Debreceni Egyetem

Informatikai Kar

Információ Technológia Tanszék

Otthon automatizáció Java-ban

Konzulens:
Vágner Anikó Szilvia
adjunktus

Szerző:
Lakatos István
programtervező informatikus

Debrecen, 2017.

Tartalomjegyzék

1. fejezet

Bevezetés

Dolgozatom témaválasztása nem volt nehéz feladat számomra, mivel korábbi érdeklődés alapján már találkoztam az otthon „okosítás” vagy automatizálás témakörével. Rendkívül le tudtak foglalni a hardverközeli, mikrokontrollereket felhasználó projektjeim és azoknak kapcsán kezdtem el például különböző környezeti jellemző mérőeszközöket készíteni szabadidőmben. Onnantól kezdve pedig rövid út vezet házak utólagos felokosításának ötletéhez. Foglalkoztat még az is, hogy milyen előnyei lehetnek egy okos otthon rendszernek. Természetesen az elsődleges dolog e szempontból a spórolás lehetősége. Mindenki szeretne lefaragni a számlák összegéből például a fűtés kikapcsolásával, mikor nem vagyunk otthon, vagy a lámpák automatikus lekapcsolásával, ha éppen nem vagyunk a szobában. Másik vonzó tulajdonsága egy okos otthonnak, hogy rengeteg kényelmi funkciót tud nyújtani. Gomb nyomásra leengedhetjük a ház összes sötétítőjét vagy a rendszer meglocsolhatja helyettünk a növényeket. Tervem, hogy ezt az általam fejlesztett rendszert a saját otthonomban használjam jelentős számú mérőeszközöket felhasználva.

Annak a háttérében, hogy mindezt a Java nyelv segítségével tervezem megvalósítani, szintén egyszerű okok állnak. Tanulmányaim során legtöbbször a Java nyelvvel találkozhattam és így szerezhettem jelentősebb belelátást a működésébe és használatába. Így mivel szinte egyedül a Java nyelvhez kapcsolódóan van megfelelő tudásom ahhoz, hogy szakdolgozathoz illő nagyságú munkát készítsek, azt választottam alapnak. Másik fontos oka választásomnak, hogy szakmai gyakorlatom alatt is Java nyelvet használtam és így tudtam ismereteket szerezni több olyan keretrendszerről, amiket használni is szeretnék tapasztalatszerzés céljából, illetve pontosan beleilletek a bennem kialakuló képbe, amit arról a rendszerről alkottam, mely ennek a dolgozatnak az alapját adja.

A dolgozatom során a fő cél egy olyan rendszer elkészítése, mely képes apró eszközök és egy központi egység segítségével automatizált feladatokat elvégezni. Ennek

eléréséhez meg kellett tervezni az apró eszközök hardveres felépítését, fejleszteni egy központi szoftvert, ami képes az eszközöktől érkező adatok feldolgozására és azok alapján a felhasználó által megadott szabályok mentén feladatokat végrehajtására. Felmerült több érdekes kérdés is, melyekre kutatásaim során találtam válaszokat és próbáltam ezen válaszokat felhasználni a fejlesztés során.

A fejlesztés alatt végig úgy hoztam a döntéseket, hogy egy egyszerű háztartásban kell működjön a rendszer. Tehát nem volt célom az, hogy nagy számú felhasználók használják egyszerre vagy éppen több száz eszköz kapcsolódjon a központi rendszerhez. Részletesebben és feladathoz kapcsolódóan is meg fogom fogalmazni az elvárásokat, célokat a rendszerrel szemben, de ezt természetesen a későbbi fejezetekben teszem majd.

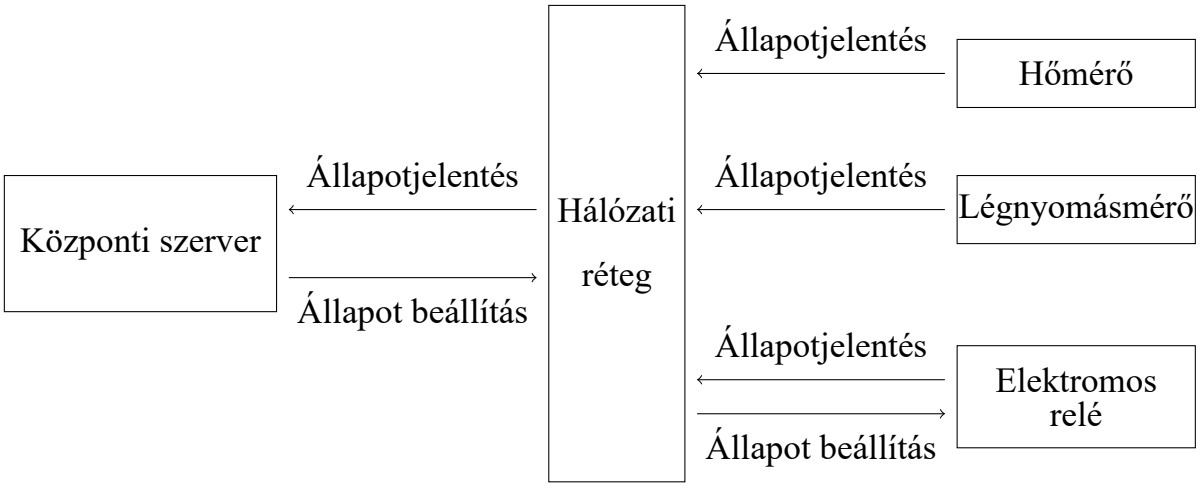
2. fejezet

Rendszer áttekintés

Ebben a fejezetben egy kezdeti képet szeretnék adni a teljes rendszer alapvető felépítéséről, illetve arról, hogy az egyes rendszer részek felé milyen elvárások vannak. Minden részt kifejtek ebben a fejezetben, de csak annyira, hogy átláthatóvá tegye a dolgozat többi részét. Későbbi fejezetekben viszont majd mélyrehatóbban nézzük meg a rendszer alapjait.

Alapvetően három részre bontható a rendszer:

- Eszközök, amely magába foglalja az összes mérőeszközt és egyéb kisebb beágyazott rendszereket. Ezek egyszerű mikrokontrollereket használó rendszerek, amiknek annyi szerepe van, hogy vagy adatot szolgáltatnak, vagy valamilyen funkcionalitást végeznek el.
- Hálózati réteg, amely egy egyszerű közvetítőként funkcionál a központi szerver és az eszközök között. Lényegében átalakítja az üzeneteket a két rész között a megfelelő formátumra.
- Központi szerver, amely végzi az adatok feldolgozását és biztosít egy felületet a felhasználó számára, hogy láthassa a rendszer állapotát vagy aktuális adatokat.



2.1. ábra. A rendszer felépítése, a rétegek és eszközök egymás közötti kommunikációja

2.1. Eszközök

Mint már említve volt, ide tartoznak azok a hardverközeli rendszerek, amik vagy adatgyűjtő szerepet töltenek be, vagy valamilyen funkcionalitást képesek végrehajtani. Lehet egy ilyen eszköz például egy hőmérő, légnyomásmérő, földnedvességmérő vagy éppen egy relé.

Ezek az eszközök két csoportba sorolhatók:

- Érzékelők, másnéven *szenzorok* vagy
- Cselekvők, másnéven *aktorok*[1]

Valójában viszont ez a két csoport nem diszjunkt és később kiderül miért is nem, ha látjuk pontosan melyik mit jelent. Közös bennük, hogy mindegyikben van egy mikrokontroller és egy kommunikációért felelős modul. E mellett mindegyik eszköz tartalmaz még egy specifikus modult is, ami meghatározza, hogy mit is csinál az adott eszköz (pl.: hőmérő esetén egy hőmérséklet- és páratartalom érzékelő), illetve, hogy melyik csoportba tartozik.

2.1.1. Szenzorok

Magától értetődően a szenzorok felelősek az adatok szolgáltatásáért. Működését tekintve a mikrokontroller vezérli az adatgyűjtés és küldés folyamatát. Az idő nagy

[1] Ezt az elnevezést magam adtam, mivel nem találkoztam kutatásaim során más megfelelő elnevezéssel.

részében alvó állapotban van energiatakarékossági szempontok miatt, viszont fix időközönként felébred. Mikor felébred lekéri a saját specifikus moduljától a jelenlegi mért állapotot, majd azt megpróbálja közvetíteni a központba. Ezt addig próbálja, amíg nem sikerül, vagyis a központ vissza nem küld egy automatikus jelzést, hogy megérkezett az adat. Szenzorok közé sorolhatók például a sok féle mérőeszközök vagy akár egy mozgásérzékelő is.

2.1.2. Aktorok

Az aktorok a rendszer olyan részei, amik segítségével a rendszer képes változtatásokat végbe vinni a fizikai világban. Hasonló a helyzet a szenzorok esetéhez, hiszen ugyanúgy a mikrokontroller kezel mindent. Ugyanúgy alvó állapotban van az eszköz az idő nagy részében, viszont jelen esetben nem fix időközönként ébred fel, hanem amikor üzenetet kap a központtól. Ezek az üzenetek állapot beállítási parancsok, vagyis utasítják az eszközt, hogy állítson át vagy cselekedjen valamit. Ha megtörtént a parancs feldolgozása, az eszköz egyből elküldi a központi rendszernek az új állapotát, állapotjelentés formájában, akárcsak a szenzorok, majd megint alvó állapotba kerül a következő parancsig. Aktor lehet például egy elektromos relé vagy egy termosztát is.

Tehát láthatjuk, hogy miért is nem szétválasztható a két csoport. Az aktorokra tekinthetünk olyan szenzorokként, amik képesek még egyéb funkcionalitást is végezni. Így tekintve a szenzorok csoportja tágabb és magába foglalja az aktorok csoportját.

2.2. Hálózati réteg

A hálózati réteg léte elsőnek nem látszik logikusnak. Feleslegesnek tűnhet egy plusz szereplőt bevonni a központi szerver és az eszközök közé. A probléma, ami mégis megköveteli azt, hogy legyen egy köztes réteg, az egy eszközökhöz kötődő döntésből fakad.

A hardveres tervezés alatt találkoztam több vezetéknélküli kommunikációs modullal is. Volt olyan, ami WiFi-t használ a kommunikációhoz, volt, ami csak simán rádiófrekvenciás modul volt. Bizonyos okok miatt, amit majd a 3.1.2. alfejezetben fogok leírni, a rádiófrekvenciás modulok mellett döntöttem. Így viszont szükség van egy központi eszközre, ami egy ugyanolyan kommunikációs modullal kell rendelkezzen, mint a többi eszköz. Ahhoz, hogy ezt a modult használni tudjuk, viszont alacsonyabb szintű eszközökhöz kell folyamodni, mint amit a Java nyújtani tud. Pont ezért a hálózati réteg egésze C++-ban íródott.

Lényegében a hálózati réteg tartja fent a kapcsolatot az eszközökkel és alakítja át az üzeneteket a fogadó félnek megfelelő formátumra. Ez az alakítás azért szükséges, mert a kommunikáció a hálózati réteg és a központi szerver között egy TCP kapcsolaton keresztül történik JSON objektumok küldésével, viszont a hálózati réteg és az eszközök között C-beli `struct`-okat küldünk rádiófrekvenciás jelek segítségével.

2.3. Központi szerver

A legnagyobb funkcionalitást természetesen a központi szerver végzi, hiszen ott lesznek feldolgozva az eszközöktől érkező állapotjelentések és a szerver küldhet állapot beállítási parancsokat is.

Mint említve volt már a központi szerver TCP kapcsolaton keresztül kommunikál a hálózati réteggel. Az onnan érkező üzeneteket a rendszer például a megfelelő formában elmenti az adatbázisban vagy az üzenet hatására elindulhatnak bizonyos döntéshozási folyamatok is.

A központi szervert is tovább tudjuk bontani több darabra funkcionalitása alapján:

- Flow-rendszer
- Statisztika
- Kezelőpanel

2.3.1. Flow-rendszer

A *Flow-rendszer* onnan kapta a nevét, hogy a felhasználó által megadható szabályokat „flow”-knak neveztem el. A név a beérkező adatok folyamából és azoknak folyamatos feldolgozásából jön. Minden flownak van feltétele és hatása. Ez a hatás akkor fog bekövetkezni, ha a feltétel teljesül. Részletesebben a 5.3. alfejezetben lesz szó a flowkról.

A *Flow-rendszer* biztosítja az új szabályok létrehozásának, régebbiek módosításának lehetőségét és elmenti az így létrejött változásokat. Másik lényeges feladata ennek a rendszernek még, hogy be is tartassa ezeket a szabályokat. A szabályok betartásáért felelős mechanizmust szintén a 5.3. alfejezetben fogom taglalni. Röviden összefoglalva, ha új adat érkezik, a rendszer megnézi melyik szabályokat érintheti az esemény és kiértékeli azokat. Tulajdonképpen ez a rendszer felelős a szerver legfontosabb és legbonyolultabb funkcióiért.

2.3.2. Statisztika

A rendszer használata során természetesen szeretnék néha visszanézni régebbi méréseket. Lehet ennek sok oka, például szeretnénk megnézni mennyivel volt hidegebb az előző hónapban, vagy leellenőrizhetjük mennyit volt bekapcsolva a fűtés. Tehát elég könnyen adja magát az elvárás, hogy tudjunk statisztikákat nézni a múltbeli adatokról.

Ehhez viszont el kell tárolni minden adatot, hogy később is tudjunk valamit mutatni a felhasználónak. Az adatbázis-rendszer kiválasztásánál ez volt az egyik legnagyobb szempont. A választás a *MongoDB*-re jutott, viszont a döntés mögött álló érveket majd a 5.1. alfejezetben fogom ismertetni. Mindemellett ahhoz, hogy az a sok adat, amit elmentenénk ne foglaljon sok tárhelyet, nem külön mentődnek el, ahogy beérkeznek a szerverhez, hanem percenként átlagolva kerül az adatbázisba. Ezzel igaz, hogy egy kicsi pontosságot vesztünk mikor visszanézzük a statisztikákat, viszont, ha jobban belegondolunk nincs is szükségünk arra, hogy például 10 másodpercenkénti részekre lebontva lássuk a múlt heti hőmérsékletet. Ami azt illeti még a percenkénti lebontás is túl aprónak tűnik bizonyos helyzetekben, de természetesen a lementett adatok segítségével elő tudunk állítani akár napi vagy havi átlagokat is. Mindezt grafikonokon és idősor diagrammokon mutatva, a felhasználó könnyedén tud következtetések levonni magának.

