

## 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

# Keretrendszer energetikai felügyelethez

DIPLOMATERV

*Készítette*  
Veress Gábor

*Konzulens*  
dr. Zsóka Zoltán

2025. szeptember 23.

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>1</b>
<b>2. Meglévő megoldásokkal összehasonlítása</b>	<b>2</b>
2.1. Meglévő ipari megoldások . . . . .	2
2.1.1. Schneider Power Monitoring Expert . . . . .	2
2.1.1.1. Alapfunkciók . . . . .	2
2.1.1.2. Előnyök . . . . .	3
2.1.2. Siemens SIMATIC Energy Suite . . . . .	3
2.1.2.1. Alapfunkciók . . . . .	3
2.1.2.2. Előnyök . . . . .	3
2.2. Saját megoldás . . . . .	4
<b>3. Keretrendszer</b>	<b>5</b>
3.1. Rendszerarchitektúra áttekintése . . . . .	5
3.2. Eszközök . . . . .	6
3.2.1. Végpontok . . . . .	6
3.2.1.1. ESP8266 és AC árammérő szenzorok . . . . .	6
3.2.1.2. Mért eszközök . . . . .	8
3.3. Kommunikáció . . . . .	10
3.3.1. ESP8266 és Szerver között (Wi-Fi és REST API) . . . . .	10
3.3.2. Modbus . . . . .	10
3.4. Standardizált rendszer . . . . .	11
3.4.1. Áttekintés . . . . .	11
3.4.2. Tervezési célok és követelmények . . . . .	11
<b>4. Komponensek megvalósítása</b>	<b>13</b>
4.1. Végpontok . . . . .	13
4.1.1. Autótöltő . . . . .	13
4.1.1.1. Bevezetés . . . . .	13
4.1.1.2. Megvalósítás . . . . .	14
4.1.1.3. Mérési adatok elküldése . . . . .	14
4.1.1.4. Main loop . . . . .	14
4.1.1.5. Kommunikáció . . . . .	14
4.1.1.6. Modbus kommunikáció vezérléshez . . . . .	14
4.1.2. Megszakító . . . . .	15
4.1.2.1. Hardver . . . . .	15
4.1.2.2. Szoftver . . . . .	15
4.2. Kontroll szerver . . . . .	16

4.3.	Adatbázis . . . . .	16
4.3.1.	Prometheus adatgyűjtés kezelése . . . . .	17
4.3.2.	Prometheus lekérdezések kezelése . . . . .	17
4.4.	Grafana alapú megjelenítés . . . . .	18
4.4.1.	A háromfázisú és a napelemes áram vizualizálása és riasztása a Grafanában . . . . .	18
4.4.1.1.	A Dashboard . . . . .	18
<b>5.</b>	<b>Alkalmazás migrálása a Docker Compose-ból a Kubernetesbe</b>	<b>20</b>
5.1.	Bevezetés . . . . .	20
5.2.	A Docker Compose és Kubernetes áttekintése . . . . .	20
5.3.	Rendszerarchitektúrája . . . . .	21
5.4.	A Docker Compose beállítások konvertálása Kubernetes manifeszteké	22
5.4.1.	Névtér- és konfigurációkezelés . . . . .	22
5.4.2.	Deployment-ek és Service-ek . . . . .	22
5.4.3.	Perzisztens tárolók kezelése . . . . .	22
5.4.4.	Szolgáltatások elérhetővé tétele és hálózati konfiguráció . . . . .	23
5.4.5.	Telepítés és tesztelés . . . . .	23
5.5.	Nagy elérhetőségű rendszer implementációja . . . . .	23
5.5.1.	Replikák megvalósítása . . . . .	24
5.5.2.	KubeADM . . . . .	25
<b>6.</b>	<b>Felhasználói felület</b>	<b>26</b>
6.1.	Szöveges be- és kimenetek a szimulációhoz . . . . .	26
6.1.1.	Cél és áttekintés . . . . .	26
6.1.2.	Bemeneti szövegfájlok . . . . .	26
6.1.2.1.	thresholds.txt – küszöbök és maximum megen- gedhető áram . . . . .	26
6.1.2.2.	esp{1..3}_schedule.txt – idősoros bemenet . . . . .	27
6.1.2.3.	sim_control.txt – futtatási állapot . . . . .	27
6.1.3.	Kimeneti szövegfájl . . . . .	27
6.1.3.1.	output.txt – idősoros kimenet . . . . .	27
6.1.4.	Időkezelés és futtatás . . . . .	28
6.1.5.	Reprodukálhatóság és feldolgozhatóság . . . . .	28
6.1.6.	Rövid példa – beállítás → kimenet (részlet) . . . . .	28
<b>7.</b>	<b>Max–min fair (water-filling) elosztás</b>	<b>29</b>
7.1.	Elméleti háttér és cél . . . . .	29
7.1.1.	Motiváció és cél . . . . .	29
7.1.2.	Definíció (max–min fair) . . . . .	29
7.1.3.	Vízbetöltés (water-filling) . . . . .	29
7.1.4.	Algoritmus és bonyolultság . . . . .	29
7.1.5.	Tulajdonságok . . . . .	30
7.2.	A vezérlőben alkalmazott megvalósítás . . . . .	30
7.2.1.	Kapcsolat a rendszer komponenseivel . . . . .	30
7.2.2.	Példák . . . . .	31
7.2.3.	Implementációs részletek . . . . .	31

<b>8. Fejlesztői panel (Dev Panel)</b>	<b>32</b>
8.1. Cél és szerep . . . . .	32
8.2. Architektúra áttekintése . . . . .	32
8.3. Fő funkciók és munkamenet . . . . .	32
8.4. Backend API (rövid kivonat) . . . . .	33
8.5. Megbízhatósági és használhatósági szempontok . . . . .	33
8.6. Biztonsági és korlátok . . . . .	33
8.7. Kiterjeszthetőség . . . . .	33
<b>Irodalomjegyzék</b>	<b>35</b>

## 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 hálózataán 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. szeptember 23.

---

*Veress Gábor*  
hallgató

# 1. fejezet

## Bevezetés

A diplomatervem célja egy olyan keretrendszer megvalósítása, amely lehetővé teszi a távoli felügyeletet és vezérlést. A rendszer egyszerű felépítésű végponti elemeket (például ESP8266 alapú érzékelő- és vezérlőmodulokat) köt össze konténerizált vezérlőkomponenssel. A célja az adatok gyűjtése, tárolása és automatizált feldolgozása, valamint a végpontok megbízható kezelésének és redundanciájának biztosítása Kubernetes környezetben.

