatnyi állapotokat, vagy időzített változásokat szimulálnak, és ezek segítségével értékelje a rendszer működését hibamentes állapotban, és egyes egyszerű hibák esetében!

Tanszéki konzulens: Dr. Zsóka Zoltán docens

Külső konzulens:

Budapest, 2025. március 3.

Dr. Imre Sándor
egyetemi tanár
tanszékvezető

Konzulensi vélemények:
Tanszéki konzulens: □ Beadható, □ Nem beadható, dátum: aláírás:
Külső konzulens: □ Beadható, □ Nem beadható, dátum: aláírás:

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék
1117. Budapest, Magyar tudósok körútja 2. I. Ép. B. 121.
Telefon: 463-3261 Fax: 463-3263
URL: http://www.hit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Keretrendszer energetikai felügyelethez

Diplomaterv

Készítette
Veress Gábor

Konzulens
dr. Zsóka Zoltán

2025. december 5.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2025. december 5.

Veress Gábor
hallgató

Kivonat

Az elosztott villamosenergia-felhasználás (elektromos járműtöltők, lokális termelők, intelligens fogyasztók) gyors terjedése új követelményeket támaszt a villamos energia elosztó hálózattal szemben. A dolgozat egy olyan, nyílt forrású és konténerizált keretrendszert mutat be, ami kis költségű eszközöket (ESP8266 alapú mérő), egy Python-alapú vezérlőt, idősoros adatkezelést (Prometheus) és vizualizációt (Grafana) integrál egy egységes megoldásba. A cél egy könnyen reprodukálható és testreszabható rendszer megvalósítása, ami képes valós idejű felügyeletre és korlátos erőforrások kezelésére.

A javasolt architektúra a mérést és a vezérlést átlátható interfészeken mutatja. A végpontok áramértékeket exportálnak Prometheus-kompatibilis metrikákként. A kontrollszerver (Python/Flask) REST API-n keresztül fogadja a méréseket, majd ipari környezetben elterjedt protokollon, Modbus/TCP-n keresztül hajt végre beavatkozásokat az energetikai eszközökön. A rendszer komponensei konténerekben futnak, fejlesztői környezetben Docker Compose biztosítja az orkesztrációt, míg nagyobb rendelkezésre állási igény esetén Kubernetes alkalmazható.

A rendszer kulcseleme egy max-min fair allokációs elv, ami egy előre rögzített globális áramkeretet tartat be több fogyasztó között. Ez a kisebb igényeket preferálja, majd a fennmaradó kapacitást olyan módon osztja szét, hogy a fogyasztás összege kiegyenlítődjön. Ez a megközelítés determinisztikus, egyszerűen paraméterezhető, és jól illeszthető valós idejű döntésekhez. A szabályozó a mérésekből származó idősoros adatokon dolgozik, és szabályok mentén (időablakok, prioritások, határértékek) állítja elő a beavatkozási parancsokat.

A megvalósítást szimulációs környezetben próbáltam. Itt ESP8266-alapú végpont gyűjtött terhelési adatokat valós idejű vizualizációval, miközben a kontrollkomponens dinamikusan korlátozta a fogyasztókat a megadott áramkeretre. A szimuláció determinisztikus bemenetekkel (küszöbök, ütemezések, vezérlési szkriptek) engedte meg különböző terhelési profilok és hibaesemények létrehozását, az algoritmus stabilitásának és működésének vizsgálatára. A tapasztalatok szerint a rendszer képes a keretek pontos követésére, a túllépések gyors csillapítására és az erőforrások igazságos elosztására.

A dolgozat hozzájárulásai a következők: (i) egységesített, Prometheus kompatibilis mérési interfész energetikai rendszerekhez (ii) max-min fair elosztást megvalósító vezérlő (iii) konténer-alapú referenciaimplementáció Docker Compose és Kubernetesben (iv) Grafana alapú üzemviteli és diagnosztikai irányítópultok (v) reprodukálható tesztkörnyezet. A keretrendszer akár ipari célokra is alkalmas, költséghatékony alternatívát kínál a zárt, gyártóspecifikus megoldásokkal szemben.

Abstract

The rapid spread of distributed electricity use (electric vehicle chargers, local producers, smart consumers) places new demands on the electricity distribution network. The thesis presents an open source and containerized framework that integrates low-cost devices (ESP8266-based meter), a Python-based controller, time series data management (Prometheus) and visualization (Grafana) into a single solution. The goal is to implement an easily reproducible and customizable system that is capable of real-time monitoring and managing limited resources.

The proposed architecture exposes measurement and control through transparent interfaces. The endpoints export current values as Prometheus-compatible metrics. The control server (Python/Flask) receives measurements via REST API, then performs commands on energy devices via Modbus/TCP, a protocol common in industrial environments. The system components run in containers, Docker Compose provides orchestration in the development environment, while Kubernetes can be used in cases of higher availability requirements.

The key element of the system is a max-min fair allocation principle, which maintains a pre-set global power limit among multiple consumers. This prefers smaller demands, then distributes the remaining capacity in such a way that the amount of consumption is balanced. This approach is deterministic, easily parameterizable, and well suited to real-time decisions. The controller works on time-series data from measurements and generates intervention commands based on rules (time windows, priorities, limit values). I tried the implementation in a simulation environment. Here, an ESP8266-based endpoint collected load data with real-time visualization, while the control component dynamically limited the consumers to the specified power frame. The simulation allowed the creation of different load profiles and fault events with deterministic inputs (thresholds, schedules, control scripts), to test the stability and operation of the algorithm. According to the experience, the system is able to accurately track the frames, quickly mitigate overshoots, and fairly allocate resources.

The contributions of the thesis are as follows: (i) a unified, Prometheus-compatible metering interface for energy systems (ii) a controller implementing max-min fair allocation (iii) a container-based reference implementation in Docker Compose and Kubernetes (iv) Grafana-based operational and diagnostic dashboards (v) a reproducible test environment. The framework is suitable even for industrial purposes and offers a cost-effective alternative to closed, manufacturer-specific solutions.

1. fejezet

Bevezetés

Az elmúlt évtizedben az energetikai szektor, és különösen a villamosenergia-rendszer, alapvető változásokon ment keresztül. A megújuló energiaforrások - elsősorban a nap- és szélenergia - exponenciális terjedése, valamint az elektromobilitás megjelenése új kihívások elé állította a hálózatüzemeltetőket. A hagyományos, egyirányú energiaáramlásra méretezett hálózatoknak ma már dinamikus, kétirányú terheléseket kell kezelniük, ahol a fogyasztás és a termelés egyensúlya folyamatosan ingadozik.

A jövő energetikája adatvezérelt: a mérés, az adatfeldolgozás és a beavatkozás ciklusidejének drasztikus csökkenése a kulcs a fenntartható üzemeltetéshez.

1.1. Motiváció és ipari környezet

Az épületautomatizálás és az ipari folyamatirányítás területén jelenleg egy érdekes kettősség figyelhető meg. A piacon domináns nagyvállalati megoldások (például Siemens, Schneider Electric) robusztus, szabványosított hardvert és szoftvert kínálnak. Ezek a rendszerek zárt ökoszisztémákra épülnek. Magas licenckköltségeik, egyedi kommunikációs protokolljaik és a gyártói függőség gyakran gátat szabnak a kisebb felhasználóknak.

Ezzel párhuzamosan, az Ipar 4.0 rendszerek részeként, az olcsó mikrokontrollerek és beágyazott rendszerek ma már olyan számítási kapacitással és kommunikációs képességekkel rendelkeznek, amelyek néhány éve még csak drága ipari PC-k kiváltságai voltak. Ez a technológiai demokratizálódás teremt alapot dolgozatom motivációjának: lehetséges-e ipari szemléletű, megbízható felügyeleti rendszert építeni ezen költséghatékony, nyílt eszközök felhasználásával?

1.2. A dolgozat célkitűzései

Jelen diplomaterv célja egy olyan moduláris, nyílt forráskódú keretrendszer tervezése és megvalósítása, amely alternatívát kínál a drága ipari megoldásokkal szemben kutatási és fejlesztési célokra. A rendszernek képesnek kell lennie a teljes mérési-beavatkozási lánc kezelésére.

A megvalósítás során az alábbi konkrét célokat tűztem ki:

- **Architektúra tervezés:** Egy egységesített, konténerizált szoftverkörnyezet kialakítása a következő elemekkel idősoros adatgyűjtés (Prometheus), vezérlés (Python) és a vizualizáció (Grafana).
- **Végponti integráció:** Költséghatékony IoT eszközök (ESP8266) illesztése a rendszerbe szabványos ipari (Modbus/TCP) és webes (REST API) protokollokon keresztül.
- **Algoritmikus vezérlés:** A water-filling allokációs algoritmus implementálása.
- **Validáció és vizualizáció:** A rendszer működésének igazolása valós idejű mérésekkel, valamint egy átlátható kezelőfelület létrehozása az üzemeltető számára.

A dolgozat bemutatja, hogy a nyílt szabványok és a modern szoftvertechnológiák megfelelő kombinációjával létrehozható egy olyan rugalmas energetikai menedzsment rendszer, amely funkcióban hasonló, mint a drágább ipari megoldásokat.

2. fejezet

Piaci megoldások áttekintése

2.1. Meglévő ipari megoldások

A piaci megoldások bemutatására két piacvezető gyártó, a Schneider Electric és a Siemens rendszereit választottam elemzésre.

Ezek a rendszerek bemutatása azért indokolt, mert jelenleg ők képviselik az ipari sztenderdet az energetikai felügyelet területén. Ezek adják a referenciaalapot, amelyhez viszonyítva reálisan értékelhető a dolgozatban bemutatott saját, költséghatékony fejlesztés teljesítménye és korlátai.

2.1.1. Schneider Power Monitoring Expert

A Schneider Electric kínálatából a Power Monitoring Expert szoftvert vizsgálom, ami a kritikus energiaellátású létesítmények felügyeletére készült. A rendszer célja, hogy átláthatóságot biztosítson az energiaelosztó hálózatban.[4]

2.1.1.1. Alapfunkciók

- Segít csökkenteni a meddő teljesítmény termelést és az ebből keletkező büntetéseket.
- Saját számlát készít, a helyi mérések alapján, hogy összehasonlítási alap legyen a számlákhoz.
- Segít elszámolhatóságot biztosítani alszámlázáshoz.
- Berendezések teljesítményét és várható élettartamát ellenőrzi.
- Valós idejű adatfigyelés, riasztás és energiafolyamatok vezérlése a létesítményen belül.
- Azonosítsa a potenciális áramminőségi problémákat a hálózatában, és értesíti erről a személyzetet.

2.1. ábra. Schneider Electric PME model[4]

2.1.1.2. Előnyök

Az energiamérési rendszer használata átlagban 24%-kal csökkentette a fogyasztást, és 30%-al a költségeket.

Mivel folyamatos megfigyelés és beavatkozás lehetséges, a problémák korai szakaszában orvosolhatóak így ezeket 22%-al lehet csökkenteni. Ez a tudatosság csökkenteni a a hiba utáni visszaállítások idejét is. Ezenkívül segít a mögöttes problémák megtalálásában is.[4]

2.1.2. Siemens SIMATIC Energy Suite

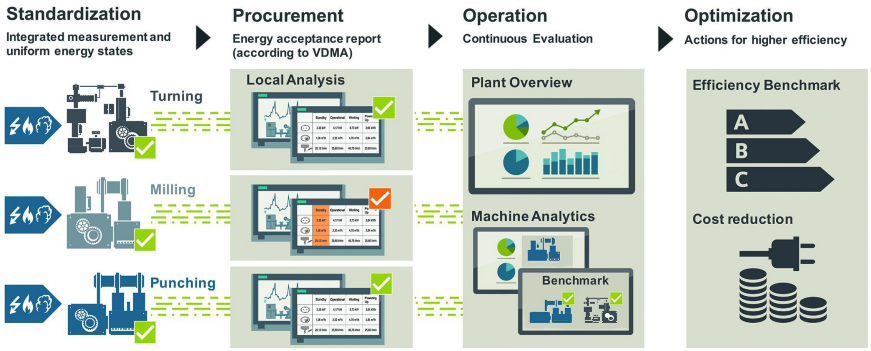
Hasonló helyzetben áll a piacon a Siemens megoldása, a SIMATIC Energy Suite, ez közvetlenül a gyártóautomatizálási környezetbe integrálódik. A rendszer hasonló funkciókkal rendelkezik mint a Schneider rendszere.

2.1.2.1. Alapfunkciók

A Siemens SIMATIC Energy Management rendszere integrált tehát nem csak megfigyelésre alkalmas hanem vezérlésre is. A már létező TIA Portal keretrendszerükbe épül és így egy helyen elérhető a többi rendszerükkel. Ez szintén egy moduláris és skálázható rendszer. Megfelel az ISO 50001 szabványnak, és ez is alkalmazható terhelés figyelésre számlázásra és rendszerelemzésre, mint az előzőleg taglalt rendszer.[23]

2.1.2.2. Előnyök

- Terepi szintű integráció saját és más eszközökkel. Figyelve itt az egyedi eszközökre.
- Gyártás szintű felügyelet. Üzem szintű energia fogyasztást lehet vele figyelni.
- Nagyobb rendszerekben vállalati szintű energiaelemzés, ahol több helyszín között is lehet felügyelni.
- Ezentúl alkalmas beavatkozásra is. Amennyiben túl nagy a fogyasztás képes fogyasztókat leválasztani távolról is akár.



2.2. ábra. Siemens EMS model[23]

2.2. Nyílt forráskódú és közösségi megoldások

Az ipari rendszerek mellett az elmúlt években megjelentek a rugalmasabb, alacsonyabb költségű, jellemzően a "Smart Home" és a kisvállalati szegmensre fókuszáló nyílt megoldások is. Ezek nem rendelkeznek ipari tanúsítványokkal, viszont egyszerűségük miatt jó összehasonlítási alapot képeznek.

2.2.1. Home Assistant

A Home Assistant jelenleg az egyik legnépszerűbb nyílt forráskódú otthonautomatizálási platform. Fő erőssége, hogy lényegében bármilyen IoT eszközt képes integrálni, ami lehetővé teszi többek között energetikai mérők kezelését is. [27]

- **Előnyök:** Ingyenes, nagy közösség, helyi működés, vizuális felület.
- **Hátrányok:** Főként otthoni felhasználásra tervezték, az idősoros adatok hosszú távú tárolása és elemzése korlátozott, nem specifikusan energetikai szabályozásra lett kitalálva.

2.2.2. OpenEnergyMonitor (Emoncms)

Az OpenEnergyMonitor projekt kifejezetten az energetikai mérésekre specializálódott. Szoftveres központja, az Emoncms egy webes alkalmazás elektromos adatok feldolgozására és vizualizációjára. [20]

- **Előnyök:** Ez már energia-fókuszú.
- **Hátrányok:** A vezérlési funkciók nem túl hangsúlyosak. A hardveres rendszer kötöttebb.

2.3. A vizsgálat tanulságai és a tervezési követelmények

A piaci körkép alapján látható, hogy létezik egy szakadék a drága, zárt ipari rendszerek és az általános célú hobbi megoldások között. Míg az ipari eszközök garantálják a pontosságot és a szabványosságot, addig költségvonzatuk megnehezíti használatukat kisebb projektekben.

A saját rendszeremmel szemben ezért nem cél a Siemens vagy Schneider megoldásaival való közvetlen verseny. A cél egy olyan köztes megoldás létrehozása, ami ötvözi a nyílt rendszerek rugalmasságát egy ipari jellegű szabályozási logikával.

Ez alapján a saját fejlesztésű rendszerrel szemben az alábbi **elvárásokat** fogalmaztam meg:

- Modularitás és Nyíltság
- Költséghatékonyság
- Aktív beavatkozási képesség
- Reprodukálhatóság

Összefoglalva: a tervezett rendszer a csináld magad (DIY) árkategóriában kíván megvalósítani egy, a funkcióit tekintve az ipari rendszereket megközelítő, zárt szabályozási rendszert.

Jellemző	**Tervezett rendszer**	**Schneider PME**	**Siemens Energy Suite**
Technológiai alapok			
Eszközök	ESP8266 mikrovezérlő + Áramváltó (CT)	PowerLogic / ION mérők, Smart megszakítók	S7-1500 PLC, Sentron PAC mérők
Kommunikáció	Wi-Fi (IEEE 802.11) REST API / JSON	Zárt ipari hálózat Modbus/TCP	Ipari Ethernet PROFINET
Adatbázis	Prometheus (Idősoros DB)	MS SQL Server (Relációs DB)	WinCC Archívum (Integrált)
Vizualizáció	Grafana (Webes Dashboard)	Power Monitoring Expert (Web kliens)	WinCC Professional (HMI / SCADA)
Funkcionalitás és Költségek			
Analitika	Alapvető mérés + Water-filling algoritmus	Energiaminőség (EN 50160), Zavarelemzés	Terhelésmenedzsment, ISO 50001 riportok
Bekerülési költség	**Alacsony** (< 1 000 €)	**Magas** (Licenc + Eszközök)	**Magas** (Licenc + PLC hardver)
Támogatás	Közösségi (garancia nélkül)	Gyártói 24/7 Support, Hivatalos tanúsítvány	Gyártói Support, TÜV / Szabványi garancia

2.1. táblázat. A tervezett rendszer összehasonlítása az ipari sztenderdekkel

3. fejezet

Keretrendszer

A tervezett rendszer többrétegű, moduláris felépítést követ, ez magában foglalja a fizikai végpontokat, a kommunikációs hálózatot és a felhőalapú vezérlőt is. A rendszer összetevőinek kapcsolatát a 3.2. ábra szemlélteti.

3.1. Standardizált rendszer és követelményei

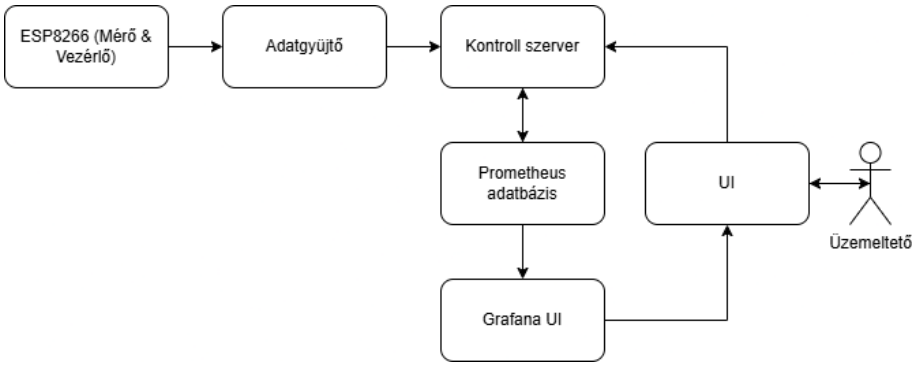
3.1.1. Áttekintés

A rendszer szabványosítja az adattárolást (Prometheus), a vezérlési interfészeket (REST API) és az adatvizualizációt (Grafana), ezzel egyszerűsítve az éles termelési környezetbe történő bevezetést (deployment).

3.1.2. Tervezési célok és követelmények

A keretrendszerrel szemben támasztott legfontosabb funkcionális és nem-funkcionális követelmények az alábbiak:

- Plug-and-Play modularitás
- Szabványosított mérési API
- Központosított vezérlés
- Konténerizált környezet
- Skálázhatóság



3.1. ábra. A keretrendszer architektúrája és Kubernetes komponensei

3.1.3. Komponensek

A rendszer telepítése és üzemeltetése Kubernetes klaszterben történik. Ez a környezet garantálja, hogy a Python vezérlő, a végpontok, valamint a Prometheus és Grafana szolgáltatások folyamatosan elérhetők legyenek.

