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. október 16.

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. október 16.

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 megfelelőséget 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 skálázhatóság, 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

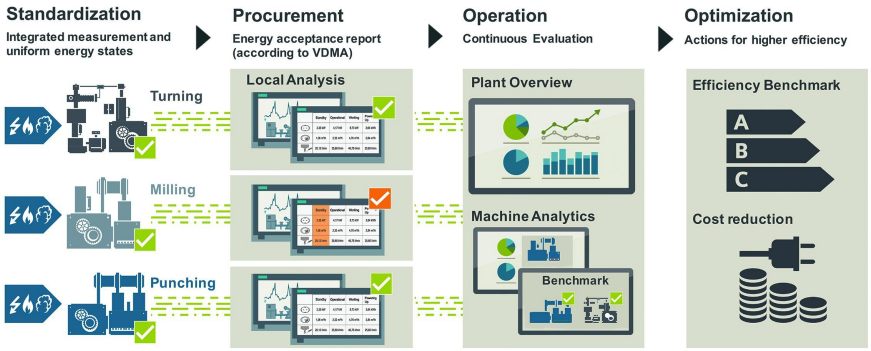
2.1.1. Schneider Power Monitoring Expert

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.

[?]

2.1. ábra. Schneider Electric PME model[?]



2.2. ábra. Siemens EMS model[**?**]

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.[**?**]

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.[**?**]

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. Saját megoldás

Egy mondatban: a saját eszközkészletem (ESP-8266 + Prometheus + Grafana + Python) sokkal olcsóbb és könyebben 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 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.

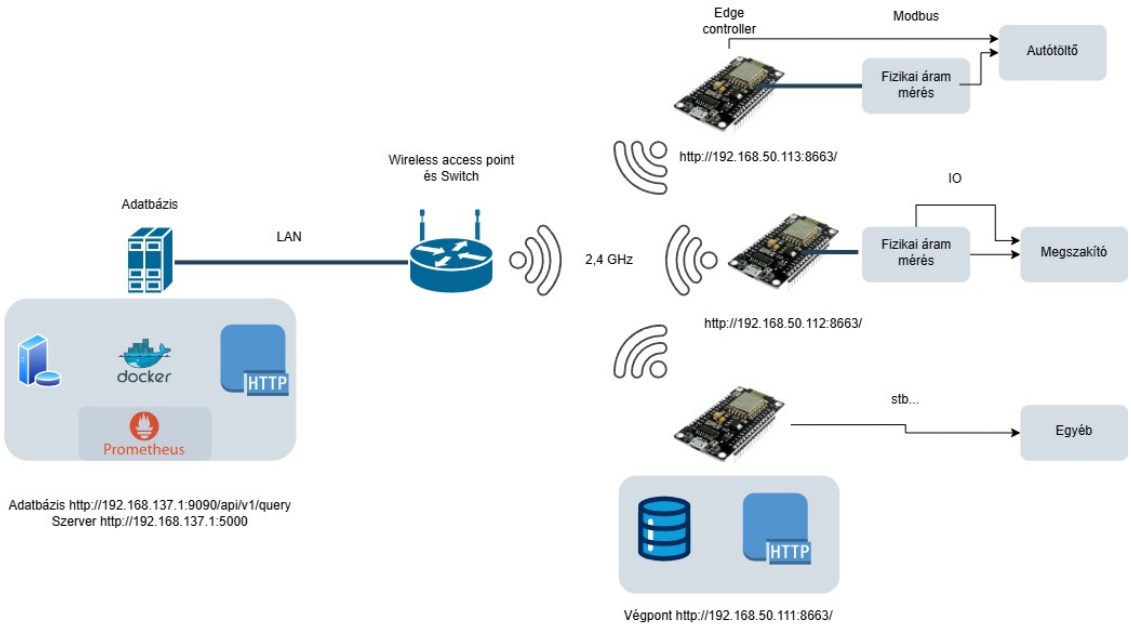
Az ESP8266 csomópontok csak az adatok két oldalú továbbadásáért felelősek, aktuális adatokat küldenek a szervernek, míg a Flask szerver döntéshozatalt hajt végre és parancsokat ad ki a töltőknek. Az érzékelés és a vezérlés szétválasztása leegyszerűsíti a végpont tervezését és a feldolgozást a szerver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

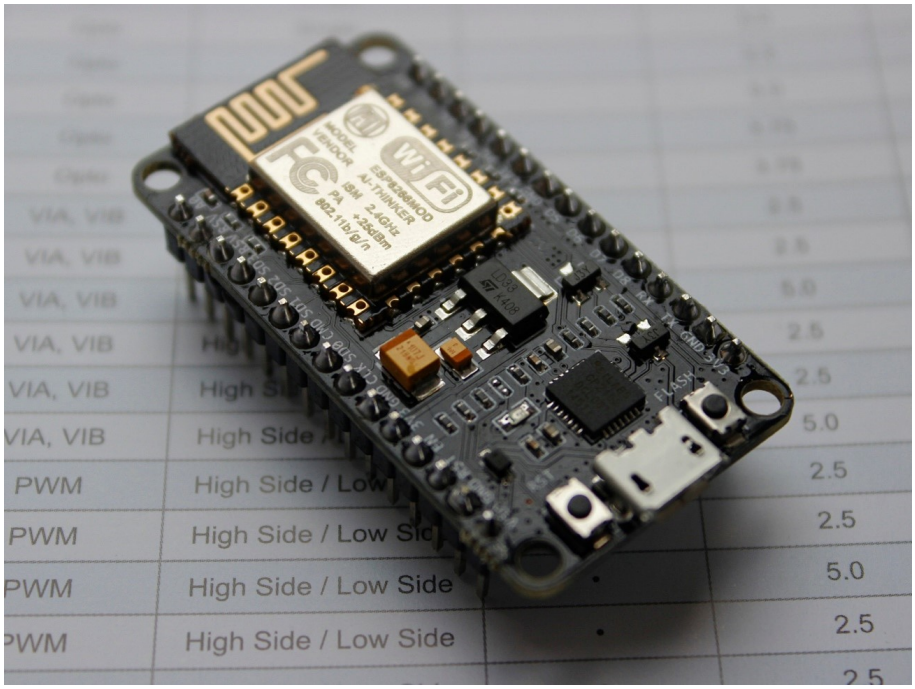
3.2. Eszközök

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 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. [**?**]



3.2. ábra. NodeMCU (ESP8266) [**?**]

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. [**?**]

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. [**?**]

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.