2.3.3. Kezelőpanel

A kezelőpanel szolgáltatja a felhasználó számára a valós idejű adatokat. Itt jelennek meg a beérkező állapotjelentések az eszközöktől legelőször, illetve aktorok esetében itt lesz lehetőségünk kézzel átállítani az eszköz állapotát, vagyis állapot beállító parancsokat küldeni.

Lehetővé válik így, hogy a felhasználó akár a munkahelyéről is megnézhesse, hogy áll az otthona, esetleg nem hagyta-e nyitva az ajtót reggel. Ez a pár másopercenként frissülő kezelő felület minden eszköztípushoz külön mini megjelenítő modulokat rendel, aminek köszönhetően akár pár pillanat alatt megtalálhatjuk a számunkra fontos információkat.

3. fejezet

Eszközök

Az előző fejezetben megtudtuk, mit csinálnak az eszközeink. Szintén megismertük, hogy mit is jelent a *szenzor* és az *aktor* fogalma, illetve, hogy miben különböznek. Ebben a fejezetben ecsetelve lesz, milyen hardverek lettek kiválasztva az eszközök összeállítására, illetve meg lesznek indokolva ezen választások is.

Ugye tudjuk szintén az előző fejezetből, hogy minden eszköz „agya" egy mikrokontroller. Ez a kicsi és alacsony teljesítményű processzor felelős az adatgyűjtés folyamatának koordinálásáért. Az *Arduino* fejlesztői platformok mellett tettem le a voksom, mert nem igényel elmélyült hardverismeretet a programozásuk.

Másik fontos alkotóelem, a vezetéknélküli modul. Ebben a választásban már természetesen megszorítás volt, hogy a kiválasztott modul kompatibilis legyen Arduinokkal. Habár nem volt egyszerű döntés, egy *nRF24L01+* adó-vevő integrált áramkört felhasználó modult választottam.

Ahogy korábban említettem harmadik része az eszközeinknek változó, még pedig az határozza meg, hogy mit szeretnénk elérni az adott szenzorral vagy aktorral. A fejlesztés elején kiválasztottam pár eszköztípust, amit el szerettem volna készíteni és támogatni a kész rendszerben. A teljesség igénye nélkül be fogok mutatni pár szenzor és aktor modult, mint például a hőmérséklet mérésére szolgáló *DHT22* hő- és légnedvesség mérő szenzor modult.

Mindezek után kitérek arra, hogy a fent említett hardvereket hogyan kell összekötni, hogy egy működő eszközt kaphassunk. Szerencsémre az összerakási folyamat nem igényelt jelentős villamosmérnöki tudást, elég volt néhány kábellel összekötni a megfelelő ki- és bemeneteket.

3.1. Hardver választás

3.1.1. Arduino

Az Arduino-k egyszerűen használható AVR mikrokontroller alapú fejlesztői platformok, melyeknek az a céljuk, hogy megkönnyítsék az elektronikával való ismerkedést az átlagemberek számára. Ezeket az eszközöket az Arduino, mint vállalat tervezte, illetve tervezi, hiszen folyamatosan újabb és újabb Arduino platformok kerülnek napvilágra. Az Arduino vállalatnak fő célja az volt az Arduino eszközök tervezésénél, hogy minél több embernek legyen elérhető az elektronikus eszközökkel való barkácsolás. Mindezt úgy próbálják elérni, hogy olcsón előállíthatóak az eszközök, hogy leegyszerűsítették a programozást, illetve, hogy minden hardver tervet nyílt forrásúvá tettek.

Az én választásom is a fent említett előnyök miatt esett az Arduino fejlesztői platformokra. A nyílt források miatt kínai Arduino klónok lepték el a piacot. Akár 1000 forintért[1] is hozzájuthatunk egy ilyen klónhoz, de akár az eredetit is beszerezhetjük szintén nem túl drágán körülbelül 6000 forintért[2] magyarországi internetes boltokból.

A legfontosabb érv számomra mégis az, hogy nagyon egyszerű programozni az Arduinokat. Ezért is figyeltem fel rájuk és töltöm sokszor szabadidőmet Arduinokat használó saját projektekkel. A platformok mellé az Arduino vállalat aktívan fejleszt egy saját programozási környezetet és egy programozási nyelvet is. Az „Arduino” programozási nyelv valójában egy olyan C/C++ könyvtár, amely egyszerűsíti az eszközök használatához kapcsolódó programozási feladatokat. Például ahhoz, hogy egy LED-et villogtassunk elég a következő kód:

```
1  // the setup function runs once when you press reset or power the board
2  void setup() {
3    // initialize digital pin LED_BUILTIN as an output.
4    pinMode(LED_BUILTIN, OUTPUT);
5  }
6
7  // the loop function runs over and over again forever
8  void loop() {
9    // turn the LED on (HIGH is the voltage level)
10   digitalWrite(LED_BUILTIN, HIGH);
11   // wait for a second
```

[1] Arduino Uno klón: https://tinyurl.com/ArduinoUnoClone (2017. április 11.)
[2] Eredeti Arduino Uno Magyarországon: https://tinyurl.com/ArduinoUnoHun (2017. április 11.)

```
12    delay(1000);
13    // turn the LED off by making the voltage LOW
14    digitalWrite(LED_BUILTIN, LOW);
15    // wait for a second
16    delay(1000);
17  }
```

3.1. ábra. Hivatalos Blink.ino Arduino példakód

Továbbá mivel C/C++ az alapnyelv az Arduino „nyelv” támogat minden olyan C vagy C++-beli nyelvi elemet, amit a `avr-gcc` fordítóprogram támogat.

3.1.2. Vezetéknélküli modulok

A vezeték nélküli modulok közti jó választás fontos pontja a rendszernek és nagy hatással van később a működés menetére vagy stabilitására. Négy nagyobb szempont szerint vizsgáltam a jelölteket: ár, áramfogyasztás, hatótáv, sebesség. Ugye a célunk az, hogy minél olcsóbban érjünk el nagy hatótávokat. A sebesség szerencsére nem fontos számunkra, hiszen alkalmanként küldünk csak egy pár bájtnyi üzenetet. Annál inkább lényegesebb az áramfogyasztás, mert természetesen ezek az eszközök elemekről működnek majd. Ezért korlátozó tényező az áramforrás kapacitása és a teljesítménye is, vagyis, hogy mekkora áramerősségre képes. Háromféle modult vettem számításba és hasonlítottam őket össze egymással a végleges döntés meghozásához.

Elsőként az *nRF24L01+* alapú modulokat mutatom be. Az *nRF24L01+* modulok olcsónak tekinthetőek, hiszen akár 500 forint körül megvásárolhatunk egyből 2 darabot.[3] A modul nem csak olcsó, hanem az áramfogyasztása is kiváló. Mind a üzenet fogadás és küldés közbeni nagyjából 12 milliamperes fogyasztás kivételesen jó. Összehasonlításnak, ha önmagában az *nRF24L01+* modult kellene működtetni egy gombelemről, akkor 10-15 órás folyamatos üzenet küldésre vagy fogadásra lennénk képesek és ha ezt úgy úgy nézzük, hogy egyszerre csak a másodperc töredékéig kell aktívan küldeni vagy fogadni nagy időközönként, akkor láthatjuk tényleg mennyire is kevés az *nRF24L01+* modul fogyasztása. Sebességet tekintve közép kategóriába sorolható, ugyanis a modul képes a maximum 2 Mb/s adatátviteli sebességre. A 2 Mb/s nem mondható túl nagynak, ugye sebességre nincs is nagy szükségünk. Ha viszont lentebb vesszük a sebességét az moduloknak 250 kb/s-ra, akkor még hatótávban is nyerhetünk pár métert. Minden közül a legfontosabb tulajdonsága egy vezeték nélküli modulnak

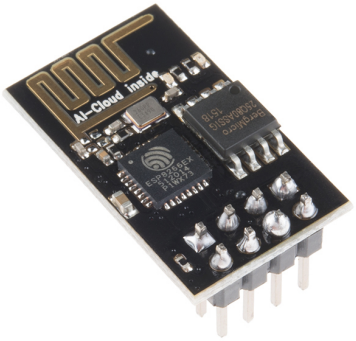
[3] nRF24L01+ vezeték nélküli modul eBay ára: https://tinyurl.com/eBayNRF24 (2017. április 18.)

az, hogy mennyire messze ér el az adatátvitel. Ezen szempontból sincs szégyenkeznivalója az *nRF24L01+* moduloknak. Természetesen a hatótávra rendkívül nagy hatással vannak a környezeti tényezők, például mennyi akadály van az adó és vevőt között. Mivel az *nRF24L01+* 2.4 GHz-es vivőfrekvenciával működik a modulok a beltéri hatótáv többszörösét tudják elérni egy szabadtéren. De szintén hátráltató tényező a WiFi hálózatok jelenléte, hiszen az is a 2.4 GHz-es frekvenciákon működik. Mindezek ellenére a modul képes akár 100 méteren keresztül is az adatátvitelre és ha azt tekintjük, hogy a rendszer célja, hogy egy háztartásban működjön, akkor még a 25 méteres hatótáv is elég lehet számunkra.

3.2. ábra. nRF24L01+ alapú vezeték nélküli modul

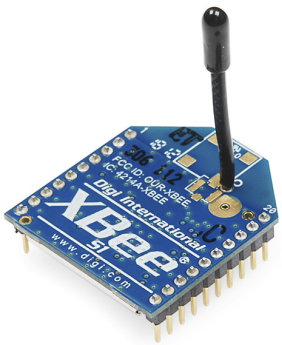
A második modul az *ESP8266*, ami egy WiFi-n keresztül kommunikáló eszköz. Azzal, hogy WiFi-képes a modul az is lehetséges, hogy közvetlen az internetre csatlakozva küldjünk üzeneteket és ezáltal nem is feltétlen lenne szükség egy közvetítőre a szenzorok, az aktorok és a központi rendszer között. Ezzel a tulajdonsággal, viszont olyan hátrányok járnak, mint aránylag rövid hatótáv és rendkívül nagy áramfogyasztás. Maga az *ESP8266* is képes WiFi hozzáférési pontként funkcionálni és jó hatótávokat elérni (akár 100 méternél is többet), viszont mi úgy szeretnénk használni, mint egy routerhez kapcsolódó eszközt, ezért a router hatótávja korlátoz minket. Egy router hatótávja viszont általában tekintve nem a legfényesebb, van, hogy alig ér be egy-egy háztartást. Áramfogyasztást tekintve üzenetküldés alatt 100 milliamper körül fogyaszt, olykori 300-350 milliamperes csúcsokkal. Ezen csúcsok miatt elég nagy teljesítményű áramforrásokra lesz szükség a működtetéséhez, amik maguk is sok áramot fogyasztanak. Ráadásul a modul fogyasztása is nagyon magas, nem is nagyon lehetséges sokáig üzemeltetni, ha véges kapacitásokkal rendelkezünk. Az árat tekintve viszont jól áll az *ESP8266*, mert 2000 forintért[4] már hozzájuthatunk.

[4]ESP-01 WiFi modul külföldi webáruházi ára: https://www.sparkfun.com/products/13678 (2017. április 18.)



3.3. ábra. ESP8266 WiFi vezeték nélküli modul

Hátramaradtak a *Zigbee* modulok. Sok féle Zigbee létezik, de én a Series 1, 1 mW-os eszközöket vettem figyelembe, mert a nagyobb fogyasztású társai az *ESP8266*-höz hasonlóan túl hamar lemerítené az áramforrásainkat. Hiába van jó hatótávja, mint az *nRF24L01+* moduloknak, még a legenergiatakarékosabb variáció is többet fogyaszt nála. Hivatalosan a *Zigbee* eszközök 50 milliampert fogyasztanak, ami 3-szor vagy 4-szer több, mint az *nRF24L01+* fogyasztása. A sebessége a moduloknak elfogadható számunkra, 250 kb/s-ra képesek, ami elég, ahogy arra már korábban kitértem. Utoljára hagytam a legnagyobb hátrányát a modulnak, ami nem más, mint az ára. Darabja a *Zigbee* eszközöknek 8000 forint[5] körül van, ami túlságosan megemeli egy-egy szenzor előállítási költségét. Egy *Zigbee* árából akár 30 *nRF24L01+* modult is vehetünk, ami elég abszurdul hangzik.



3.4. ábra. Zigbee Series 1 1mW vezeték nélküli modul

3.1. táblázat. Vezeték nélküli modulok összehasonlítása

Modulok	nRF24L01+	ESP8266	Zigbee
Sebesség	250 kb/s-2 Mb/s	1-5 Mb/s	250 kb/s

[5]Zigbee Series 1 1mW vezeték nélküli modul külföldi webáruházi ára: https://www.sparkfun.com/products/8665 (2017. április 18.)

Modulok	nRF24L01+	ESP8266	Zigbee
Fogyasztás	~ 12 mA	~ 100 mA	50 mA
Hatótáv	30-100 m	10-30 m	30-100 m
Ár	~ 250 Ft/db	~ 2000 Ft/db	~ 8000 Ft/db

A fejezet elején már említettem, hogy az *nRF24L01+* modulokat választottam, amiben főleg az ára és az áramfogyasztása játszott nagy szerepet. Mind a három eszköz megfelelt a sebesség és hatótáv feltételeimnek, így könnyen látható a 3.1. táblázatból, hogy a maradék két szempontot nézve egyértelműen az *nRF24L01+* eszközök a nyerőek.