3.1.3.1. Control Server (Vezérlő)

A központi logikát tartalmazó Python alapú vezérlő Kubernetes `Deployment` formájában jön létre, amelyet egy `Service` tesz elérhetővé a klaszteren belül. A kommunikáció biztonságosságát a `Deployment` manifestjében definiált, TLS tanúsítványokat tartalmazó `Secret` biztosítja.

A Flask alapú REST API két fő végpontot kínál:

- `/metrics`: Ez prometheus kompatibilis formátumban mutatja az aktuális rendszerállapotot és fogyasztási adatokat.
- Vezérlő végpontok: Itt fogadja a rendszer a beavatkozásra adott parancsokat.

3.1.3.2. ESP8266 Szenzor

Az ESP8266 szenzorok és a rendszer közötti kapcsolatot mikroszolgáltatások biztosítják. Ezeket a Kubernetes Podokat HTTPS-en keresztül látja a vezérlő, ezeknek folyamatosan figyeli a `/metrics` végpontját.

Az elvárt "Plug-and-Play" működést a Kubernetes annotációi teszik lehetővé. Ha egy új eszköz kerül telepítésre akkor ennek a manifest tartalmazza a `prometheus.io/scrape: "true"` és `prometheus.io/path: "/metrics"` értékeit, a Prometheus, pedig automatikusan, új konfiguráció beállítása nélkül is fel fogja venni ezt az eszközt.

3.1.3.3. Adattárolás és Vizualizáció

Az adatbiztonság miatt a Prometheus nem egyszerű Deployment, hanem `StatefulSet`. Az adatbázist egy `PersistentVolumeClaim` segítségével tartós tárhelyre menti, így egy újraindítás után is megmaradnak az adatok. A scrape config-ot, `ConfigMap`-ekben lehet definiálni.

A Grafana vizualizációs felület is tartós tárhelyet használ a dashboard-ok és felhasználói beállítások mentésére.

3.2. Rendszerarchitektúra áttekintése

Az áramérzékelők (áramváltó bilincsek) mérik az elektromos áramot és az adatokat egy ESP8266 gyűjti, ezek pedig a központi vezérlőhöz táplálják, amely vezérlőparancsokat küld visszafelé az elektromos járművek töltőinek a megengedett áram beállításához.

Az ESP8266-alapú érzékelőcsomópontok mindegyik elektromos töltőnél el vannak helyezve, hogy valós időben mérjék a töltőáramot. Ezek pedig Wi-Fi-n keresztül küldik el az adatokat egy Python Flask alkalmazást futtató vezérlőhöz, amely összesíti a méréseket és kiadja a vezérlőparancsokat.

Maguk az töltők Modbus kommunikációval rendelkeznek, ezért a Modbus protokollon keresztül fogadják a távolról érkező utasításokat (esetünkben a megengedett áram beállítását). Mindegyik töltő saját címmel rendelkezik a soros hálózaton.

Az ESP8266 csomópontok csak az adatok két oldalú továbbadásáért felelősek, aktuális adatokat küldenek a szervernek, míg a Flask szerver döntéshozatalt hajt végre és parancsokat ad ki a töltőknek. Az érzékelés és a vezérlés szétválasztása leegyszerűsíti a végpont tervezését és a feldolgozást a szerver oldalon központosítja, ami növeli a robosztusságot.

A Flask szerver egy Prometheus idősoros adatbázissal dolgozik (ami külön konténer alapú szolgáltatásként fut), ez naplózza az összes mérést a megfigyeléshez és elemzéshez. Az összes kiszolgálóoldali összetevő (a Flask alkalmazás, a Modbus interfész és a Prometheus) a Docker használatával van konténerben tárolva a felhőalapú környezetben történő egyszerű telepítés érdekében. Az architektúra a következőket tartalmazza:

- ESP8266 érzékelő csomópontok: Wi-Fi csatlakozású mikrokontrollerek minden végponton (legyen az töltő, megszakító, stb. . .), amelyek a csatlakoztatott érzékelőkön keresztül mérik a váltakozó áramot.

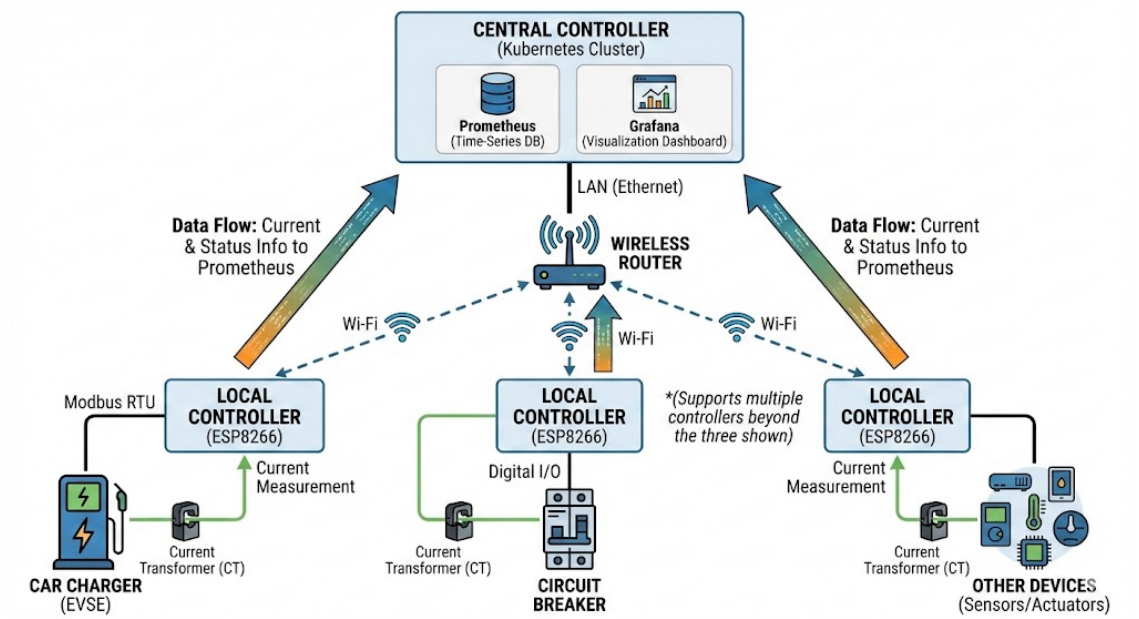
- Wi-Fi hálózat: Ami biztosítja, hogy a végpontok tudjanak kommunikálni a központi szerverrel. Mindegyik csomópont csatlakozik a helyi Wi-Fi-hez és HTTPS-kéréseken keresztül adatokat küld a szerver REST API-jának.

- Flask alapú központi szerver: Helyi szerveren vagy cloud környezetben is futhat. Mérési adatokat fogad az ESP8266 csomópontoktól, feldolgozza és tárolja azokat és ahogy már említettem Modbus segítségével vezérlőjeleket küld az EV-töltőknek.

- Modbus kommunikációs kapcsolat: Összekapcsolja a végpontokat a töltőkkel. Ez esetünkben Modbus/TCP over Ethernet. A végpontok Modbus masterként működnek, és minden elektromos töltő egy Modbus slave eszköz.

- Prometheus adatbázis: Idősoros adatbázis, amely összegyűjti és tárolja a mért értékeket (pl. áramok, töltőállapotok, megszakító állapotok) a Flask szerverről valós idejű megfigyeléshez és későbbi elemzéshez.

- Docker containerek: A Flask szerver és a Prometheus Docker-tárolókban fut, így megvalósul a mikroszolgáltatás alapú összeállítás, ami akár helyi szerveren, akár modern felhő natív rendszeren jól fut. A Docker biztosítja, hogy a konténer taralmazza az összes szükséges függőséget (Python-könyvtárak stb.), így megkönnyítve az üzemeltetés dolgát, és lehetővé teszi a rendszer megbízható méretezését vagy replikálását.



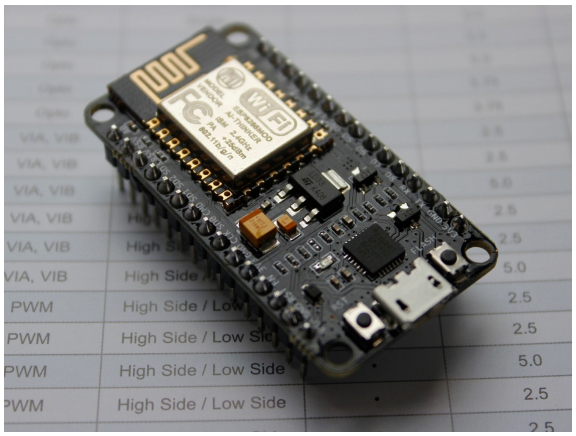
3.2. ábra. Rendszerarchitektúra

3.3. Eszközök

3.3.1. ESP8266 és AC árammérő szenzorok

A pontos árammérés minden elektromos töltőnél kritikus a rendszer számára. Az ESP8266-ot (NodeMCU) aminek az io felosztása 3.5 ábrán látszik, nem invazív váltakozóáram-érzékelőkkel párosítva használjuk a megfelelő áramkörök által felvett áramerősség mérésére. Egy megfelelő érzékelő az YHDC SCT-013 sorozatú bilincses áramtranszformátor, például az SCT-013-030 modell, ami 30 A AC feszültségre van méretezve. Az SCT-013 egy osztott magú áramváltó így könnyű a csatlakozása, ez a tápkábelnek feszültség alatt álló vezetéke köré kerül, és nincs szükség közvetlen elektromos érintkezésre a vezetővel. Ez az érzékelő a kábelen átfolyó árammal arányos kis váltakozó feszültséget ad ki. Különösen az SCT-013-030 körülbelül 0-1 V AC (effektív) kimenetet produkál 0-30 A mérésekor. [9]

Ez a feszültségtartomány kompatibilis az ESP8266 analóg-digitális átalakítójával az analóg bemenetén, amely a legtöbb ESP8266 kártyán 0-1 V-ot tud olvasni (a NodeMCU kártyák tartalmaznak beépített feszültségosztót, amely lehetővé teszi a 3,3 V-os bemenetet). Így az SCT-013-030 0-1 V-os kimenete közvetlenül az analóg bemenetre rakható. Az SCT-013 érzékelők, amelyek feszültségkimenettel rendelkeznek, már rendelkeznek belső ellenállással, így nincs szükség további terhelésre. [19]

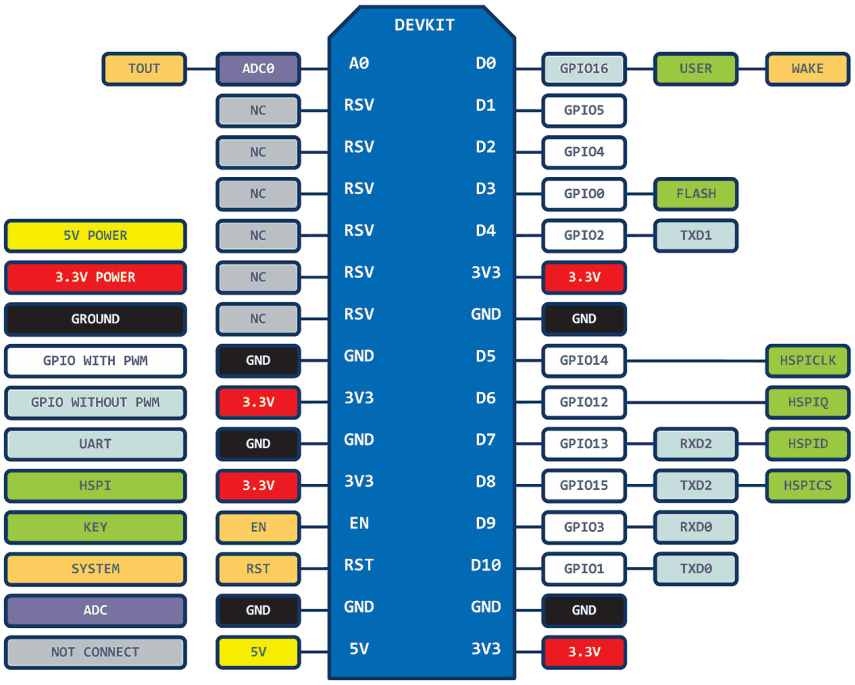


3.3. ábra. NodeMCU (ESP8266) [21]

3.4. ábra. SCT-013 áramváltó [17]

Mindkét esetben szükséges egy csatoló áramkör az érzékelőhöz: A CT AC kimenete 0 V ha nincsen semmi behatás, de az ESP8266 ADC nem tudja leolvasni a negatív feszültséget. Ezért elkell tolnunk az értékeket ehhez kell két ellenállás, amelyek feszültségosztót alkotnak a 3,3 V-os tápegységgel, hogy az érzékelő kimenetét a skála közepére rakjuk. Lényegében az érzékelő két vezetéke csatlakozik: az egyik az ADC bemenethez, a másik pedig a középponthoz körülbelül 1,65 V a 3,3 V-os táp miatt. [19]

Mindegyik ESP8266 tápellátást kap lehetőleg 5 V-os USB-adapterrel az EV-töltő kiegészítő tápellátásával és az analóg bemeneten keresztül olvassa le a CT-érzékelőjét. A mikrokontroller a megírt kódot futtatja, csatlakozik a Wi-Fi-hez, és folyamatosan méri az áramerősséget. Ezt úgy küldi a szervernek, hogy már könnyű legyen prometheusnak tovább küldeni.



3.5. ábra. Pinout [8]

3.3.2. Mért eszközök

A rendszer a hálózat különböző pontjain elhelyezkedő és akár eltérő funkciójú eszközök felügyeletét látja el. Mivel az egyes eszközök működése eltér, a monitorozás során gyűjtött paraméterek is eszerint változnak.

A rendszer az alábbi eszközöket és adatpontokat különbözteti meg:

- **Megszakítók:**
  - Pillanatnyi áramerősség
  - Állapotjelzés
  - Hibajel
  - Túlterhelés figyelmeztetések
- **Autótöltők:**
  - Pillanatnyi áram
  - Állapot (csatlakoztatva, tölt, hiba, stb...)
- **Szekrények:**
  - Hőmérés
  - Gázelemzés (füst érzékelés)

A mérések egy mikrokontrollerbe vannak beprogramozva, sok esetben, hogy a megfelelő és helyileg feldolgozható jelet kapjunk valamilyen hardverre van szükség,

ez átalakítja az eredeti jelet. Ilyen például az áram méréséhez használt áramváltó és sönt ellenállás, jellemzően a nagyobb áramokat 5 A-re transzformáljuk egy áramváltóval.

Esetünkben maga az ESP8266 chip az analóg bemenetén 0 és 1 volt közötti jelszintet vár, viszont a nodeMCU környezet már végez az áramkörön feszültség áttalakítást így a bemeneti skála változik 0 és 3,3 voltra.

Ha árammméréseket áramváltóval akarjuk megvalósítani akkor az áramváltó 5 A-es maximum kimenetét kell a kontroller 3,3 v-os maximum bemenetére alakítani. Ezt egy sönt ellenállással tudjuk megvalósítani.

$$R = \frac{U}{I} = \frac{3.3\,\mathrm{V}}{5\,\mathrm{A}} = 660\,\mathrm{m\Omega} \tag{3.1}$$

1. egyenlet: Árammméréshez használt sönt ellenállás értéke

$$P = U \times I = 3.3\,\mathrm{V} \times 5\,\mathrm{A} = 16.5\,\mathrm{W} \tag{3.2}$$

2. egyenlet: A sönt ellenállás

A számítások után látszik, hogy olyan ellenállásra van szükség, ami $R = 660m\Omega$ ellenállással rendelkezik és legalább 16,5 W teljesítményt el tud diszipálni folyamatos terhelés melett is.

3.4. Kommunikáció

A rendszer kommunikációs topológiáját és az adatfolyam irányait a 3.2. ábra szemlélteti. Ezután ennek a megvalósításával folytatom a két fő protokoll, a HTTP és a Modbus mentén.

3.4.1. Helyivezérlő és Szerver között (Wi-Fi)

A kommunikációhoz az ESP8266 végpontok Wi-Fi-t használnak a mérések továbbítására a vezérlő Flask szerverre. Indításkor minden ESP8266 csatlakozik a konfigurált Wi-Fi hozzáférési ponthoz. A kód első lépése a kapcsolat felépítése a konfigurált Wi-Fi hozzáférési ponttal. Ehhez az ESP8266 standard könyvtárát használtam, ahogy az alábbi firmware-részlet is mutatja:

```
(pl. WiFi.begin(ssid, jelszó))
```

[26] A csatlakozást követően a csomópont képes HTTP vagy esetünkben HTTPS kéréseket küldeni a szerver IP-címére. Egy egyszerű RESTful API-t implementáltam a Flask szerveren az adatok fogadásához. Minden fizikai végpont Prometheus adatbázis jellegű kommunikációhoz is használt végponton hirdeti a mért adatait.

```
app.run(host="0.0.0.0", port=6000, ssl_context=('cert.pem', 'key.pem'))
```

3.1. lista. A Flask szerver indítása Python-ban

```
http://<szerver_ip>:6000/metrics
```

3.2. lista. Az API végpont URL struktúrája

A válaszgeneráló logika az alábbi Python kódrészletben látható:

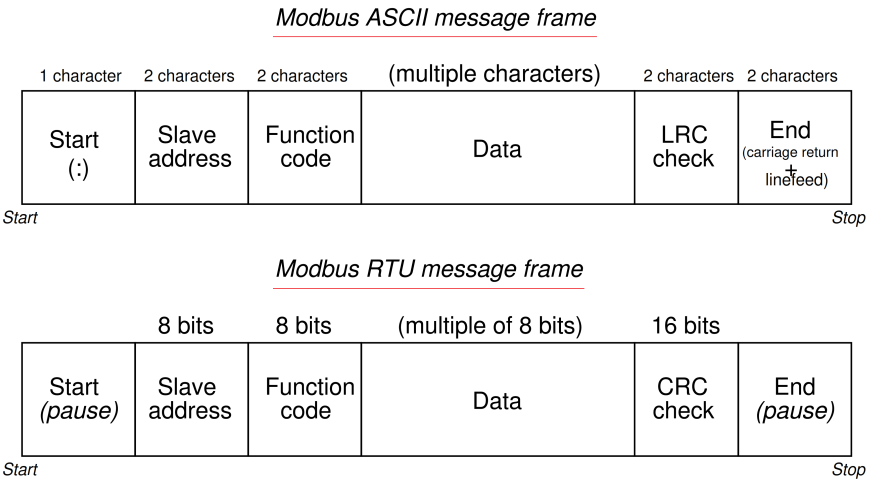
```
def metrics():
    return jsonify({
        "simulator_id": simulator_id,
        "current": current_value,
        "state": "plugged in" if charger_on else "plugged out",
        "max_current": max_current
    })
```

3.3. lista. A JSON választ generáló Python függvény

A Wi-Fi kommunikáció itt nem követeli meg, hogy az ESP8266 ismerje a szerver címét, mert csak GET parancsokat használtam. Itt a kiszolgáló fix IP-címmel rendelkezhet a LAN-ban. Elég viszont, ha a szerver ismeri a végpontok IP címeit, amit viszont könnyű megadni és frissíteni. Kezdetben titkosítatlan HTTP-t használtam, viszont ezt később frissítettem a valódi telepítési környezethez hasonló HTTPS-el. Szerencsére az ESP8266 képes kezelni a TLS-t.