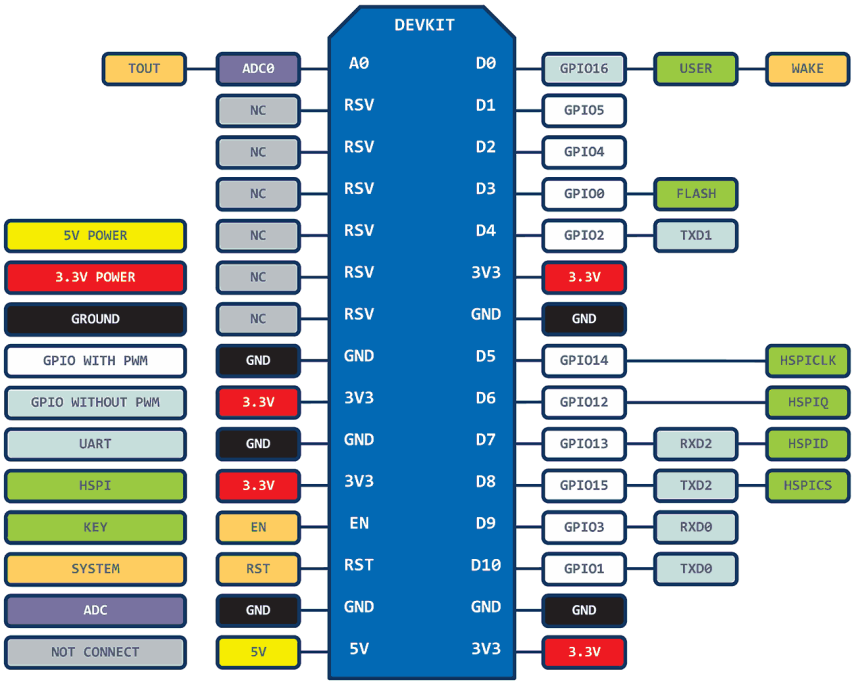
3.3. ábra. SCT-013 áramváltó [**?**]

3.2.1.2. Mért eszközök

Különböző eszközök mérését hajtok 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 [**?**]

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.

3.3. 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. [**?**] 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:

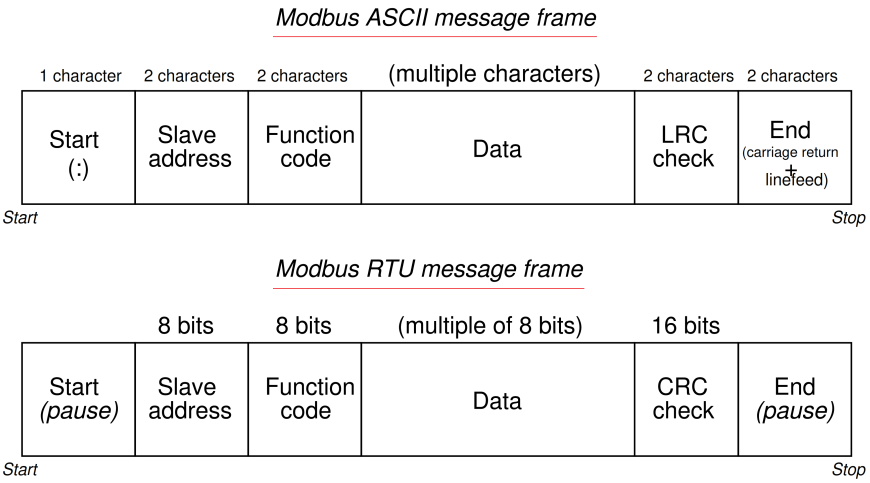
```
def metrics():
return jsonify({
    "simulator_id": simulator_id,
    "current": current_value,
    "state": "plugged in" if charger_on else "plugged out",
    "max_current": max_current
})
```

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 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.

3.3.2. 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. [**?**]

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 [**?**]

3.4. Standardizált rendszer

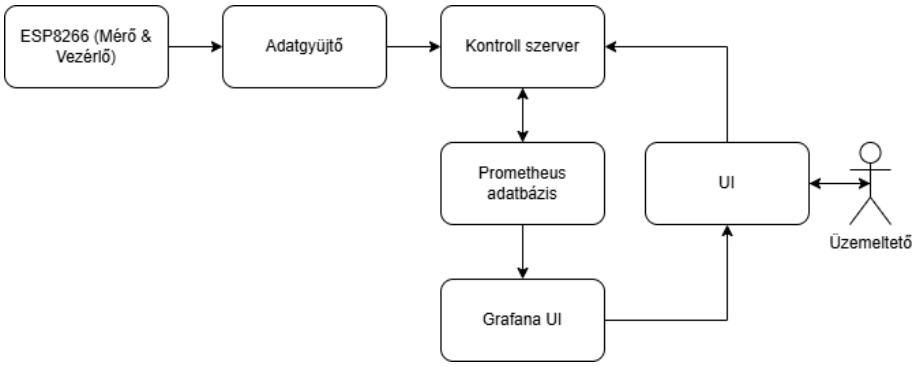
3.4.1. Áttekintés

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 orchestrálást, 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: mindegyik 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 ConfigMapeken 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 kibőtve külső linux VM-re.

4. fejezet

Max–min fair (water-filling) elosztás

4.1. Elméleti háttér és cél

4.1.1. Motiváció és cél

A szimulált fogyasztók áramigénye (d_i) időben változik. Adott egy globális, maximum áramérték $B = \texttt{ALLOC\_MAX\_TOTAL}$ 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.

4.1.2. 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. [**?**]

4.1.3. Feltöltés (water-filling)

A max–min fair kiosztás felírható egyetlen paraméterrel:

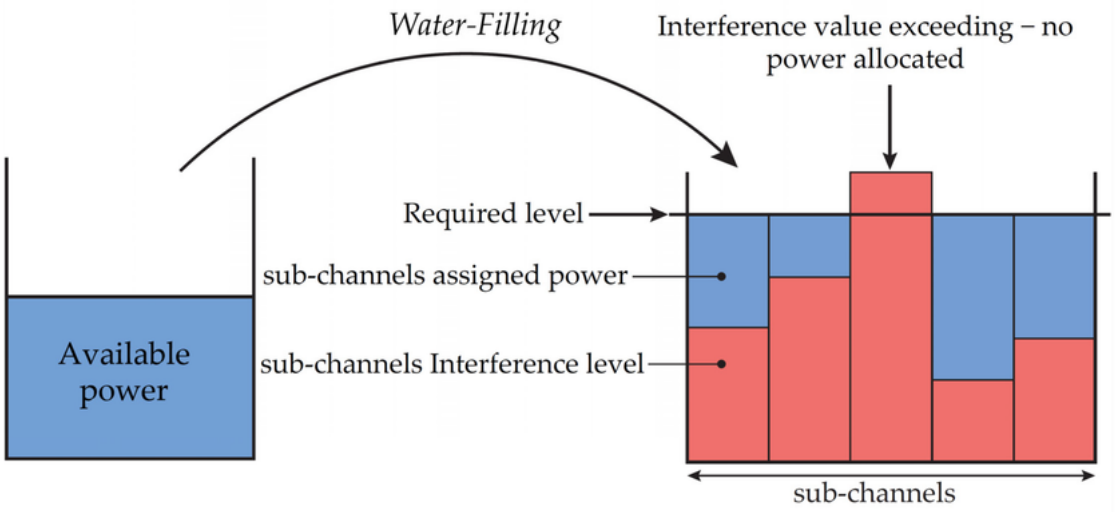
$$a_i = \min\{\, d_i,\ \lambda \,\}, \qquad \text{ahol} \ \sum_{i=1}^{n} \min\{d_i, \lambda\} \ = \ B. \tag{4.1}$$

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).

4.1.4. Algoritmus és bonyolultság

Gyakorlati, determinisztikus eljárás (progresszív töltés):

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



4.1. ábra. Water-filling elve telekomunikációban. [**?**]

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.

4.1.5. 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 $a_i = 0$.

4.2. A vezérlőben alkalmazott megvalósítás

4.2.1. 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.

4.2.2. 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.

4.2.3. 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. fejezet

Komponensek megvalósítása

5.1. Végpontok

5.1.1. Autótöltő

5.1.1.1. 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.