A villamosenergetikában az elmúlt években nagy igény jelentkezett a hálózati elemek, például autótöltők vagy megszakítók valós idejű felügyeletére és távoli vezérlésére. Ez fontos energiahatékonyság, hálózati biztonság és a zavartűrés szempontjából is. A modern ipari rendszerekben alkalmazott szoftverek, mint a Schneider EcoStruxure Power Monitoring Expert vagy a Siemens SIMATIC Energy Suite széleskörű funkcionalitást biztosítanak, szabványoknak megfelelést és 24/7 gyártói támogatást kínálnak, de jelentősen drágábbak.

A munkámban bemutatni kívánt saját megoldás ezzel szemben nyílt forráskódú komponensekre épít (ESP8266 mikrokontrollerek, Prometheus idősoros adatbázis, Grafana vizualizáció és Python Flask vezérlőszerver), ez költséghatékony és jól testre szabható az előzőkkel szemben. A rendszerem moduláris felépítése miatt az érzékelők plug-and-play módon csatlakoztathatóak és Kubernetes segíti a skálázhatóságot, a redundancia és a magas rendelkezésre állást.

A dolgozat a következő részekből áll:

- **Első fejezet:** A meglévő ipari megoldások ismertetése és összehasonlítása a saját fejlesztéssel.
- **Második fejezet:** A keretrendszer tervezésének bemutatása.
- **harmadik fejezet:** A rendszer egyes komponenseinek részletes megvalósítása.
- **Negyedik fejezet:** Nagy rendelkezésre állás megvalósítása hibrid klaszter topológiával.

## 2. fejezet

# Meglévő megoldásokkal összehasonlítása

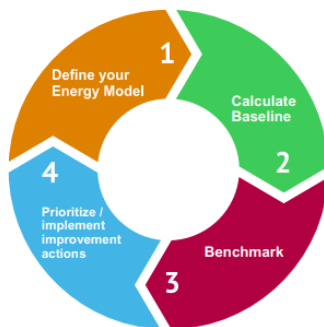
### 2.1. Meglévő ipari megoldások

#### 2.1.1. Schneider Power Monitoring Expert

##### 2.1.1.1. Alapfunkciók

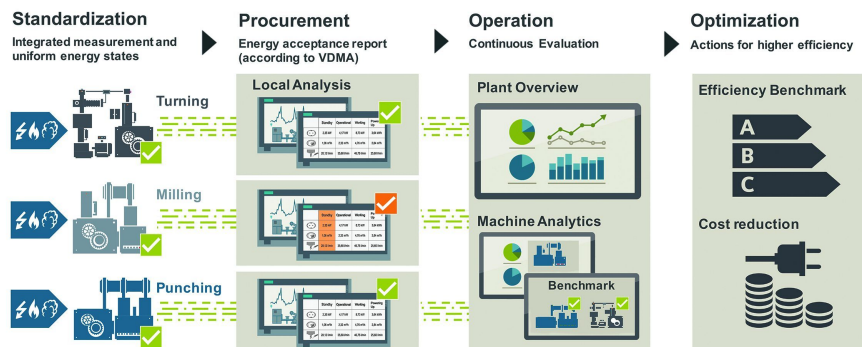
- Segít csökkenteni a meddő teljesítmény termelést és az ebből keletkező büntetéseket.
- Saját számlát készít, a helyi mérések alapján, hogy összehasonlítási alap legyen a számlákhoz.
- Segít elszámolhatóságot biztosítani alszámlázáshoz.
- Berendezések teljesítményét és várható élettartamát ellenőrzi.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

[3]



2.1. ábra. Schneider Electric PME model[3]





2.2. ábra. Siemens EMS model[15]

### 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 csökkenti a hiba utáni visszaállítások idejét is. Ezenkívül segít a mögöttes problémák megtalálásában is.[3]

## 2.1.2. Siemens SIMATIC Energy Suite

### 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.[15]

### 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.

[15]

## 2.2. Saját megoldás

Egy mondatban: a saját eszközkészletem (ESP-8266 + Prometheus + Grafana + Python) sokkal olcsóbb és könnyebben módosítható, de a Schneider EcoStruxure Power Monitoring Expert (PME) és a Siemens SIMATIC Energy Suite olyan pontosságot, energiaminőség-elemzést, ISO-50001-megfelelőséget és 24/7-es gyártói támogatást biztosít, aminek megvalósítása nagy munkát és pénzt igényelne.

Jellemző	Nyílt forráskódú megoldás	Schneider PME	Siemens Energy Suite
Peremi eszközök	ESP8266 + CT	PowerLogic / ION & PowerTag mérők, megszakítók, átjárók	S7-1500 PLC + Sentron PAC, 7KM PAC, megszakítók
Adatátvitel	Wi-Fi és HTTPS REST	Modbus/TCP	PROFINET
Adatbázis	Prometheus	Beépített SQL Express	Integrált WinCC SQL archívum
Vizualizáció	Grafana	Webalapú HTML5 irányítópult	WinCC HMI képernyő
Analitika	Ami lekódolásra kerül	Harmonikus, villódzás, EN 50160 megfelelés	Automatikus terheléskikapcsolás ISO 50001
Licenc költségek	Nincs	Eszközcsomagok: 5-től korlátlanig; 50-es csomag tízezer eurós nagyságrend	Futtatási licenc eszközönként ezer eurós nagyságrend
Tipikus ár 50 mérőpontra	kb. 1 000 € (panelek + szenzorok + szerver)	kb. 10 ezer € (mérők + licenc + szerver)	kb. 10 ezer € (mérők, PLC, licencek, TIA Portal)
Támogatás	Közösségi támogatás; nincs hivatalos tanúsítvány	Gyártói 24/7, ISO 50001	Gyártói 24/7, TÜV EN 13849

**2.1. táblázat.** Rendszeráttekintés - összehasonlítás

## 3. fejezet

# Keretrendszer

### 3.1. 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 össze-síti 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.

