FELADATKIÍRÁS

Hibatűrő digitális ikrek a KubeEdge platformon

Az „*edge computing*” (perem számítástechnika) rohamosan fejlődő platform-támogatása a taszkok konténerizált formában a „terep” és a „felhő” határán dinamikus telepítése mellett ma már azt is lehetővé teszi, hogy a terepi eszközök és folyamatok digitális ikreit (*Digital Twins, DT*) a platformon belül hozzuk létre. Így a digitális ikerként kezelt tervezési- és futásidejű adatok (pl. szenzoradatok) közvetlenül elérhetővé válnak a telepített alkalmazások számára.

Ez az absztrakció azonban elégtelen abban az értelemben, hogy az elemi hozzáférés-fedésen túl jellemzően nem támogatja a perem számítástechnikában kiemelt jelentőséget élvező *szenzoradat-fúziót* és *algoritmikus hibatűrést* (*Algorithm-Based Fault Tolerance, ABFT*).

A hallgató feladatai a szakdolgozat-feladat keretében a következők.

- Szakirodalmi források alapján adjon egy áttekintést az alapvető szenzoradat-fúziós és ABFT megközelítésekről!
- A KubeEdge platformon tervezzen meg, majd implementáljon egy megoldást, mely létező, regisztrált digitális ikrekre támaszkodva képes olyan „virtuális” digitális ikreket létrehozni, melyek a szenzoradat-fúzió/hibatűrési megoldások rejtését szolgálják!
- Megoldásában implementáljon egy alkalmasan választott szenzoradat-fúziós/hibatűrési mintakészletet! Amennyiben lehetséges, teremtsen lehetőséget a létrehozott hibatűrő digitális ikrek hibatűrési tulajdonságainak a KubeEdge DT-menedzsmentjében kifejezésére is!
- Megoldását demonstrálja is egy (fizikai) minta-rendszeren, ide értve az egyszerű, de célzott hibainjektálást is.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Hibatűrő digitális ikrek a KubeEdge platformon

Készítette
Herjavecz Zoltán

Konzulens
Dr. Kocsis Imre

2021

TARTALOMJEGYZÉK

Összefoglaló 6
Abstract 7
1. Bevezetés 8
2. Felhasznált technikák 10
2.1. Algoritmikus hibatűrés (ABFT) 10
2.1.1. Alkalmazási területek 11
2.1.2. Triple Modular Redundancy 12
2.2. Szenzoradat-fúzió 13
2.2.1. Motiváció 14
2.2.2. Az adatfúzió megvalósításának néhány technikája 15
2.2.3. A szenzoradat-fúzió néhány alkalmazási lehetősége 15
3. Felhasznált szoftverkomponensek 17
3.1. Kubernetes 17
3.1.1. Fejlődéstörténet 17
3.1.2. Előnyei 19
3.1.3. Felépítése 20
3.2. KubeEdge 23
3.2.1. Célkitűzés 24
3.2.2. Megvalósítás 25
4. Feladat áttekintése 29
4.1. Célkitűzés 29
4.2. Architektúra 29
4.3. Fizikai megvalósítás 32
4.4. Implementációhoz felhasznált projektek 33
4.4.1. KubeEdge Examples Github projekt 33
4.4.2. Xiaomi BLE MQTT project 34
5. A projekt implementációja 35
5.1. Eszközmodell leírás Kubernetesben 35
5.2. Eszközpéldány leírók a Kubernetesben 37
5.3. Hő- és páratartalommérő csatlakoztatása 41

5.3.1. A Bluetooth átjáró KubeEdge telepítési leírója 45
5.4. A mapper program bemutatása 49
5.5. A jelaggregátor komponens bemutatása 54
5.5.1. Fizikai szenzorok méréseinek lekérdezése 55
5.5.2. A harmadik, hibás mérési érték generálása 56
5.5.3. Az aggregált jel előállítása 57
5.6. A projekt fájlszintű felépítése 62
5.7. Tapasztalatok, tanulságok 64
6. Összefoglalás 65

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2021. 12. 09.

.....

Herjavecz Zoltán

Összefoglaló

A mai modern magas fokon összekapcsolt világunkban rengeteg adat termelődik az otthonainkban, közterületeken található eszközökön, az Internet of Things, továbbiakban IoT, magyarul a dolgok internetének köszönhetően gyakorlatilag mindenhol. Ezen adatok feldolgozása, kellő szűrése, megfelelő hasznosítása, nagy kihívást állít a hálózati rendszereink felé.