5.1. ábra. Autótöltő [**?**]

5.1.1.2. Megvalósítás

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.

5.1.1.3. 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.

5.1.1.4. 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.

5.1.1.5. 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.

5.1.1.6. 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 [**?**] 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.

5.1.2. Megszakító

5.1.2.1. 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.

5.1.2.2. 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).

5.2. ábra. Megszakító [**?**]

5.2. Kontroll szerver

5.3. 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. [**?**] [**?**]

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

5.3. ábra. Prometheus [**?**]

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']).
```

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 konfugorációs 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 gyújt 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 metrikanévvel 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.

5.3.1. 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);
```

5.3.2. 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 másodpercre állítottam, hogy ne terhelje a próbák során feleslegesen a hálózatot, de gyorsabb válaszidő érdekében ez növelhető.

5.4. 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.

5.4.1. 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
```

5.4.1.1. 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"}
```

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.

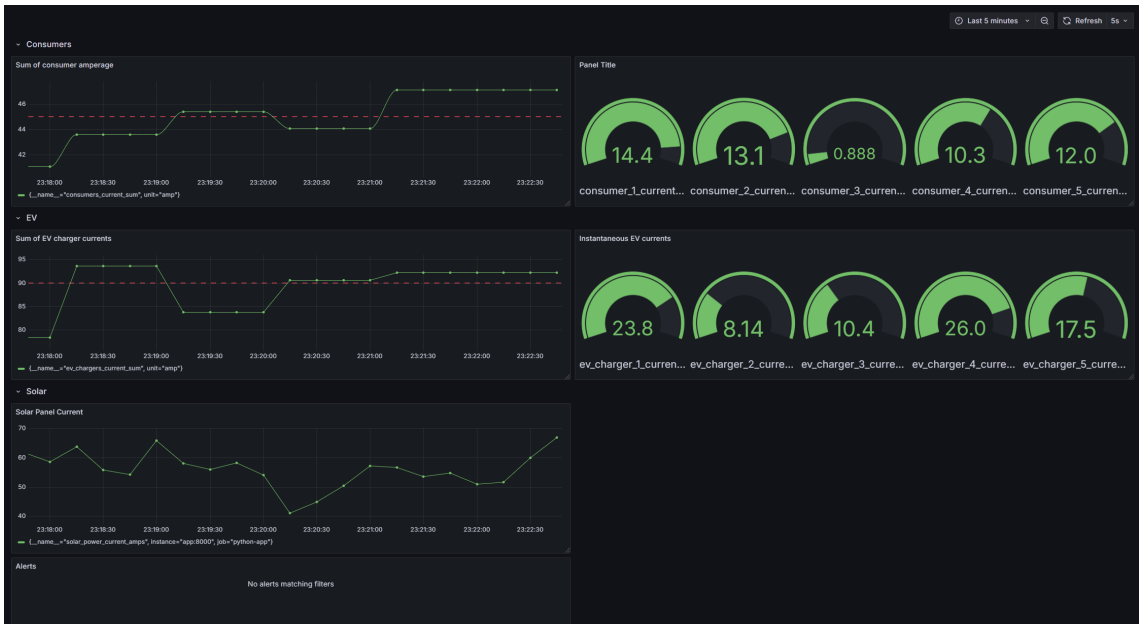
A panel hasonlóan az előző ponthoz jeleníti meg az adatokat, amely a pillanatnyi és max értéket mutatja.

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

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
```



5.4. ábra. Általam készített Grafana dashboard

6. fejezet

Alkalmazás migrálása a Docker Compose-ból a Kubernetesbe

6.1. 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 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.

6.2. A Docker Compose és Kubernetes áttekintése

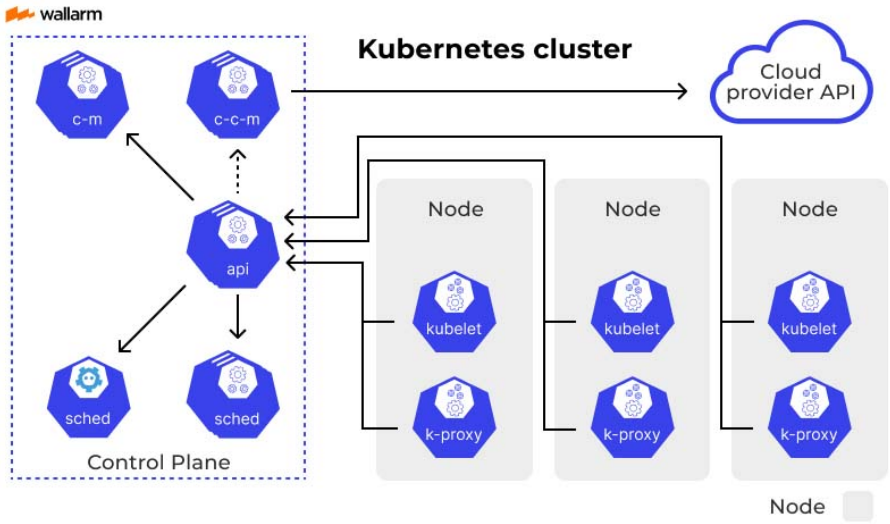
Docker Compose Ezzel több konténert tartalmazó Docker alkalmazásokat lehet definiálni és futtatni. Konfigurációja egy YML fájlban tárolt, ahol a szolgáltatásokat, hálózatokati kapcsolatokat, köteteket és függőségeket lehet megadni. A Docker Compose leegyszerűsíti a konténerek egyetlen hoszton történő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

Kubernetes A Kubernetes 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 foglalnak 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 terheléselosztást.
- **ConfigMap és Secret:** Mechanizmus a konfiguráció és az imagek szétválasztására.
- **PersistentVolumeClaim (PVC):** Absztrakció adattárolásra.

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



6.1. ábra. Kubernetes architektúra [**?**]

6.3. Rendszerarchitektúrája

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. [**?**]

6.4. 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. [**?**]

6.4.1. 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']
```

6.4.2. 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 deployement-ekké alakulnak át, amelyek környezeti változókat és portkonfigurációkat adnak meg.

6.4.3. 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
```

6.4.4. 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.

6.4.5. 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:

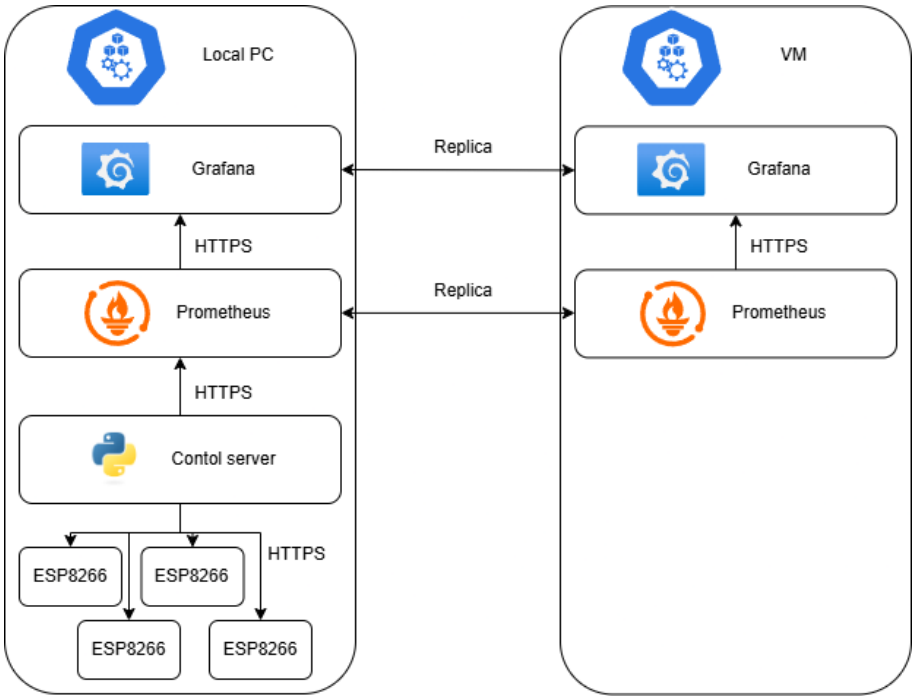
```
kubectl get pods -n monitoring
```

6.5. 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.



6.2. ábra. Hibrid kubernetes topológia

6.5.1. 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 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 }
```

6.5.2. 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.