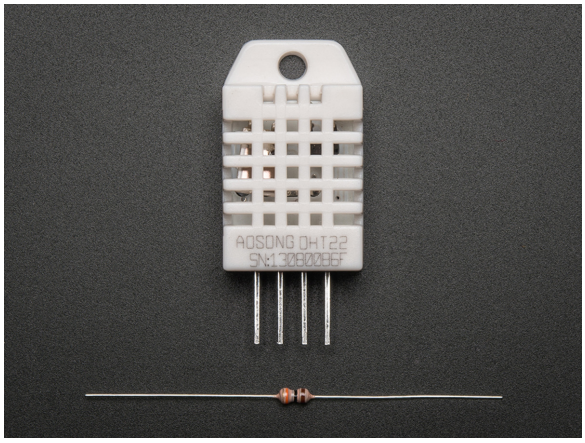
3.1.3. Eszköz specifikus hardver

Közös megszorítás volt természetesen minden eszköz specifikus modul kiválasztásánál, hogy könnyedén lehessen használni Arduinokkal. Ez általában abban nyilvánult meg, hogy az interneten találtam-e Arduinokhoz írt könyvtárakat, amik az adott modul képességeinek használatát teszik lehetővé. Választási szempontom volt még, hogy mennyire drága az eszköz. Néhol érdemes feláldozni a mérési pontosságot ahhoz, hogy ne kelljen sok pénzt kiadni. Mivel minden eszköztípus más érzékelőt vagy más funkciót elvégző alkatrészt tartalmaz, illetve mivel konkrétan akármilyen funkciójú eszközt megtervezhetünk, ezért nem tudok minden eszköz specifikus modult bemutatni. Viszont korábban említettem, hogy kiválasztottam pár szenzort és aktort, amit megterveztem és támogattam a rendszer fejlesztése során. Ezek a következőek:

- Hő- és páratartalom szenzor
- Mozgásérzékelő szenzor
- Föld nedvesség szenzor
- Elektromos relé aktor

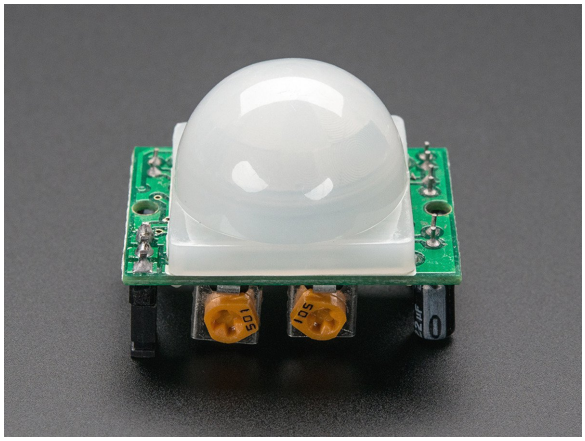
A *hő- és páratartalom szenzor* elkészítéséhez az úgynevezett `DHT22` modult használtam, ami egyszerre képes hőmérsékletet és páratartalmat mérni. Eredetileg nem terveztem, hogy egy szenzor akár többféle adatot is tud szolgáltatni a központi rendszernek, de mivel egy ilyen kettő-az-egyben modult találtam, ezért hamar változtattam az eredeti elképzeléseimen. Azért ezt a DHT22 modult választottam, mert szinte csak ehhez találtam Arduinot használó példákat. Pontosságát tekintve a ±2%-os páratartalom és ±0.5°C-os hőmérséklet hibahatár bőven megfelelt az én céljaimra. A modul ára is

aránylag kedvező, hiszen körülbelül 3000 forintért[6] beszerezhető egy-egy darab.



3.5. ábra. DHT22 hő- és páratartalom érzékelő modul

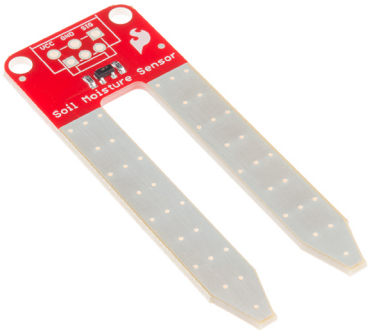
A *mozgásérzékelő szenzor* elkészítéséhez `PIR szenzort` használtam. A PIR feloldása a „*passive infrared*”, vagyis az infravörös fények változását képes érzékelni a szenzor. Ilyen PIR szenzorokat használnak például a mozgásra felkapcsoló kertilámpák is. Ezek egyszerű eszközök, általában működésük annyiból áll, hogy amikor mozgást érzékelnek elindul egy időzítő és az amíg le nem jár, addig a modul magas jelet ad. Ahogy az időzítő lejárt eltűnik a jel. Ennek a jelnek az értelmezése rendkívül egyszerű Arduinokkal. Elég, ha rákötjük a PIR szenzor kimenetét az Arduino egyik bemenetére és figyeljük mikor változik a bemeneten a jel. Az időzítő hossza és a modul érzékenysége kézzel állítható az eszközön. Ahogy az előző esetben is, most se magas az ára egy ilyen modulnak. Körülbelül ezek a szenzor modulok is 3000 forintba[7] kerülnek.



3.6. ábra. PIR mozgásérzékelő modul

[6]DHT22 hő- és páratartalom érzékelő: https://www.adafruit.com/product/385 (2017. április 11.)
[7]PIR mozgásérzékelő: https://www.adafruit.com/product/189 (2017. április 11.)

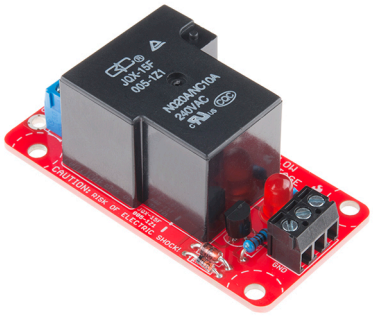
A *föld nedvesség szenzor* még a *mozgásérzékelő szenzornál* is egyszerűbb mérőeszközt igényel. A legegyszerűbb módja annak, hogy a föld nedvességét megmérjük az, hogy két föld alá dugott fémlemez között megmérjük a föld ellenállását. Minél kisebb az ellenállás, annál nedvesebb a föld. Ezt az elvet használó érzékelő modulokat könnyedén lehet találni és nem is drágán. Egy-egy ilyen érzékelő ára 1500 forint[8] körül mozog. Ahhoz, hogy működésre bírjük ezeket az eszközöket, szintén csak annyi a dolgunk a kimenetet az Arduino egyik bemenetére kötjük és onnan leolvassuk az ellenállás értékét. Abból tudunk következtetni mennyire száraz vagy nedves a föld.



3.7. ábra. Föld nedvességmérő modul

Az *elektromos relé aktor* esetében egy 5 voltos relére volt szükségem. Ha magasabb feszültséget igénylő relét akartam volna használni, akkor plusz alkatrészekre lett volna szükségem ahhoz, hogy Arduinoval irányíthassam. Egy relé alapvetően úgy működik, mint egy akármilyen villanykapcsoló, a különbség csak annyi, hogy elektromos jellel lehet kapcsolni. A relé egyik oldalára az irányítani kívánt áramkört kell bekötni, a másikra a mi esetünkben az Arduino egy kimenetét és az említett 5 voltot. Így a kimenet fel- és lekapcsolásával a bekötött áramkört is kapcsolgatjuk. Az ára a reléknek rendkívül változó tud lenni, annak függvényében, hogy mekkora teljesítményt bírnak ki. Én egy 10 ampert kibíró relé mellett döntöttem, ami 2500 forint[9] körüli összegbe kerül.

[8]Föld nedvességmérő: https://www.sparkfun.com/products/13322 (2017. április 11.)
[9]Elektromos relé: https://www.sparkfun.com/products/13815 (2017. április 11.)



3.8. ábra. Elektromos relé modul

3.2. Felépítés

Így, hogy minden alkatrészt kiválasztottunk a következő feladat, hogy össze is rakjuk az eszközeinket. Az első és legfontosabb lépés, hogy áramot kapjon a fő alkatrészünk, az Arduino. Magán az Arduinon elhelyezett feszültség szabályzónak köszönhetően, képesek vagyunk akár egy 9V-os elemmel is megoldani az áramellátást. A példa kedvéért most maradjunk ennél a megoldásnál annak okán, hogy ne kelljen fölösleges feszültség átalakításokkal foglalkozni. Valójában úgy terveztem viszont az eszközt, hogy egy tölthető, 3.7V-os lithiumion-akkumulátor üzemeltesse a szenzorjainkat, aktorjainkat.

Ha megoldottuk az áramellátás kérdését, jöhet a kommunikáció felállítása a központi szerverrel. Ehhez az úgynevezett SPI („*Serial Peripheral Interface*”) buszt kell használjuk az Arduino és az nRF24L01+ közötti kommunikációhoz. Sajnos magam sem ismerem mélyrehatóan az SPI technológiát, hiszen programtervező informatikusnak tanulok, viszont röviden összefoglalva az SPI egy soros kommunikációs interfész, amely *master-slave* architektúrára épül. Tehát van egy *master* eszköz és elviekben tetszőleges számú *slave*, ahol a *master* eszköz irányítja teljes egészében a kommunikációt és a *slave* eszközök csak a *master* jelére kezdenek el adatot küldeni. Összesen négy kábel elég ahhoz, hogy a kapcsolat létrejöjjön az Arduino és az nRF24L01+ között. Ahogy összekötöttük őket a full-duplex kapcsolaton keresztül máris tudunk adatot küldeni más nRF24L01+-t használó eszközöknek.

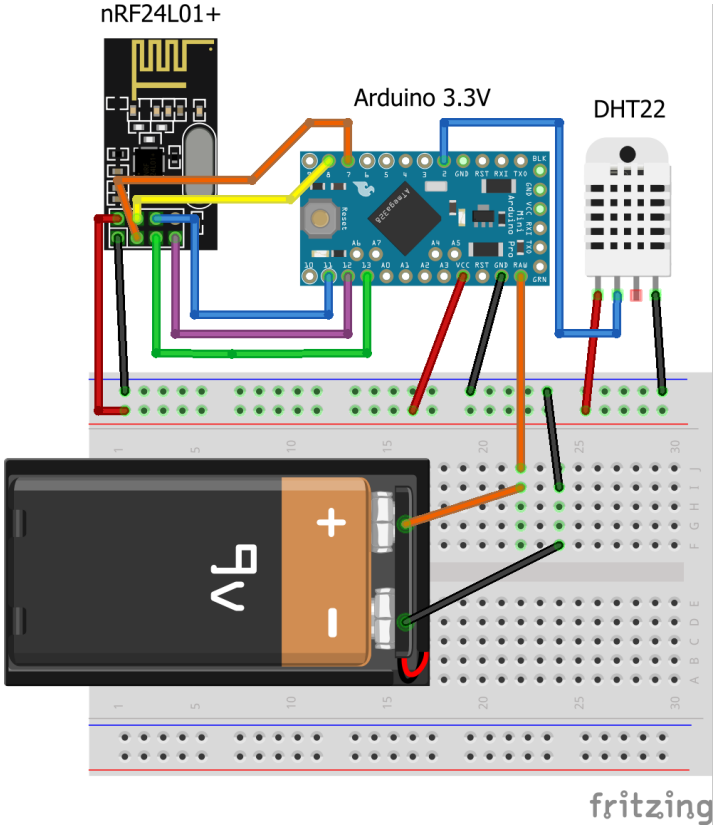
Hátra van még az eszköz specifikus rész bekötése. Természetesen ez eszközről eszköre változik, viszont mivel szükség lesz az Arduino megszakítás rendszerére, az egyetlen megszorítás, hogy olyan bemenetre kell bekötni a specifikus eszközt, ahol az Arduino támogatja a megszakításokat. A megszakításokat később, a 3.3.1. alfejezetben

fogom ismertetni.

3.2.1. Példák kész eszközökre

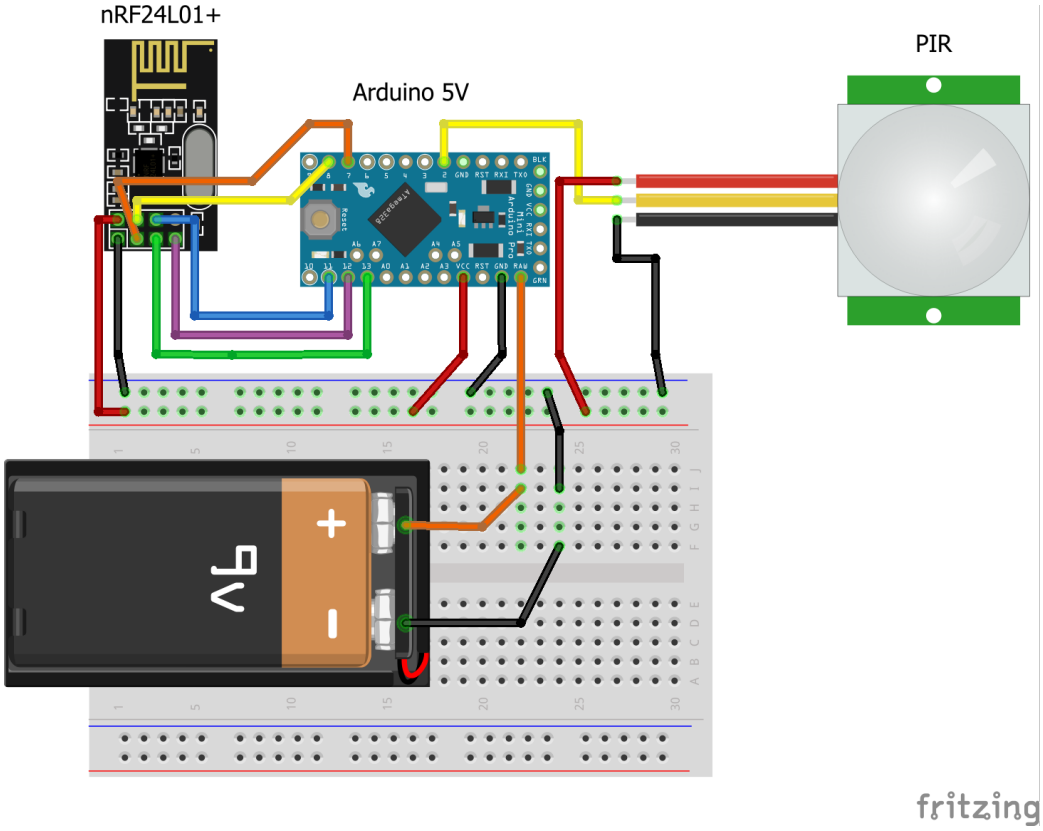
Csak felsorolásszerűen bemutatnám képekben annak a pár eszköznek a felépítését, amelyeket korábban említettem, mint a rendszer által támogatottak. A képek nem feltétlen fedik le a valós, összerakott eszközök felépítését. Ennek oka annyi, hogy másképp nem feltétlen lennének jól értelmezhetőek az ábrák. Például bizonyos esetekben az nRF24L01+ kommunikációs modul nem megfelelő tápfeszültséget kap a képeken, viszont ez nem változtat semmit a valódi bekötési ábra számunkra is lényeges részeihez képest.