3.4.2. Energetikai eszköz és helyivezrlő között (Modbus)

A vezérlő oldalon a szerver a Modbus protokollt használja az EV-töltőkkel való kommunikációhoz. A Modbus egy széles körben elterjedt protokoll az ipari rendszerekben elektronikus eszközök csatlakoztatására. Eredtileg PLC-k közötti Kommunikáció kialakítására használták. A mi beállításunkban a szerver Modbus masterként van konfigurálva, és minden EV töltő Modbus slave eszköz (Modbus/TCP). A Modbuson keresztül a szerver képes regisztereket olvasni a töltőkről, és regisztereket írni. Ezzel áttudtam írni a töltőben a maximális áramértéket amit engedélyezett. [25]



3.6. ábra. Modbus adatstruktúra [10]

A képen Modbus RTU soros kommunikáció összeállítása látszik. Ebben a rendszerben Modbus TCP rendszert használunk. Ez igazából csak annyit csinál, hogy TCP keretekbe foglalja a már előbb felsorolt kommunikációt.

4. fejezet

Komponensek megvalósítása

Ebben a fejezetben a megtervezett rendszerarchitektúra gyakorlati megvalósítását ismertetem. Részletesebben megmutatom a hardveres és szoftveres komponenseket, a kommunikációs protokollok implementációját, valamint az adatgyűjtő és megjelenítő rétegeket. A fejezet a fizikai rétegtől a "végpontoktól" halad a magasabb szintű szoftveres rétegekig (adatbázis, vizualizáció).

4.1. Végpontok

A rendszerben a fizikai réteget a végpontok alkotják, amik közvetlen kapcsolatban állnak a villamos hálózattal. Feladatuk a kommunikáció biztosítása: egyrészt a mérési adatok (áramfelvétel, státusz) továbbítása a vezérlő felé, másrészt a beavatkozó parancsok végrehajtása a fizikai eszközökön.

4.1.1. Autótöltő

4.1.1.1. Hardver

A rendszer egy ESP8266 mikrokontroller köré épül, amely két elsődleges funkciót lát el:

- Árammérés: Folyamatosan méri az autótöltők által felvett elektromos áramot. Összegyűjti és a Prometheus, egy népszerű nyílt forráskódú felügyeleti rendszerrel kompatibilis formátumban jelenti ezeket a mérési adatokat, hogy a rendszerben egységes adatstruktúrákat használjunk.
- Vezérlő interfész: Emellett olyan mechanizmust biztosít, ami Modbus parancsokon keresztül vezérli az autótöltőket.

- **RS485:** Az ESP8266 natívan nem támogatja az RS485 kommunikációt, viszont tudunk használni egy RS485 adó-vevőt (pl. MAX485). Ez az ESP8266 UART jeleit RS485-re alakítja, ami az ipari kommunikációban elterjedt szabvány, ezért jellemzően a töltőkben és egyéb épületinformatikai eszközökben is megtalálható.
- **ModbusMaster könyvtár:** Itt az open source ModbusMaster könyvtárat [29] használtam a továbbítás egyszerűsítésére.

4.1. ábra. Autótöltő [5]

- **Átviteli vezérlés:** Az előbb említett adó-vevőnek szüksége van egy úgynevezett DE/RE (Driver Enable/Receiver Enable) vezérlőpinre. Amit viszont egyszerű megvalósítani az ESP8266-on egy digitális pin segítségével amire itt a D2 lett használva. Ezzel tudunk később adó és vevő módok között kapcsolni. Küldéshez a pin HIGH (adási mód), ezután a vételhez, pedig (vételi mód) állapotba kerül, ekkor LOW.

4.1.1.2. Szoftver

Itt az ESP8266 firmware főbb részeit elemzem.

WiFi és HTTP-kiszolgáló beállítása Kezdésnek az ESP8266 csatlakozik WiFi-re és ezzel a helyi hálózatra a megadott SSID és jelszóval. A csatlakozást követően az eszköz az ESP8266WebServer könyvtár segítségével inicializál egy HTTPS-kiszolgálót. Ez a szerver egy kijelölt porton (pl. 8663) figyel, és a /metrics végpontot teszi közzé, ahol közli az adatokat a központ vezérlővel.

Mérési adatok elküldése A `sendMetricsToEndpoint()` függvény formázza a méréseket Prometheus-szerű szöveges formába. A metrikák a következőket tartalmazzák:

- **esp8266_current0**: A mért áramértéket mutatja.
- **esp8266_connection**: Az ESP8266 kapcsolati állapotát jelzi, pl.: csatlakozva vagy nem.

Ez a funkció a Prometheus-kompatibilis mért érték és címkézési formátummal küldi el a mérést. Amikor például a vezérlő szerver lekéri a `/metrics` végpontot, a HTTPS-kiszolgáló `200 OK` státusszal küldi vissza ezeket a formázott metrikákat, amennyiben minden rendben ment.

Main loop A loop() funkcióban az ESP8266 folyamatosan kezeli a bejövő HTTPS kéréseket és 30 másodpercenként az eszköz meghívja a queryPrometheus() függvényt, hogy frissítse az összesített metrikát. Ez az időszakos lekérdezési mechanizmus biztosítja, hogy a helyi mérések folyamatosan frissek legyenek és döntéshozatal alapjául lehessen venni őket.

Kommunikáció A rendszer itt is a biztonságos adatátvitel érdekében minden hálózati kommunikációhoz HTTPS protokollt használ. A legfontosabb adatáramlások a következők:

- **Mérések közzététele:** Az ESP8266 összegyűjti az aktuális méréseket, és azokat a /metrics végponton olyan formátumban teszi elérhetővé, ami már alkalmas Prometheus alapú adattárolásra.
- **Visszacsatolási hurok:** Az ESP8266 vezérlési értékeket kap a szervertől, amiket aztán modbuson ad tovább az eszközöknek.

4.1.1.3. Modbus kommunikáció vezérléshez

Ez a funkció az autó töltő áramhatárának beállítására szolgál. Az itt használt Modbus RTU használatával az ESP8266 lesz a master, ami „Write Single Register" parancsot ad az autó töltőnek (Modbus slave). Az autós töltő áramkorlátja egy előre meghatározott regiszterben található.

4.1.2. Megszakító

4.1.2.1. Hardver

A felügyelet- és vezérlésben minden megszakító egy ESP8266 modulhoz van csatlakoztatva, ami megkapja az aktuális állapotot, és ki-/bekapcsolást tud végezni. Legfontosabb komponensek és munkafolyamatok:

- Állapotérzékelés: Az ESP8266 digitális bemenete a megszakító egy segédérintkezőjéhez van kötve. Ha a megszakító zárva van, az érintkező bezár és az ESP bemenetét magasra húzza, ha nyitva van, a bemenet alacsony. Egy sima RC-szűrő és szoftveres pergésmentesítéssel (pl. 50 ms) lehet biztosítani a tiszta és zaj mentes átmeneteket.
- Parancskimenet: Egy GPIO pin egy relét húz meg, ami a megszakító kioldó/-becsukó tekercsét aktiválja.

4.1.2.2. Szoftver

Az ESP8266 arduino alapokon fut, és HTTPS segítségével csatlakozik a LAN-hoz Wi-Fi-n keresztül. Minden megszakító interakció RESTful API hívásokon keresztül történik a Python vezérlő szerverhez:

```
https://<control-server>/api/breakers/<id>/state
```

```
{ "breaker_state": 1 }
```

4.2. ábra. Megszakító [6]

A metrika mezők használatával a Python szerver fordítás nélkül le tudja képezni a bejövő JSON-t a Prometheus-nak megfelelő formátumra (breaker_state és breaker_command).

4.2. Kontroll szerver

4.2.1. Szerepe és technológiai háttere

A Kontroll Szerver a keretrendszer központi döntéshozó egysége. Míg az ESP végpontok feladata a mérés és a fizikai beavatkozás, addig a szerver felelős az adatkezelésért, a korlátok kezeléséért és az erőforrás-allokációs algoritmus futtatásáért.

A megvalósításhoz a **Python** nyelvet és a **Flask** keretrendszert választottam. Azért,erre esett a választásom, mert sok előre elkészített könyvtár érhető el hozzá, és így nem nekem kelett megírnom az egész REST API-t a semmiből. Ez, pedig így könnyen integrálható a konténerizált (Docker/Kubernetes) környezetbe.

4.2.2. Működési logika

A szerver működése eseményvezérelt, a HTTPS kérések indítják el a vezérlési ciklust. A folyamat a következő lépésekből áll:

1. Adatfogadás
2. Globális paraméterek olvasása
3. Allokáció számítása
4. Válasz és vezérlés

4.3. Adatbázis

A rendszer által generált adatok tárolásához egy Prometheus adatbázist használok. A Prometheus egy nyílt forráskódú idősoros adatbázis, ami inkább felhő környezetben ismert, de ugyanolyan hasznos az IoT-telemetria számára. Minden adatot időbélyegzett értéksorozatként kezel. Ezeket lehet tárolni és lekérdezni. [7] [22]

Esetemben minden metrika tárhelyeként szolgál. Ez lehetővé teszi, hogy megőrizzem a töltési áramok történetét és ez alapján irányítsam a rendszert.

4.3. ábra. Prometheus [18]

A Flask szerver-ből könnyű továbbítani az adatokat. A megközelítés amit én használtam hogy egy HTTPS /metrics végpont elérhetővé tettem. Amin prometheus által olvasható formában hirdettem az adatokat. Például a Flask alkalmazás tudja továbbítani a mért számokat:

```
current_gauge = prometheus_client.Gauge(
    'ev_charger_current',
    'Current draw of EV charger',
    ['charger']
)
```

4.1. lista. Prometheus metrika deklarálása a Flask szerveren

A Prometheus adatgyűjtési modellje alapvetően a „pull” elvre épül, az adatbázis előre megadott időközönként lekérdezi a konfigurált végpontokat. Alap estben minden ESP8266 IP-címét külön célpontként kellene felvenni a `prometheus.yml` konfigurációs fájlba.

Ez a megoldás azonban rugalmatlan, mivel a Prometheus konfigurációja statikus: ha új eszközt szeretnénk felvenni ez a konfigurációs fájl szerkesztését és a Prometheus szolgáltatás újraindítását igényli.

Ennek kiküszöbölésére a Kontroll Szervert egy dinamikus aggregációs rétegként alkalmazom. A Prometheus konfigurációjában így egyetlen stabil, statikus célpont szerepel: maga a Kontroll Szerver.

A szerver belső memóriában tartja nyilván a végpontokat és azok legfrissebb méréseit. Amikor a Prometheus lekérdezi a szerver `/metrics` végpontját, a Flask alkalmazás dinamikusan generálja le a választ az összes jelenleg csatlakoztatott eszköz adataiból.

4.3.1. Prometheus adatgyűjtés kezelése

A mikrokontroller több metrikát is mér, amit belső változókba elment. Jelenleg teszt célokból ezek, csak kézzel megadott számok.

```
{
  "# HELP": "esp8266_current Current sensor reading.",
  "# TYPE": "esp8266_current gauge",
  "esp8266_current0": 1.20,
  "esp8266_current1": 2.50
}
```

4.2. lista. Prometheus számára formázott szöveges kimenet példa

Ez a formátum megengedi, hogy ezt a /metrics endpointon a prometheus folyamatosan lekérdezze a mikrokontrollerektől.

A formátumot a következő függvény hozza létre és küldi:

```
sendMetricsToEndpoint();
// ... metrikak osszeallitasa ...
server.send(200, "text/plain", metrics);
```

4.3. lista. Válaszadás a /metrics végponton (ESP8266)

4.3.2. Prometheus lekérdezések kezelése

```
queryPrometheus()
```

4.4. lista. Adatgyűjtő függvény hívása

Ez a függvény egy HTTP GET kérést küld a Prometheus szervernek, amely a esp8266_total_current metrikát kérdezi le és a prometheusValue változóba írja be.

```
/api/v1/query?query=esp8266_total_current
```

4.5. lista. Prometheus lekérdezési URL formátuma

A fentebbi endpointon.
A lekérdezés sikerességét a httpCode ellenőrzésével teszem amennyiben ez 200-at ad vissza az értéket eltárolom és kiírom a soros kommunikáción ellenőrzés céljából.

```
if (httpCode == HTTP_CODE_OK) {
  String payload = http.getString();
  Serial.println("Response from Prometheus:");
  Serial.println(payload);

  DynamicJsonDocument doc(1024);
  DeserializationError error = deserializeJson(doc, payload);

  if (error) {
    Serial.print(F("JSON deserialization failed: "));
    Serial.println(error.c_str());
    return;
  }
}
```

4.6. lista. HTTP válasz fogadása és JSON deserializáció

Mivel a lekérdezés egy JSON formátumú változót ad vissza és ennek feldolgozása nehézkes ezért ezt rögtön szám formátumba alakítom későbbi feldolgozás céljából.

```
const char* status = doc["status"];
if (String(status) == "success") {

  // Az ertek a data -> result -> value tombben talalhato
  const char* valueStr = doc["data"]["result"][0]["value"][1];
```

```
    prometheusValue = String(valueStr).toFloat();

    Serial.print("Extracted Prometheus Value: ");
    Serial.println(prometheusValue);
  }
```

4.7. lista. JSON válasz feldolgozása és az érték kinyerése

A fenti rész kinyeri az adatot JSON formátumból és szám formátumba írja.

Természetesen az egész queryPrometheus loop-ban ismétlődve fut, hogy a kontroller folyamatosan frissítse az értékeket. Jelenleg a gyakoriságot 30 másodpercre állítottam, hogy ne terhelje a próbák során feleslegesen a hálózatot, de gyorsabb válaszidő érdekében ez növelhető.

4.4. Grafana alapú megjelenítés

A szerver automatikusan beavatkozik szükséges esetben, viszont emellett továbbra is szükséges a működtető személyzetnek látnia, a rendszer működését. Ezt folyamatosan ellenőrizni és amennyiben nem megfelelő működés lép fel. Akár nem működik az automatizmus akár rosszul működik, szükséges beavatkozni manuálisan.

4.4.1. Az áramok vizualizálása és riasztások a Grafanában

Ebben a fejezetben bemutatom, hogy a nyers árammérések az egyes EV-töltők és egyéb terhelések hogyan oszlanak meg három fázison, valamint a napelemek bemeneti áramai hogyan jelennek meg Grafanában, és hogyan történik a túláram vagy más veszélyes állapotok automatikus vagy manuális kezelése. Minden eszköz a Prometheus metrikákat exportálja a következőképpen:

```
ev_charger_current_phase_a_amplitude{charger="ev1"} 12.3
ev_charger_current_phase_b_amplitude{charger="ev1"} 11.8
ev_charger_current_phase_c_amplitude{charger="ev1"} 12.1

equipment_current_phase_a_amplitude{device="pump1"} 5.4
...
solar_input_current_amplitude 8.7
```

4.8. lista. Példa az exportált metrikák címkézésére

4.4.2. A Dashboard

1. sor EV töltők áramai amik a $charger változóval vannak jelölve, ez felsorolja az összes ide tartozó cimke értékét (pl. „ev1”, „ev2”, ...). Ezután az idősoros panel: ábrázolja az összegzett értéket három fázison.

```
  ev_charger_current{charger="$charger"}
```

4.9. lista. Lekérdezés az EV töltők áramfelvételére

Itt ugyanazon a tengelyen láthatóak a három fázis összegzett értékei, különböző színnel és elnevezéssel. Az úgynevezett "mérőpanelen" a pillanatnyi fázisáramokat három kis mérő formájában lehet látni igazából továbbra is a a fenti lekérdezésseket használva, pillanatnyi csak üzemmódban. A küszöb értékeket állítottam be a

könnyeb vizualizáció érdekében a töltő névleges áramának, 80 %-ánál (sárga) és 100 %-ánál (piros) vannak beállítva.

2. sor Segédberendezések áramai A $device változóban keressük ezeket a metrikákat.

A panel hasonlóan az előző ponthoz jeleníti meg az adatokat, amely a pillanatnyi és max értéket mutatja.

```
max_over_time(equipment_current_phase_a_amplitude{device="$device"}[1m])
```

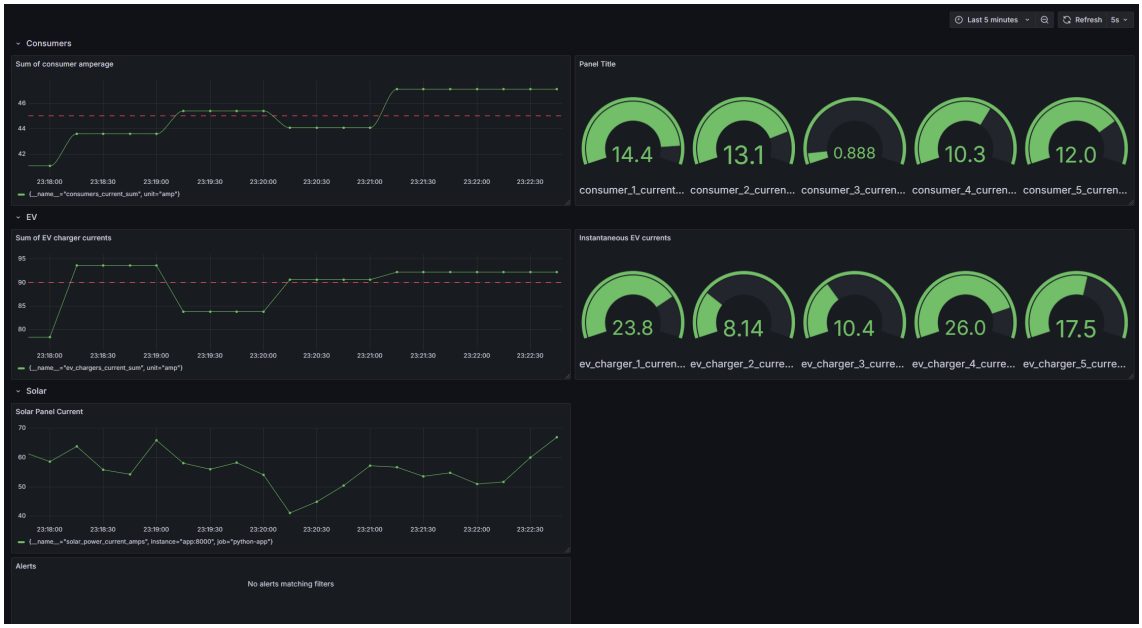
4.10. lista. Maximum kiválasztása időablakban (segédberendezések)

A maximumot minden fázisra az elmúlt percben mutatja, az idetartozó megfelelő színküszöbökkel. Mellette raktam egymás mellé csoportosító oszlopdiagramot a gyors összehasonlításokra.

3. Sor Napelem bemeneti áram Itt szintén egy idősoros panelt alkalmaztam a megjeleníthetőség érdekében.

```
solar_input_current_amplitude
```

4.11. lista. Napelem bemeneti áram lekérdezése



4.4. ábra. Általam készített Grafana dashboard

5. fejezet

Központi vezérlő felépítése

5.1. Flask alapú vezérlő

A központi vezérlő a rendszer "agya", amely a Kubernetes cluster-en belül, konténerizált környezetben fut. A szoftver fejlesztéséhez a **Python** programozási nyelvet és a **Flask** keretrendszert választottam, mivel ezek rugalmas és gyors fejlesztést tesznek lehetővé, valamint kiváló támogatást nyújtanak a RESTful API-k megvalósításához.

A vezérlő architektúrája három fő rétegre bontható:

1. **Kommunikációs réteg:** A bejövő HTTP kéréseket (mérésadatok) és a kimenő válaszokat (aktuális áramlimitek) kezeli. Az Ingress Gateway felelős a forgalom megfelelő pod-hoz való irányításáért.

2. **Adatkezelési réteg:** A beérkező mérési adatokat a rendszer a Prometheus idősoros adatbázisba továbbítja, illetve onnan olvassa ki a historikus adatokat a döntéshozatalhoz.

3. **Logika réteg:** Itt történik a tényleges számítás és döntéshozatal.

Ez az alfejezet kizárólag a szoftverkomponensek technikai felépítését és az adatkapcsolatokat mutatja be. A vezérlő tényleges döntési mechanizmusát, az alkalmazott *water-filling* algoritmust és az allokációs stratégiákat részletesen a következő alfejezetben fejtem ki. A rendszer valós idejű működésének korlátait, különös tekintettel a hálózati késleltetésre (latency) és a szabályozási köridőre, az ezt követő alfejezet tárgyalja.