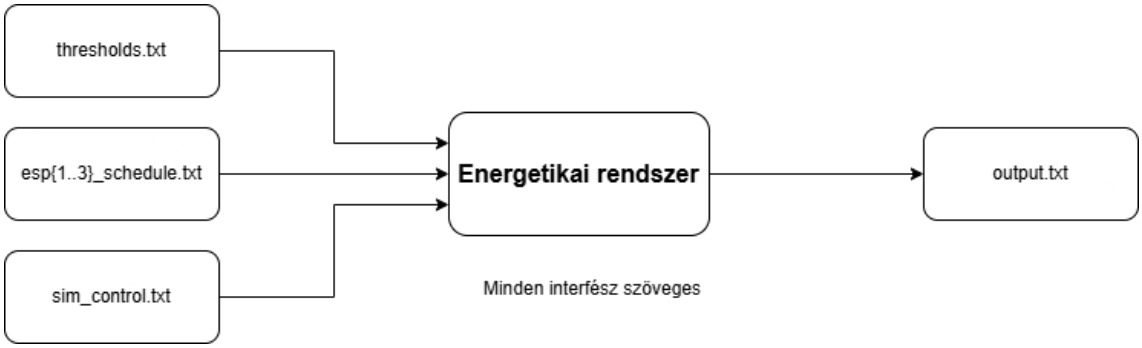
7. fejezet

Szöveges interfészek a szimulációhoz

7.1. Cél és áttekintés

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

7.2.1. `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.

7.2.2. `esp{x}_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.

7.3. `sim_control.txt` – futtatási állapot

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

7.4. Kimeneti szövegfájl

7.4.1. `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 SIM-enként:
  `espX:raw=..., effective=..., cap=...`
  ahol `raw` = menetrendi igény, `effective` = tényleges áram, `cap` = küldött maximum.
- `breakers` – megszakítók állapota `on`/`off`, vesszővel elválasztva.

7.5. 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).

7.6. 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).

7.7. 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 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. Fő funkciók és munkamenet

Start/Stop. A `sim_control.txt` fájlba írja a panel a `RUNNING` vagy `STOPPED` értéket. STOP módban az esp szimulátorok virtuális ideje megáll, a vezérlő nem küld max értékeket és nem kapcsol megszakítókat, csak mérést és naplózást végez.

Reset (t=0). A panel a STOP beállítás után meghívja minden esp szimulátor `/reset_time` végpontját, a virtuális idejük nulláról indul újra.

Clear output.txt. A `data/output.txt` törlése. A vezérlő a következő ciklusban automatikusan újra létrehozza a fejlécet 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, fel- és lefutás, lépcső, szinusz és random walk idősorok képezhetők a `esp{x}_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 be- és a kimenet letölthető megőrzéshez.

8.4. Backend API (elérések)

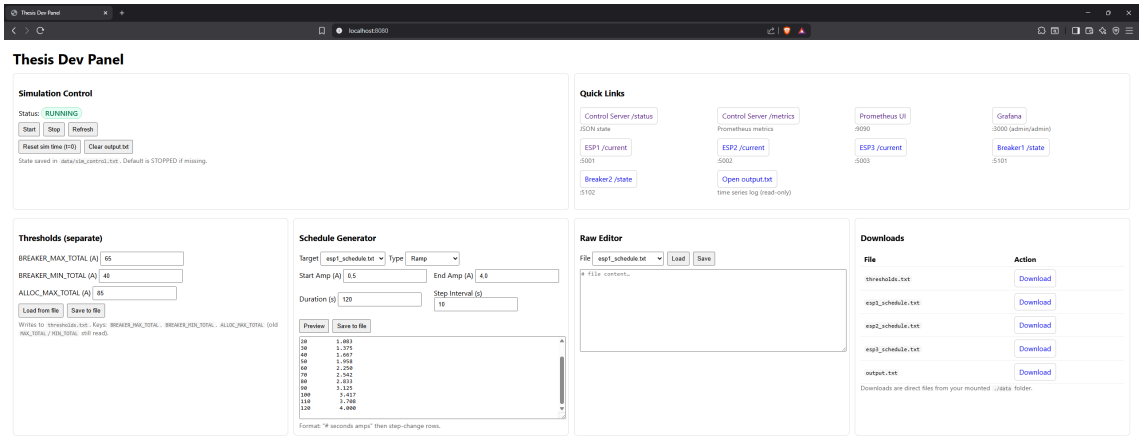
Végpont	Funkció
`GET /api/read?name=...`	Fájl beolvasása
`POST /api/write`	Fájl írása
`GET /api/download?name=...`	Közvetlen letöltés
`GET /api/sim_state`	Globális állapot lekérdezése
`POST /api/sim_state`	RUNNING/STOPPED beállítása
`POST /api/clear_output`	`output.txt` törlése
`POST /api/reset_sim_time`	Minden szimulátor időnullázása

8.5. 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.

8.6. Kiterjeszthetőség

A panel könnyen bővíthető új be- és kimenetekkel: 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.