- Hő- és páratartalommérő szenzor



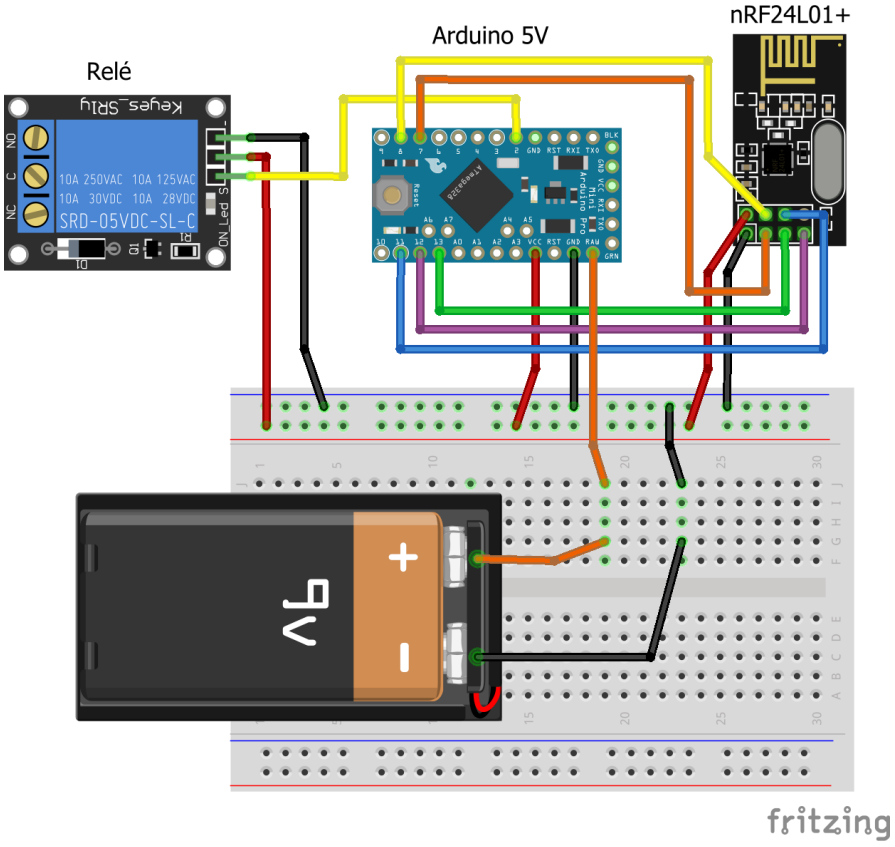
3.9. ábra. Hő- és páratartalommérő szenzor

- Mozgásérzékelő szenzor



3.10. ábra. Mozgásérzékelő szenzor

- Elektromos relé aktor



3.11. ábra. Elektromos relé aktor

3.3. Firmware működés

3.3.1. Megszakítások

A fejlesztés közben szembesültem azzal, hogy különbséget kell tegyek szenzorok közt is a kívánt működésüket tekintve. Pontosabban fogalmazva abban különbözhet egyik szenzor a másiktól, hogy mikor kell adatot küldjön a központi szervernek. Gondoljunk bele abba, hogy egy hőmérő szenzor elég, ha fix időközönként elküldi az aktuális állapotát. Ha viszont tegyük fel egy mozgásérzékelő szenzorról beszélünk, akkor azt szeretnénk, hogy ha mozgás van a szobában, akkor egyből elküldje az új adatot a szenzor. Ugyan ez az helyzet például egy ajtózárt irányító aktor esetében is. Azonnal szeretnék értesülni arról, hogy kinyitották a bejárati ajtót, nem pedig akkor, ha éppen úgy esik a fix időközönkénti állapot küldés. A fix időközönként közvetítő szenzort időzített szenzoroknak neveztem el, míg a azokat amik állapotváltozást igényelnek, hogy elküldjék az adatot, reaktív szenzoroknak.

Az eszközök felépítésének bemutatásánál említettem a megszakítások fogalmát és azt, hogy szükségünk lesz rájuk. A megszakítások olyan külső vagy belső jelek, amelyek a feldolgozó egység számára szólnak, hogy azonnali figyelmet igényel valamilyen hardver vagy szoftver. Az Arduinok esetében egy ilyen megszakítás lehet egy bemeneti jel változás vagy esetleg egy belső időzítő lejárta. Másik fontos tulajdonsága a megszakításoknak az Arduinoknál, hogy egy megszakítás hatására az eszköz képes felébredni alvó állapotból. Így viszont meg is oldottuk a szenzorok közti különbség problémáját, hiszen, ha az Arduino normál állapotban van, akkor leolvassuk az aktuális állapotát az eszköznek, elküldjük a központi szerverhez az adatot, majd alvó módba lépünk. Ahhoz, hogy az időzett szenzorok fix időközönként közvetítsenek a belső időzítőt kell használni az eszköz felébresztésére, a reaktív szenzorok esetében egy olyan bemenetre kell kötni az érzékelő modult, ahol az Arduino támogatja a megszakításokat. Így elértük azt, hogy ha például valamilyen mozgást érzékelünk az eszköz egyből felébred és elküldi az új állapotot.

Itt egy picit kitérnék az aktorok működésére is, igaz szorosan nem a szenzor típusokhoz kapcsolódik, viszont a megszakításokhoz annál inkább. Mivel az aktorok nem csak üzennek a központi szervernek, hanem fogadnak is onnan érkező parancsokat, az eszköznek normál állapotában kell működnie ahhoz, hogy a parancsot fel tudja dolgozni. Ennek ellenére szeretnénk, ha az aktorok is tudnának alvó állapotban lenni

energiatakarékosság miatt. Szerencsére az nRF24L01+ képes nekünk megszakításokat generálni, ha valamilyen üzenet érkezett az eszköz számára. Ez pont tökéletes számunkra, hiszen csak akkor fogjuk így felkelteni az Arduinot alvó állapotból, ha fel kell dolgozni a központól érkezett parancsot.

3.3.2. Alacsony fogyasztás

Mindegyik eszköznek, működése folyamán, van olyan időköze, amikor nem kell semmilyen feladatot elvégezzen. Szenzoroknál például két állapot leolvasás közt vagy aktorok esetén két parancs feldolgozás közt. Felmerül a kérdés, hogy amíg nincs szükségünk az eszközök teljes számítási kapacitására, addig tudnánk-e valamilyen módon energiát megtakarítani annak érdekében, hogy kevesebbszer kelljen elemet cserélni bennük. Korábban már említettem, hogy az Arduinok képesek *alvó módba* lépni és ezzel elérjük azt, hogy spóroljon az energiával. Ha például a legalacsonyabb energiafelhasználású módba tesszük az eszközt akár 40-szer kevesebb áramot fogyaszthat az Arduinon elhelyezett AVR mikrokontroller. Ez viszont jelentős üzemidő javulást jelenthet, így akár pár napnyi működés helyett hónapok is lehetnek. Lehetőségünk van még arra is, hogy kikapcsoljuk az AVR mikrokontroller nem használt részeit. Kikapcsolhatjuk például az analóg-digitális átalakítót vagy bizonyos belső időzítőket. Alkalmazhatjuk például úgy ezt a módszert, hogy mielőtt alvó módba lép az eszköz kikapcsoljuk minden részét a mikrokontrollernek és majd csak akkor visszakapcsoljuk azokat, ha felébred. Illetve azokat a részeket véglegesen kikapcsolhatjuk, amik olyan funkciókért felelnek, amiket sose használunk.

Tovább tudjuk faragni a fogyasztást bizonyos hardveres módosításokkal. Minden Arduinon található egy LED, ami folyamatosan ég amíg az eszköz áramot kap. Ha ezt a LED-et eltávolítjuk szintén jelentős mennyiségű energiát megspórolhatunk és igazából szükségünk sincs arra a jelző fényre. Található még az Arduinokon egy feszültség szabályzó integrált áramkör, ami általában nem a legjobb hatékonysággal rendelkezik. Természetesen ennek az az oka, hogy ahol csak lehet az olcsóbb alkatrészeket választják a gyártási költségek minimalizálása miatt. A feszültség szabályzó cseréje egy hatékonyabbra szintén segíthet valamennyit. Akár a teljes eltávolítása is megoldás lehet még, csak sajnos a mi esetünkben szükség van rá, így ez nem járható út.

4. fejezet

Hálózati réteg

Az áttekintő fejezeteben említésre került, hogy miért van szükség a hálózati rétegre. Ez a mondhatni közvetítő réteg annak ellenére, hogy nem végez el sok funkcionalitást, mégis sok munkát igényelt a fejlesztés során. A nehézség nem feltétlen csak a szoftver előállításából adódott, hanem a megfelelő hardverek megtalálásából és azok összehangolásából. Ahhoz, hogy fogadni tudjuk az eszközök üzeneteit ugyanolyan *nRF24L01+* adó-vevő modult kell használni a hálózati rétegben is, mint az eszközökben. Mivel az ahhoz való kapcsolódás és kommunikáció alacsony szintű hardverhozzáférést igényel és SPI buszt egy hétköznapi számítógép nem megfelelő számunkra. Léteznek viszont úgynevezett *single-board* számítógépek, amelyek pont az ilyen esetekre lettek kitalálva. Egy-egy ilyen számítógépen található egy aránylag nagy teljesítményű processzor, pár megszokott ki- és bemenet perfiréria számára (pl.: USB portok) és ami a legfontosabb számunkra GPIO pinek vagy másnéven *általános-célú ki- és bemenetek*. A GPIO-n keresztül lesz lehetőségünk az adó-vevő modullal kommunkálni. Az én választásom a *Raspberry Pi 3* apró számítógépre esett.

Szó lesz még a fentiek mellett a hálózat felépítéséről, hogy miként is kapcsolódnak és kapnak címet az eszközök. Habár mindezt az *nRF24L01+* modulhoz fejlesztett külső programkönyvtár végzi, mégis jelentős részét képezi a hálózati réteg működésének. A hálózat maga *mesh alapú*, viszont ez csak a programkönyvtárnak köszönhető, ezért nevezném inkább *pszeudo-mesh* hálózatnak, amit meg is fogok magyarázni miért. Ha már tudjuk, hogyan kapcsolódunk a hálózathoz, akkor már csak az üzenetek küldése hiányzik. Ismertetni fogom, hogy milyen felépítésűek az állapot jelentő vagy állapot beállító üzenetek a szerver és hálózati réteg között, illetve a hálózati réteg és az eszközök között.

Ezek után röviden bemutatom, hogy miként is épül fel a hálózati réteg szoftvere, hogy hol voltak gondjaim a fejlesztés során. Illetve említésre kerül még a C++ *Boost*

könyvtárcsomag, mivel rendkívül megkönnyítette számomra a TCP kapcsolat kezelését a központi szerverrel.

4.1. Mesh hálózat

Ahogy említettem a teljes eszköz hálózat irányítását egy külső programkönyvtár végzi. Elvégez olyan feladatokat, mint például az automatikus címkiosztás vagy éppen az útkeresés két eszköz között az üzenetküldéshez. Ezeknek köszönhetően az eszközök automatikusan be tudnak csatlakozni a hálózatba, annak bármely pontján és el tudnak érni bármilyen eszközt, ami aktív része a hálózatnak. Ezáltal a végfelhasználónak nem kell konfigurációval foglalkoznia a hálózat miatt, hiszen minden eszköz miután bekapcsoltuk saját maga végzi el a szükséges feladatokat, ahhoz, hogy a hálózathoz csatlakozhasson. Ezt a programkönytárat a *TMRh20* névre hallgató GitHub felhasználó készítette el és az *RF24Mesh* nevet adta neki,[1] amely az *nRF24L01+* adó-vevő modullal való munkát könnyíti meg.

4.1.1. Felépítés

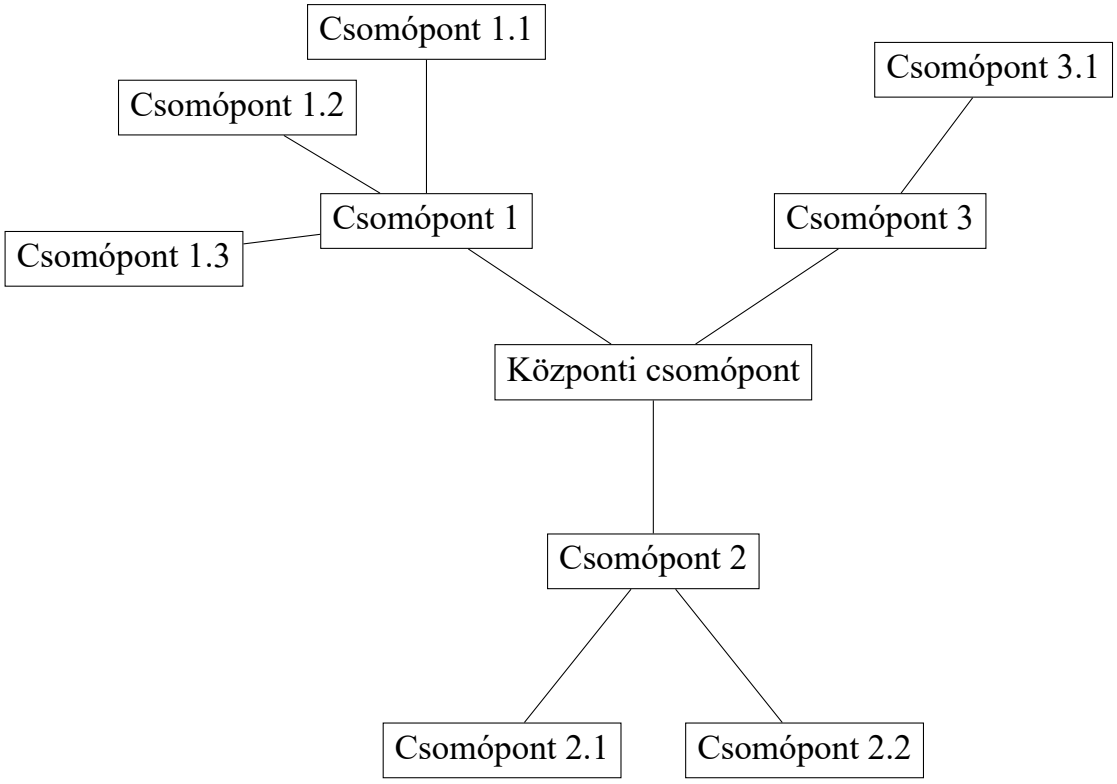
A *mesh* hálózatok legfontosabb tulajdonságai közé tartozik, hogy minden csomópontja a hálózatnak résztvesz az adatforgalom továbbításában, illetve, hogy a hálózat maga oldja meg az esetleges kapcsolati problémák okozta hálózati elérhetetlenséget. Mindez úgy valósulhat meg, hogy a csomópontok között akár több útvonal is létezik és ha egy csomópont elérhetetlenné válik, akkor újrakonfigurálja magát, hogy a hálózat másik pontján csatlakozhasson fel újra.