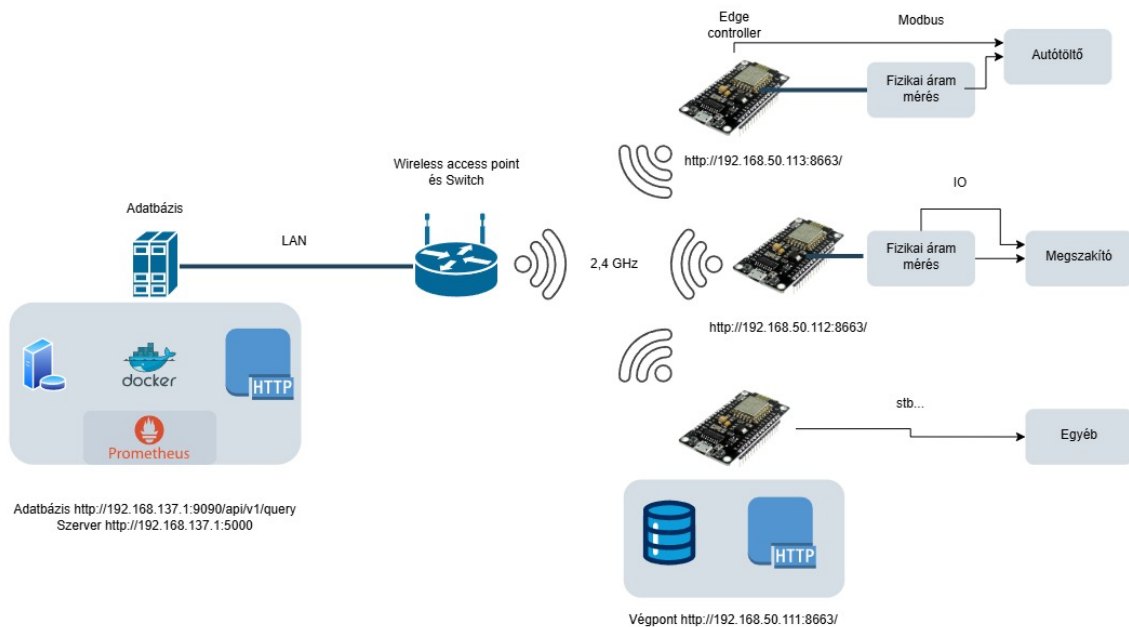
Az ESP8266 csomópontok csak az adatok két oldalú továbbadásáért felelősek, aktuális adatokat küldenek a szervernek, míg a Flask szerver döntéshozatalt hajt végre és parancsokat ad ki a töltőknek. Az érzékelés és a vezérlés szétválasztása leegyszerűsíti a végpont tervezését és a feldolgozást a szerver 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 végponon (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 fut-hat. 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 való idejű megfigyeléshez és későbbi elemzéshez.
- Docker containerek: A Flask server 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 az összes szükséges függőséget (Python-könyvtárak stb.) így megkönnyítve az üzemeltetés dolgát, és lehetővé teszi a rendszer megbízható méretezését vagy replikálását.



3.1. ábra. Rendszerarchitektúra

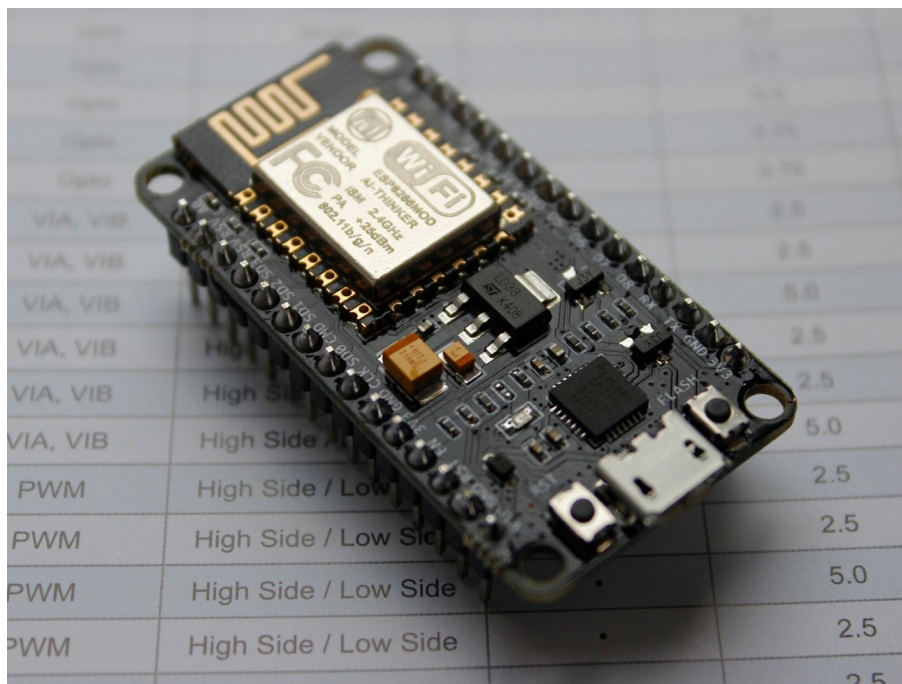
## 3.2. Eszközök

### 3.2.1. Végpontok

#### 3.2.1.1. ESP8266 és AC árammérő szenzorok

A pontos árammérés minden elektromos töltőnél kritikus a rendszer számára. Az ESP8266-ot (NodeMCU) 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 vezetőke köré kerül, és nincs szükség közvetlen elektromos érintkezésre a vezetővel. Ez az érzékelő a kábelben 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. [8]



**3.2. ábra.** NodeMCU (ESP8266) [13]

Ez a feszültségtartomány kompatibilis az ESP8266 analóg-digitális átalakítójával az analóg bemeneten, amely a legtöbb ESP8266 kártyán 0-1 V-ot tud olvasni (a NodeMCU kártyák tartalmazznak 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. [12]

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 el kell 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 vezetőke csatlakozik: az egyik az ADC bemenethez, a másik pedig a középponthoz körülbelül 1,65 V a 3,3 V-os táp miatt. [12]

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.



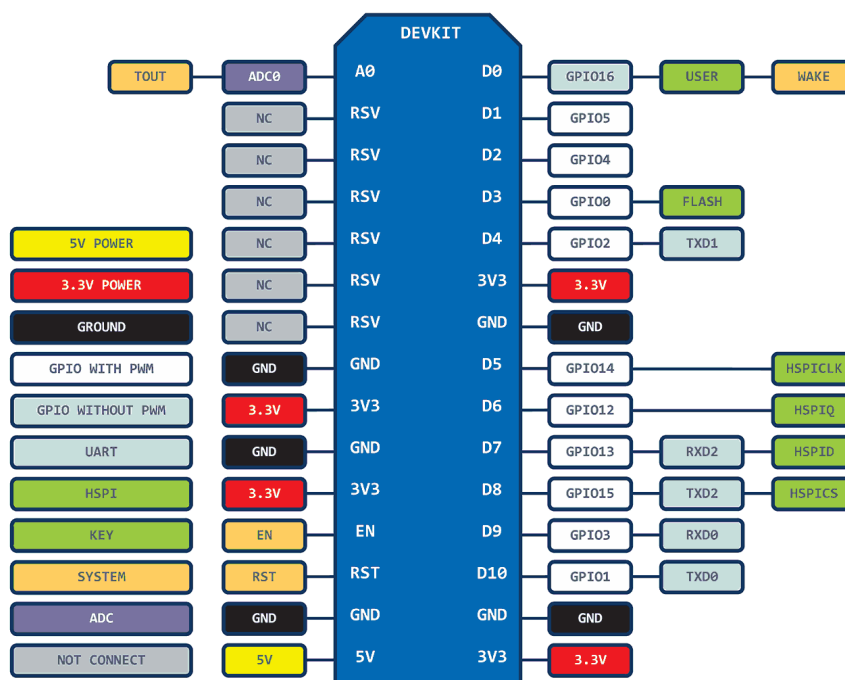
**3.3. ábra.** SCT-013 áramváltó [10]

#### **3.2.1.2. Mért eszközök**

Különböző eszközök mérését hajtók végre egy hálózatban, amiknél, más paraméterek mérésére van szükségünk.