8.1. ábra. Devpanel

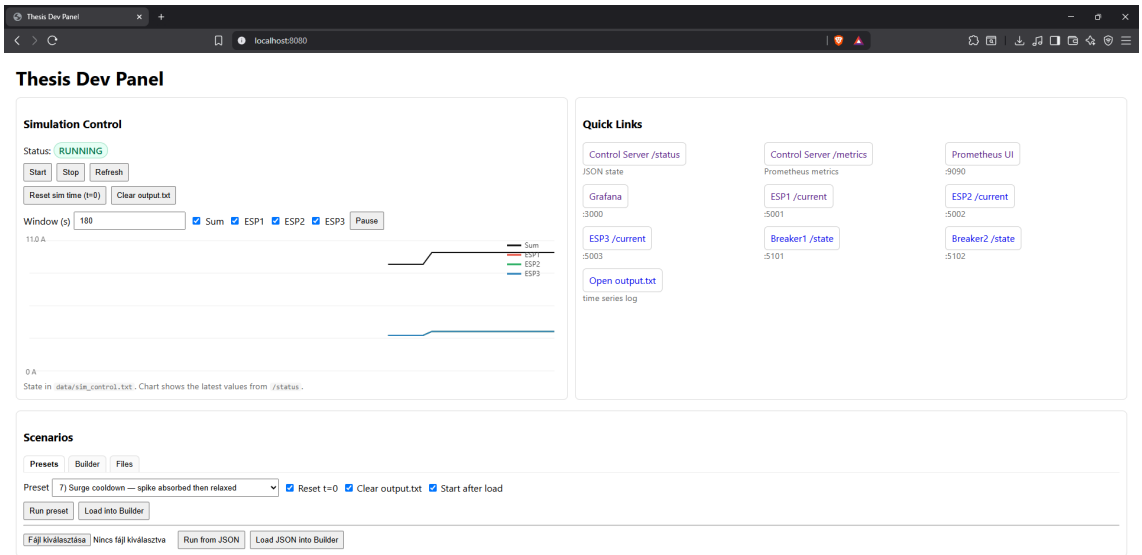
8.7. Dev Panel módosítások

8.7.1. Motiváció és cél

A fejlesztői panelt a készítésnek ebben a fázisában úgy bővítettem, hogy a tesztesetek egy kattintással reprodukálhatóak legyenek, a kézi teszt generátor pedig ugyanazon a felületen, azonnali futtatással történjen és opciót adjon a kimentésre . A fő cél: *egységes teszteset kezelés* (presetek, és könnyen generálható egyedi scenariók), *automatizált tesztelés*, és *idősoros megfigyelés* egy nézetben.

8.7.2. Fő fejlesztések

- **Egységesítettem a *Scenarios* kártyát** három füllel: *Presets*, *Builder*, *Files*.
- **Auto Test Runner**: beépített demók (presetek) egy kattintással futtathatóak, illetve betölthető és kimenthető JSON-ba egy teljes scenarió.
- **Összevont szerkesztés**: a menetrend-generátor és a nyers szövegszerkesztő panelt egységesítettem (*Builder*).
- **Scenarió-könyvtár**: megvalósítottam a teljes tesztet magába foglaló JSON mentését sajátgépre vagy szerverre. (`/data/scenarios/`).
- **Atomikus írás és állapotkezelés**: bemeneti fájlok biztonságos felülírása, `sim_control.txt` kezelés, opcionális *reset* és naplótörlés.
- **Élő grafikon és gyorshivatkozások**: valós idejű összáram és szimulátoronkénti külön görbe, közvetlen linkek is itt találhatóak meg a `/status`, `/metrics`, Prometheus, Grafana nézetekhez.



8.2. ábra. Devpanel v2

8.7.3. Felépítés és API változások

A Dev Panel backend része új végpontokkal bővült, ezek:

- `POST /api/run_scenario` - teljes scenarió alkalmazása (küszöbök + menetrendek + vezérlési opciók).
- `GET/POST /api/read`, `/api/write` - bemeneti állományok (txt) kezelése.
- `POST /api/clear_output`, `POST /api/reset_sim_time`, `POST /api/sim_state`.
- `GET /api/list_scenarios`, `GET /api/read_scenario`, `GET /api/download_scenario`, `POST /api/save_scenario_json`, `DELETE /api/delete_scenario`.

Az általam használt JSON séma a tesztek definiálására:

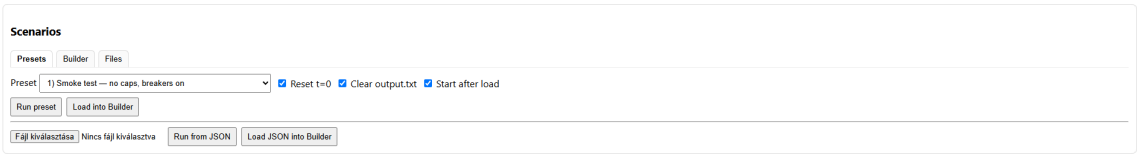
```
{
  "name": "Demo",
  "thresholds": {
    "BREAKER_MAX_TOTAL": 12,
    "BREAKER_MIN_TOTAL": 2,
    "ALLOC_MAX_TOTAL": 9
  },
  "schedules": {
    "esp1": [[0, 50], [60, 10]],
    "esp2": [[0, 50], [60, 10]],
    "esp3": [[0, 50], [60, 100]]
  },
  "control": {
    "reset_time": true,
    "clear_output": true,
    "start_after_load": true
  }
}
```

A `/api/run_scenario` hívás menete:

1. `sim_control.txt` → `STOPPED`.
2. Opcionálisan lehet törölni az `output.txt` tartalmát, szimulátor időket reset-elni *reset*.
3. `thresholds.txt`, `esp{1..3}_schedule.txt` újra írása.
4. Végállapot a futattás után: `RUNNING` vagy `STOPPED` (beállítástól függően).

8.7.4. UI szervezés (*Scenarios* kártya)

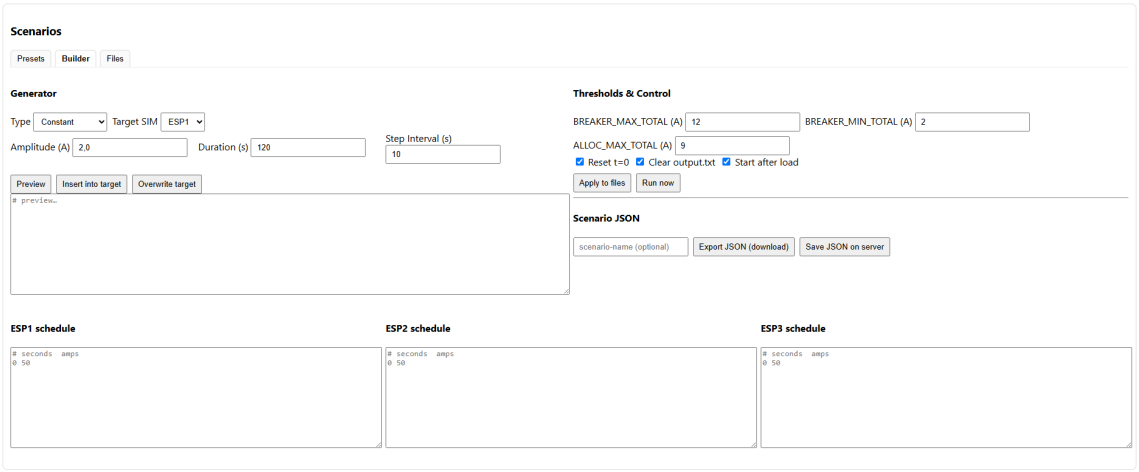
Presets. Előre definiált demókat tartalmaz (smoke test, under budget, fair split 3/3/3, dinamikus újraelosztás, hiszterézis, STOPPED invariánsok) Ezek futtathatóak az említett opciókkal: *Reset t=0*, *Clear output.txt*, *Start after load*. Külső JSON is betölthető/futtatható.