A felszínről nézve egy *mesh* hálózatot kapunk az *RF24Mesh* segítségével összekapcsolt eszközök csoportjából. Azonban a belső működés egy *fa-alapú* hálózatra épít és azt dinamikusan alakítva kapjuk meg a *mesh* hálózatot. Ezen működésnek nagy árulkodó jele például, hogy szükség van egy központi csomópontra, ahhoz, hogy a címkiosztás működjön. Ezen ok miatt neveztem *pszedo-mesh*-alapúnak a hálózatot a fejezet elején. Sajnos annak, hogy fa-alapra épül valójában a hálózatunk vannak következményei. A fa topológiából adódik, hogy minden eszközt csak egy úton tudunk elérni és ezért elveszítjük a redundás utak előnyeit. Megvan az oka, viszont, hogy miért a fa topológia lett mégis felhasználva a belső működésre. Az *nRF24L01+* egyszerre összesen 6 kapcsolatot tud fenntartani, tehát, ha több út vezetne egy csomóponthoz, akkor

[1] TMRh20 RF24Mesh programkönyvtárja: https://github.com/nRF24/RF24Mesh

hamar kifogyhatunk abból a 6 kapcsolatból. Ami azt illeti, még így is rendkívül hamar kevésnek tud bizonyulni.

Szerencsénkre nem minden eszköznek kell folyamatosan aktív része lenni a hálózatnak. Mivel az összes szenzor csak időszakosan kell rendelkezzen hálózati kapcsolattal, ezért mielőtt alvó módba lépnek lecsatlakozhatnak a hálózatról. Ezt csak azért léphetjük meg, mert a szenzorokat nem akarja elérni egy csomópont se. Ezzel ellentétben az aktoroknál nincs ilyen szerencsénk, hiszen az aktoroknak szólhatnak állapot beállító parancsok.



4.1. ábra. A hálózat felépítése

4.1.2. Címzés

Ahogy említettem a központi csomópont kezeli minden egyes hálózati eszköz címének kiosztását, a címek tárolását és a címek feloldását is. Minden eszköznek van egy egyedi, fix, 0-től 255-ig terjedő csomópont azonosítója. Tekinthetünk erre az azonosítóra, akárcsak egy MAC címre. Ezzen azonosító segítségével tudunk majd üzeneteket címezni az adott csomópontnak. Amikor egy csomópont felcsatlakozik a hálózatra a központi csomópont kioszt neki egy hálózati címet, amelyet el is ment magának. A hálózati cím már viszont az IP címekhez hasonlítható. Közvetlen üzenetet küldeni csak a hálózati cím birtokában tudunk. Az automatikus cím kiosztás úgy működik, hogy az

újonnan csatlakozott csomópont egy kérést indít a központ felé. A központ mindig a 0-ás csomópont azonosítóval rendelkezik, így az újonnan kapcsolódott csomópont is tudhatja hova kell küldeni a címkérést. Ez nem csak a címkérésnél érvenyes természetesen, ha akármilyen üzenetet szeretnénk küldeni a központi csomópontnak akkor csak a 0-ás címnek kell címezni az üzenetet. Azonban, ha a címzett nem a központi csomópont, akkor szükséges egy kérést intézni a központi csomóponthoz, hogy lekérjük a csomópont azonosítóhoz tartozó hálózati címet, majd a megkapott hálózati címre elküldhetjük az üzenetet.

4.2. Üzenetek felépítése

A hálózati réteg legfontosabb funkciója, hogy átalakítsa az üzeneteket megfelelő formátumra a központi szerver és az eszközök között. Erre azért van szükség, mert míg a központi szerverrel egy TCP kapcsolaton keresztül JSON üzenetekkel kommunikálunk, addig az eszközökkel nyers bitfolyamokon keresztül C-beli `struct`-okat használva tudunk. Egy üzenet típusa vagy állapot jelentés vagy állapot beállítás lehet.

Az üzenetekben használt csomópont azonosító megegyezik a 4.1.1. alfejezet csomópont azonosítóival. Az adat típus sztring literálok megmondják, hogy az adott eszköz milyen specifikus hardverrel rendelkezik, például a `temperature` adat típust a hőmérő szenzor használja. Fontos még megemlíteni, hogy logikusan az állapot jelentő üzenetek mindig egy eszköztól indulnak és az állapot beállító üzenetek mindig a központi szervertől.

4.2.1. Kommunkáció a központi szerverrel

Azért esett a választásom a TCP kapcsolaton keresztüli kommunikációra a központi szerver és a hálózati réteg közt, mert jelentősen egyszerűbb megoldani, mint más technológiákkal. Például megoldás lett volna még, hogy közös megosztott memória területen dolgozzon a központi szerver és a hálózati réteg. A hátránya az lett volna az osztott memóriának, hogy kötelező egy eszközön futnia a kapcsolat mindkét felének. Hasonló indokok állnak a mögött is, hogy miért JSON-t választottam az üzenetek formai alapjának. Szintén ez volt a legkönnyebben és leghamarabb megvalósítható megoldás. Habár itt már voltak más életképes megoldások is, mint például a Google által fejlesztett *Protobuf*, ami egy programozási nyelvfüggetlen adatcsere formátum. A JSON, azért tűnt mégis jobbnak számomra, mert könnyebben tudtam változtatni az üzenetek formátumán és kevesebb velejáró programozást igényelt.

- Állapot jelentés üzenet:

```
{
  id: [csomópont azonosító],
  type: [adat típusa],
  value: [adat érték],
  category: [aktor vagy szenzor]
}
```

- Állapot beállító üzenet:

```
{
  id: [csomópont azonosító],
  targetState: [kívánt állapot]
}
```

4.2.2. Kommunkáció az eszközökkel

Több különbség is észrevehető a központi szerver üzeneteihez képest. Elsőnek azt láthatjuk meg, hogy sehol nincs `id` mező, de nincs is rá szükség, mivel az üzenetet csak `id` csomópont azonosítóval rendelkező eszköznek küldjük el. Második különbség, hogy nincs `category` mező az állapot jelenő üzenetben. Erre azért nincs szükség, mert a hálózati kapcsolatot irányító programkönyvtár megengedi, hogy felcimkézzük az üzeneteket. Ezt kihasználva minden eszköz, mivel tudja magáról, hogy melyik kategóriába tartozik megfelelően cimkézi az üzeneteket. Ez a cimke lesz átalakítva mezővé a központi szervernek küldött üzenetekben.

- Állapot jelentés üzenet:

```
struct state_update_message {
  float data; // adat érték
  char type[15]; // adat típus
};
```

- Állapot beállító üzenet:

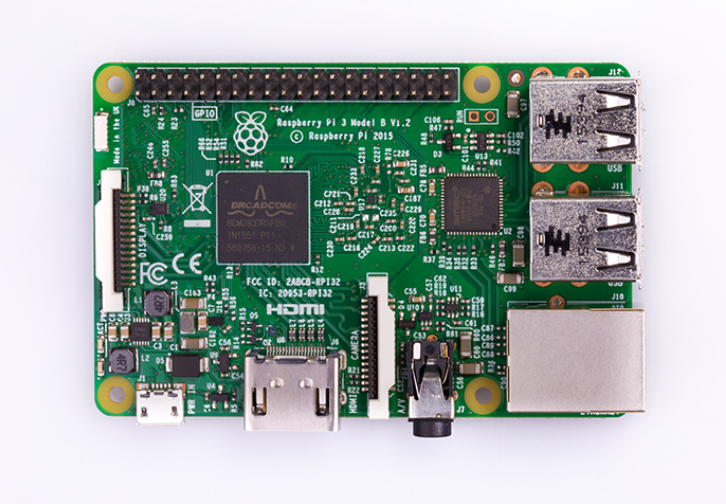
```
struct state_set_message {
  float targetState; // kívánt állapot
};
```

4.3. Raspberry Pi

A *Raspberry Pi* eszközcsalád rendkívül nagy hírnevet szerzett a *single-board* számítógépek világában. Mára, akit érdekel az IoT vagy az otthoni elektronikai barkácsolás, az bizonyára hallott a *Raspberry Pi* eszközcsalád egyik tagjáról. Jelenleg a legnagyobb teljesítménnyel a *Raspberry Pi 3* rendelkezik. Azon kívül, hogy a nagy gyártói és közösségi támogatás miatt nagyon egyszerű ezeket az eszközöket felhasználni saját projektjeinkben, még nagyon jó áron is hozzájuk juthatunk. Akár Magyarországon is 15000 forintért[2] kézhez kaphatunk egy *Pi 3*-at. Léteznek viszont olcsóbb testvérek is a *Pi 3*-hoz képest. Vegyük például a *Raspberry Pi Zero W*-t, ami úgy híresült el, mint az első 5 amerikai dollárba kerülő számítógép a világon. Azaz átszámolva 1500 forint környékén hozzájuthattunk egy olyan hardverhez, ami képes futtatni egy könnyedebb Linux disztribúciót és akár mindennapi irodai munkát is el lehet végezni rajta. Pontosan a fenti okok győztek meg arról, hogy egy ilyen eszköz teljes mértékben megfelel a hálózati réteg futtatásához. Illetve az is fontos döntési pont volt, hogy az általam választott *RF24Mesh* programkönyvtár teljes mértékben támogatja a *Raspberry Pi*-t.

4.3.1. Használat

A *Raspberry Pi 3* annak ellenére, hogy olcsón beszerezhető, aránylag jó hardver konfigurációval rendelkezik. A központban egy *ARMv8* architektúrájú 64 bites processzor van, amely 4 darab 1.2GHz-es magból van felépítve. A processzor mellé 1 GB RAM van párosítva, amely bőven elég a legtöbb felhasználási módra. Mindemellett WiFi 802.11n és Bluetooth 4.1 LE képes az eszköz.



[2] Raspberry Pi 3 magyarországi ár: https://malnapc.hu/yis/raspberry-pi-3-model-b (2017. április 13.)

4.2. ábra. Raspberry Pi 3

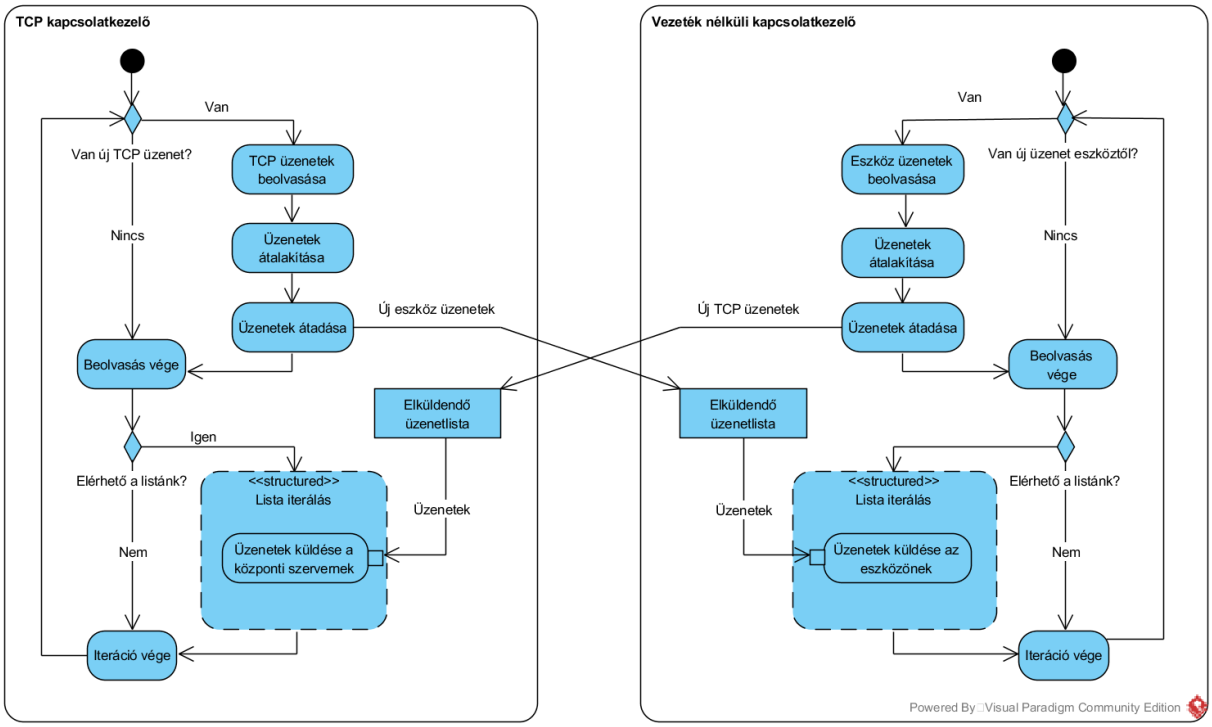
Ahhoz, hogy elkezdhessük használni szükségünk lesz egy SD kártyára, egy 5 voltos áramforrásra, egy HDMI képes monitorra és billentyűzetre, egérre. Az első lépés, hogy egy másik számítógéppel feltelepítsünk egy Linux disztribúciót az SD kártyára. Ajánlott magához a *Pi*-hez készített *Raspbian* Linuxt feltenni. Ha bele dugtuk az összes perifériát és az SD kártyát a *Pi*-be, majd áram alá helyezzük rövid időn belül egy teljes Linuxos felület fogad minket.