Ilyen eszközök a megszakítók, itt érzékelnünk kell:

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



3.4. ábra. Pinout [7]

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 áttalakitást így a bemeneti skála változik 0 és 3,3 voltra.

Ha áramméréseket 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.

$$R = \frac{U}{I} = \frac{3.3 \text{ V}}{5 \text{ A}} = 660 \text{ m}\Omega \quad (3.1)$$

**1. egyenlet:** Áramméréshez használt sönt ellenállás értéke

$$P = U \times I = 3.3 \text{ V} \times 5 \text{ A} = 16.5 \text{ W} \quad (3.2)$$

**2. egyenlet:** A sönt ellenállás

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

## 3.3. Kommunikáció

### 3.3.1. ESP8266 és Szerver között (Wi-Fi és REST API)

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.

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

Itt az ESP8266 beépített WiFi könyvtárat használtam. [18] 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'))
```

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

A JSON adatstruktúra a következő:

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

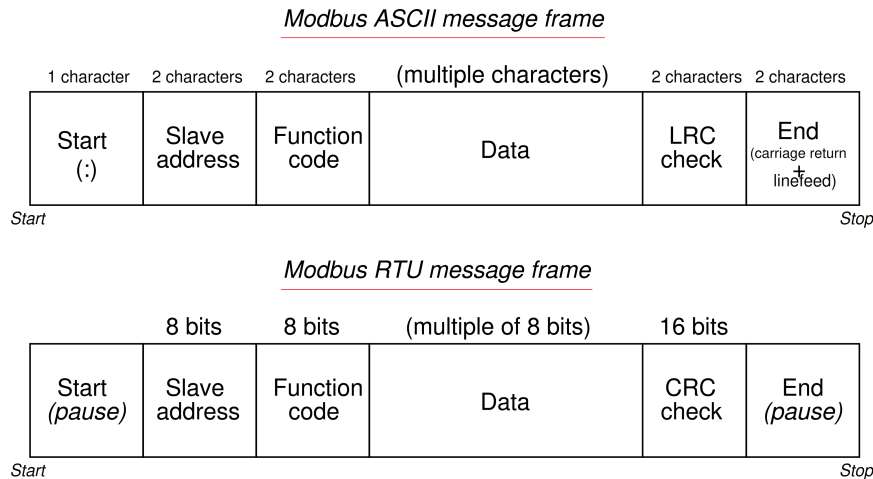
A Wi-Fi kommunikáció itt nem követeli meg, hogy az ESP8266 ismerje a szerver címét, mert csak GET parancsokat használtam. Itt a kiszolgáló fix IP-címmel rendelkezhet a LAN-ban. Elég viszont, ha a szerver ismeri a végpontok IP címét, amit viszont könnyű megadni és frissíteni. Kezdetben titkosítatlan HTTP-t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

### 3.3.2. 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. [17]

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.





3.5. ábra. Modbus adatstruktúra [9]

## 3.4. Standardizált rendszer

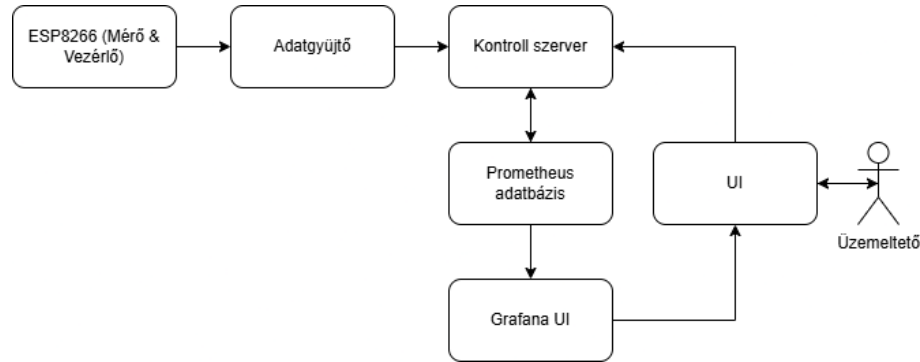
### 3.4.1. Áttekintés

Direkt úgy állítottam össze a rendszert, hogy az moduláris, konténeres keretrendszert tudjon képezni. Ez így lehetővé teszi az épületek energiagazdálkodásához szükséges komponensek gyors integrációját, leginkább az EV-töltőkre és a megszakítópanelekre fókuszálva. Az adat tárolást (Prometheus), vezérlő API-kat és vizualizációt (Grafana) szabványosítja ebben az esetben, hogy egyszerűsítse a demo környezeteket és a termelési környezetben bevezetéseket is.

### 3.4.2. Tervezési célok és követelmények

- Plug-and-Play alkatrészek: A felhasználók új érzékelőket vagy vezérlőket önálló szolgáltatásként telepíthet, amíg az kompatibilis a keretrendszer eszközeivel.
- Szabványosított mérési API: Itt egyszerűen minden komponens Prometheus-formátumú metrikákat exportál HTTPS-n keresztül.
- Vezérlő: A Python-alapú ControlServer REST protokollon keresztül irányítja az eszközöket.
- Konténerizált környezet: Minden mikroszolgáltatás konténerekben fut; az orkestrálást, pedig Kubernetes segítségével oldottam meg.
- Skálázhatóság és bővíthetőség: Könnyedén skálázható ez a környezet , hozzá lehet adni új erőforrástípusokat, és integrálni új eszközöket, mint ütemezés vagy akár valamilyen ML elemzés.

Mivel kubernetesben telepíthető a keretrendszer minden komponense konténerként fut és a cluster szolgáltatásai gondoskodnak arról, hogy a Python ControlServer, az ESP8266-hoz kapcsolódó adapterek, valamint a Prometheus és Grafana mindig elérhetők és skálázhatók legyenek.



**3.6. ábra.** Keretrendszer architektúra

A Python ControlServer egy Deployment formájában jön létre, ezt egy Service köti össze a belső hálózaton belül. A Deployment manifest-jében TLS tanúsítványokat tartalmazó Secret hivatkozik a HTTPS tanúsítványokra, így minden REST hívás titkosított csatornán zajlik. A control serveren belül a Flask alapú REST API két fő végpontot kínál: az egyik a `/metrics`, ami Prometheus kompatibilis formátumban szolgáltatja az aktuális fogyasztási metrikákat, a másik pedig a vezérlőhívások fogadására van fenntartva ahol lehet új áramkorlátokat beállítani az EV töltők számára.

Az ESP8266 szenzorok microservice-ként jelennek meg a rendszerben: mind-egyik egy Deployment, ami tartalmazza a hardverrel kommunikáló sorospor adaptert. Ezek a podok HTTPS-en jelentkeznek be a ControlServernél, és folyamatosan kiszolgálják a `/metrics` végpontjukat. A Prometheus-ban lekonfiguráljuk, hogy a Prometheus scrapper felvegye őket a targetek közé. A felhasználó, ha új eszközt telepít és megadja a `prometheus.io/scrape: "true"` és `prometheus.io/path: "/metrics"` értékeket. Így plug and play módon csatlakozható tetszőleges új érzékelők vagy vezérlők.

A Prometheus egy StatefulSet formájában fut és PersistentVolumeClaim segítségével tárolja el az adatbázist, így újraindítás esetén se vesznek el az adatok. A scrape konfiguráció paraméterezését ConfigMapben lehet megtenni. A Grafana Deployment mellé szintén egy PVC került a dashboard-ok mentéséhez és ConfigMapen keresztül lehet dashboard-okat definiálni. Az útválasztó SSL-terminációja után a felhasználó hozzá fér az irányítópulthoz.

A skálázhatóságot HorizontalPodAutoscaler biztosítja: ha egy pod CPU- vagy memóriahasználata átlépi a beállított küszöböt, Kubernetes automatikusan podokat indít. Ez a felépítés lehetővé teszi, hogy a cluster pillanatnyi terhelésének megfelelően bővítsük a kapacitást, akár kibővtve külső linux VM-re.

## 4. fejezet

# Komponensek megvalósítása

### 4.1. Végpontok

#### 4.1.1. Autótöltő

##### 4.1.1.1. Bevezetés

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.



4.1. ábra. Autótöltő [4]

#### 4.1.1.2. Megvalósítás

Itt az ESP8266 firmware főbb részeit elemzem.

**WiFi és HTTP-kiszolgáló beállítása** Kezdsnek 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.

#### 4.1.1.3. 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.

#### 4.1.1.4. 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.

#### 4.1.1.5. 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.

#### 4.1.1.6. 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.

## Hardver

- **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 [20] használtam a továbbítás egyszerűsítésére.
- **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.

### 4.1.2. 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 pergesmentesítéssel (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 }
```