8.3. ábra. Devpanel Presets fül

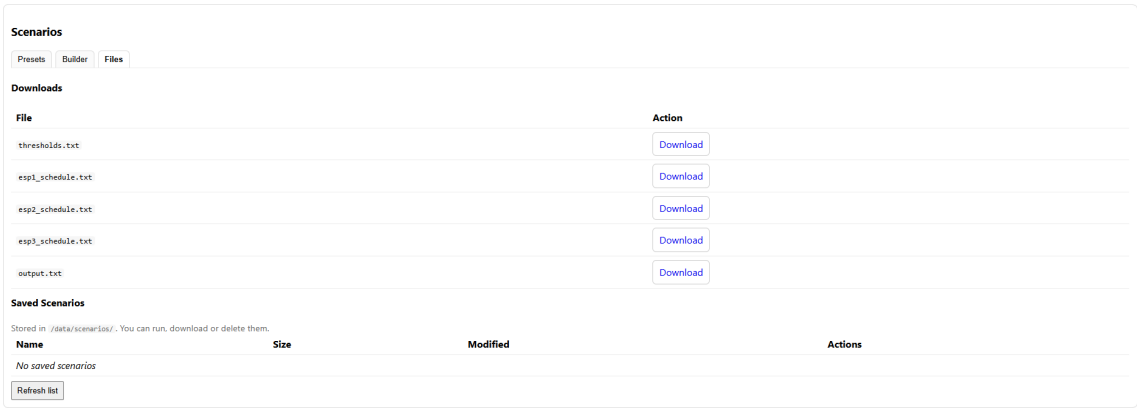
Builder. Max értékek (`BREAKER_MAX_TOTAL`, `BREAKER_MIN_TOTAL`, `ALLOC_MAX_TOTAL`) és vezérlési opciók mellett szerkesztőablakok a menetrendekhez. Különböző menetrend típusok választhatóak (állandó, rámpa, lépcső, szinusz, random walk) *Insert/Overwrite* móddal. Kettő opció van ezután:

- *Apply to files*: csak fájlok írása (nem indul a szimuláció).
- *Run now*: fájlok írása és azonnali futtatás az opciók szerint.



8.4. ábra. Devpanel Builder fül

Files. Gyors letöltések a txt-állományokhoz, illetve a szerveren tárolt scenariók listája (futtatás, letöltés, betöltés a Builderbe, törlés).



8.5. ábra. Devpanel Files fül

8.7.5. Használati munkafolyamat

A kipróbáláshoz javasolt lépések folyamata:

1. *Presets*: kiválasztás → opciók (*Reset/Clear/Start*) → *Run preset*.
2. *Builder*: menetrend generálása/szerkesztése → *Run now*.
3. *Export/Save*: scenarió JSON letöltése vagy mentése saját gépre vagy a szerverre későbbi felhasználás céljából.
4. *Files*: `output.txt` letöltése eredmények tárolásához.

9. fejezet

Rendszertesztek és bemutató szcenáriók

9.1. 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 $T_c = 3\,\mathrm{s}$.

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.

9.2. Bemenetek és állapot

- `thresholds.txt`:
  - `BREAKER_MAX_TOTAL`
  - `BREAKER_MIN_TOTAL`
  - `ALLOC_MAX_TOTAL`

  Ezek fentebb említett módon kerülnek használatra.
- idő - áramerősség párokat tartalmazzó menetrendek minden végponthoz:
  - `esp1_schedule.txt`
  - `esp2_schedule.txt`
  - `esp3_schedule.txt`
- `sim_control.txt`:
  - RUNNING a szimulációhoz használt merők belső órája megy
  - STOPPED a szimulációhoz használt mérők belső órája megáll

  alapértelmezés: STOPPED.

9.3. 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.

9.4. Szcenáriók és elfogadási kritériumok

9.4.1. Alaptesztek: Start/Stop/Reset/Clear

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: $\text{effective}_i = d_i$.