4.3.2. Raspberry Pi Foundation

A *Pi* eszközcsalád mögött a *Raspberry Pi Foundation* brit jótékonysági szervezet áll. Az a céljuk ezeknek a csodás eszközöknek a tervezésével, hogy olyan helyekre is eljuthasson a világon a technológia és annak tanítása, ahova pénzügyi okok miatt amúgy nem tudna. Rendkívül nagy sikerre tettek szert, hiszen több mint 10 millió *Pi* lett már értékesítve a világon és olyan cégekkel léptek partneri kapcsolatba, mint például a Microsoft. A sok munka, amit a platform fejlesztésével töltenek, viszont nem csak a világ szegényebb részeit segíti. Mivel ezek az eszközök egyszerűen használhatók, olcsóak és ráadásul nagy közösséget is tudtak maguk mögé állítani, akárki neki tud állni valamilyen elektronikai projektnek szabadidejében.

4.4. Szoftver működés

A hálózati réteg működése könnyen átlátható és annál is egyszerűbben implementálható. A fejezet korábbi részei alapján magától értetődő, hogy a program egyszerre kell kezeljen egy vezeték nélküli kapcsolatot az eszközökkel és egy TCP kapcsolatot a központi szerverrel. Ha ezt a két feladatot egy végrehajtási szálon szeretnénk elvégezni nagy eséllyel gondokba ütköznénk, aminek az lenne az oka, hogy az egyik kapcsolat eseményei nem lennének kellően lekezelve, amíg a másik kapcsolat kapja a figyelmet. Példa eset, az egyik kapcsolaton keresztül épp olvasás történik és a másikon üzenet érkezik. Viszont ez még nem azt jelenti, hogy a feladat megcsinálhatatlan egy futási szálon. Bizonyára lehetséges, de én úgy döntöttem, hogy teljesítményi okokból inkább kettő szálon kezelem a kapcsolatokat. A futtató hardverünk nem fog gátakat szabni e szempontból, tehát nyugodtan használhatunk szálkezelést.



4.3. ábra. Hálózati réteg szoftver működési ábra

Van akkor két végrehajtási szálunk, egy a TCP kapcsolatnak, egy a vezeték nélküli kapcsolatnak. Ha akármelyiken valamilyen üzenet érkezik, azt át kell alakítani a másik fél üzenetformájára és azon az oldalon tovább küldeni. Figyelnünk kell viszont, mivel, ha az egyik szál közvetlen bele szól a másik oldal kapcsolatába, könnyen lehet, hogy elrontja azt. Ami azt illeti elsőnek elkövettem ezt a hibát és nem is működött túl megbízhatóan a rendszer. Megoldani a problémát nem volt nehéz, hiszen elég volt mindkét szálon létrehozni egy listát, amelyben ideiglenesen el lesznek tárolva azok az üzenetek, amiket majd el kell küldeni. Vagyis, ha az egyik szálon érkezik egy üzenet, átalakítja azt, majd belerakja a másik szál listájába. Mikor a másik szál oda kerül, hogy üzeneteket küld, akkor kiveszi a listából a várakozó üzeneteket és kézbesíti. A listához való hozzáadás mindig lezárja a hozzáférést a másik oldal számára, így az csak akkor tudja olvasni és elküldeni az üzeneteket, ha senki nem ad hozzá éppen semmit. Ezzel így teljes mértékben megoldottuk a szálkezelést és minden üzenet szinte azonnal kézbesítésre kerül.

4.4.1. Programkönyvtárak

Említésre került a fejezet elején, hogy a *Boost* könyvtárcsomag egy részét használtam a TCP kapcsolat irányítására. A *Boost.Asio* programkönyvtár adatok aszinkron feldolgozására van fejlesztve és többek között képes `socket`-ek kezelésére is. Mivel a *Bo-*

ost könyvtárcsomag, olyan nagy jelentősséggel bír a C++ fejlesztésben, hogy szinte egy kiterjesztett Standard könyvtárként lehet tekinteni rá, egyértelmű választás volt, mint TCP kapcsolatkezelő könyvtár. Jelentős könnyebbséget hoz a C-beli `socket` programozáshoz képest. Ahhoz, hogy teljesítse a feladatát a TCP kapcsolat kezelő végrehajtási szálunk elsőnek azt kell megnéznie, hogy van-e beérkező üzenetünk a központi szervertől. Ha van, akkor addig olvassuk a bejövő adatot, amíg nem találkozunk egy sorvége karakterrel, mert így jelöltem az üzenet végét. Ehhez a `boost::asio::read_until()` függvényt használtam. A beérkezett üzenetet átfuttatva az átalakító logikánkon belerakjuk a vezeték nélküli kapcsolatot kezelő szál listájába. Ha mindez meg volt vagy esetleg nem is volt beérkező üzenet, akkor megpróbáljuk elérni a saját elküldendő üzenet listánkat. Ennek sikere attól függ, hogy éppen történik-e módosítása a listának. Sikeres elérés esetén végig megyünk a listán és egyenként elküldünk minden üzenetet. A küldést a `boost::asio::ip::tcp::socket.write_some()` függvény segítségével lehet megvalósítani. [BOOST 1]

A fenti logikához képest csak egy kicsivel van több dolga a vezeték nélküli kapcsolatot kezelő végrehajtási szálnak. Az előző alfejezetekből tudhatjuk már, hogy az eszközökkel való kommunikáció az *RF24Mesh* programkönyvtárnak köszönhető. A különbség a másik oldalhoz képest annyi, hogy az *RF24Mesh* esetén az érkező üzenetek beolvasását megbonyolítja az üzenetek cimkézése. Már említettem, hogy ezeket a címkéket arra használjuk, hogy az eszközök kategóriáját (szenzor vagy aktor) meg tudjuk mondani. A beolvasás azzal kell kezdődjön, hogy megnézzük, milyen címkével rendelkezik az üzenet. Ez úgy történik, hogy az `RF24Network::peek()` függvény egy `RF24NetworkHeader` objektumba belerakja az üzenet fejléc részét. A fejléc rész olyan információkat tartalmaz, mint a küldő és a címzett hálózati címe, egy üzenet azonosító és az üzenet címke. Ezután a `RF24Network::read()` függvény a fejléc segítségével az teljes üzenetet belerakja egy általunk megadott változóba. Az olvasás befejezése után minden a TCP kapcsolat kezeléséhez hasonlóan zajlik, csak az üzenetküldéshez a `RF24Mesh::write()` függvényt kell használni. [RF24 1; RF24 2]

5. fejezet

Központi rendszer

Elértünk a teljes rendszer legfontosabb és legbonyolultabb részéhez, mivel, ahogy az gondolható a központi szerver kezel mindent. Itt történik az összes beérkező adat mentése, a felhasználói felületek biztosítása, az adatok alapján történő döntéshozás és a statisztikák elkészítése.

Az egyik lényegesebb döntést az adatbázis tekintetében kellett meghozni, hiszen a rendkívül nagy mennyiségű adatot el kell mentenünk. A kihívást az jelentette, hogy miként lehet kevés tárhely használatával eltárolni a látszólag kevés rendszerességet tartalmazó beérkező állapot jelentéseket. Tehát lesz szó arról, hogy milyen adatbáziskezelő rendszerek jöhettek szóba és melyiknek milyen előnyei, illetve hátrányai lettek volna a fejlesztésnél. Mivel már említettem a 2.3. alfejezetben, hogy a *MongoDB*-t választottam az adatok perzisztens tárolására, ezért kitérek majd arra is, hogy milyen séma alapján lesznek mégis rendszerezve az eszköz adatok és, hogy miért jó az említett adatbáziskezelő rendszer az ilyen szituációkban.

Ahhoz, hogy a központi szerver létrejöhessen több keretrendszerre is szükségem volt. Így például a szerver alapját a *Spring* alkalmazásfejlesztési keretrendszer előre felkonfigurált verziója a *Spring Boot* adja. Nagy előnye a *Spring Boot*-nak, hogy nagyon gyorsan el lehet kezdeni fejleszteni az alkalmazás lényegi részét és nem kell sok időt tölteni a konfigurációval vagy éppen úgynevezett boilerplate kód írásával. A felhasználói felület megvalósításához a *Vaadin* webes UI keretrendszer volt segítségemre. Használata előre megírt komponensek felhasználásával felépített felületek tervezéséből áll. A *Vaadin* mellett szólt még, hogy könnyedén használható *Springes* webalkalmazások készítéséhez, mivel a készítők figyelmet fordítottak a kompatibilitásra és külön támogatják *Spring Boot* alapú webalkalmazások fejlesztését.

Nagy hangvételű részt kap a *Flow-rendszer*, hiszen csak említve voltak a fogalmak

hozzá kapcsolódóan, viszont rendesen megmagyarázva nem. A *Flow-rendszer* felelős a felhasználó által megadott szabályok kezeléséért és betartásáért. A működés gyors menetének megtartásáért kellett alkalmazni néhány megoldást, amik akár gyorsítótáraknak is tekinthetőek, amelyekre azért volt szükség, hogy ne kelljen túl nagy terhelést rakni az adatbázisra feleslegesen, még akkor se ha az adatbázis sokkal nagyobb terhelést is kibírna. Szintén lesz szó arról, hogy mi történik akkor, ha beérkezik új adat a rendszerbe, milyen folyamatok indulnak el.

5.1. Adatbázis

5.1.1. Adatbázis-kezelő rendszer kiválasztása

Az adatbázis-kezelő rendszer megfelelő kiválasztása jelentős befolyással van arra, hogy a fejlesztés során mennyire lesz egyszerű dolgunk az adatkezeléssel. Pont ezért fontos, hogy jól végiggondolt döntést hozzunk, hiszen ha a fejlesztés közben jövünk rá, hogy mégse a jó fejlesztői eszközt választottuk, akkor nehézkes és sok munkát igénylő lesz az átállás.

Mivel a rendszerbe folyamatosan temérdek mennyiségű adat érkezik és ezek az adatok nem éppen a legrendezettebbek vagy éppen sémába illeszthetőek, érdekes és újfajta kihívásokat jelentő feltételeknek kell megfelelnie a kívánt adatbázis-kezelő rendszer. Továbbá a beérkező adatok, amiket tárolni szeretnénk általában magukban értelmezhető kis egységeket alkotnak és így kevés vagy akár semennyi kapcsolat nem létezik köztük. Ezen okok miatt arra kellett jussak, mikor a megfelelő adatbázis-kezelő rendszer után kutattam, hogy egy relációs adatbázis egyáltalán nem lenne kedvező választás. Habár az ilyen típusú adatbázisok mára csodálatra méltó teljesítményre képesek és valószínűleg meg lehet oldani velük is alig rendezett adatok tárolását inkább a NoSQL adatbázisok felé tereltem figyelmemet.

A NoSQL adatbázisokról hallhattunk úgy, mint a modern kor adattárolásának megmentői, hiszen pont a régi, reláció alapú adatbázisok okozta problémákat próbálják orvosolni. Egyre több alkalmazás mögött áll elképesztő mennyiségű adat, aminek kezelése nagy kihívások elé állítja a fejlesztőket. Az egyik legfontosabb szemponttá vált a skálázhatóság az alkalmazások megtervezésénél és míg a relációs adatbázisok nehézkes és vertikális skálázhatóságot biztosítottak csak, addig a legtöbb NoSQL adatbázis beépítetten támogatja a horizontális skálázhatóságot. Ezt az előnyt tovább fokozza az a tény, hogy a szoftverfejlesztés egyre jobban halad a felhő alapú architektúrák használata felé, aminek jelentősen kedvez egy horizontálisan skálázható, sok csomópontból

álló adatbázis klaszter.

NoSQL adatbázisokon belül is van többféle, lehet gráf adatbázis, kulcs-érték adatbázis vagy éppen dokumentumorientált adatbázis. Mindegyiknek meg van a maga előnye, például egy gráf adatbázisban könnyedén ábrázolhatunk kapcsolatokat, összefüggéseket az adataink között vagy egy dokumentumorientált tárolás esetén lehetőségünk van egy folyamatosan változó, sémamentes módon kezelni az adatainkat. A mi esetünkben megfelelő egy dokumentumorientált adatbázis használata, mert a sémamentes tárolás nagy szabadságot ad, amire szükségünk lesz az eszközöktől érkező állapotjelentő üzenetek strukturálatlansága miatt.

Tegyük fel, hogy mégis egy relációs adatbázisban szeretnénk tárolni az adatainkat, annak érdekében, hogy láthassuk milyen problémák lépnek fel és, hogy a dokumentumorientáltság miért is kell számunkra. Tehát az a feladatunk, hogy a másodpercenként több eszköztől is érkező adatot egy táblában el legyenek mentve. Minden bejövő üzenet tartalmaz egy eszköz azonosítót, az eszköz típusát, az aktuális időpontot és egy mérési értéket. A legnaivabb megoldás az lenne, ha készítenénk erre egy táblát, ami pontosan ezeknek az adatoknak megfelelő típusú oszlopokat tartalmazza. Természetesen ez rendkívül tárhelyigényes megoldás lenne, hiszen feltételezve, hogy minden másodpercben csak egy eszköz küld mérési adatot, akkor körülbelül 11 nap futási idő után máris egy millió sort kellene kezeljünk a táblában. Ezt természetesen tovább rontja, hogy egynél több eszköz is lesz a rendszer által kezelve. Ráadásul, mivel az adat szinte csak ömlesztve van egy táblába hosszú és költséges lenne akármilyen lekérdezést is futtatni rajta. A hagyományos szemlélet szerint egy tábla valamely típus egyedeit tartalmazza, melyek relációban állnak más egyedekkel. Ezzel szemben jelen esetben a táblában tárolt adatok heterogének abban a tekintetben, hogy különböző típusú szenzorok mért értékeit írják le. Természetesen következik ebből, hogy hozzunk létre különböző táblákat a különböző eszköztípusok számára, vagy akár közvetlenül a különböző eszközök méréseinek tárolására. Ez azonban ahhoz vezetne, hogy dinamikusan kell táblákat létrehozni, új eszközök, vagy eszköztípusok hozzáadásakor. Ez viszont nem kívánatos. Illetve nincs is szükségünk arra, hogy az adataink relációban álljanak, hiszen az önmagában is értelmesek, ezért sem érdemes használjunk relációs adatbázist és inkább dokumentumorientált tárolásra lesz szükségünk.