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. ábra. Megszakító [5]

## 4.2. Kontroll szerver

## 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-telemetry számára. Minden adatot időbélyegzett értéksorozatként kezel. Ezeket lehet tárolni és lekérdezni. [6] [14]

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



4.3. ábra. Prometheus [11]

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 hirdetem 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'])
```

Ha olvasás érkezik, a szerver frissíti a számokat (egyébként ezt periodikusan is megteszi)

```
current_gauge.labels(charger=id).set(value).
```

A Prometheus-nak előre megkell adni az ip-címeket a konfigurációs filejában (a scrape konfigurációján keresztül), hogy időszakonként megnézze a Flask szerver /metrics URL-jét. Ez azért előnyösebb mert utólag ezeket már nem lehet állítani a prometheusban indítás után. A szerver, pedig egy stabil IP címen van. A sok fizikai végpontról, pedig a szerver gyűjt ahol elértem, hogy üzem közben is lehessen új végpontokat hozzáadni vagy módosítani.

Amikor a Prometheus olvas, a Flask az összes aktuális értéket szöveges Prometheus metrika formátumban adja ki. A Prometheus ezután ezeket az értékeket a metrikánévvel és címkékkel indexelve tárolja. Ez a lehívás alapú felügyelet jól illeszkedik a Prometheus működéséhez. A Prometheus adatai megjeleníthetők a Grafana által is és összetett lekérdezések írhatók például a teljes áram kiszámítására, amihez szükségem is volt nekem rendszer irányításához.

### 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
}
```

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

### 4.3.2. Prometheus lekérdezések kezelése

```
queryPrometheus()
```

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
```

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); \texttt{Adat JSON-be nyomtatása}

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

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

```

        Serial.println(error.c_str());
        return;
    }
}

```

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") {

    const char* valueStr = doc["data"]["result"][0]["value"][1];

    prometheusValue = String(valueStr).toFloat();

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

```

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 másodperc-re 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. A háromfázisú és a napelemes áram vizualizálása és riasztása 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

```

#### 4.4.1.1. A Dashboard

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



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

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 lekérdezéseket 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.

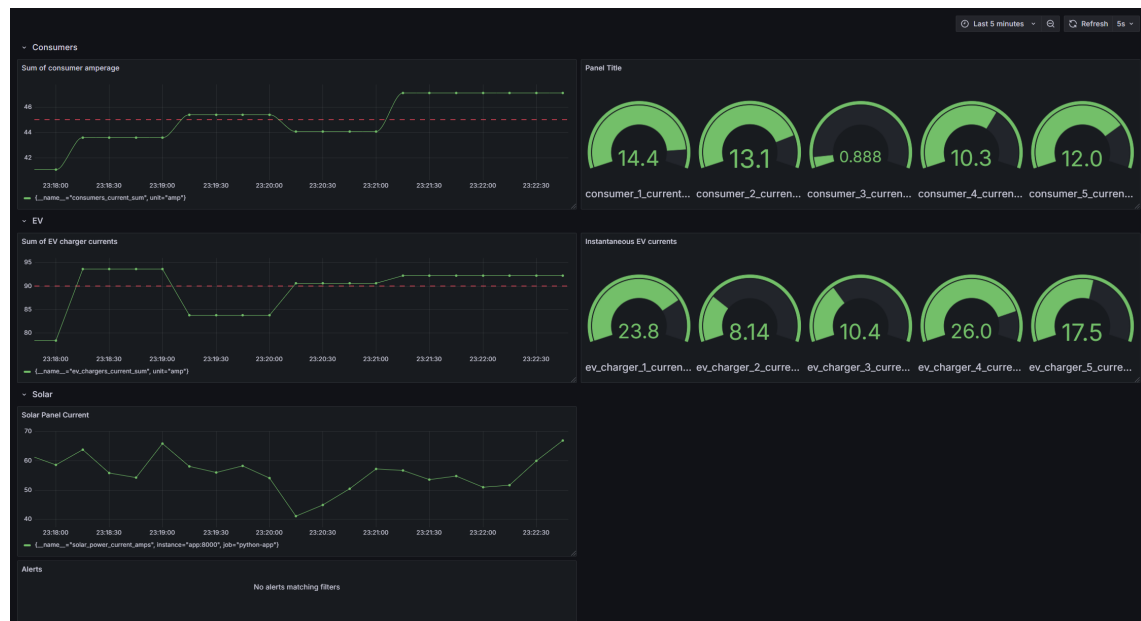
A panel hasonlóan az előző ponthoz jeleníti meg az adatokat, amely a pillanatnyi és max értéket mutatja.

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

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
```



4.4. ábra. Általam készített Grafana dashboard

## 5. fejezet

# Alkalmazás migrálása a Docker Compose-ból a Kubernetesbe

### 5.1. Bevezetés

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 jelennek meg, mint a Prometheus a felügyelethez, a Grafana a vizualizációhoz, több szimulátorszolgáltatás és egy vezérlőszerver. Itt bemutatom a Docker Compose konfigurációk Kubernetes manifeszttekbe való átforgatásának kihívásait.