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

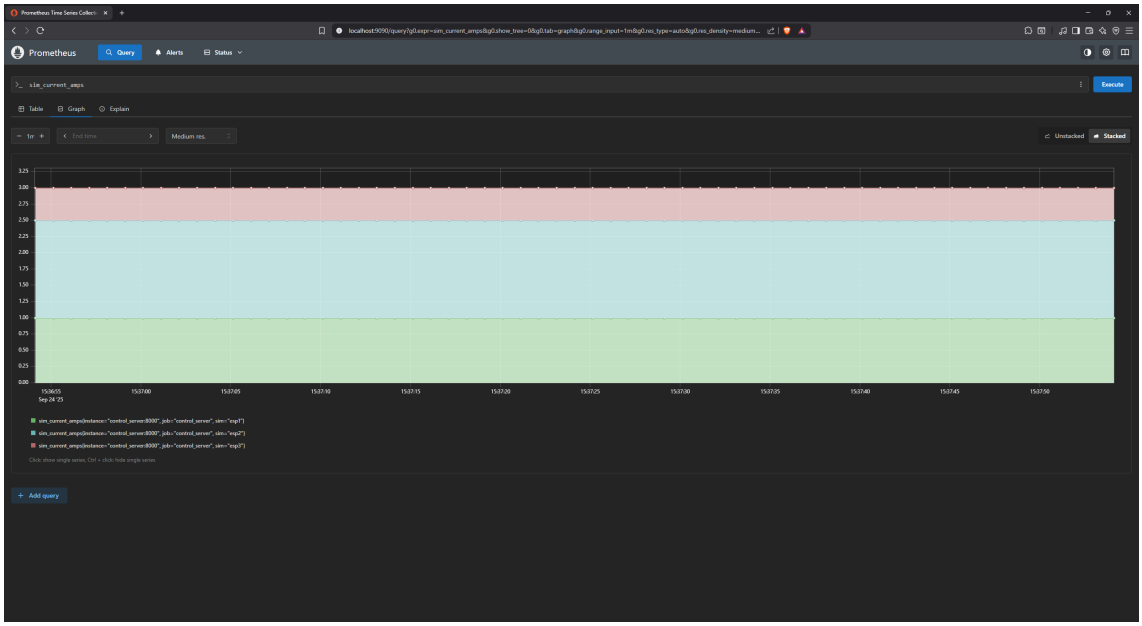
- *Nincs korlát*: $\text{effective}_{1,2,3} = \{1{,}0; 1{,}5; 0{,}5\}$ A, a korlátokat nagy értékek ($\sim 10^9$) jelölik.
- *Összáram*: `sum_current_amps` $\approx$ 3,0 A stabilan, ez így vízszintes vonal az élő grafikonon.
- *Megszakító*: bekapcsolt állapotban van, mert 3,0 < `BREAKER_MAX_TOTAL` = 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=
breakers=brk1:on,brk2:on
```

Siker kritérium: az élő grafikon 3 A vízszintes görbét mutat az `output.txt` egyezően jelzi, hogy effective$_i = d_i$, korlátokk nincsenek érvényben, a megszakítók on; mindez 1-2 ciklus (3-6 s) után stabilan látszik.



9.1. ábra. Alaptesztek

9.4.2. Alulterhelés: nincs korlátozás

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

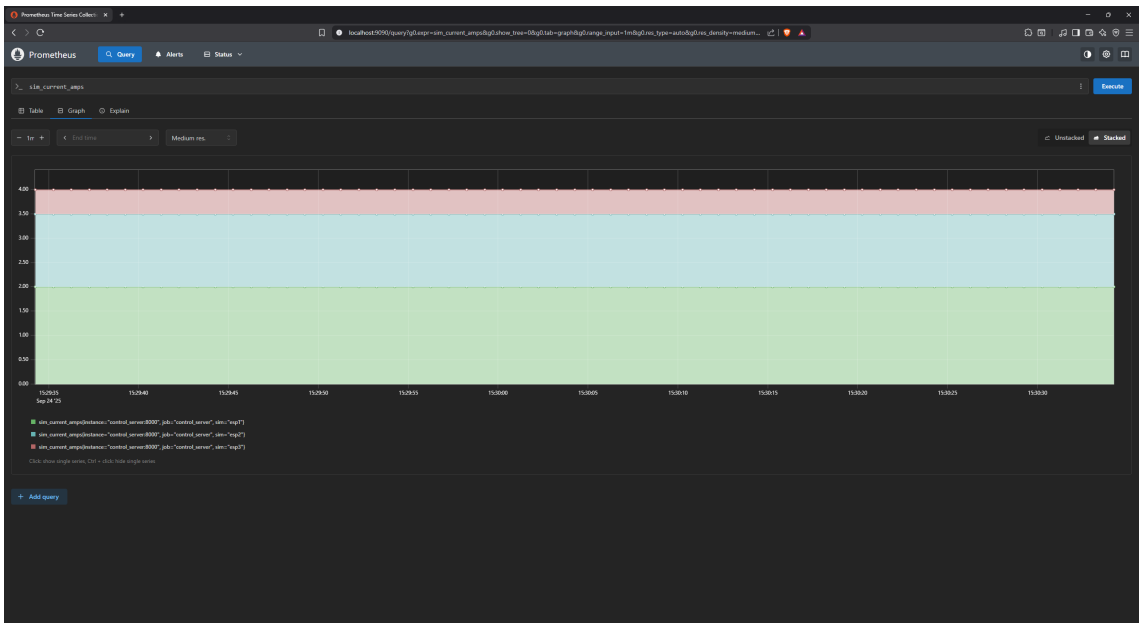
- *Nincs korlát*: $\text{effective}_{1,2,3} = \{2{,}0; 1{,}5; 0{,}5\}$ A, a korlátok nagy értékekkel ($\sim 10^9$) jelzik a „nincs limit” állapotot.
- *Összáram*: `sum_current_amps` $\approx 4{,}0$ A $\Rightarrow$ vízszintes vonal az élő grafikonon.
- *Megszakító*: bekapcsolva, mert $4{,}0 <$ `BREAKER_MAX_TOTAL`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

9.4.3. Túlterhelés, azonos igények: fair 3/3/3 (részletezve)

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

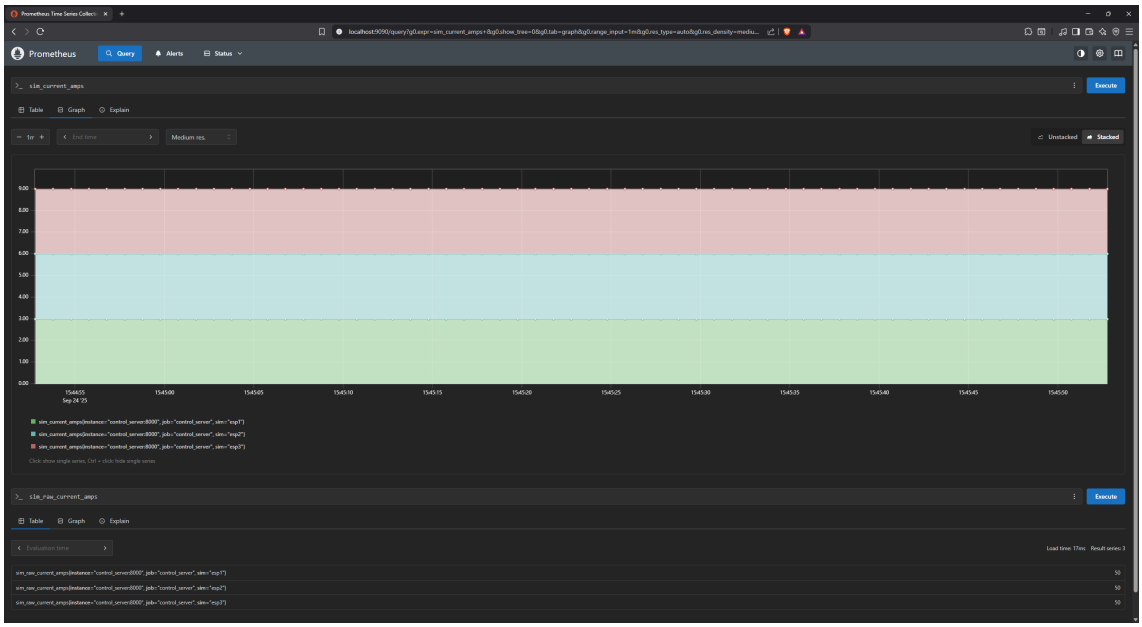
- *Korlátok*: mindhárom mérőnél `cap` $= 3$ A; $\text{effective}_{1,2,3} = \{3, 3, 3\}$ A.
- *Összáram*: `sum_current_amps` $= 3{+}3{+}3 = 9$ A $\Rightarrow$ egyenes vonal a grafikonon.
- *Megszakító*: bekapcsolva, mert $9 <$ `BREAKER_MAX_TOTAL` $= 12$ és $9 >$ `BREAKER_MIN_TOTAL` $= 2$.

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
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

9.4.4. Dinamikus újraelosztás: a nagy felhasználó kap teret (részletezve)

Bemenetek:

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

Miért ez a beállítás? Az első szakaszban az igények azonosak (50, 50, 50 A), ami túl nagy a 90 A kerethez képest, ezért *fair* elosztás lép életbe: [30, 30, 30] A. A második szakaszban két mérő visszaesik 10 A-ra, a harmadik 100 A-t kér; a felszabaduló 80 A-ból 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, a menetrendváltás $t = 60$ s-nál pontosan értelmezhető.
2. **START** — a vezérlő 3 s-os ciklusokban számolja az allokációt; a $t = 60$ s utáni váltás az azt követő első ciklusban jelenik meg.
3. **Megfigyelés** 0..80 **s** — két jól elkülönülő állapotot várunk a grafikonon.

Várt rendszerállapot

- 0..60 *s*: cap-ek és tényleges értékek [30, 30, 30] A, összáram = 90 A.
- 60+ *s*: cap-ek és ténylegesek [10, 10, 70] A, összáram = 90 A (a két kisebb igény teljesül, a maradék a nagy igényűhöz kerül).
- *Megszakító*: végig on, mert 90 < `BREAKER_MAX` = 120 és 90 > `BREAKER_MIN` = 10.

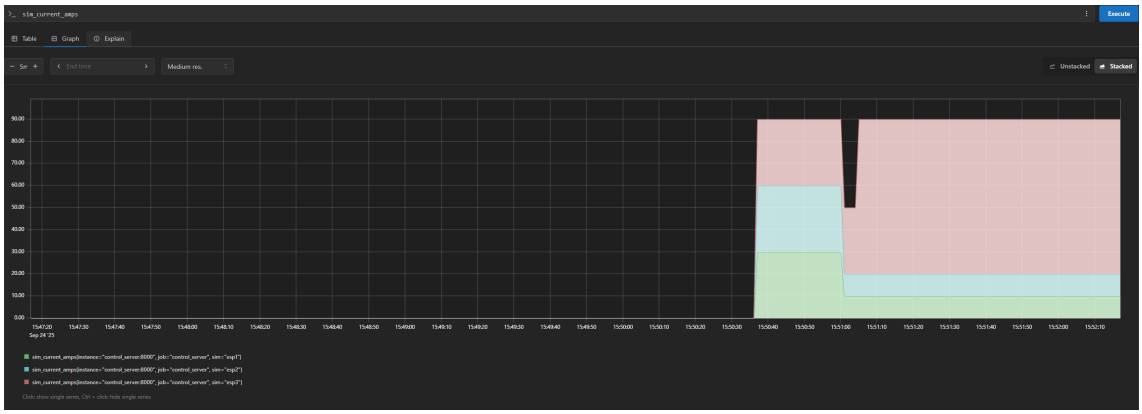
Hol ellenőrizhető?

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

```
 t3s   \(első szakasz\)
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

 t63s  (második szakasz, menetrendváltás után)
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: az élő grafikon két platót mutat: előbb 30/30/30 A, majd a $t = 60$ s utáni első ciklusban 10/10/70 A; az `output.txt` ugyanígy jelzi a cap és effective értékeket, az összáram végig 90 A, a megszakítók on.