5.1.1. A Flask alkalmazás felépítése

A Python alkalmazás alapja egy REST API interfész, ami Get kérések küld a lokális vezérlőknek. Minden kérés tartalmazza az adott végpont azonosítóját és az általa mért pillanatnyi áramfelvételt. Az alkalmazás állapotmentes módon működik, az állapotokat a Prometheus adatbázis és a memóriában tárolt rövid távú cache kezeli.

5.2. Max–min fair (water-filling) elosztás

5.2.1. Elméleti háttér és cél

5.2.1.1. A szabályozási feladat és definíciók

A dolgozat rendszerének fő feladata, hogy a rendelkezésre álló, korlátos villamos teljesítményt dinamikusan ossza el a fogyasztók (esetemben elektromos autótöltők) között úgy, hogy a hálózat fizikai védelme ne oldjon le, így a felhasználható áramon belül maradva maximalizáljuk a lehetőségeket.

Az elosztási algoritmus ismertetéséhez, szükséges definiálni a rendszerparamétereket, amik a vezérlő működésének alapját képzik. A rendszerben két védelem stalálható:

- Fizikai korlát (`BREAKER_MAX_TOTAL`)
- Allokációs keret (`ALLOC_MAX_TOTAL`)

A bemutatott *max–min fair* algoritmus fő feladata tehát az, hogy a fogyasztók igényeit úgy elégítse ki, hogy összegük soha ne lépje át a szoftveres `ALLOC_MAX_TOTAL` keretet.

5.2.1.2. Motiváció és cél

A szimulált fogyasztók áramigénye (d_i) időben változik. Adott egy globális, maximum áramérték $B =$ `ALLOC_MAX_TOTAL` amperben, ennél a tényleges összáram nem lehet nagyobb. A cél egy olyan kiosztás a_i meghatározása, amely (i) nem lépi túl az egyes igényeket ($0 \le a_i \le d_i$), (ii) a teljes kereten belül marad ($\sum_i a_i \le B$), (iii) és *fair* a kis igényűekkel szemben, azaz a kis igények teljesülnek először, a fennmaradó kapacitás pedig egyenlő alapról oszlik meg.

5.2.1.3. Definíció (max–min fair)

Egy $a = (a_1, \ldots, a_n)$ kiosztás *max–min fair*, ha bármely más megengedett y esetén, ha létezik i úgy, hogy $y_i > a_i$, akkor létezik j olyan, hogy $a_j \le a_i$ és $y_j < a_j$. Intuíció: csak a *már kisebb* részesedésűek rovására lehet növelni bárki juttatását. [2]

5.2.1.4. Feltöltés (water-filling)

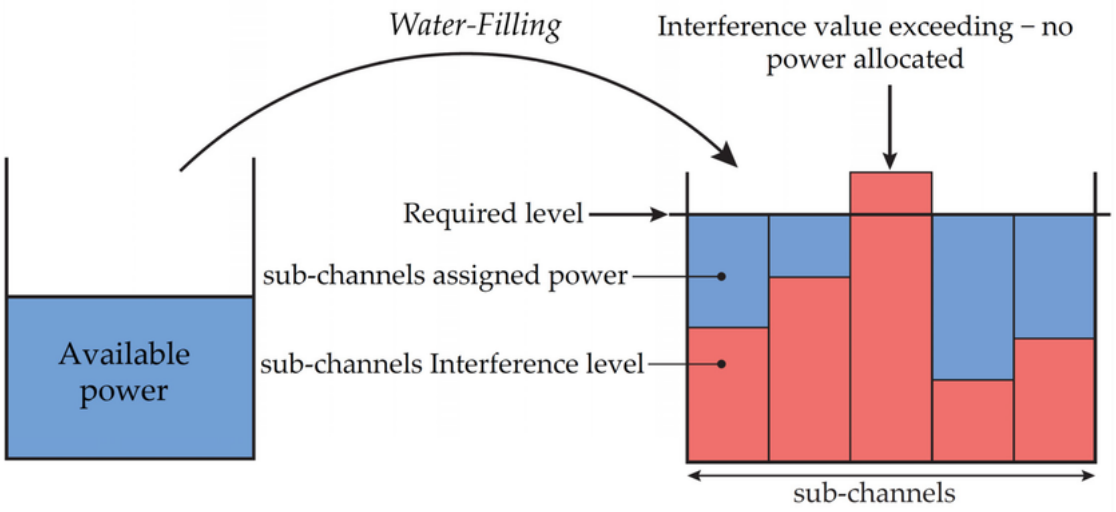
A max–min fair kiosztás felírható egyetlen paraméterrel:

$$a_i = \min\{\, d_i,\ \lambda \,\}, \qquad \text{ahol} \ \sum_{i=1}^{n} \min\{d_i, \lambda\} \ = \ B. \tag{5.1}$$

A λ *vízszint* úgy választandó, hogy a keret pont kiteljen (vagy ha $\sum_i d_i < B$, akkor $\lambda \ge \max_i d_i$, vagyis nincs korlát).

5.2.1.5. Algoritmus és bonyolultság

Gyakorlati, determinisztikus eljárás (progresszív töltés):



5.1. ábra. Water-filling elve telekomunikációban. [24]

1. Rendezzük az igényeket növekvő sorrendbe: $d_{(1)} \leq \cdots \leq d_{(n)}$.
2. Iteráljuk $k = 1..n$: feltételezzük, hogy az első k igény teljesül ($a_{(i)} = d_{(i)}$, $i \leq k$), és a maradék $B_k = B - \sum_{i=1}^{k} d_{(i)}$ egyenlő szinten oszlik meg a még nyitott $n - k$ elemre. A jelölt vízszint: $\lambda_k = B_k/(n - k)$.
3. Ha $\lambda_k \leq d_{(k+1)}$, megtaláltuk a vízszintet: az összes hátralévő $a_{(i)} = \lambda_k$ (és a korábbiak $d_{(i)}$).
4. Ha minden $d_{(i)}$ teljesül és még marad keret, akkor nincs korlátozás: $a_i = d_i$.

A rendezés miatt az időbonyolultság $O(n \log n)$. A megvalósított vezérlőben egy ekvivalens, iteratív *progresszív* algoritmus fut, amely kis elemszámon szintén gyors és stabil.

5.2.1.6. Tulajdonságok

- **Egyenlő szint elve:** a λ alatti igények teljes, a λ felettiek λ-ig kapnak. Így a kis igényűek sosem szenvednek hátrányt.
- **Monotonitás:** ha a keret B nő, akkor λ nem csökken, és senki kiosztása nem csökken.
- **Határhelyzetek:** ha $\sum_i d_i \leq B \rightarrow$ nincs cap (végtelen korlát). Ha $B = 0 \rightarrow$ minden $a_i = 0$.

5.2.2. A vezérlőben alkalmazott megvalósítás

5.2.2.1. Kapcsolat a rendszer komponenseivel

A vezérlő igényekből (`raw_current`) számolja a limiteket a fenti elv szerint a `ALLOC_MAX_TOTAL` kereten. A *megszakító* (breaker) logika ettől független, a *mért, tényleges* áramhoz viszonyít (`BREAKER_MAX_TOTAL`, `BREAKER_MIN_TOTAL`) biztonsági rétegként.

5.2.2.2. Példák

Klasszikus példa. $d = [10, 10, 100]$, $B = 90 \Rightarrow a = [10, 10, 70]$ (a két kicsi teljesül, a maradék egy szinten oszlik meg).

Vegyes igények. $d = [3, 8, 8, 20]$, $B = 25 \Rightarrow$ rendezve az első igény (3) teljesül, a maradék 22 három felé oszlik: $a = [3,\ 7.33,\ 7.33,\ 7.33]$ A.

5.2.2.3. Implementációs részletek

A limitek csak $\pm 10^{-3}$ A változás felett frissülnek a fogyasztók felé (zajcsillapítás), a „nincs korlát" állapotot nagy `INF_CAP` érték reprezentálja. Ha a nyers igény összeg a keret alá esik, a limitek feloldódnak.

5.2.2.4. Alternatív allokációs stratégiák

A globális keret elosztására több lehetőség is fennáll. A megfelelő algoritmus választása sokat tud javítani a rendszer hatékonyságán, ezért nagyon fontos. A `max-min fair` elv mellett kettő alternatívát találtam.

Proporcionális elosztás. Ennél a megközelítésnél minden fogyasztó a teljes igény ($\sum_j d_j$) arányában részesül a rendelkezésre álló keretből, amennyiben $\sum_j d_j > B | B = \textit{Globálkeret}$. A kiosztás képlete: $a_i = d_i \times (B / \sum_j d_j)$.

- *Példa:* Ha a keret $B = 90$ A, és az igények $d = [50, 50, 100]$, a teljes igény itt ekkor, $\sum d_j = 200$ A. Az arányos kiosztás $a \approx [22.5, 22.5, 45]$ A lenne.
- *Előnye:* Az egyik legegyszerűbb algoritmus.
- *Hátrány:* Ez a stratégia a "nagy" fogyasztóknak kedvez, és a kis igényűeket "éhezteti" (starvation).

Prioritásos (súlyozott) elosztás. Ebben az esetben minden fogyasztó kap w_i prioritási szintet (súlyt). Ha túlterhelés lépne fel a rendszer először a magasabb prioritású fogyasztókat elégíti ki, és a maradék keretből kapnak az alacsonyabb prioritású fogyasztók.

- *Előny:* Lehetővé teszi a "prémium" ügyfél fogalmát pl.: adott emberek autótöltője mindig kap áramot.
- *Hátrány:* Szintén bevezeti az "éheztetés" (starvation) problémáját, ahol egy alacsony prioritású végpont akár soha nem kap erőforrást. A rendszer komplexitása nő, és a konfiguráció nehézkessé válik.

A választás indoklása. Az alternatívákkal szemben azért esett a választás a **max–min fair** elosztásra mert ez biztosítja a legjobb egyensúlyt az egyszerűség és az igazságosság között. Motivációk között megfogalmazott pontokat teljesíti:

1. *Védi a kis fogyasztókat:* Garantálja, hogy aki keveset kér, az megkapja, amíg a keret ezt egyáltalán lehetővé teszi. Ezzel elkerüli az arányos elosztás fő problémáját az "éheztetést".

2. *Igazságos a nagy fogyasztók között:* A fennmaradó kapacitást egyenlően osztja szét a nagy igényű fogyasztók között, anélkül, hogy bonyolult prioritási problémákat kéne megoldani.

3. *Determinisztikus és stabil:* Az algoritmus viselkedése kiszámítható, és könnyen implementálható.

A tárgyalt tulajdonságok miatt esett a választás a Max–Min fair elvre.

5.3. Vezérlési késleltetés és rendszer-reakcióidő

5.3.1. Bevezetés: A reakcióidő kritikus szerepe

A dolgozatban bemutatott keretrendszer nem csak egy statikus eszköz, hanem egy aktív beleszól a rendszer elemek működésébe Ezentúl vannak egyéb fizikai eszközök amik a felügyeleti rendszertől függetlenül üzemelnek.

Energetikai rendszerek felügyeleténél a reakcióidő kritikus tényező, mert egy zárlat esetén rövid idő alatt nagyon nagy energiák szabadulnak fel. A keretrendszer célja, hogy a szoftveres energia allokációval a teljes keretet(`ALLOC_MAX_TOTAL`) úgy ossza fel, hogy megelőzze a fizikai védelmi eszközök (megszakítók) leoldását. A rendszer versenyt fut az idővel: a szoftvernek gyorsabban kell beavatkoznia, mint ami a fizikai megszakító védelmén van beállítva leoldási időnek. Ha a késleltetés túl nagy, a megszakítók leoldanak, mielőtt a szoftveres korlátozás érvénybe léphetne.

5.3.2. A vezérlés komponenseinek késleltetése

Ahhoz, hogy a teljes rendszer reakcióideét (T_{total}) megértsük, fel kell bontanunk a teljes vezérlési hurkot az egyes komponensek által hozzáadott késleltetésekre. Egy túlterhelés észlelése és kezelése a következő lépésekből áll:

1. **Mérés ($T_{measure}$):** Az `ESP8266` árammérő szenzora (pl. SCT-013) fizikailag megméri az áramot. Ez kb. valós időben történik, ez a késleltetés elhanyagolható.

2. **Adat küldés ($T_{publish}$):** Az `ESP8266` frissíti a mért értéket és elérhetővé teszi azt a `/metrics` végponton. Ez elég szintén elég gyors, hogy valós idejűnek nevezzük .

3. **Adatgyűjtés (T_{scrape}):** A Prometheus szerver a kontrol szerveren keresztül, gyűjti az adatokat. Mivel a kontrol szerverrel szinkronben kérdeznek, ezért szükséges mindekettőnek a konfigurációjában (`scrape_interval`) változót előre és egységesen meghatározni, ez időközönként lekérdezi a mérő végpontjának a `/metrics` URL-jét. *Ez a rendszer egyik legjelentősebb késleltetési tényezője.* Ha egy tipikus 5 másodperces időközről beszélünk a mérés és a Prometheus általi észlelés között, átlagosan 2.5, rosszabb esetben 5 másodperc is eltelhet.

4. **Adatfeldolgozás (T_{query}):** A Kontroll Szerver lekérdezi az adatbázisból az áramokat. A lekérdezés gyors, viszont ez a lépés is a Prometheus belső frissítési ciklusától függ.

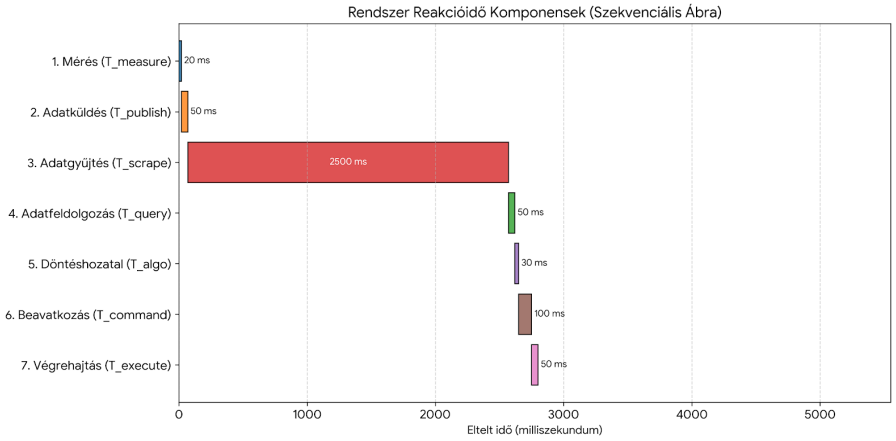
5. **Döntéshozatal (T_{algo}):** A Python szerver az adatokra lefuttatja a `max-min fair` algoritmust. Az elosztást tagalaló fejezetben láttuk, hogy ennek bonyolultsága $O(n \log n)$, ez kis ($n = 3..10$) fogyasztószám esetén elhanyagolható.

6. **Beavatkozás ($T_{command}$):** A szerver a számolás után korlátokat küld a végpontoknak. Itt egyedül a hálózati késleltetések lépnek fel (Wi-Fi, LAN).

7. **Végrehajtás ($T_{execute}$):** Az `ESP8266` fogadja a parancsot és a Modbus-os fizikai eszközök beállítják az autótöltő vagy egyéb eszköz maximális áramát. Ez is egy gyors műveletnek számít.

Így jön ki, hogy a következő lesz a teljes késleltetés:

$$T_{total} = T_{measure} + T_{publish} + T_{scrape} + T_{query} + T_{algo} + T_{command} + T_{execute} \tag{5.2}$$



5.2. ábra. késleltetés eloszlása

5.3.3. A fő késleltetési tényezők

Az előző pont lánca jól mutatja, hogy a késleltetést nem az algoritmus limitálja, hanem az adatgyűjtés architektúrája.

A teljes késleltetés nagy részét (T_{total}) kettő tényező dominálja:

1. **Prometheus Scrape Interval (T_{scrape}):** Ez a rendszer "szívverése". A vezérlőhurok nem tud gyorsabban reagálni, mint amilyen sűrűn friss adathoz jut. Ha ez az érték 5 másodperc, a rendszer fizikailag "vakon" repül 5 másodpercig, és csak utólag értesül a korábbi eseményekről.

2. **Hálózati késleltetés ($T_{command}$):** A Wi-Fi-alapú kommunikáció további, változó hosszúságú (jitter) késleltetést vihet a rendszerbe, terhelt vagy zajos hálózati környezetben, ez különösen problémás hiszen nehéz számolni a dinamikusan változó késleltetéssel.

Míg a hálózati késleltetés általában tíz-száz milliszekundum nagyságrendű, a `scrape_interval` másodperces nagyságrendű alap beállítás szerint. Viszont ezt lecsökkenthetjük akár 100ms-re is, ez növeli a hálózati forgalmat és a szerver terhelését,

de jelentősen csökkenti a teljes késleltetést. Így beállítható a megszakítók védelmi szintje ennél nagyobb értékre, például 500ms-re. Így már nem kell aggódnunk, hogy a késleltetés miatt a fizikai védelem leold.

5.3.4. Védelmi beállítások késleltetések alapján

A késleltetés számítása után lehetséges beállítani a védelem értékeit. A rendszerben kettő kvázi független védelmi szint létezik:

- **1. Szoftveres Védelem (Allokáció):** Ezt a `ALLOC_MAX_TOTAL` kerettel tudjuk beállítani. Célja a *komfort* és a *hatékonyság* biztosítása, a keretek a felhasználható teljesítmény folyamatos kihasználása és a fizikai leoldások *megelőzése*. Reakcióideje megfelelően pár száz milliszekundum (T_{total}).
- **2. Fizikai Védelem (Megszakító):** Ezt a megszakító védelmének fizikai beállítása határozza meg. Célja a hálózat fizikai védelme. Reakcióideje azonnali viszont a védelemben késleltetést állítunk be, amit körülbelül a szoftveres érték kétszeresére állítunk.

A rendszer stabil működésének feltétele, hogy e két szint megfelelően legyen összehangolva, a kétszeres érték valószínűleg egy megfelelő mértékű eltávolítás, ami enged helyet még a jitter-nek és egyéb hibáknak a vezeték nélküli hálózaton.

Konfigurációs példa. Tegyük fel, hogy:

- A szoftveres keret `ALLOC_MAX_TOTAL` $= 90\,A$.
- A Prometheus `scrape_interval` és a teljes hurok késleltetése $T_{total} = \mathbf{200\,ms}$.
- A csatlakoztatott eszközök (pl. autótöltők) képesek 200 milliszekundum alatt további 20 A terhelést bekapcsolni (pl. egy új autó csatlakozik).
- Ebben az esetben a fizikai megszakító védelmi beállítása legyen legalább $T_{total} = \mathbf{400\,ms}$.

Ebben az esetben az esetben megfelelő a kettő késleltetés távolsága, hogy ne történjen véletlen leoldás.

6. fejezet

Kubernetes integráció

A konténerizáció nagy előnyt nyújt, mivel szabványosított, elszigetelt környezetet kínál a szoftverek futtatásához. A Docker Compose elterjedt a helyi, több konténert tartalmazó alkalmazásokhoz, egyszerűsítve az összekapcsolt szolgáltatások definiálását és futtatását. Mivel azonban sokszor skálázódásra van szükség, és olyan funkciókra, mint a nagy rendelkezésre állás, az automatikus skálázás és a kifinomult orkesztráció, a Kubernetes vált a konténer orkesztráció szabványává.