### 5.2. A Docker Compose és Kubernetes áttekintése

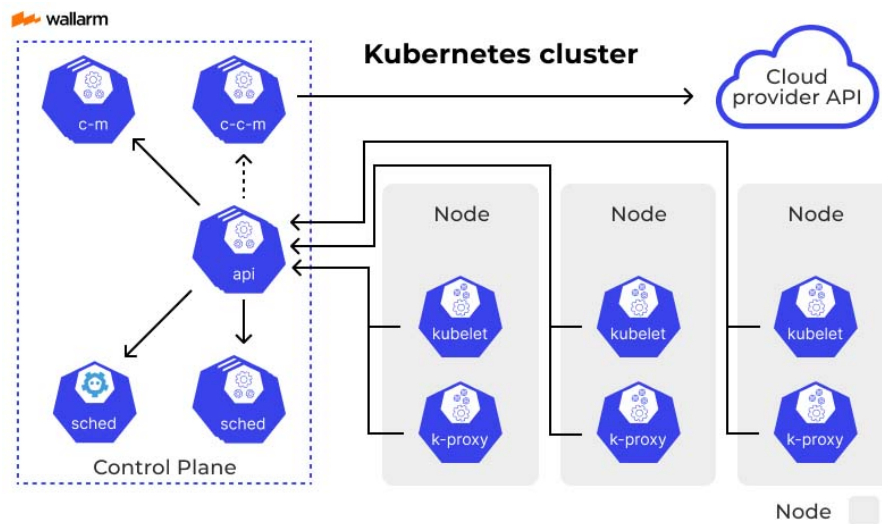
Docker Compose Ezzel több konténert tartalmazó Docker alkalmazásokat lehet definiálni és futtatni. Konfigurációja egy YAML fájlban tárolt, ahol a szolgáltatásokat, hálózati kapcsolatokat, köteteket és függőségeket lehet megadni. A Docker Compose leegyszerűsíti a konténerek egyetlen hoszton történő orkesztrációját,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

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

- **Pod:** Ez a legkisebb telepíthető egység, amely egy vagy több konténert foglal magukba.
- **Deployment:** Állapot nélküli alkalmazások kezelésére szolgáló objektumok, amelyek olyan funkciókat kínálnak, mint a gördülő frissítések és a visszaállítás.

- **Service:** Végpontokat biztosítanak a podok eléréséhez, segítve a felfedezést és a terheléelosztást.
- **ConfigMap és Secret:** Mechanizmus a konfiguráció és az imagek szétválasztására.
- **PersistentVolumeClaim (PVC):** Absztrakció adattárolásra.

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



5.1. ábra. Kubernetes architektúra [1]

## 5.3. Rendszerarchitektúrája

A rendszer a már megismert következő részeket tartalmazza:

- **Prometheus:** Egyéni prometheus.yml fájljal 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. [2]

## 5.4. A Docker Compose beállítások konvertálása Kubernetes manifeszteké

A Docker Compose-ról a Kubernetesre való áttérés magában foglalja az alkalmazás architektúrájának újragondolását a podok, deployment-ek, szolgáltatások és más Kubernetes objektumok szerint. [19]

### 5.4.1. 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']
```

### 5.4.2. 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 deployment-ekké alakulnak át, amelyek környezeti változókat és portkonfigurációkat adnak meg.

### 5.4.3. 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
```