Rengeteg dokumentumorientált NoSQL adatbázis létezik a világon, ezért csak arra a pár jelentősebbre fogok kitérni, amik illenek a mi használati esetünkhöz. Ezek név szerint a *MongoDB* és a *Couchbase* adatbázis-kezelő rendszerek, melyek rendre az 1. és a 3. legelterjedtebb dokumentumorientált adatbázisok a világon [DB 1]. A 2. helyen áll az Amazon által fejlesztett *DynamoDB*, ami csak azért marad ki, mert logikus

módon erősen épít az Amazon által nyújtott felhőszolgáltatásokra és nem célunk, hogy ilyen felhő alapú rendszert készítsünk. A *MongoDB* és a *Couchbase* közötti első szembetűnő különbség az adatelérésben rejlik. A *MongoDB* SQL-hez hasonló lekérésekkel dolgozik kollekciókon, amik megfelelnek a relációs adatbázis tábláinak, a *Couchbase* esetében pedig különböző nézeteket határozhatunk meg előre, más-más szempont szerint. A *Couchbase* előre meghatározottságából adódik, hogy csak akkor érdemes használni, ha ritkán kell változtatni az adatok struktúráját és ha tudjuk előre, hogy mit és milyen formában szeretnénk lekérni az adatokat. Ezzel ellentétben áll a *MongoDB* rugalmas lekérdező nyelve, mely az SQL-hez hasonlóan tartalmaz logikai vagy éppen matematikai operátorokat. Illetve ezt a rugalmasságot kiegészíti még a *MongoDB* úgynevezett aggregációs keretrendszere, mely csővezetékszerűen egymás után kapcsolható szűrők és transzformációk végrehajtását teszi lehető az adatokon. Ez által aránylag kevés munka befektetésével hamar tudunk olyan lekérdezéseket írni, amiket felhasználva következtetéseket tudunk levonni az adataink alapján.

Ezek alapján úgy láttam, hogy a *MongoDB* lesz az az adatbázis-kezelő rendszer, ami legjobban kielégíti az igényeinket. Mivel aránylag közel áll a relációs adatbázisok fogalomvilágához, könnyedén át tudtam térni a használatára. Az aggregációs keretrendszer tökéletes eszköz akármilyen statisztika elkészítésére, amit a felhasználóknak megjelenítünk. Dokumentumorientáltságának és sémamentességének köszönhetően tudtam olyan tárolási megoldást találni, ami leegyszerűsíti a folyamatosan bejövő adat kezelését, de ezt majd a következő alfejezetben fogom részletezni.

5.1.2. Tárolási séma

A *MongoDB* egyszerre több adatbázist is tud kezelni, amelyek megegyeznek a relációs adatbázisoknál vett adatbázisok fogalmával. Az adatbázisokon belül kollekcióba szerveződnek a dokumentumaink, tehát a kollekciók megfeleltethetőek a tábláknak és a dokumentumok a tábla sorainak. A különbség az, hogy a dokumentumok felépítésének nem szükséges megegyezniük egy kollekción belül. Egy dokumentum kulcs-érték párokból áll, ahol a kulcs mindig egy sztring, az érték pedig lehet a többféle adattípus közül egy, tömb vagy akár dokumentum is. Ezzel az eszköz készlettel képesek vagyunk könnyedén olyan dokumentumokat alkotni, amik magukban alkotnak független egységet.

Legegyszerűbben egy példán keresztül lehet bemutatni, hogy milyen struktúrát választottam az adatok elmentéséhez. Minden dokumentum egy eszköz egy órányi adatát reprezentálja percenkénti átlagok formájában. A percenkénti átlag tárhely spórolási

okokból lett bevezetve és nincs is szükségünk 10 másodpercenkénti mérések tárolására. Tehát egy dokumentum tárolja az adatot szolgáltató eszköz azonosítóját, típusát, a küldés dátumot, amit órákra van csonkolva és egy tömböt, amiben a percenkénti átlagok vannak. Minden tömb elem tárolja, hogy melyik percet reprezentálja, hogy abban a percben hány mérés érkezett és a mérések összegét. A 5.1. ábrán látható dokumentum egy hőmérőtől származó adatokat tárolja, ahol az küldő eszköz azonosítója `4` és az adatokhoz tartozó dátum `2017-01-01 12:00`. Látható, hogy csak a 30. és 31. percről vannak méréseink, még pedig mindkét esetben 6 darab. Logikusan a percekre lebontott átlag hőmérséklet úgy jön ki, ha a `sum` értékét elosztjuk a `num` értékével.

```
{
        "_id" : ObjectId("587b98c8ffea6d2aece24250"),
        "dev_id" : 4,
        "date_hour" : ISODate("2017-01-01T12:00:00Z"),
        "type" : "temperature",
        "values" : [
                {
                        "minute" : 30,
                        "num" : 6,
                        "sum" : 127.50000495910645
                },
                {
                        "minute" : 31,
                        "num" : 6,
                        "sum" : 128.388999809265137
                }
        ]
}
```

5.1. ábra. A tárolási séma példán keresztül

Az az előnye ennek a sémának egy olyan sémával szemben, ahol egy mérés egy dokumentum lenne, hogy a *MongoDB-ben* sokkal jobb egy dokumentumot frissíteni, mint mindig újat létrehozni. Ez részben abból ered, hogy jelentősen kevesebb dokumentumot kell egyszerre beolvasni a memóriába, illetve kevesebb tárhelyet is foglalnak maguk a dokumentumonként generált azonosítók is. Nem beszélve arról, hogy ezzel az összevont sémával nem kell mindig eltárolni például az küldő eszköz adatait.

Továbbá a percenkénti összegzés elvégzése az adat mentésénél kevesebb számítási erőforrást igényel, mintha akkor kéne összeadni az egybe tartozó értékeket és elosztani a darab számmal, mikor lekérnénk statisztikákat. Ezt a sémát a *MongoDB* egyik solution arhitectjének egy előadása alapján hoztam létre [DB 2].

Eddig azt ismertettem miért kellett dokumentumorientáltnak lennie az adatbázisunknak, de korábban kiemeltem azt is, hogy a sémamentesség is remek előnyökkel járhat. Például változtatás esetén nem szükséges migrálni az adatokat az új rendszerbe. A fejlesztés során a sémamentességet explicit módon nem használtam ki, hiszen valamilyen szinten strukturálni tudtam az adatokat. Azonban volt eset arra, hogy fejlesztés közben kellett változtatni a struktúrán és meg tudtam tartani minden addigi eszköz mérést különösebb nehézség nélkül. Emellett jövőbeli fejlesztés során bármikor jöhet olyan eszköz, amelynek másféle tárolási mód kedvező. Ezen „jövő-állósága” miatt is döntöttem a sémamentesség mellett.

5.2. Keretrendszerek

5.2.1. Spring Boot

A teljes szerver architektúrális alapját a *Spring Boot* adja, amely a *Spring* alkalmazásfejlesztési keretrendszer egy olyan verziója, ami előre megoldja nekünk a *Spring* konfigurációját. Természetesen ez csak úgy lehetséges, ha elfogadunk néhány konvenciót, hogy miként épüljön fel az elkészített alkalmazásunk. A fő célja a *Spring Bootnak*, hogy a fejlesztés a lehető leghamarabb elérje azt a fázist, ahol már az alkalmazás logikát lehet készíteni. Java webalkalmazások fejlesztésénél időről-időre előfordul, hogy nagyon hasonló kódot kell írni bizonyos háttérműködések futásához. Ezen boilerplate kód írásának a kihagyása sok időt és ezáltal pénzt tud megspórolni vállalati fejlesztési környezetben. Én is azért választottam a *Spring Bootot*, mert csak arra kellett figyeljek, ami lényeges volt a szerver működésének szempontjából és így nem vesztegettem az örökké szűkös fejlesztési időt.

A *Spring* keretrendszer magában nem több olyan konfigurációs és programozási modelleknél, amik hasznosak tudnak lenni Java-alapú vállalati alkalmazások fejlesztésénél. Önmagában a *Spring* olyan feladatokat lát el számunkra, mint a tranzakciókezelés, adatkapcsolat elérése, függőség kezelés és injektálás, illetve webes servletek létrehozása. Ezek közül számomra főleg a függőség injektálás és a webes servletek voltak nagy hasznomra, igaz a servletek esetében plusz absztrakciót ad a *Vaadin* keretrendszer. Azonban ez csak a *Spring* alapcsomagja, emellett léteznek olyan modulok

hozzá, mint a *Spring Data*, amely többféle adatelérési technológiát tesz egységesen használhatóvá vagy éppen a *Spring Security*, ami gondoskodik az alkalmazásunk autentikációjáról és autorizációjáról. Ezeknek a moduloknak a segítségével és az alap *Spring* csomaggal tudunk igazán gyorsan és könnyedén webalkalmazásokat létrehozni.

A *Spring Boot* tudja mindazt, amit fent leírtam és ahhoz, hogy használni tudjuk elég egy *Maven* függőséget és egy szülő projektet megadni a mi alkalmazásunk `pom.xml`-jében. Az 5.2. ábrán látható `pom.xml` részlet megadása mellett további *starter* függőségek hozzáadásával a már korábban említett külső modulokat is konfigurációmentesen tudjuk használni az alkalmazásunkban. Ha például a *Spring Data JPA* implementációjának *Spring Boot* verzióját szeretnénk hozzáadni az alkalmazásunkhoz, akkor a `spring-boot-starter-data-jpa` *Maven* függőségre lesz csak szükségünk.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.3.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

5.2. ábra. Szükséges Maven pom.xml részlet Spring Boot használatához

5.2.2. Vaadin

A felhasználói felület teljes egészében a *Vaadin* UI keretrendszer segítségével lett összerakva. Azért összerakva, mivel a *Vaadin* egy komponens alapú rendszer, vagyis különféle előre elkészített felület darabok segítségével és Java kód írásával, rövid idő alatt komplex megjelenítést vagyunk képesek létrehozni az alkalmazásunkhoz. Például az 5.3. ábrán látható Java kódrészlet egy szöveg dobozt és egy gombot fog megjeleníteni a felületen. Észrevehetjük még, hogy a gomb kattintásának lekezelése is *Java-ban* van implementálva és nem valamilyen kliens-oldali nyelven. Ezt a *Vaadin* szerver-vezérelt fejlesztési modelljének köszönhetjük, ami azt jelenti, hogy a böngészőben futó kliens kód folyamatosan kommunikál a szerverrel *HTTP* vagy *WebSocket*

üzenetek formájában. Tehát, ha a felhasználó rákattint a gombra, egy üzenet generálódik a szervernek a gomb lenyomásról és erre a szerver egy értesítés megjelenítési üzenetet küld vissza, amit a kliens egyből végrehajt. A szerver-vezérelt modell rengeteg előnnyel járhat, javíthatja akár az alkalmazásunk biztonságát vagy éppenséggel könnyen változtathatóvá teszi felületet működtető logikát.

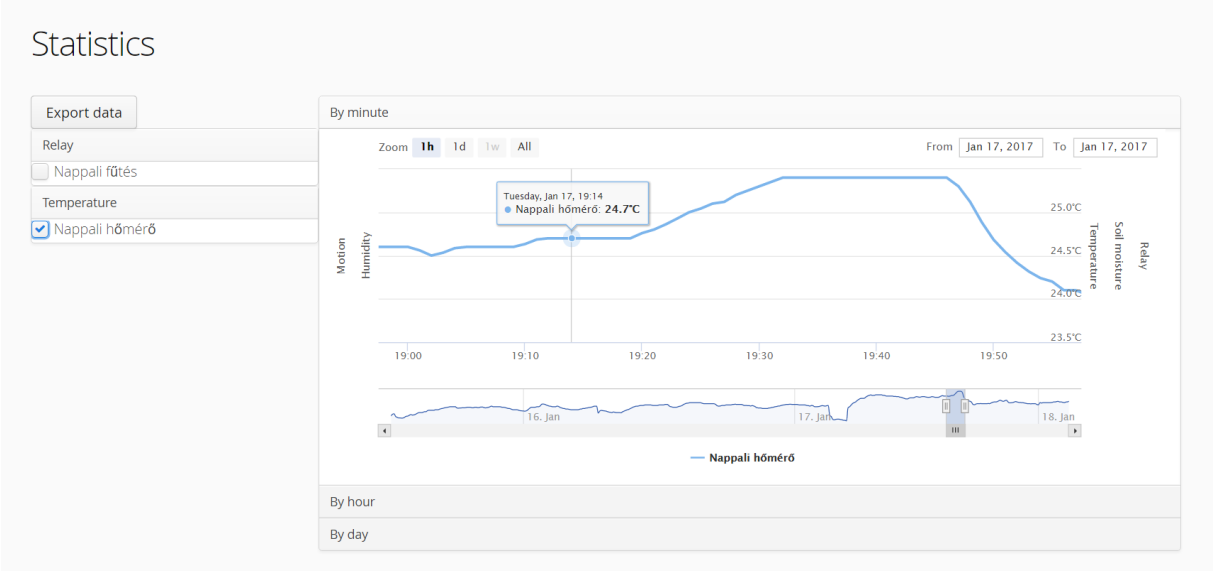
```
TextField name = new TextField("Your Name", "Vaadin");

Button greetButton = new Button("Greet");
greetButton.addClickListener(event ->
    Notification.show("Hello " + name.getValue());

VerticalLayout layout = new VerticalLayout(name, greetButton);
```

5.3. ábra. Példa kód a Vaadin használatára

Természetesen vannak bonyolultabb beépített komponensek, mint egy szövegdoboz, például lenyíló menük vagy akár táblázatok. Azonban ha valamilyen oknál fogva nem találjuk a megfelelő komponenst, akkor sajátokat is létrehozhatunk. Az 5.4. ábrán a rendszer statisztikákért felelős felülete látható, aminek fejlesztése során szükségem volt egy idősor grafikont megjelenítő modulra. Egy kliens-oldali grafikonrajzoló külső programkönyvtár segítségével sikerült is létrehozni az egyedi grafikon megjelenítő komponenst. Ez a példa tökéletesen bemutatja a *Vaadin* rugalmasságát és könnyű bővíthetőségét.