Ebben a fejezetben megmutatom, hogy az eredetileg a Docker Compose segítségével definiált rendszeremet, hogyan migráltam Kubernetes környezetbe. A rendszeremben a már meglévő szolgáltatások jelenek meg, mint a Prometheus a felügyelethez, a Grafana a vizualizációhoz, több szimulátorszolgáltatás és egy vezérlőszerver. Itt bemutatom a Docker Compose konfigurációk Kubernetes manifesztekbe való átforgatásának kihívásait.

6.1. A Docker Compose és Kubernetes áttekintése

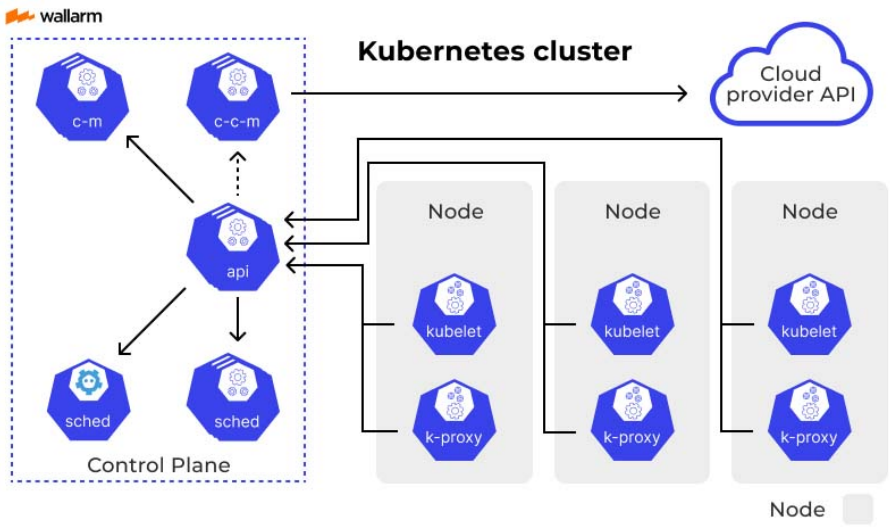
Docker Compose Ezzel több konténert tartalmazó Docker alkalmazásokat lehet definiálni és futtatni. Konfigurációja egy YML fájlban tárolt, ahol a szolgáltatásokat, hálózatokati kapcsolatokat, köteteket és függőségeket lehet megadni. A Docker Compose leegyszerűsíti a konténerek egyetlen hoszton történő orkesztrációját, így segíti a fejlesztést és tesztelést.

Kubernetes A Kubernetes (K8) viszont egy robusztus, open source platform a konténerek telepítésének, skálázásának és üzemeltetésének automatizálására hostokon. A Kubernetes új absztrakciókat vezet be:

- **Pod:** A Kubernetes legkisebb futtatható egysége, amely egy vagy több konténert foglal magában. [14]
- **Deployment:** Az alkalmazás skálázását és frissítését kezelő vezérlőobjektum. [12]
- **Service:** Stabil hálózati végpontot és terheléselosztást biztosít a dinamikus Podok számára. [16]
- **ConfigMap és Secret:** A konfigurációs adatok és a bizalmas információk (pl. jelszavak) tárolására szolgáló elemek. [11] [15]

- **PersistentVolumeClaim (PVC):** Tartós tárhely igénylésére szolgáló absztrakció, amely függetleníti az adattárolást a hardvertől. [13]

A migráció során ezeket képeztem le docker-ből k8-ba.



6.1. ábra. Kubernetes architektúra [1]

6.2. Rendszerarchitektúra

A rendszer a már megismert következő részeket tartalmazza:

- **Prometheus:** Egyéni prometheus.yml fájllal konfigurált idősoros adatbázis. Ennek szerencsétlensége, hogy az újra konfiguráció csak újra indítással lehetséges.
- **Grafana:** Vizualizációs eszköz, ami közvetlen a Prometheushoz kapcsolódik megjelenítéséhez.
- **ESP8266 szimulátorok:** Itt épen három példány szimulálja a különböző szimulátorazonosítókkal rendelkező eszközöket.
- **Breaker Simulators:** Más jellegű, de hasonló célú szimulátor.
- **Vezérlőszerver:** Lebonyolítja az eszközök közötti interakciókat, vezérlést és adatok továbbítását.
- **System Simulator:** A rendszer általános viselkedését emuláló központi szolgáltatás.

A Docker Compose alkalmazásban ezek az összetevők hálózaton és socketeken keresztül kapcsolódtak össze, és meghatározott végpontokon jelenítettek meg. [3]

6.3. A Docker Compose beállítások konvertálása Kubernetes manifesztekké

A Docker Compose-ról a Kubernetesre való áttérés magában foglalja az alkalmazás architektúrájának újragondolását a podok, deployement-ek, szolgáltatások és más Kubernetes objektumok szerint. [28]

6.3.1. Névtér- és konfigurációkezelés

Itt létrehoztam egy névteret (pl. monitoring) ez izolációt biztosít az alkalmazás számára. A ConfigMap a Prometheus konfiguráció tárolására szolgál (a prometheus.yml tartalma), lehetővé téve a konfiguráció frissítését a konténerek image-einek újbóli legenerálása nélkül.

```
    apiVersion: v1
kind: Namespace
metadata:
  name: monitoring
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |-
    global:
      scrape_interval: 15s
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets: ['localhost:9090']
```

6.3.2. Deployment-ek és Service-ek

Minden szolgáltatás Docker Compose-ban egy Deployment és egy Service formájában jelenik meg a Kubernetesben. A Deployment kezeli az alkalmazásban a podokat, a Service ezeket a podokat teszi elérhetővé.

Például a Prometheus szolgáltatás egyetlen replikával rendelkezik. Konfigurációja a ConfigMap-ról van mountolva, a perzisztens adatai pedig egy PersistentVolumeClaim (PVC) segítségével tárolom. Hasonlóképpen, más szolgáltatások, például az ESP8266 szimulátorok és a vezérlő szerver deployement-ekké alakulnak át, amelyek környezeti változókat és portkonfigurációkat adnak meg.

6.3.3. Perzisztens tárolók kezelése

A Docker Compose-ban gyakran definiálnak volume-okat az adattárolására. A Kubernetesben ezt a PersistentVolumeClaims biztosítja. A készített rendszeremben a Prometheus, mind a Grafana perzisztens tárolót igényelt az adatok megőrzéséhez, amiket a PVC-k létrehozásával és konténerekhez kötésével értem el.

```
  apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: grafana-data
  namespace: monitoring
spec:
```

```
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

6.3.4. Szolgáltatások elérhetővé tétele és hálózati konfiguráció

A Docker Compose-ban a portok hozzárendelését a konfigurációban végezzük. A Kubernetesben a portok meghatározást a Service-ek kezelik, ezek lehetnek NodePort típusúak a külső hozzáféréshez vagy ClusterIP típusúak a belső kommunikációhoz. A migráció során a konténerek portjait le kellett képezni a hosztokra, hogy a külső interfész ugyanaz maradjon az eredeti Docker Compose-hoz képest.

Például a Docker Compose-ban az 5000-es porton található vezérlő szervert egy olyan Kubernetes Service replikálja, amely egy adott NodePort-ot rendel hozzá, például 30050-et.

6.3.5. Telepítés és tesztelés

A Kubernetes manifeszt a kubectl apply -f paranccsal kerül alkalmazásra. Ez telepíti az összes komponenst a névtérben. A telepítés után a szabványos Kubernetes-parancsok (pl. kubectl get pods, kubectl logs, kubectl describe) a podok állapotának ellenőrzésére szolgálnak. Így iteratívan lehet tesztelni az új rendszert és később szolgáltatás kimaradás nélkül frissíteni.

Telepítéséhez a következő parancsot használjuk:

```
kubectl apply -f monitoring.yaml
```

És hogy megvizsgáljuk a telepített podokat:

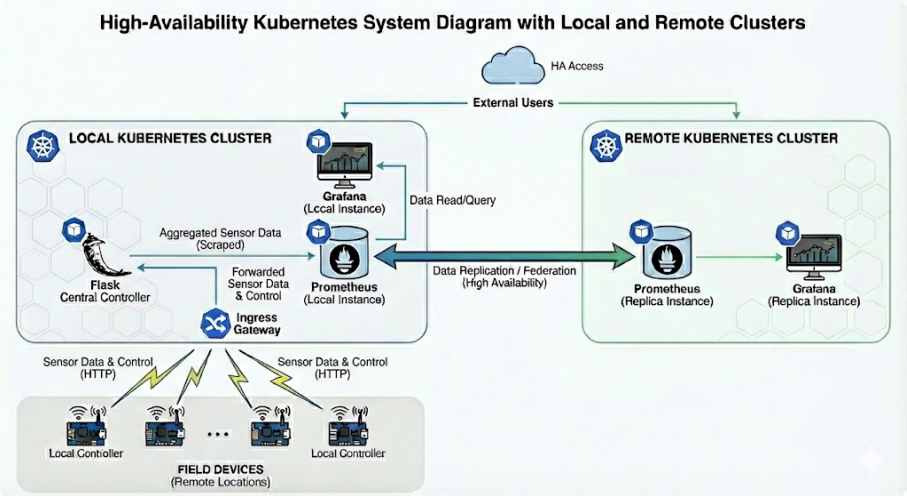
```
kubectl get pods -n monitoring
```

6.4. Nagy elérhetőségű rendszer implementációja

Az aktív elsődleges és passzív készenléti minta csökkenti a komplexitást és emellett megbízható felügyeletet biztosít:

- A Prometheus folyamatos adatreprodukciója mindent megőriz a második helyszínen.
- A replikán keresztül biztosított a Grafana-B azonnali használhatósága.
- Az átállást csak a DNS/szolgáltatás frissítési sebessége korlátozza.
- Ez a topológia megfelel a megbízhatósági céloknak a monitorozási környezetbe.

A Prometheus-A minden célpontot lekérdez, és elvégzi az összes értékelést. A Prometheus-B távoli írást kap A-tól (A hálózat felesleges terhelésének elkerülése érdekében nem scrapel közvetlen). A Grafana-B csatlakozik a Prometheus-B-hez, és a dashboardokat inen frissíti (ez közvetlenül nem érhető el). Egyetlen DNS név mutat az A ingressre. A Kubernetes és egy külső állapotellenőrzés frissíti a DNS-t a B oldalra, amikor az A leáll.



6.2. ábra. Hibrid kubernetes topológia

6.4.1. Replikák megvalósítása

A VM-n a prometheus konfigurációja a következő képen történik.

```
  apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus-b
  namespace: monitoring
spec:
  replicas: 1
  selector: { matchLabels: { app: prometheus-b } }
  template:
    metadata: { labels: { app: prometheus-b } }
    spec:
      nodeSelector: { site: "b" }
      containers:
      - name: prometheus
        image: prom/prometheus:v2.49
        args:
          - --config.file=/etc/prometheus/prometheus.yml
          - --web.enable-lifecycle
        volumeMounts:
          - name: data
            mountPath: /prometheus
        readinessProbe: { httpGet: { path: /-/ready, port: 9090 } }
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: prometheus-b-data
---
kind: PersistentVolumeClaim
metadata:
  name: prometheus-b-data
  namespace: monitoring
spec:
  accessModes: [ReadWriteOnce]
  storageClassName: cloud-ssd
  resources: { requests: { storage: 50Gi } }
```

Az eredeti A prometheus-ból pedig a B-be folyamatosan írunk.

```
  remote_write:
- url: http://prometheus-b.monitoring.svc.cluster.local:9090/api/v1/write
  queue_config:
    capacity: 10000
    max_shards: 5
    max_samples_per_send: 1000
```

```
  batch_send_deadline: 5s
```

A grafana megvalósítása során igazából csak egy ugyanolyan deployement-et hozunk létre. Ez egy másolat a másikról amire ha kell áttudunk bármikor térni.

```
spec:
replicas: 1
template:
  metadata: { labels: { app: grafana-b } }
  spec:
    nodeSelector: { site: "b" }
    containers:
    - name: grafana
      image: grafana/grafana:11.0.0
      env:
        - name: GF_DATABASE_URL     # same secret as primary
          valueFrom: { secretKeyRef: { name: grafana-db, key: db_url } }
        - name: GF_SECURITY_SECRET_KEY
          valueFrom: { secretKeyRef: { name: grafana-db, key: secret } }
      readinessProbe:
        httpGet: { path: /api/health, port: 3000 }
```

6.4.2. KubeADM

A projektemben a kubeadm-re támaszkodtam, hogy kubernetes klasztert készítsek a linux vm-et bevonva. Ez megkönnyítette a folyamatot mert magasabb színtű tervezésre volt csak szükség és ez megoldotta magától az alacsonyabb szintű problémákat.

```
sudo kubeadm init --config=/etc/kubeadm/config.yaml
```

Az inicializálás után csak egy tokent kellett adni a nodenak, hogy csatlakozzon a clusterhez. Ezután a a további folyamatokat kezelte is a Kubeadm.

```
sudo kubeadm join 10.200.0.1:6443 \
  --token <token> \
  --discovery-token-ca-cert-hash sha256:<hash>
```

Ennek köszönhetően egy hasonló rendszerben, ha a egy node meghibásodik akkor a másik átveszi a helyét és felhasználói oldalról nem érzünk kiesést. A helyre állítás során, pedig csak egy parancsot kell kiadnunk:

```
kubeadm join
```

Ezután újra csatlakoztattuk is a node-ot és újonnan felépíthetjük a clusterben.

7. fejezet

Szöveges interfészek a szimulációhoz

A rendszer működésének igazolásához szükséges egy tesztkörnyezet, ami képes a valós hardver eszközök viselkedését szoftveresen emulálni. A fizikai tesztelés önmagában nem fedné le a szélsőséges terhelési eseteket (például a túlterhelését), és nem tenné lehetővé a hosszú távú, reprodukálható vizsgálatokat.

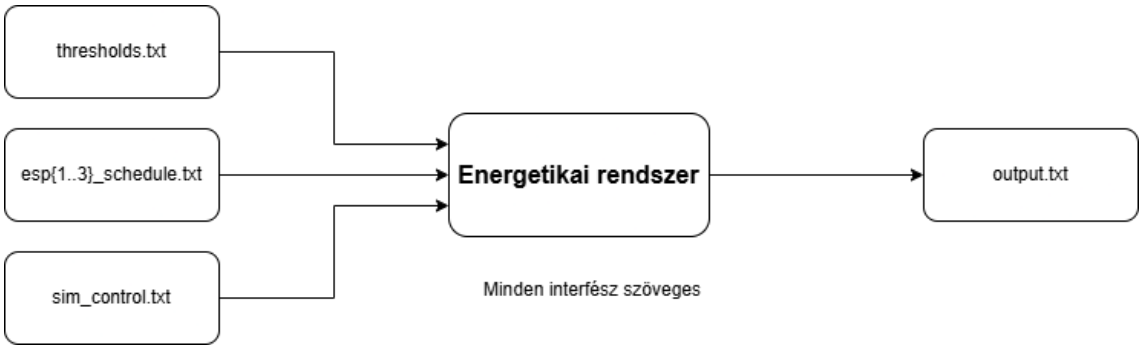
7.1. Cél és áttekintés

A szimulációs alrendszer vezérlése és adatgyűjtése kizárólag szöveges interfészeken , gyakorlatban .txt fájlokon alapul.

Ezek biztosítják a kapcsolatot a felhasználó és a futó környezet között: a bemeneti fájlok definiálják a teszteseteket, míg a kimeneti fájlok rögzítik az adatokat.

A cél, hogy a szimuláció kiegészítő eszközök (pl. CSV-konverzió) nélkül, egyszerű szövegszerkesztővel kiértékelhető legyen.
Rövid összefoglaló:

- **Bemenetek:**
  - `thresholds.txt` - itt találhatóak meg a maximum és minimum értékek, amit elérhet különböző pontjain a rendszer.
  - `esp{1..3}_schedule.txt` - ebben találhatóak meg a mérőpontok napirendjei.
  - `sim_control.txt` - itt találhatóak meg a futtatási állapotok parancsai.
- **Kimenet/napló:** `output.txt` (idősoros; egy sor = egy vezérlési ciklus) Ebben található meg az összes lényeges mérőszám és állapot minden ciklusban.
- **Webes felület:** „Dev Panel” (localhost:8080) a fájlok szerkesztéséhez, generálásához, letöltéséhez, a futás indításához/megállításához, az idő nullázásához és a napló törléséhez használható.



7.1. ábra. Interfészek

7.2. Bemeneti szövegfájlok

A vezérlő paramétereket és a szimulált környezetet három szöveges állomány határozza meg. Ezeket a fájlokat a rendszer ciklikusan olvassa, így a módosítások újraindítás nélkül érvényesülnek.

7.2.1. `thresholds.txt` – küszöbök és maximum megengedhető áram

Ez a fájl tartalmazza a globális határértékeket, amiket a vezérlő algoritmusnak be kell tartania. A fájl kulcs–érték párokat tartalmaz:

```
# Kuszobertekek a vezerlo szerverhez
BREAKER_MAX_TOTAL=65.0    # [A] - Megszakito lekapcsolasi aram
BREAKER_MIN_TOTAL=35.0    # [A] - Megszakito bekapcsolasi aram
ALLOC_MAX_TOTAL=95.0      # [A] - Max aram ertek
```

Feldolgozás módja: A kontroll Szerver minden vezérlési ciklus elején beolvassa ezt a fájlt. Ez lehetővé teszi, hogy szimuláció közben is változtathassuk a rendelkezésre álló áramkeretet (`ALLOC_MAX_TOTAL`), így tesztelve a szabályozó algoritmust.
Megjegyzések:

- A *megszakítók* (breakerek) logikája az *aktuálisan mért hatásos* összáramhoz viszonyít (`BREAKER_MAX_TOTAL`, `BREAKER_MIN_TOTAL`).
- A SIM-ekre küldött korlátok (cap) a *nyers igényekből* számítódnak *max-min fair* elv szerint, az `ALLOC_MAX_TOTAL` keret figyelembevételével.

7.2.2. `esp{x}_schedule.txt` – idősoros bemenet

Ezek a fájlok írják le a szimulált fogyasztók (pl. autótöltők) viselkedését, vagyis azt, hogy az idő függvényében mekkora áramigénnyel lépnek fel. Az adott másodpercben + kívánt áram (A). A menetrendben a legutóbbi időponthoz tartozó érték érvényes a következő megadásig. Itt fontos megemlíteni, a szimulációs vizsgálatok hatékonysága érdekében a környezetben bevezetésre került a *virtuális idő* fogalma. Ez a mechanizmus egy szoftveres óra segítségével leválasztja a lokális vezérlőket a valós fizikai időtől, így lehetővé teszi a hosszú távú folyamatok (pl. egész napos terhelési ciklusok) nagyságrendekkel gyorsabb, de adatkonzisztens lefuttatását. A virtuális órákat a kontrol szerver szinkronizálja.

```
# seconds  amps
0         1.0
30        2.5
120       0.8
```

Feldolgozás: Ezeket a fájlokat, az **ESP szimulátorok** olvassák be. A szimulátor a saját belső virtuális órájához igazodva hozza létre a terheléseket.

7.2.3. `sim_control.txt` – futtatási állapot

Ez a fájl a szimuláció globális főkapcsolója. Egyetlen szót tartalmazhat: `RUNNING` vagy `STOPPED` (Az alapértelmezés `STOPPED`).
Feldolgozás: Ez a fájl egy szinkronizációs pont az egész rendszer számára.

- A **Vezérlő** csak `RUNNING` állapotban futtatja az allokációs algoritmust és küld beavatkozó parancsokat.
- A **Szimulátorok** csak `RUNNING` állapotban léptetik a virtuális idejüket. Ha az állapot `STOPPED`-ra vált, a rendszer "lefagyasztja" a pillanatnyi állapotot, lehetővé téve a naplók és az állapotok statikus elemzését.

7.3. Kimeneti szövegfájl

7.3.1. `output.txt` – idősoros kimenet

A vezérlő minden ciklusban *egy sort* ír. A fájl alapértelmezetten *append-only* a véletlen szerkesztést elkerülendő; a Dev Panel „Clear output.txt” művelete törli amennyiben ez szükséges, és a vezérlő legközelebb automatikusan újra létrehozza a fejlécet.
Formátum: `kulcs=érték` párok szóközzel elválasztva.