#### 5.4.4. 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.

#### 5.4.5. 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:

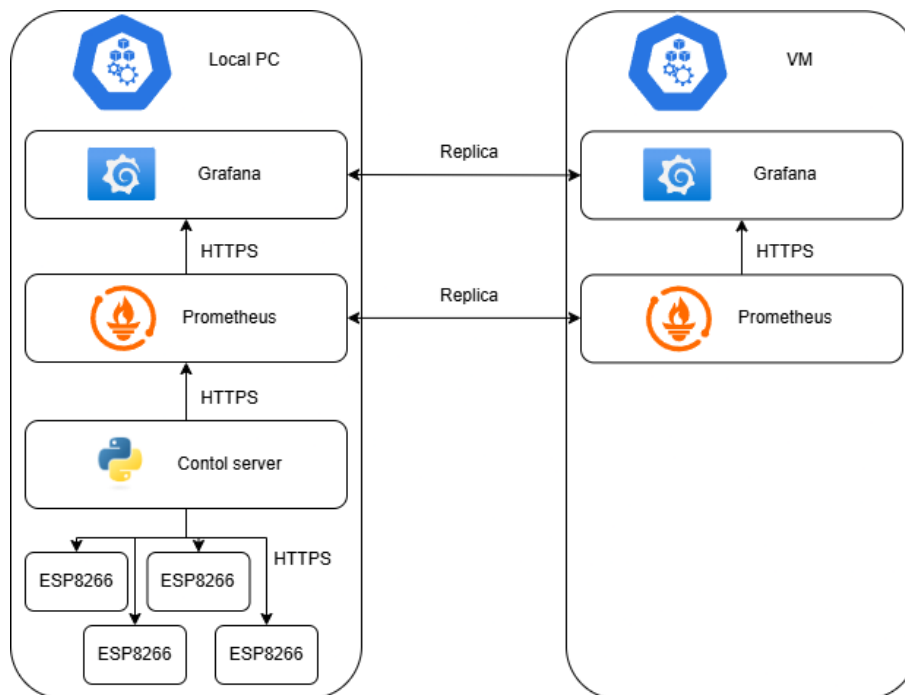
```
kubectl get pods -n monitoring
```

### 5.5. 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.



5.2. ábra. Hibrid kubernetes topológia

### 5.5.1. 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 deployment-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 }
```

### 5.5.2. KubeADM

A projektemben a kubeadm-re támaszkodtam, hogy kubernetes klasztert készítsek a linux vm-et bevonva. Ez megkönnyítette a folyamatot mert magasabb szintű 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 token kellett adni a nodenak, hogy csatlakozzon a clusterhez. Ezután 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.

## 6. fejezet

# Felhasználói felület

### 6.1. Szöveges be- és kimenetek a szimulációhoz

#### 6.1.1. Cél és áttekintés

A rendszer szöveges alapú beállításra és eredménygyűjtésre alkalmas. A cél, hogy a szimuláció *kiegészítő eszközei* (pl. curl, CSV-konverzió) nélkül, egyszerű szövegfájlokkal legyen vezérelhető és kiértékelhető.

Rövid összefoglaló:

- **Bemenetek:** thresholds.txt, esp{1..3}\_schedule.txt, sim\_control.txt
- **Kimenet/napló:** output.txt (idősoros; egy sor = egy vezérlési ciklus)
- **Webes felület:** „Dev Panel” (localhost:8080) a fájlok szerkesztéséhez, generálásához, letöltéséhez, a futás indításához/megállításhoz, az idő nullázásához és a napló törléséhez.

#### 6.1.2. Bemeneti szövegfájlok

##### 6.1.2.1. thresholds.txt – küszöbök és maximum megengedhető áram

A vezérlő szerver minden ciklusban beolvassa. Kulcs-érték párok, tizedes ponttal:

```
# Küszöbértékek a vezérlő szerverhez
BREAKER_MAX_TOTAL=65.0    # [A] - Megszakító lekapcsolási áram
BREAKER_MIN_TOTAL=35.0    # [A] - Megszakító bekapcsolási áram
ALLOC_MAX_TOTAL=95.0      # [A] - Max áram érték
```

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.



### 6.1.2.2. `esp{1..3}_schedule.txt` – idősoros bemenet

Formátum: időpillanat másodpercben + kívánt áram (A). A menetrend *lépcsős*: a legutóbbi időponthoz tartozó érték érvényes a következő megadásig.

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

Irányelvek: tizedes elválasztó pont; tetszőleges szóköz; a sorok idő szerint rendezve mint minden szöveges ki- és bemeneti file-ban.

### 6.1.2.3. `sim_control.txt` – futtatási állapot

Egyetlen szó: RUNNING vagy STOPPED (Az alapértelmezés STOPPED). A SIM-ek „virtuális órája” csak RUNNING állapotban megy.

## 6.1.3. Kimeneti szövegfájl

### 6.1.3.1. `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 fejléct. 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 összárám (cap után).
- `alloc_max_total_amps` – allokációs keret (A).
- `max_total_amps / min_total_amps` – breaker küszöbök (legacy nevek).
- `sims` – | jellel szeparált lista SIM-enként:  
    `espX:raw=..., effective=..., cap=...`  
    ahol `raw` = menetrendi igény, `effective` = tényleges áram, `cap` = küldött maximum.
- `breakers` – megszakítók állapota on/off, vesszővel elválasztva.

#### 6.1.4. Időkezelés és futtatás

- **Virtuális idő:** minden SIM saját menetrendi ideje csak RUNNING állapotban növekszik.
- **STOPPED** módban a SIM-ek ideje megáll; a vezérlő nem küld új cap-et és nem kapcsolgat megszakítót, csak mér és naplóz.
- **Reset (t=0):** a Dev Panel „Reset sim time (t=0)” gombja az összes SIM virtuális idejét nullázza (a panel előbb STOP-ra állít, majd resetel).

#### 6.1.5. Reprodukálhatóság és feldolgozhatóság

A bemenetek (küszöbök, menetrendek, futtatási állapot) veriozhatók 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).

#### 6.1.6. Rövid példa – beállítás → kimenet (részlet)

**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 a nyers összeg  $150\text{ A} > 9\text{ A}$ . 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.

## 7. fejezet

# Max–min fair (water-filling) elosztás

### 7.1. Elméleti háttér és cél

#### 7.1.1. Motiváció és cél

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

#### 7.1.2. Definíció (max–min fair)

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

#### 7.1.3. Vízbetöltés (water-filling)

A max–min fair kiosztás felírható egyetlen paraméterrel:

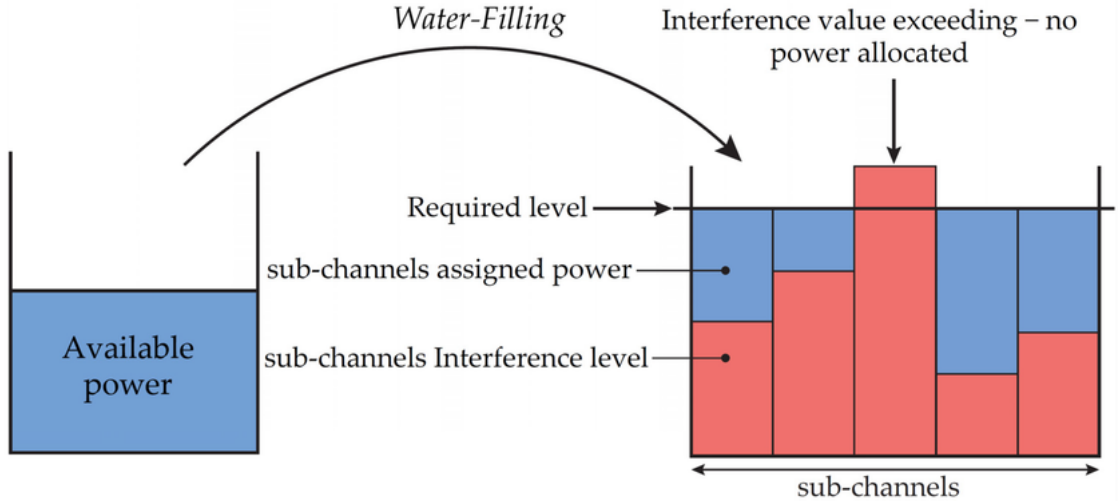
$$a_i = \min\{d_i, \lambda\}, \quad \text{ahol} \quad \sum_{i=1}^n \min\{d_i, \lambda\} = B. \quad (7.1)$$

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

#### 7.1.4. Algoritmus és bonyolultság

Gyakorlati, determinisztikus eljárás (progresszív töltés):

1. Rendezzük az igényeket növekvő sorrendbe:  $d_{(1)} \leq \dots \leq d_{(n)}$ .



7.1. ábra. Water-filling elve telekommunikációban. [16]

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.

### 7.1.5. Tulajdonságok

- **Egyenlő szint elve:** a  $\lambda$  alatti igények teljes, a  $\lambda$  felettiek  $\lambda$ -ig kapnak. Így a kis igényűek sosem szenvednek hátrányt.
- **Monotonitás:** ha a keret  $B$  nő, akkor  $\lambda$  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  $a_i = 0$ .

## 7.2. A vezérlőben alkalmazott megvalósítás

### 7.2.1. Kapcsolat a rendszer komponenseivel

A vezérlő *nyers* igényekből (raw\_current) számolja a cap-eket a fenti elv szerint a ALLOC\_MAX\_TOTAL kereten. A *megszakító* (breaker) logika ettől független, a *mért, hatásos* összához viszonyít (BREAKER\_MAX\_TOTAL, BREAKER\_MIN\_TOTAL) biztonsági réteggént.

### 7.2.2. 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.

### 7.2.3. Implementációs részletek

A cap-ek csak  $\pm 10^{-3}$  A változás felett frissülnek a SIM-ek felé (zajcsillapítás), a „nincs korlát” állapotot nagy INF\_CAP érték reprezentálja. Ha a nyers összeg a keret alá esik, a cap-ek feloldódnak.

## 8. fejezet

# Fejlesztői panel (Dev Panel)

### 8.1. Cél és szerep

A Dev Panel egy könnyű, belső használatú webes felület, amely a szöveges bemenetek és kimenetek kezelését, a futtatás indítását/megállítást, 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` kötetben lévő fájlokat kezeli (atomikus írással), és hálózaton hívja a SIM konténerek belső `/reset_time` végpontjait. A vezérlő külön, a saját portján (8000) fut; a Prometheus és Grafana eléréséhez gyorslinkek állnak rendelkezésre.

### 8.3. Fő funkciók és munkamenet

**Start/Stop.** A `sim_control.txt` állományba írja a panel a RUNNING/STOPPED értéket. STOP módban a SIM-ek virtuális ideje megáll, a vezérlő nem küld cap-et és nem kapcsol megszakítót, csak mér és naplóz.

**Reset (t=0).** A panel a STOP beállítás után meghívja minden SIM `/reset_time` végpontját; a virtuális idő nulláról indul (RUNNING állapotban onnantól ismét telik).

**Clear output.txt.** A `data/output.txt` törlése. A vezérlő a következő ciklusban automatikusan újra létrehozza a fejléceket és folytatja a naplózást.

**Thresholds szerkesztés.** A `thresholds.txt` beolvasása/írása a panelről: BREAKER\_MAX\_TOTAL, BREAKER\_MIN\_TOTAL, ALLOC\_MAX\_TOTAL.

**Menetrend-generátor.** Konstans, rámpa, lépcső, szinusz és random walk idő-sorok képezhetők a `esp{1..3}_schedule.txt` fájlokba (formátum: `seconds amps`). A generált tartalom előnézetben ellenőrizhető.

**Raw editor & Letöltés.** Tetszőleges bemeneti fájl közvetlen szerkeszthető; az `output.txt` csak olvasható. Minden bemenet és a kimenet letölthető archiváláshoz.

## 8.4. Backend API (rövid kivonat)

Végpont	Funkció
GET /api/read?name=...	Fájl beolvasása (csak engedélyezett lista)
POST /api/write	Fájl írása (atomikus csere, <code>output.txt</code> kivétel)
GET /api/download?name=...	Közvetlen letöltés
GET /api/sim_state	Globális állapot lekérdezése
POST /api/sim_state	RUNNING/STOPPED beállítása
POST /api/clear_output	<code>output.txt</code> törlése
POST /api/reset_sim_time	Minden SIM időnullázása

## 8.5. Megbízhatósági és használhatósági szempontok

- **Atomikus írás:** ideiglenes fájl + átnevezés, így elkerülhetők a félig írt állapotok.
- **Önjavító napló:** a vezérlő törlés után fejléceket generál, a naplózás folytatódik.
- **Deklaratív bemenet:** minden konfiguráció egyszerű, verziózható szövegfájl.

## 8.6. Biztonsági és korlátok

A panel *belső* használatra készült. Nincs többfelhasználós jogosultság- és CSRF-kezelés; éles környezetben ezeket pótolni szükséges. A fájlműveletek engedélyezett listához kötöttek, a végpontok nem tesznek lehetővé tetszőleges fájlhozzáférést.

## 8.7. Kiterjeszthetőség

A panel könnyen bővíthető: pl. súlyozott fair-elosztás bemenete (`weights.txt`), előre definiált menetrend-sablonok, vagy beépített grafikon a `output.txt` vizualizálására. A funkcionalitás változtatása különösen egyszerű, mivel az állapot *szöveges fájlokban* van deklarálva, és a vezérlő *idempotens* módon dolgozza fel.

## Összegzés

A Dev Panel egységes bejáratot ad a szöveges I/O-hoz és a futtatási műveletekhez, miközben a max-min fair elosztás biztosítja, hogy a rendelkezésre álló keret igazsá-

gosan és determinisztikusan oszoljon meg a szimulált fogyasztók között. A megoldás kifejezetten kutatási és tesztelési használatra optimalizált: egyszerű, átlátható, és könnyen megismételhető futásokat tesz lehetővé.



# 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] Docker Inc.: Deploy on kubernetes with docker desktop, 2025. URL <https://docs.docker.com/desktop/features/kubernetes/>. Megnyitva: 2025-04-05.
- [3] Ecostruxure™ power monitoring expert. <https://www.se.com/hu/hu/product-range/65404-ecostruxure-power-monitoring-expert/#overview>, 2025. Megnyitva: 2025-03-17.
- [4] 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.
- [5] Schneider Electric: Masterpact mtz product range, 2025. URL <https://www.se.com/hu/hu/product-range/63545-masterpact-mtz/#products>. Megnyitva: 2025-05-18.
- [6] 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.
- [7] ElectronicWings: Nodemcu development kit/board, 2023. URL <https://www.electronicwings.com/nodemcu/nodemcu-development-kitboard>. Megnyitva: 2025-05-01.
- [8] 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.
- [9] 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.

- [10] 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.
- [11] Creator name or YouTube channel name if known: Title of video, 2025. URL [https://www.youtube.com/watch?v=pcGg-U5d\\_n8](https://www.youtube.com/watch?v=pcGg-U5d_n8). Megnyitva: 2025-03-25.
- [12] OpenEnergyMonitor: Interface with arduino, 2025.  
URL <https://docs.openenergymonitor.org/electricity-monitoring/ct-sensors/interface-with-arduino.html>. Megnyitva: 2025-03-22.
- [13] 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.
- [14] Prometheus: Prometheus - dimenzionális adatok: kulcs-érték párokon alapuló modell, 2025. URL <https://prometheus.io/>. Megnyitva: 2025-03-25.
- [15] Simatic energy management software. <https://www.siemens.com/global/en/products/automation/industry-software/automation-software/energymangement.html>, 2025. Megnyitva: 2025-03-17.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] The Kubernetes Authors: Kubernetes documentation, 2025.  
URL <https://kubernetes.io/>. Megnyitva: 2025-04-05.
- [20] Doc Walker: Modbusmaster: Arduino library for modbus communication, 2016.  
URL <https://github.com/4-20ma/ModbusMaster>. Megnyitva: 2025-04-07.