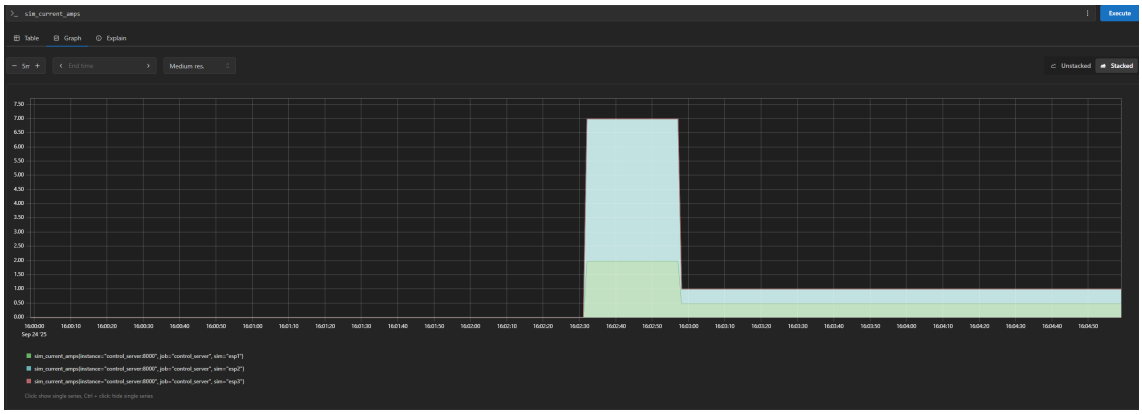


9.4. ábra. Dinamikus újraelosztás áramerősség

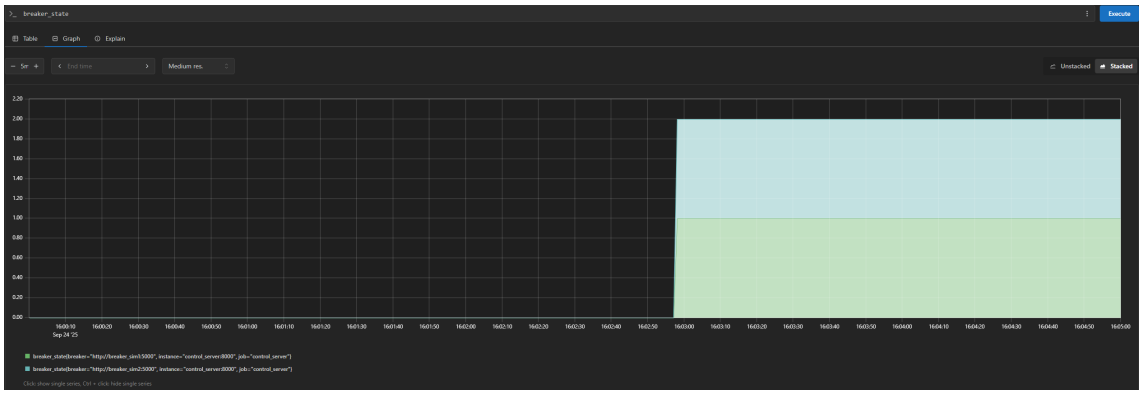
9.5. ábra. Dinamikus újraelosztás igények

9.4.5. Megszakító hiszterézis

Bemenetek:
`ALLOC_MAX_TOTAL=50`, `BREAKER_MAX_TOTAL=6`, `BREAKER_MIN_TOTAL=3`. Menetrendek:
ESP1: 0 s → 2.0 A, 40 s → 0.5 A; ESP2: 0 s → 5.0 A, 40 s → 0.5 A; ESP3: 0 s → 0.0 A.
Lépések:
Reset t=0, START, megfigyelés 0..60 s.
Várt eredmény:
0..40 s Sum = 7 A ⇒ `off`; 40+ s Sum = 1 A ⇒ on.
Siker:
`breakers= 0 s:off, 40 s:off` → `on`, on váltás az `output.txt`-ben és `/status`-ban.



9.6. ábra. Megszakító áramerősség

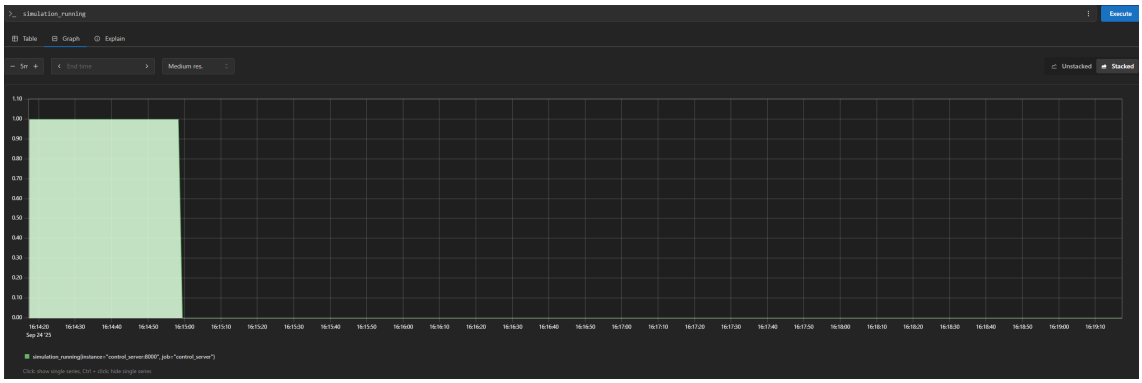


9.7. ábra. Megszakító állapotok

9.4.6. STOPPED invariánsok

Bemenetek:
induljunk a 3. szcenárió állapotából (3/3/3 cap).
Lépések:
`STOPPED` módba váltás; módosítsuk `ALLOC_MAX_TOTAL=6`-ra, ESP1 menetrendjét 1 A-ra; várjunk $\sim$ 2 ciklust.
Várt eredmény:
a cap-ek és a megszakítóállapot **nem** változik (`STOPPED` alatt nem történik beavatkozás).
Siker:
`/status.sim_state=STOPPED`; a `cap` és a `breakers` mezők változatlanok.

9.8. ábra. stopped állapot áramerősség



9.9. ábra. stopped állapot állapotok

9.10. ábra. stopped állapot maximális áramok

Összegzés

A tesztek ellenőrzik, hogy (i) az allokáció a max-min fair elvet követi-e, (ii) a megszakító hiszterézise a küszöbértékekhez képest működik-e, (iii) a `STOPPED` állapot működik-e, és (iv) a rendszer minden ciklusban önmagát leíró idősoros naplót állít elő. Ezek együttesen biztosítják az elvárt funkcio nális helyességet és transzparens viselkedést.