```
# One record per line; fields are key=value separated by spaces
timestamp=1758199200 sim_state=RUNNING sum_current_amps=5.7 \
alloc_max_total_amps=6.0 max_total_amps=6.0 min_total_amps=1.0 \
sims=esp1:raw=2.0,effective=2.0,cap=2.0|esp2:raw=1.7,effective=1.7,
cap=2.0|esp3:raw=2.5,effective=2.0,cap=2.0 \
breakers=brk1:on,brk2:on
```

Kulcsok a kimeneti file-ban:

- `timestamp` – UNIX időpecsét (s).
- `sim_state` – globális állapot: `RUNNING`/`STOPPED`.
- `sum_current_amps` – mért hatásos összáram (cap után).
- `alloc_max_total_amps` – allokációs keret (A).
- `max_total_amps` / `min_total_amps` – breaker küszöbök (legacy nevek).
- `sims` – `|` jellel szeparált lista mérési pontonként (SIM-enként):
  `espX:raw=..., effective=..., cap=...`
  ahol `raw` = menetrendi igény, `effective` = tényleges áram, `cap` = küldött maximum.
- `breakers` – megszakítók állapota `on`/`off`, vesszővel elválasztva.

7.4. Időkezelés és futtatás

- **RUNNING (Aktív futás):** Ilyenkor a virtuális óra minden ciklusban előrelép. A szimulált végpontok (ESP-k) a t_{sim} időpillanat alapján keresik ki a menetrendjükből (`schedule.txt`) az aktuális áramigényt. A vezérlő szerver dolgozza fel az adatokat és számolja az allokációt, küldi ki az új korlátokat.
- **STOPPED (Felfüggesztés):** Ekkor a virtuális idő „befagy" ($t_{sim} = \text{const}$). A szimulátorok megálnak, tartják az utolsó beállított értéket. A vezérlő továbbra is mér és naplózza, de nem történik beavatkozás nem küld új korlátokat és nem kapcsol megszakítót.

7.4.1. A STOPPED állapot szerepe a vezérlésben

A rendszerben bevezetett `STOPPED` állapot a futás szüneteltetését szolgálja és a szimulációs környezet konzisztenciáját segíti. Bevezetésének kettő oka volt:

- **Állapot-befagyasztás:** Mivel a vezérlő felfüggeszti az aktív beavatkozást, a rendszer pillanatnyi belső állapotát lehet ellenőrizni statikus körülmények között.
- **Determinisztikus újraindítás:** Ez az állapot a kiindulópontja a `RESET` műveletnek is, biztosítva, hogy minden komponens pontosan ugyanabból a $t = 0$ időpillanatból induljon.

Ez a mechanizmus garantálja, hogy a tesztelés során a vezérlő szoftver viselkedése determinisztikus és bármikor reprodukálható legyen.

7.5. Reprodukálhatóság és feldolgozhatóság

A bemenetek (küszöbök, menetrendek, futtatási állapot) verziózhatók és mellékelhetők. A kimeneti `output.txt` önleíró; minden rekord tartalmazza az adott ciklus lényeges paramétereit. A formátum egyszerűen feldolgozható bármely nyelven (kulcs=érték párok; `sims` és `breakers` mezők jól definiált szeparátorokkal).

7.6. Rövid példa – beállítás → kimenet

thresholds.txt

```
BREAKER_MAX_TOTAL=9.0
BREAKER_MIN_TOTAL=2.0
ALLOC_MAX_TOTAL=9.0
```

esp1_schedule.txt

```
# seconds  amps
0  50
60 10
```

esp2_schedule.txt

```
0  50
60 10
```

esp3_schedule.txt

```
0  50
60 100
```

Várható kiosztás a 0–60 s szakaszban: mindhárom SIM korlátozott, mivel az igény 150 A $>$ 9 A. 60 s után az igények `[10, 10, 100]` $\Rightarrow$ kiosztás `[10, 10, 70]`. A `cap` és az `effective` értékek ennek megfelelően jelennek meg az `output.txt`-ben.

8. fejezet

Fejlesztői panel (Dev Panel)

8.1. Cél és szerep

A Dev Panel egy könnyű használatú webes felület, amely a szöveges bemenetek és kimenetek kezelését, a futtatás indítását/megállítását, az idő nullázását és a napló törlését teszi lehetővé. Célja a *gyors kísérletezés* és a *reprodukálható* tesztfutások támogatása külön eszközök nélkül.

8.2. Architektúra áttekintése

A panel egy Flask-alapú backendből (`/api/*`) és statikus frontendből (HTML+CSS+JS) áll. A backend közvetlenül a `./data` mappában található fájlokat kezeli, és hálózaton hívja az esp-t szimuláló konténerek végpontjait. A vezérlő külön, a saját portján (8000) fut; a Prometheus és Grafana eléréséhez gyorslinkek állnak rendelkezésre.

8.3. Felhasználói felület és funkciók

A grafikus felület két fő funkcionális egységre tagolódik: a globális vezérlőkre (Simulation Control) és a tesztesetekre (Scenarios).

8.3.1. Simulation Control (Vezérlőpult)

Itt lehet a szimuláció globális állapotát menedzselni.

- **Start/Stop:** A gombok a `sim_control.txt` fájl írásával vezérlik a futtatást. `STOP` módban a szimulátorok virtuális ideje megáll, a vezérlő nem küld új parancsokat.
- **Reset (t=0):** A funkció leállítja a futást, és üzenettel nullázza minden szimulátor belső óráját, így a teszt biztosan nulláról indul.
- **Clear output:** Törli a naplófájl tartalmát, hogy a következő futás adatait ne keveredjenek a korábbiakkal.

- **Élő monitorozás:** A felület valós időben megjeleníti az aktuális összáramot és a rendszer státuszát.

8.3.2. Scenarios (Szcenárió-kezelő modul)

Lehetővé teszi a komplex tesztesetek egységes kezelését. A modul három fülre bomlik:

8.3.2.1. Presets (Előre definiált tesztek)

A gyakran használt tesztesetek (pl. statikus túlterhelés, hiszterézis-vizsgálat, dinamikus átrendeződés) beépített sablonként érhetők el.

A felhasználó egyetlen kattintással betöltheti és elindíthatja ezeket. Ez a funkció ("Auto Test Runner") gyorsítja a demonstrációt.

8.3.2.2. Builder (Szerkesztő és Generátor)

Ez a felület szolgál az egyedi tesztek összeállítására:

- **Menetrend-generátor:** Segítségével grafikusan állíthatóak be a terhelési görbék (konstans, rámpa, lépcső, szinusz, random walk). Ezek azonnal megjelennek a szerkesztőablakban.
- **Thresholds beállítás:** A globális korlátok (`ALLOC_MAX`, `BREAKER_MAX`) közvetlen szerkesztése.
- **Azonnali futtatás:** A "Run now" gombbal a beállított paraméterek mentésre kerülnek a fájlokba, és a szimuláció azonnal elindul az új értékekkel.

8.3.2.3. Files (Fájlkezelés)

A panel lehetőséget biztosít a teljes konfiguráció (menetrendek + határértékek + vezérlési opciók) JSON formátumú mentésére és visszatöltésére.

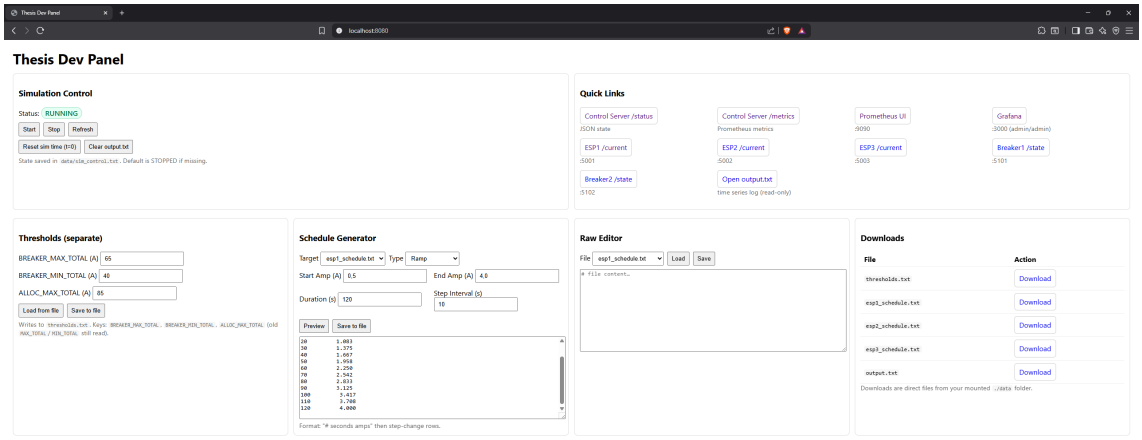
A "Saved Scenarios" listából a korábban elmentett tesztek bármikor újra előhívhatók, a "Downloads" szekció pedig lehetővé teszi a nyers bemeneti és kimeneti (`output.txt`) fájlok letöltését archiválás céljából.

8.4. Backend API interfész

A Dev Panel backendje az alábbi REST végpontokon keresztül érhető el, amelyeket a frontend vagy külső automatizációs szkriptek is használhatnak:

Végpont	Funkció
`GET /api/sim_state`	A globális állapot (RUNNING/STOPPED) lekérdezése
`POST /api/sim_state`	A futtatási állapot módosítása
`POST /api/reset_sim_time`	Minden szimulátor órájának nullázása
`POST /api/clear_output`	A naplófájl törlése
`POST /api/run_scenario`	Teljes teszteset (JSON) alkalmazása és indítása
`POST /api/save_scenario_json`	Aktuális beállítások mentése a szerverre
`GET /api/read /api/write`	Nyers fájlműveletek a `/data` mappában

8.1. táblázat. A Dev Panel legfontosabb API végpontjai



8.1. ábra. Devpanel

9. fejezet

Rendszertesztek és bemutató szcenáriók

9.1. Tesztelés módszertan

9.1.1. Tesztek megvalósítása

A cél itt annak igazolása volt, hogy a rendszer komponensei megfelelően működnek. A vizsgálat során *idősoros* bemeneti és kimeneti fájlt (`thresholds.txt`) használtam. Ebben az esetben a kontrollciklus periódusa $T_c = 3\,\mathrm{s}$.

Fontos hangsúlyozni, hogy a vizsgálatok során nem valós fizikai eszközökön (töltőkön) mértem áramot, hanem a szimulátorok által generált és a vezérlő-monitorozó rendszeren keresztül, mért értékeket elemeztem. Ez a megközelítés biztosította, hogy a vezérlés a valósághoz hű, de biztonságos és reprodukálható környezetben vizsgázhasson, ahol a "tényleges" érték a rendszer által érzékelt állapotot jelenti.
Mérőszámok és ellenőrzési pontok:

- **Mérőnkénti tényleges áram** (`effective`) ez nem az igényelt hanem a ténylegesen megkapott áramerősség, a vezérlő `/status` végpontján és az `output.txt`-ben.
- **Összáram** A Mérőnkénti tényleges áramok összege (`sum_current_amps`)
- **Korlátok (cap)**: az allokált teljesítmény a végpontokon (max–min fair) eredményei.
- **Küszöbök**:
  - `ALLOC_MAX_TOTAL` - Ez a teljes teljesítmény keret, amit a vezérlő ki tud osztani, a kiosztott áramok összege legfeljebb ennyi lehet.
  - `BREAKER_MAX_TOTAL` - A védelem kapcsolásának küszöbe, ha az összáram meghaladja ezt az értéket, a megszakítók lekapcsolnak (OFF).
  - `BREAKER_MIN_TOTAL` - A védelem automatikus visszakapcsolásának küszöbe, csak akkor kapcsol vissza (ON) a megszakító, ha az összáram ez alá csökken.

- **Megszakító állapot**: Itt csak `on/off` értéket figyelünk a védelmet ellátó megszakítókon.

Ezeket `output.txt` idősoros naplóban ellenőriztem, itt volt a legegyszerűbb, mert itt egy sor egy ciklus.

9.1.2. Várt viselkedés

1. Ha $\sum_i d_i \leq$ `ALLOC_MAX_TOTAL`: *nincs korlát*, ezért $\text{effective}_i = d_i$ minden mérőre és az összáram egyszerűen $\sum_i d_i$. Ilyenkor a vezérlő nem „oszt újra", a kiosztás megegyezik az igényekkel és a megszakító-logika csak akkor lép működésbe, ha az összáram véletlenül mégis átlépi a védelmi küszöböt.

2. Ha $\sum_i d_i >$ `ALLOC_MAX_TOTAL`: *max–min fair* elosztás lép életbe, vagyis egy λ szintet keresünk úgy, hogy $a_i = \min\{d_i, \lambda\}$ és $\sum_i a_i =$ `ALLOC_MAX_TOTAL`. Azok a mérők, amelyek igénye $d_i \leq \lambda$, teljes igényüket megkapják, a nagyobb igényűek pedig λ-nál „levágódnak", a vezérlő ezt 3 s-onként újraszámolja, így ha szabadul fel kapacitás ez automatikusan átcsoportosul.

3. Megszakító: ha az *összáram* sum $\geq$ `BREAKER_MAX_TOTAL`, a megszakító kikapcsol (védelmi leoldás) és csak akkor kapcsol vissza, ha sum $\leq$ `BREAKER_MIN_TOTAL`.

4. `STOPPED` állapotban a virtuális idő nem halad, a vezérlő nem küld új korlátokat és nem ad megszakító-parancsokat, ilyenkor a bemeneti fájlok szabadon szerkeszthetők, és a következő `RUNNING` ciklus kezdetekor az új konfiguráció lép életbe, ha ez be van kattintva időnullázással és naplóürítéssel.

9.2. Szcenáriók és elfogadási kritériumok

9.2.1. Alaptesztek: Start/Stop/Reset/Clear

Az első teszt a vezérlő és a szimulációs környezet alapvető funkcióinak (indítás, leállítás, újraindítás) működését validálja. Itt a cél annak ellenőrzése, hogy a rendszer alapállapotban stabil, a vezérlő parancsokra megfelelően reagál. A bemeneti paramétereket szándékosan úgy állítottam be, hogy ne lépjen fel semmilyen áramkorlátozás vagy megszakítási esemény, így itt kizárólag az alapvető vezérlés ellenőrzésén van a hangsúly.

Bemenetek:

- `BREAKER_MAX_TOTAL=12`
- `BREAKER_MIN_TOTAL=2`
- `ALLOC_MAX_TOTAL=30`
- ESP1=1,0 A
- ESP2=1,5 A

- ESP3=0,5 A

Miért ez a beállítás? Az igények összege 1,0 + 1,5 + 0,5 = 3,0 A $\ll$ `ALLOC_MAX_TOTAL`, ezért nem várható korlátozás: effective$_i = d_i$.

Lépések és jelentésük:

1. **STOP** — a `sim_control.txt` `STOPPED`-ra állítása, a vezérlő nem küld új korlátotokat és nem is kapcsol megszakítót.
2. **Reset** t=0 — minden szimulátor idejét nullázzuk, a szimuláció elejéről kezdünk.
3. **START** — a vezérlő elindul, a következő ciklusban kiírja az állapotot és beállítja a korlátokat, de ebben az estben nem kell.
4. *(Opcionális) Clear* `output.txt` törölhető a napló amennyiben tiszta fájlt szeretne valaki látni.

Várt rendszerállapot

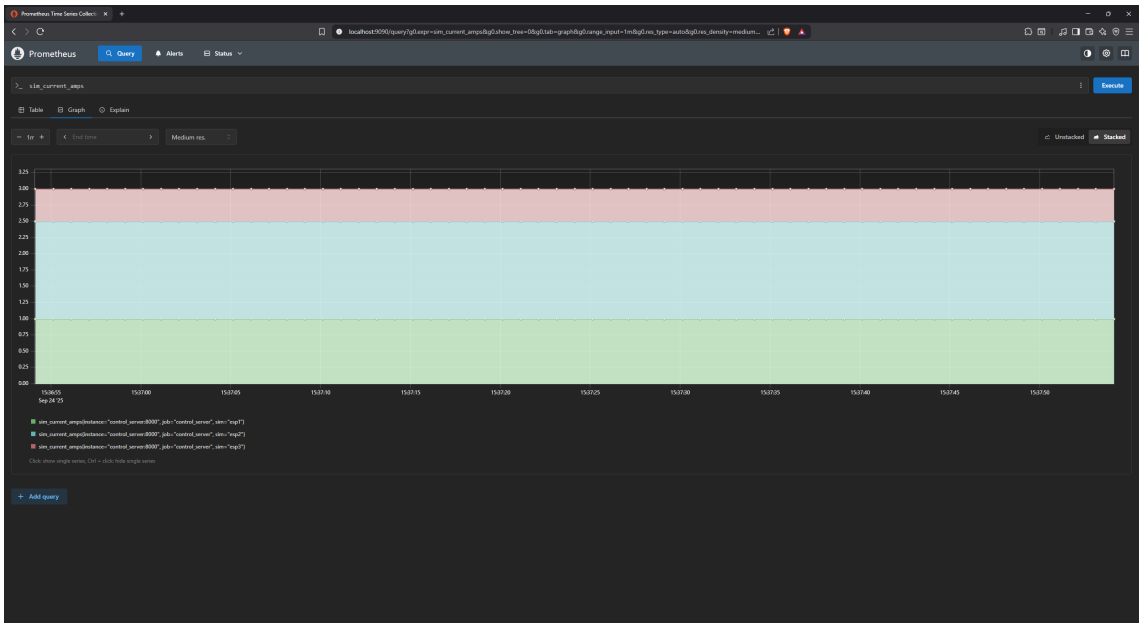
- *Nincs korlát*: effective$_{1,2,3}$ = {1,0; 1,5; 0,5} A, a korlátokat nagy értékek ($\sim 10^9$) jelölik.
- *Összáram*: `sum_current_amps` $\approx$ 3,0 A stabilan, ez így vízszintes vonal az élő grafikonon.
- *Megszakító*: bekapcsolt állapotban van, mert 3,0 < `BREAKER_MAX_TOTAL` = 12 és 3,0 > `BREAKER_MIN_TOTAL` = 2.

Hol ellenőrizhető?

- `output.txt`: 3 s-onként új sor, pl.:

```
sim_state=RUNNING sum_current_amps=3.0
sims=esp1:raw=1.0,effective=1.0,cap=1e9|esp2:raw=1.5,
effective=1.5,cap=1e9|esp3:raw=0.5,effective=0.5,cap=1e9
breakers=brk1:on,brk2:on
```

Siker kritérium: A grafikon 3 A változatlan értékű görbét mutat az `output.txt-ből` ugyanezt tudjuk kiolvasni, hogy effective$_i = d_i$, korlátok nincsenek érvényben, a megszakítók be vannak kapcsolva ez 1-2 ciklus (3-6 s) után stabilan látszik.



9.1. ábra. Alaptesztek

9.2.2. Alulterhelés: nincs korlátozás

Ebben a tesztben azt a normál körülmények között is előforduló esetet vizsgálom, amikor a fogyasztói igények összege ($\sum d_i$) kevesebb, mint a maximálisan engedélyezett teljes fogyasztható áram. Itt a cél annyi, ellenőrizni kell, hogy a vezérlő helyesen ismeri fel az alulterhelt állapotot, és nem aktivál semmilyen korlátozási mechanizmust. A várt működés, hogy minden fogyasztó a teljes igényelt áramát kapja meg (effective$_i$ = d_i), és a megszakítók is bekapcsolt állapotban maradnak, mivel a terhelés a megengedett sávon belül van.

Bemenetek:

- `ALLOC_MAX_TOTAL=6`
- `BREAKER_MAX_TOTAL=12`
- `BREAKER_MIN_TOTAL=2`
- ESP1=2,0 A
- ESP2=1,5 A
- ESP3=0,5 A

Miért ez a beállítás? Az igények összege 2,0 + 1,5 + 0,5 = 4,0 A $\leq$ `ALLOC_MAX_TOTAL` = 6, ezért *nem* indul korlátozás (max–min fair kiosztásra nincs szükség), így effective$_i$ = d_i. A 4,0 A az `BREAKER_MIN` és `BREAKER_MAX` között van, ezért a megszakítók *bekapcsolt* állapotban maradnak.

Lépések és jelentésük:

1. **START** — a vezérlő elindul, és kiírja az állapotot, mivel $\sum d_i \leq$ `ALLOC_MAX`, a korlátokat nem kell érvényesíteni.

2. **Várakozás $\sim$ 2 ciklus** — 6–7 s múlva a naplóban stabilan láthatóak a beállítások.

Várt rendszerállapot

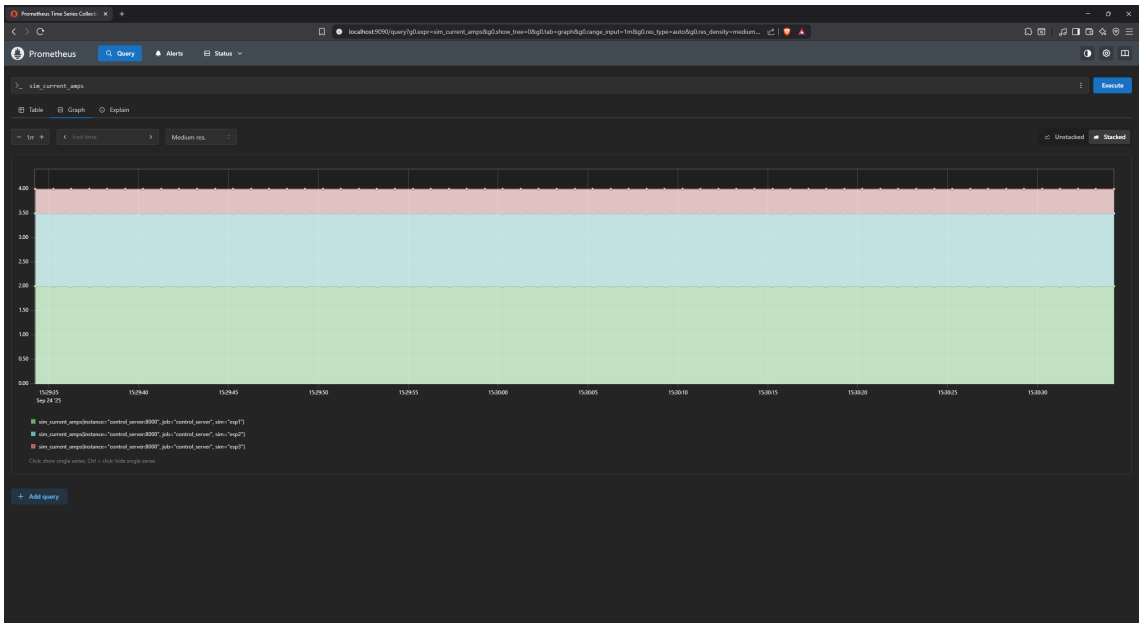
- *Nincs korlát*: $\text{effective}_{1,2,3} = \{2{,}0; 1{,}5; 0{,}5\}$ A, a korlátok nagy értékekkel ($\sim 10^9$) jelzik a „nincs limit” állapotot.
- *Összáram*: `sum_current_amps` $\approx 4{,}0$ A $\Rightarrow$ vízszintes vonal az élő grafikonon.
- *Megszakító*: bekapcsolva, mert $4{,}0 <$ `BREAKER_MAX_TOTAL` $= 12$ és $4{,}0 >$ `BREAKER_MIN_TOTAL` $= 2$.

Hol ellenőrizhető?

- `output.txt`: 3 s-onként új sor, pl.:

```
sim_state=RUNNING sum_current_amps=4.0
sims=esp1:raw=2.0,effective=2.0,cap=1e9|esp2:raw=1.5,
effective=1.5,cap=1e9|esp3:raw=0.5,effective=0.5,cap=1e9
breakers=brk1:on,brk2:on
```

Siker kritérium: az élő grafikon 4 A vízszintes görbét mutat, az `output.txt` egyezően jelzi, hogy $\text{effective}_i = d_i$, korlát nincs érvényben, a megszakítók be vannak kapcsolva mindez 1–2 ciklus (3–6 s) után stabilan látszik.



9.2. ábra. Alulterhelt eset

9.2.3. Túlterhelés, azonos igények: fair 3/3/3 allokáció

Itt jelentős túlterhelést vizsgálok, az összesített igény (150 A) ez nagyon meghaladja a globális allokációs keretet (9 A). A teszt célja, hogy ellenőrizze a "max-min fair"

elosztási algoritmus helyes működését. A forgatókönyv lényege, hogy minden fogyasztó azonos, a keretnél jóval magasabb igénnyel jelentkezik, így a várt eredmény egy tökéletesen egyenlő, 3 A-es elosztás a fogyasztók között.

Bemenetek:

- `ALLOC_MAX_TOTAL=9`
- `BREAKER_MAX_TOTAL=12`
- `BREAKER_MIN_TOTAL=2`
- ESP1=50 A
- ESP2=50 A
- ESP3=50 A

Miért ez a beállítás? Az igények összege $50 + 50 + 50 = 150$ A $\gg$ `ALLOC_MAX_TOTAL` $= 9$, ezért a max–min fair elosztás: egy közös λ szintet keresünk úgy, hogy $a_i = \min\{d_i, \lambda\}$ és $\sum a_i = 9$. Azonos igények mellett $\lambda = 9/3 = 3$ A, tehát minden mérő 3 A-t kap.

Lépések és jelentésük:

1. **START** — a vezérlő elindul, kiszámítja λ-t és beállítja a korlátokat.
2. **Várakozás $\sim$ 2 ciklus** — 6 s alatt a napló stabilan tükrözi a 3/3/3 kiosztást.

Várt rendszerállapot

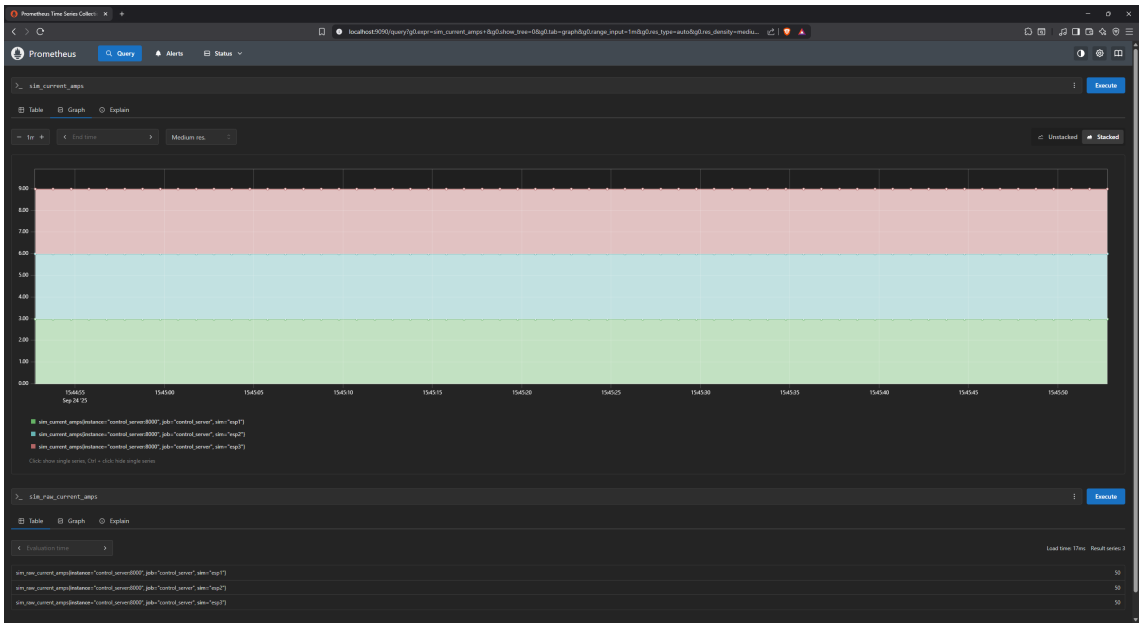
- *Korlátok*: mindhárom mérőnél `cap` = 3 A; $\text{effective}_{1,2,3} = \{3, 3, 3\}$ A.
- *Összáram*: `sum_current_amps` = 3+3+3 = 9 A $\Rightarrow$ egyenes vonal a grafikonon.
- *Megszakító*: bekapcsolva, mert $9 <$ `BREAKER_MAX_TOTAL` $= 12$ és $9 >$ `BREAKER_MIN_TOTAL` $= 2$.

Hol ellenőrizhető?

- `output.txt`: 3 s-onként új sor, pl.:

```
sim_state=RUNNING sum_current_amps=9.0
sims=esp1:raw=50.0,effective=3.0,cap=3.0|esp2:raw=50.0,
effective=3.0,cap=3.0|esp3:raw=50.0,effective=3.0,cap=3.0
breakers=brk1:on,brk2:on
```

Siker kritérium: az élő grafikon három, közel azonos (3 A) szintet és 9 A összáramot mutat, az `output.txt` `cap` = 3 A értéket jelez mindhárom mérőnél, a megszakítók bekapcsolva vannak mindez 2 ciklus (6 s) után stabil.



9.3. ábra. Túlterhelt eset

9.2.4. Dinamikus újraelosztás: a nagy felhasználó kap teret

Ez a teszt a dinamikus viselkedést vizsgálja. A cél annak demonstrálása, hogy a rendszer nemcsak statikus túlterhelést tud kezelni, hanem képes valós időben reagálni a fogyasztói igények drasztikus változására is. Ez a forgatókönyv azt modellezi, ahogy egyes fogyasztók igénye lecsökken, a felszabaduló kapacitást pedig a rendszer automatikusan és a "max-min fair" elvnek megfelelően (lásd 5.2. fejezet) újraosztja egy másik, nagy igénnyel jelentkező fogyasztóhoz, miközben az összáram végig a globális kereten belül marad.

Bemenetek:

- `ALLOC_MAX_TOTAL=90`
- `BREAKER_MAX_TOTAL=120`
- `BREAKER_MIN_TOTAL=10`
- Menetrendek:
  - ESP1: 0 s $\rightarrow$ 50 A, 60 s $\rightarrow$ 10 A
  - ESP2: 0 s $\rightarrow$ 50 A, 60 s $\rightarrow$ 10 A
  - ESP3: 0 s $\rightarrow$ 50 A, 60 s $\rightarrow$ 100 A

Miért ez a beállítás? Az elején az igények azonosak $(50, 50, 50$ A), ez túl nagy a 90 A kerethez képest, ezért *fair* elosztás lép életbe: $[30, 30, 30]$ A. Egy idő után két mérő visszaesik 10 A-ra, a harmadik 100 A-t kér; a felszabaduló 80 A-ből a keret kitöltéséhez a harmadik kap 70 A-t, így $[10, 10, 70]$ A lesz a kiosztás.

Lépések és jelentésük:

1. **Reset** t=0 — szinkron kezdet mindenki biztosan 0-ról indul.
2. **START** — a vezérlő 3 s-os ciklusokban számolja újra az allokációt, a $t = 60$ s utáni váltás az ezt követő ciklusban fog megjelenni.
3. **Megfigyelés** 0..80 **s** — várjuk $t = 60$ s-nél az újra osztást a grafikonon.

Várt rendszerállapot

- 0..60 *s*: tényleges értékek $[30, 30, 30]$ A, összáram = 90 A.
- 60+ *s*: ténylegesek $[10, 10, 70]$ A, összáram = 90 A (a két kicsi igény teljesül, a maradék, pedig a nagyhoz kerül).
- *Megszakító*: végig be van kapcsolva, mert $90 <$ `BREAKER_MAX` $= 120$ és $90 >$ `BREAKER_MIN` $= 10$.

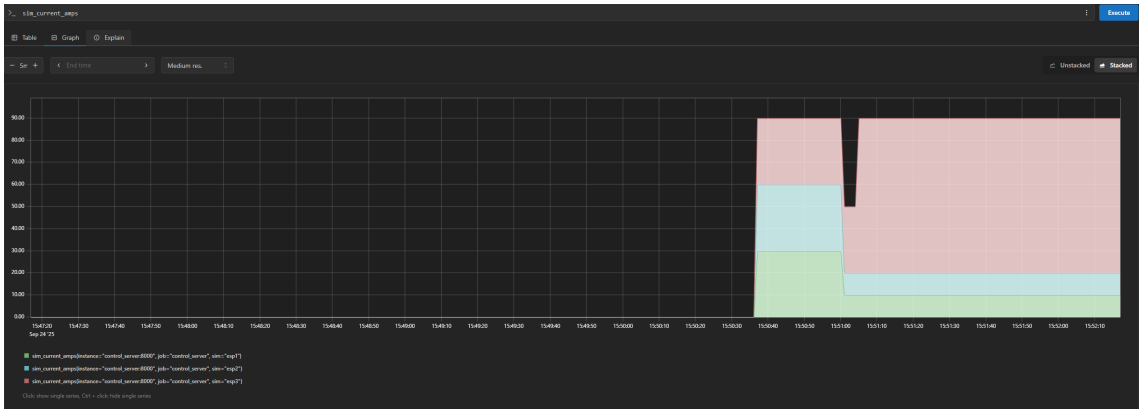
Hol ellenőrizhető?

- `output.txt`: 3 s-onként új sor; jellemző minták:

```
# t=3s  (elso szakasz)
sim_state=RUNNING sum_current_amps=90.0
sims=esp1:raw=50.0,effective=30.0,cap=30.0|esp2:raw=50.0,
effective=30.0,cap=30.0|esp3:raw=50.0,effective=30.0,cap=30.0
breakers=brk1:on,brk2:on

# t=63s (masodik szakasz, igényváltozás utan)
sim_state=RUNNING sum_current_amps=90.0
sims=esp1:raw=10.0,effective=10.0,cap=10.0|esp2:raw=10.0,
effective=10.0,cap=10.0|esp3:raw=100.0,effective=70.0,cap=70.0
breakers=brk1:on,brk2:on
```

Siker kritérium: a grafikon két állapotba áll be: előbb 30/30/30 A, majd a $t = 60$ s után 10/10/70 A, az összáram végig 90 A, a megszakítók végig bekapcsolva maradnak.



9.4. ábra. Dinamikus újraelosztás áramerősség

9.5. ábra. Dinamikus újraelosztás igények

9.2.5. Megszakító hiszterézis

Itt ebben a tesztben a megszakító vezérlési logikáját vizsgálom. A cél annak megmutatása, hogy a rendszer helyesen kezeli a kétlépcsős kapcsolást: a megszakító leold, ha az összáram meghaladja a felső küszöbértéket (`BREAKER_MAX_TOTAL`), de csak akkor kapcsol vissza, ha az áramfelvétel egy másik alacsonyabb érték (`BREAKER_MIN_TOTAL`) alá esik. Ez a viselkedés biztosítja, hogy a rendszer stabil, sűrű ki-be kapcsolás nem lesz a határérték körüli terhelésingadozás esetén sem. A teszt során az allokációs korlátozás szándékosan nem aktív.

Bemenetek:

- `ALLOC_MAX_TOTAL=50`
- `BREAKER_MAX_TOTAL=6`
- `BREAKER_MIN_TOTAL=3`
- Menetrendek:
  - ESP1: 0 s $\rightarrow$ 2,0 A, 40 s $\rightarrow$ 0,5 A
  - ESP2: 0 s $\rightarrow$ 5,0 A, 40 s $\rightarrow$ 0,5 A
  - ESP3: 0 s $\rightarrow$ 0,0 A

Miért ez a beállítás? Kezdetben az összáram $2{,}0+5{,}0+0{,}0 = 7{,}0$ A, ami nagyobb, mint `BREAKER_MAX_TOTAL` $= 6$ A, ezért a megszakító *leold*, 40 s után a terhelések $0{,}5+0{,}5+0{,}0 = 1{,}0$ A-ra esnek, ami kissebb, mint `BREAKER_MIN_TOTAL` $= 3$ A, így a megszakító *visszakapcsol*. Az `ALLOC_MAX_TOTAL`=50 A bőven a terhelések fölött van, ezért *allokációs korlátozás nem várható*.

Lépések és jelentésük:

1. **Reset** t=0 — szinkron indulás.
2. **START** — a vezérlő 3 s-os ciklusokban értékeli ki az adatokat. Az első ciklusban a ≥ 6 A miatt ki van kapcsolva, a 40 s utáni első ciklusban a ≤ 3 A miatt bekapcsol.

3. **Megfigyelés** 0..60 **s** — a naplóban egy `off` → `on` átmenet látszik 40 s-nél.

Várt rendszerállapot

- 0..40 s: összáram ≈ 7,0 A ⇒ megszakító kikapcsolva.
- 40+ s: összáram ≈ 1,0 A ⇒ megszakító bekapcsolva.
- *Allokáció*: nincs korlát (nagy szám), mert $\sum d_i \ll$ `ALLOC_MAX_TOTAL`.

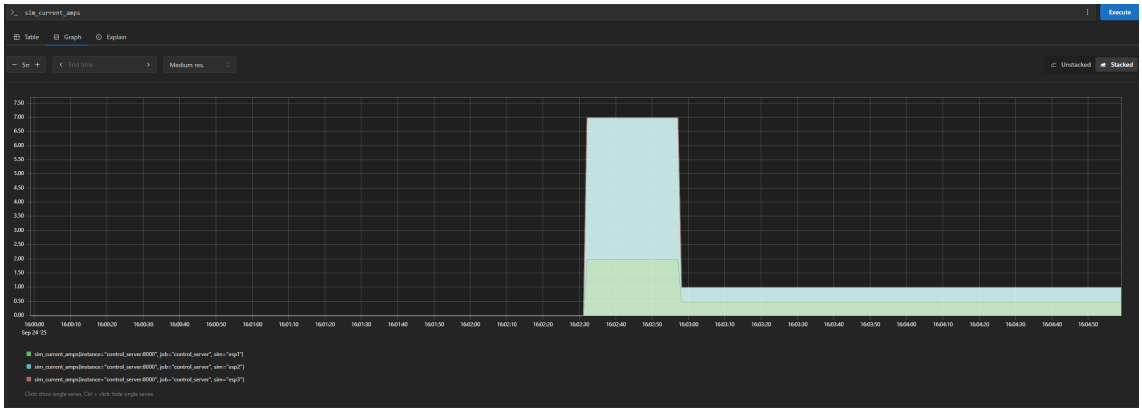
Hol ellenőrizhető?

- `output.txt`: 3 s-onként új sor; tipikus minták:

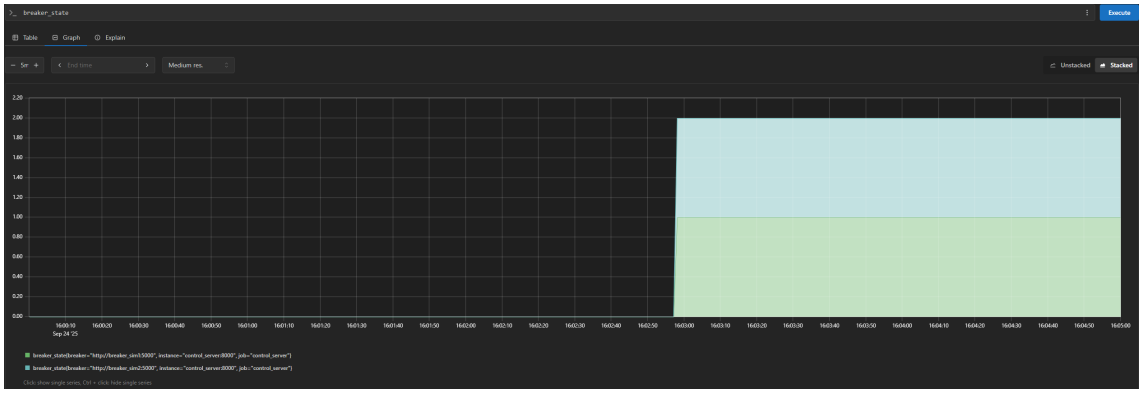
```
# t=3s  (kezdeti szakasz)
sim_state=RUNNING sum_current_amps=7.0
sims=esp1:raw=2.0,effective=2.0,cap=1e9|esp2:raw=5.0,
effective=5.0,cap=1e9|esp3:raw=0.0,effective=0.0,cap=1e9
breakers=brk1:off,brk2:off