5.4. ábra. Eszköz statisztikák felülete

Számomra azért volt jó választás a *Vaadin*, mert a komponens alapúsága tényleg egyszerűvé tette a felület létrehozását. Illetve, mivel teljes egészében *Java-ban* lehet fejleszteni, nem volt szükséges megtanuljak egy számomra teljesen új kliens-oldali nyelvet, mint például a *JavaScript*. A beépített komponensekhez előre meg voltak írva a kinézetek, így azzal se kellett sok időt vesztegessek, hogy a felület stílusát formázgassam.

5.3. Flow rendszer

A *Flow-rendszer* legfontosabb elemei a már említett *flowk*, melyekre gondolhatunk úgy akár egy-egy szabályra. Ezek a szabályok két listából állnak, egy feltétel listából és egy hatás listából. A feltétel lista olyan kikötéseket tartalmaz, amik ha teljesülnek, akkor végrehajtódik a *flow* hatás listája. A hatás lista pedig tekinthető egy akció sorozatnak, amit szekvenciálisan végigjárva megkapjuk, hogy mi kellett történjen a *flow* teljesülésének időpontjában.

A feltétel lista részei lehetnek például egyes eszközök állapotára vonatkozó megszorítások (pl.: 20°C-nál nagyobb a hőmérsékletet mutat a nappaliban elhelyezett hőmérő) vagy a felhasználó vagy külső rendszer által indított kérés az alkalmazáshoz (pl.: gomb nyomás a kezelőfelületen vagy HTTP kérés egy bizonyos címen), illetve egyéb rendszer állapot feltétel (pl.: időponthoz kötődő feltétel). Fontos megjegyezni, hogy a feltétel lista elemei között logikai ÉS kapcsolat áll, tehát mindnek teljesülnie kell, ahhoz, hogy a *flow* hatása végrehajtódjon.

A hatás lista elemei között szerepelhetnek eszközöknek célzott állapot beállító parancs küldés (pl.: kapcsoljon fel egy lámpa), más rendszerhez történő kérés (pl.: HTTP kérés) és egyéb segéd akció (pl.: késleltetés). Ezeknek a hatás elemeknek egymás után történő végrehajtása adja az adott flow hatását.

Azonban a fejlesztési idő végessége miatt nem jutott idő az összes feltétel és hatás típus implementálására. A feltételek típusok közül az eszköz állapot megszorítások és a hatás típusok közül az eszköz állapot beállítások lettek megvalósítva. Szerencsére ezzel az egy-egy feltétel és hatás típussal is teljes mértékben működő képes a rendszer. Azzal, hogy például nincs lehetőség HTTP kérések indítására és fogadására csak annyit vesztettünk, hogy egyelőre nem tudunk külső rendszerekkel együtt dolgozni.

A rendszer célja az, hogy a *flowk* segítségével a felhasználó szabályokat/feladatokat tud leírni, amelyeket a rendszer majd végrehajt és így lehetséges

lesz bizonyos házkörüli feladatok automatizálása. Azzal, hogy pár egyszerű *flowt* létrehozunk megkönnyíthetjük a mindennapjainkat. Elsőre mindig csak kényelmi funkciókra gondolnánk, amit elvégeztethetünk a rendszerrel (pl.: automatikusan lekapcsoló lámpák), viszont megfelelő mennyiségű idő feláldozása után jelentős pénzösszeget is megspórolhatunk (pl.: időzített fűtésrendszer, ami csak akkor fűt ha otthon vagyunk).

Pár példa a rendszer használatára:

- ha adott szobában nincs érzékelt mozgás, akkor lekapcsolódik a villany
- ha több hőmérő is alacsony értéket mutat, akkor automatikusan fentebb megy a fűtés
- ha egy virág földje kiszáradna, akkor víz engedődik a virág alá
- ha reggeli időpont van, akkor beindul a kávéfőző
- ha a levegő szén-monoxid tartalma átlép egy határt, akkor elindul egy jelző berendezés
- ha besötétedik és van mozgás, akkor a sötétítők leengednek és felkapcsol egy villany

5.3.1. Flow végrehajtás

A *Flow-rendszer* működésének talán legizgalmasabb része a folyamatos ellenőrzés, hogy melyik *flow* lett aktív egy éppen beérkezett üzenet hatására. Az első megoldás, ami eszünkbe juthat, hogy egy végtelen ciklusban futna egy ellenőrző algoritmus, de tudjuk, hogy nem ez a legelegánsabb megoldás. Jelen esetben viszont csak állapot jelentés üzenetek formájában érheti a rendszert olyan külső hatás, ami miatt le kell ellenőrizni a *flowkat*. Ezért ahelyett, hogy lenne egy folyamatosan futó ellenőrző algoritmusunk sokkal hatékonyabb megoldást jelent az, ha az állapot jelentés üzenetek beérkezése váltaná ki az ellenőrzés egyszeri lefutását. Ezzel a mechanikával nem fektetünk folytonosan magas terhelést a futtató hardverre és adatbázisra.

Azzal, hogy csak beérkező adat esetén indítunk ellenőrzést tovább egyszerűsíthetjük a helyzetünket. Ha egy üzenet érkezik természetesen azt is tudjuk, hogy melyik eszköz küldte. Ezen plusz információ segítségével akár célirányosan tudjuk ellenőrizni azokat a *flowkat*, amikben „szerepel” a küldő eszköz. Egy eszköz akkor „szerepel” egy *flowban*, ha annak a *flownak* a feltétel listája tartalmaz, olyan megszorítást, ami az adott eszköz állapotára vonatkozik. Tehát egy beérkezett állapot jelentés hatására csak azokat a *flowkat* kell megnézni, hogy aktívak lettek-e, amelyeknek a feltétel listájában van hivatkozás az üzenetet küldő eszközre. Ezen egyszerűsítés hatására jelentősen kevesebb

flowt kell végignézzünk, hogy végre kell-e hajtani.

Ha tovább szeretnénk optimalizálni az ellenőrzés folyamatát, még van rá lehetőségünk. Azzal, hogy célirányosan kérjük le a kiértékelendő *flowkat* az adatbázistól jelentős javulást értünk el az adatbázis terhelés tekintetében, de még mindig sok lehet, ha akár másodpercenként érkezik új adat. Valószínűleg ezen szintű terhelést nevetve kibírna a választott adatbáziskezelő rendszerünk, viszont mindig jobb előre megoldani problémákat, minthogy később valós gondokat szüljön. Ha bevezetünk egy gyorsítótárként funkcionáló mechanizmust tulajdonképpen teljesen megszüntethetjük az adatbázishoz irányuló azon lekérdezéseket, amik egy adott eszközhöz tartozó *flow* listát kérnek le. Ez a gyorsítótár egy olyan táblázat, melynek kulcsai az eszköz azonosítók és a kulcsokhoz tartozó értékek egy-egy listát tartalmaznak, melyekben a kulcsban szereplő eszközhöz tartozó *flowk* találhatóak. Lényegében listákat tartunk arról, hogy melyik eszköz állapot jelentése esetén melyik *flowkat* kell leellenőrizni. A listák frissen tartása se nagy feladat, hiszen egy *flow* csak akkor változhat, ha azt a felhasználó módosította. Ezáltal ha egy *flowt* módosítanak vagy törölnek csak annyi a dolgunk, hogy kivesszük az összes gyorsítótár listából és végigfutva a feltétel listáján belerakjuk a megfelelő eszköz azonosítójú kulcshoz tartozó listába. Így már csak annyi dolgunk lesz egy állapot jelentés beérkezése esetén, hogy a gyorsító táblázatból lekérjük a küldő eszköznek megfelelő kulcs-érték párt és az abban lévő *flow* listát egyenként ellenőrizzük.

5.3.2. Flow létrehozás

A *flow* létrehozás fejlesztése során több lehetőség közül kellett válasszak, melyek egymáshoz képest könnyebbek-nehezebbek voltak fejlesztési szempontból és felhasználó használati szempontból is. Megpróbáltam egy közép utat találni a lehetőségek közt, hogy ne is legyen sok munkát igénylő fejlesztés és mégis hamar megérthető legyen a felhasználóknak.

A lehető legflexibilisebb beviteli módszer valamilyen egyedi programozási nyelvszerű bemenet lenne. Tehát lehetne egy saját szintaxisa annak, hogy miként kell megadni egy *flowt*. Ezzel a lehetőséggel inkább azokat a felhasználókat lehetne megcélozni, akik hajlandóak akár órákat is tölteni azzal, hogy megvalósítsák a maguknak tökéletes rendszert és ezért képesek lennének megtanulni egy rendkívül leegyszerűsített programozási nyelvet, amit csak erre a célra találtak ki. Természetesen ez nem mindenkinek megfelelő módja annak, hogy *flowt* hozzon létre, nem sokak akarnak ennyi időt elfecsérelni egyszerű szabályok létrehozására. De nem csak a felhasználó szemszögéből rendkívül bonyolult ez a bemenet, hanem fejlesztési szempontból is. Egy saját kód-

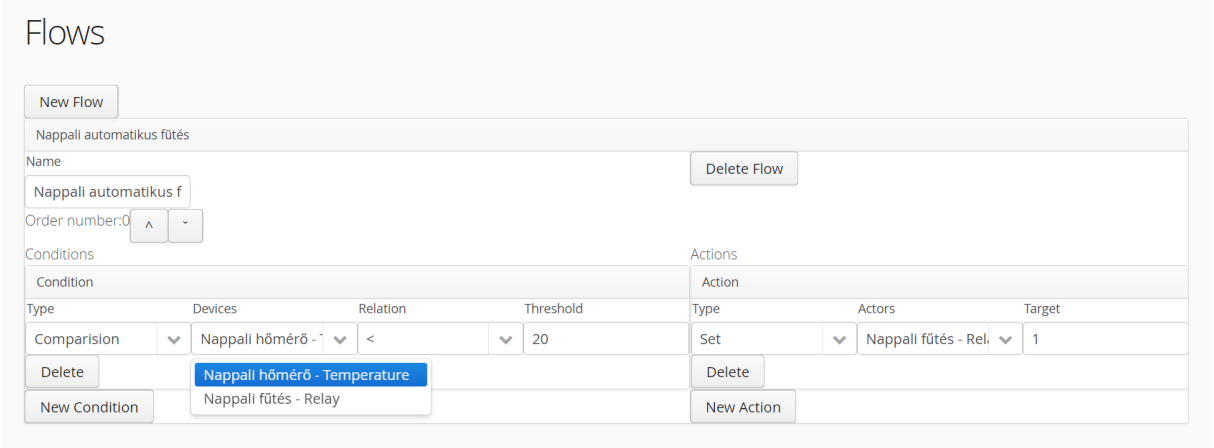
elemzőt kéne írni, ami képes lenne feldolgozni a neki beadott programot. Fel kellene oldani az eszköz neveket magukra az eszközökre vagy esetleg elvégezni matematikai feladatokat, amiket a felhasználó beleírhat a kódba. Sejthető, hogy nem ezt a megoldást választottam, mint végleges *flow* beviteli mód.

```
{
  name: "Nappali fűtés"
  when:
    "Nappali hőmérő" is under '20' AND
    TIME is later than 13:50
  then:
    set "Nappali fűtés 1" to '1',
    set "Nappali fűtés 2" to '1'
}
```

5.5. ábra. Példa program egy lehetséges megvalósításhoz

Ha a felhasználónak akarunk nagyon kedvezni, akkor létrehozhatnánk egy *drag'n'drop* felülettel a létrehozást. Minden feltétel vagy hatás típus a képernyő szélén létrehozhatnánk külön, majd behúzzuk egérrel oda, ahova szeretnénk tenni. Rendkívül átlátható lenne és pár perc alatt össze lehet rakni vele akár aránylag bonyolult szabályokat is. Ami mégis ezen mód ellen szól, hogy elég bonyolult feladat lehet a felületen ezt megvalósítani. Sajnos a felületet kezelő *Vaadin* keretrendszer nem támogat egyszerű *drag'n'drop* elemeket, ezért úgy döntöttem, hogy ez a bemeneti mód se felel meg elvárásaimnak. Túl bonyolult és aránytalanul hosszas lett volna implementálni számomra a rendszer többi részéhez képest.

Végül egy olyan felülettel oldottam meg a problémát, ahol egy-egy oszlop jelképezi a feltételek és hatások listáját minden *flowban* egyenként. A 5.6. ábrán látható egy példán keresztül is. Az oszlopokba egyesével lehet hozzáadni és törölni a feltételeket és hatásokat. Mivel minden feltétel vagy hatás típusnak más bemenetekre van szüksége, ezért ahogy változtatjuk egy feltétel vagy hatás típusát, úgy változik a mellette megjelenő bemeneti mezők kinézete is. Meg lehet még adni egy rendezési számot is, ami amolyan prioritási sorrendnek tekinthető. Erre azért van szükség, ha egy beérkező adat kivált több *flow* végrehajtást is, akkor ne nem-determinisztikus sorrendben lesznek lefuttatva, hanem elsőnek a kisebb rendezési számmal rendelkező és felfele haladva a többi.



5.6. ábra. Képernyőkép flow létrehozásról

6. fejezet

Összefoglalás

Rengeteg eszköztípus van, amit még meg lehetne valósítani és meg is szeretnék még. Néhány példa:

- Légnyomásmérő szenzor
- Eső érzékelő szenzor
- Fény érzékelő szenzor
- Szén-monoxid érzékelő szenzor
- Ablaksötétítő aktor

Köszönetnyilvánítás

Irodalomjegyzék

[BOOST 1] *Boost.Asio Reference Documentation 1.63* URL: http://www.boost.org/doc/libs/1_63_0/doc/html/boost_asio/reference.html (2017. április 19.)

[RF24 1] *RF24Network Class Documentation* URL: http://tmrh20.github.io/RF24Network/classRF24Network.html (2016. április 19.)

[RF24 2] *RF24Mesh Class Documentation* URL: http://tmrh20.github.io/RF24Mesh/classRF24Mesh.html (2016. április 19.)

[DB 1] *DB-Engines Ranking* URL: https://db-engines.com/en/ranking/document+store (2017. április 22.)

[DB 2] *MongoDB for Time Series Data* URL: https://www.mongodb.com/presentations/mongodb-time-series-data-part-1-setting-stage-sensor-management (2017. április 22.)