# t=42s (menetrendváltás utáni első ciklus)
sim_state=RUNNING sum_current_amps=1.0
sims=esp1:raw=0.5,effective=0.5,cap=1e9|esp2:raw=0.5,
effective=0.5,cap=1e9|esp3:raw=0.0,effective=0.0,cap=1e9
breakers=brk1:on,brk2:on
```

Siker kritérium: az élő grafikonon 0..40 s között ∼ 7 A körüli összáram mellett kikapcsolt állapot látszik, 40 s után ∼ 1 A mellett bekapcsolt, az `output.txt` ezt a `off` → `on` váltást 2-3 cikluson belül egyértelműen tükrözi.



9.6. ábra. Megszakító áramerősség



9.7. ábra. Megszakító állapotok

9.2.6. Leállított mód (STOPPED)

Ez a teszt azt vizsgálja, hogy a vezérlő 'STOPPED' állapotban megfelelően viselkedik-e. A forgatókönyv igazolja, hogy a rendszer leállítása után a vezérlő valóban felfüggesztésre kerül, és a rendszer "befagyasztja" az utolsó ismert stabil állapotot. Annak ellenére, hogy a leállás alatt a háttérben a bemeneti konfigurációk (pl. allokációs keret) megváltoztatható, a teszt sikeres, ha a vezérlő nem reagál ezekre, nem számol új korlátokat, és nem frissíti a kimeneti naplót.

Bemenetek:

- Kiindulás: a 3. szcenário stabil állapota
  - `ALLOC_MAX_TOTAL` = 9
  - `BREAKER_MAX_TOTAL` = 12
  - `BREAKER_MIN_TOTAL` = 2
  - effective = [3, 3, 3] A
  - `sum` = 9 A, megszakítók on
- Módosítások STOPPED állapot alatt *csak fájlba írva*:
  - `ALLOC_MAX_TOTAL` = 6
  - ESP1 menetrend 1,0 A

Miért ez a beállítás? A cél annak igazolása, hogy `STOPPED` módban a vezérlő *nem avatkozik be*: nem számol ad új korlátokat, nem kapcsol megszakítót, a virtuális idő nem halad és a napló is megáll.

Lépések és jelentésük:

1. **STOPPED** — a `sim_control.txt` `STOPPED`-re áll, a vezérlési ciklus megáll, nincs új allokáció vagy megszakítóparancs.
2. **Konfiguráció módosítása** — `ALLOC_MAX_TOTAL` = 6 és ESP1 1,0 A változtatások *mentése* (de ez még nem lép életbe).
3. **Várakozás ∼ 2 ciklusnyi időtartam** — mivel a rendszer áll, nem várható új státusz- vagy naplósor.

Várt rendszerállapot

- *Korlátok és effektív értékek*: változatlanul [3, 3, 3] A (a leállítás *előtti* állapot szerint).
- *Megszakító*: változatlanul bekapcsolt állapotban van (nem történik átkapcsolás).
- *Idő és napló*: a virtuális idő nem halad, az `output.txt` *nem* bővül.

Hol ellenőrizhető?

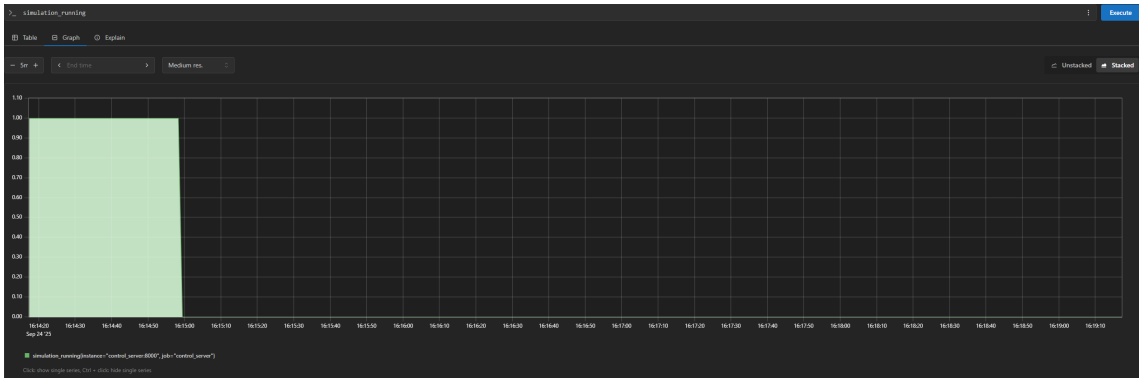
- `output.txt`: az utolsó `RUNNING` sor után nem jelennek meg új bejegyzések `STOPPED` alatt, pl.:

```
# utolsó sor STOPPED előtt
sim_state=RUNNING sum_current_amps=9.0
sims=esp1:raw=50.0,effective=3.0,cap=3.0|esp2:raw=50.0,effective=3.0,cap=3.0|esp3:raw=50.0,
     effective=3.0,cap=3.0
breakers=brk1:on,brk2:on

# STOPPED alatt nincs új sor
```

Siker kritérium: `/status.sim_state=STOPPED`, a korlátok és megszakító értékek megegyeznek a leállítás *előtti* állapottal és az `output.txt` nem bővül a várakozás ideje alatt.

9.8. ábra. stopped állapot áramerősség



9.9. ábra. stopped állapot állapotok

9.10. ábra. stopped állapot maximális áramok

9.2.7. Magas rendelkezésre állás (High Availability) tesztje

Ez a vizsgálat nem az energetikai szabályozást, hanem az infrastruktúra robusztusságát ellenőrzi. A cél annak demonstrálása, hogy a Kubernetes környezet képes automatikusan kezelni a kritikus komponensek (pl. a Kontroll Szerver) meghibásodását, és emberi beavatkozás nélkül helyreállítani a szolgáltatást.

Tesztkonfiguráció: A teszt során a Kontroll Szerver `Deployment`-je 2 replikával fut. A rendszer `RUNNING` állapotban van, folyamatos mérés és szabályozás zajlik.

Lépések:

1. **Alapállapot ellenőrzése:** A `kubectl get pods` paranccsal ellenőrzöm, hogy a vezérlő fut.
2. **Hiba szimulálása:** A `kubectl delete pod <pod-id>` paranccsal szándékosan lelövöm az aktív vezérlő konténert, szimulálva egy szoftveres összeomlást.
3. **Megfigyelés:** A devpanelen és a `kubectl get pods -w` kimeneten figyelem az újraindulást.

Eredmények: A hiba pillanatában a kapcsolat az ESP eszközökkel rövid időre megszakadt. A Kubernetes azonban azonnal észleli, hogy a futó podok száma (0) eltér a kívánt állapottól. A rendszer automatikusan új konténert indít. Mivel a konfigurációt (`thresholds.txt`) és az állapotot PVC-k vagy ConfigMap-ek tárolják, az új példány az előzővel azonos beállításokkal lép működésbe.

Konklúzió: A teszt sikeresen igazolja, hogy a konténerizált környezet *öngyógyító*. A kiesési idő nagyságrendileg összemérhető a rendszer ciklusidejével, így a felügyelet folytonossága a gyakorlatban nem sérül.

9.3. Összegzés és a vizsgálatok értékelése

A fejezetben bemutatott tesztesetek célja a rendszer működésének validálása volt különböző terhelési viszonyok és üzemállapotok mellett. Ezek a forgatókönyvek lefedik a legfontosabb működési módokat, amik a preset-ek között definiáltunk.

A kimenetek elemzése alapján megállapítható, hogy a rendszer minden esetben az elvárásoknak megfelelően működött:

- **Allokációs pontosság:** Túlterhelés esetén (10.4.3. és 10.4.4. eset) a vezérlő sikeresen tartotta a globális áramkeretet (`ALLOC_MAX`), és igazságos elosztást eredményezett.
- **Védelmi logika:** A megszakító-vezérlés (10.4.5. eset) pontosan követte a beállításokat, megakadályozva a fizikai határérték átlépését.
- **Dinamikus viselkedés:** A rendszer a terhelés hirtelen változásaira (10.4.4. eset) a következő vezérlési ciklusban azonnal reagált, bizonyítva a valós idejű beavatkozás képességét.
- **Stabilitás:** A `STOPPED` állapot (10.4.6. eset) helyes működése igazolta, hogy a rendszer állapota bármikor befagyasztható és elemezhető anélkül, hogy a vezérlés elromlana.

10. fejezet

Összefoglalás és kitekintés

A dolgozat célja egy nyílt forrású, konténerizált energetikai felügyeletre képes keretrendszer tervezése volt, ami alacsony költségű végponti eszközöket (ESP8266 mérő), Python-on alapuló vezérlő komponenst, idősoros adattárolást (Prometheus) és ezenkívül vizualizációt (Grafana) tartalmaz. A rendszer az interfészeken szét választja a mérést és a döntéshozatalt, a beavatkozást pedig ipari protokollon (Modbus/TCP) keresztül valósítja meg a villamos hálózati eszközökön. Az üzemeltetési környezet teszt állapotban, ahogy én is használtam általában Docker Compose, nagyobb rendelkezésre állási igény esetén Kubernetes.

Eredmények és tanulságok. A szimulációs vizsgálatok azt mutatták, hogy a keretrendszer képes:

- a mért villamos mennyiségek folyamatos, Prometheus-kompatibilis exportjára és azok megjelenítésére
- determinisztikus beavatkozásra, vezetéknélküli hálózaton keresztül, ez gyors reagálást és reprodukálhatóságot biztosít
- globális áramkeret pontos követésére, adott esetben a túllépések gyors csillapítására és az erőforrások igazságos újraelosztására a max-min fair elv alapján
- skálázható, konténer-alapú üzemeltetésre, ami nagyban megkönnyíti a telepítést, a frissítést és a diagnosztikát.

Gyakorlati tapasztalat, hogy az egyszerű vezérlő rendszer fontosabb a túlbonyolított modellnél, a mérési zaj, a késleltetések és a végpontok nemlineáris viselkedése mellett a robusztus, determinisztikus vezérlő stabilabban teljesített. A komponensek laza csatolása és a metrika-alapú rendszer érdemben csökkenti a hibaelhárítás bonyolultságát.

A munka fő műszaki hozzájárulásai:

1. Egységesített mérési/vezérlési interfész energetikai rendszerekhez Prometheus-formátumú metrikákkal és REST és Modbus átalakítóval.
2. Max-min fair elosztású beavatkozási szabályozó integrációja ipari rendszerbe.
3. Konténer-alapú referenciaimplementáció Docker Compose vagy Kubernetes.

4. Grafana-alapú üzemeltetési, illetve diagnosztikai irányítópultok, előre tervezett riasztások és alap-telemetria.

5. Reprodukálható szimulációs rendszer, ami determinisztikusan teszteli a rendszer beállításait.

Korlátok és érvényességi fenyegetések. A vizsgálatok kontrollált környezetben zajlottak; a terepi viszonyok (hálózati zavarok, különböző végponti firmware-ek, szélsőséges terhelési profilok) további kihívásokat jelenthetnek. A biztonsági réteg alapértelmezetten a helyi hálózatra és egyszerű hitelesítésre támaszkodik; nagyvállalati környezetben szükséges a végpont- és szolgáltatásoldali tanúsítványkezelés és kulcsforgatás beépítése. A Modbus/TCP protokoll korlátai (nincs beépített titkosítás, korlátozott hibakezelés) szintén megfontolandók.

Jövőbeli munka. A rendszer fejlesztésének lehetséges irányai:

- **Biztonság és megbízhatóság:** mTLS alapú végpont-hitelesítés, kulcsforgatás, jogosultságkezelés.
- **Protokoll-tágítás:** IEC 60870-5-104 / IEC 61850 gateway, illetve modern ipari mezőbuszok támogatása a heterogén eszközparkhoz.
- **Fejlettebb szabályozás:** modellprediktív vagy hibrid (MPC + max-min) vezérlő vizsgálata időkorlátos optimalizálásra, költség- és hálózati korlátok együttes kezelésére.
- **Edge-képességek:** lokális döntésképesség és *graceful degradation* hálózati problémák esetén, OTA frissítési csatorna az ESP eszközökre.
- **Megfigyelhetőség és üzemeltetés:** trace-alapú hibakeresés (OpenTelemetry), automatikus eszköz-felfedezés, konfiguráció- és verziókezelés.
- **Valós terepi pilot:** több, egymást zavaró terhelés és megosztott hálózati infrastruktúra mellett végzett hosszú távú mérések, SLA-k és üzemeltetési költségek becslésével.

Az elkészült keretrendszer egy átlátható, bővíthető és költséghatékony alapot ad az energetikai felügyeleti rendszer elkészítéséhez. Az egyszerű és robusztus szabályozási elvek, a metrika-központúság és a konténeres üzemeltetés együtt olyan eszköztárat alkotnak, ami ipari környezetekben is gyors bevezetést és megbízható működést tesz lehetővé.

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

[1] Mukhadin Beschokov: How to work with a kubernetes cluster? guide by wallarm, 2025. URL https://www.wallarm.com/what/what-is-a-kubernetes-cluster-and-how-does-it-work. Megnyitva: 2025-04-05.

[2] Jean-Yves Le Boudec: Rate adaptation, congestion control and fairness: A tutorial. https://web.archive.org/web/20230422115954/https://ica1www.epfl.ch/PS_files/LEB3132.pdf, 2005. EPFL.

[3] Docker Inc.: Deploy on kubernetes with docker desktop, 2025. URL https://docs.docker.com/desktop/features/kubernetes/. Megnyitva: 2025-04-05.

[4] Ecostruxure™ power monitoring expert. https://www.se.com/hu/hu/product-range/65404-ecostruxure-power-monitoring-expert/#overview, 2025. Megnyitva: 2025-03-17.

[5] Schneider Electric: Charging station evlink, 2025. URL https://www.se.com/hu/hu/product/EVB3S07NC0/charging-station-evlink-pro-ac-ac-metal-7-4kw-32a-1p+n-t2-attached-cable-rdcdd-6ma-mnx-aux-/. Megnyitva: 2025-05-18.

[6] Schneider Electric: Masterpact mtz product range, 2025. URL https://www.se.com/hu/hu/product-range/63545-masterpact-mtz/#products. Megnyitva: 2025-05-18.

[7] electrofunsmart: Iot szerver prometheus és grafana monitorozással egy esp8266 esetén, 2025. URL https://www.hackster.io/electrofunsmart/iot-server-with-prometheus-and-grafana-monitoring-a-esp8266-9e0661. Megnyitva: 2025-03-25.

[8] ElectronicWings: Nodemcu development kit/board, 2023. URL https://www.electronicwings.com/nodemcu/nodemcu-development-kitboard. Megnyitva: 2025-05-01.

[9] Simply Explained: Home energy monitor esp32 ct sensor emonlib, 2025. URL https://simplyexplained.com/blog/Home-Energy-Monitor-ESP32-CT-Sensor-Emonlib/. Megnyitva: 2025-03-22.

[10] Instrumentation Tools: Background of modbus ascii and rtu data frames, 2025. URL https://instrumentationtools.com/background-of-modbus-ascii-and-rtu-data-frames/. Megnyitva: 2025-03-25.

[11] Kubernetes Authors: ConfigMap. URL https://kubernetes.io/docs/concepts/configuration/configmap/. Dokumentáció.

[12] Kubernetes Authors: Deployments. URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Dokumentáció.

[13] Kubernetes Authors: Persistent Volumes. URL https://kubernetes.io/docs/concepts/storage/persistent-volumes/. Dokumentáció.

[14] Kubernetes Authors: Pods. URL https://kubernetes.io/docs/concepts/workloads/pods/. Dokumentáció.

[15] Kubernetes Authors: Secret. URL https://kubernetes.io/docs/concepts/configuration/secret/. Dokumentáció.

[16] Kubernetes Authors: Service. URL https://kubernetes.io/docs/concepts/services-networking/service/. Dokumentáció.

[17] Mikroelektronik: Yhdc sct013 100a 1v felfüggesztés típusú osztott magos áramérzékelő, 2025. URL https://mikroelektronik.hu/elektronikus-osszetevok/126660-yhdc-sct013-100a-1v-felfuggesztes-tipusa-osztott-magos-aramerzekelo-html. Megnyitva: 2025-03-22.

[18] Creator name or YouTube channel name if known: Title of video, 2025. URL https://www.youtube.com/watch?v=pcGg-U5d_n8. Megnyitva: 2025-03-25.

[19] OpenEnergyMonitor: Interface with arduino, 2025. URL https://docs.openenergymonitor.org/electricity-monitoring/ct-sensors/interface-with-arduino.html. Megnyitva: 2025-03-22.

[20] OpenEnergyMonitor: Emoncms: Open-source web application for energy, temperature and other environmental data visualisation. https://emoncms.org/. Elérés dátuma: 2025. november 18.

[21] Darshil Patel: Getting started with nodemcu (esp8266) on arduino ide, 2020. URL https://projecthub.arduino.cc/PatelDarshil/getting-started-with-nodemcu-esp8266-on-arduino-ide-b193c3. Megnyitva: 2024-11-08.

[22] Prometheus: Prometheus - dimenzionális adatok: kulcs-érték párokon alapuló modell, 2025. URL https://prometheus.io/. Megnyitva: 2025-03-25.

[23] Simatic energy management software. https://www.siemens.com/global/en/products/automation/industry-software/automation-software/energymanagement.html, 2025. Megnyitva: 2025-03-17.

[24] Jan Sláčík – Petr Mlynek – Martin Rusz – Petr Musil – Lukas Benesl – Michal Ptáček: Broadband power line communication for integration of energy sensors within a smart city ecosystem. *Sensors*, 21. évf. (2021) 10. sz., 3402. p. See Fig. 2 for the principle of the water-filling algorithm.

[25] Erich Styger: Controlling an ev charger with modbus rtu, 2022. URL https://mcuoneclipse.com/2022/12/31/controlling-an-ev-charger-with-modbus-rtu/. Megnyitva: 2025-03-24.

[26] TechTutorialsX: Esp8266 posting json data to a flask server on the cloud, 2017. URL https://techtutorialsx.com/2017/01/08/esp8266-posting-json-data-to-a-flask-server-on-the-cloud/. Megnyitva: 2025-03-24.

[27] The Home Assistant Community: Home Assistant: Open-source home automation that puts local control and privacy first. https://www.home-assistant.io/. Elérés dátuma: 2025. november 18.

[28] The Kubernetes Authors: Kubernetes documentation, 2025. URL https://kubernetes.io/. Megnyitva: 2025-04-05.

[29] Doc Walker: Modbusmaster: Arduino library for modbus communication, 2016. URL https://github.com/4-20ma/ModbusMaster. Megnyitva: 2025-04-07.