Felmerült a hálózat perifériáin keletkező adatok helyben történő feldolgozásának lehetősége több okból is kifolyólag. Ezen adatok feldolgozása a modernebb eszközeinknek erőforrásainak köszönhetően akár helyben is megvalósítható lenne, nem lenne szükséges minden adatot feldolgozatlanul a felhőbe visszaküldeni, különös tekintettel a mozgó eszközök hálózati kapcsolat bizonytalanságára. Ennek a megvalósítását szándékozza elérhetővé tenni a KubeEdge platform, amely autonómiával ruházhatja fel a perifériákon, a terepen elhelyezkedő IoT eszközöket.

A dolgozatomban bemutatom a KubeEdge platform néhány alkalmazási lehetőségét, megosztom a tapasztalaimat erről a mai napig fejlesztés alatt álló termékről, illetve bemutatok egy megoldást a szenzoradatfúziós és algoritmikus hibatűrés felhő peremein történő alkalmazására, terepi eszközök autonómiájának előnyeire.

Ismertetem továbbá az algoritmikus hibatűrés sok területre kiterjedő előnyeit, hasznát az űrtechnikától otthonainkig.

Abstract

In our current, highly connected world the amount of data being produced every second is incomprehensible. Nowadays data is being produced by our devices in our homes, the devices planted in our streets, basically everywhere thanks to all IoT (Internet of Things) devices. The way how to process, filter, utilize all that that puts a huge challenge in front of us.

It was brought up for multiple reasons that the data produced at the edges of the network should be processed locally. The handling of all that data is now a possibility thanks to our modern devices, thanks to them not all the data has to necessarily be sent to the cloud, especially on moving devices with unreliable connectivity. KubeEdge aims to make it a possibility that the edge devices can be autonomous, that they can handle some of the data by themselves.

In my work, I'll demonstrate some of the utilization possibilities of the KubeEdge platform, I'll share my experiences of this highly developed product and I'll bring you some examples on how to apply algorithm-based fault-tolerance and sensor data fusion, to the benefit of the autonomy of the edge devices. Furthermore, I'll show you some of the benefits of algorithm-based fault-tolerance from space technology to our homes.

1. Bevezetés

Az elmúlt 20 évben hatalmas fejlődésen esett át az informatika világa, amelynek köszönhetően az emberiség olyan dolgokra is képes lett, amikről régebben még álmodni sem mert volna. Ma már szinte mindenki rendelkezik okostelefonnal, a társadalom egyre nagyobb része szereli fel otthonát okoseszközökkel az életük megkönnyítése érdekében, valamint a klímaváltozás elleni harcban is hatalmas szerepe van és lesz is az informatikának. Gondoljunk csak bele az életünk mennyi aspektusát befolyásolja az informatika elterjedése. Ma már mindenki számítógépekkel dolgozik, a kapcsolattartást is megreformálta az újabb és újabb informatikai vívmányok elterjedése, sőt, informatikai algoritmusok befolyásolják már azt is, hogy hogyan járunk munkába és ez az informatikai befolyás csak nőni fog, optimalizációk tömkelegének tárházát elénk tárva.

Számos iparágat gyökeresen felforgatott az informatikai vívmányok elterjedése, a technológia fejlődése, így van ez például a gyártással, ahol a robotizációnak köszönhetően precízebb és gyorsabb lett a gyártás folyamata a kiszámíthatóságot növelve egyaránt. Az informatika iparágában is hatalmas változásokat hozott el és vetít elő a technológia rohamos fejlődése. A telekommunikációban folyamatosan gyorsabb és gyorsabb adatátvitelt biztosító technológiák kerülnek kifejlesztésre, ezzel folyamatosan bővítve a lehetőségek tárházát. Ennek köszönhetően egyre több eszköz található a megszokott szerverfarmokon, telephelyeken kívül, amik adatok feldolgozására is alkalmasak lehetnek, viszont kapcsolatuk a központtal, a felhővel nem megbízható, de az igény a megbízhatóságra, a folyamatos működésre ennek ellenére is fennáll.

A jelen szakdolgozat célja egy, a felhő perifériáin működőképes rendszer bemutatása, amely megbízható szenzoradatokat szolgáltat mind a helyi, mind a központi rendszerek felé nem professzionális kategóriájú, olcsó hardverek felhasználásával. Ehhez az adatok helyben történő folytonos feldolgozása szükséges, erre jelen szakdolgozatban a KubeEdge által kínált lehetőségeket használtam fel.

A második fejezetben ismertetem az algoritmikus hibatűrés és szenzoradat-fúzió néhány alkalmazását, valamint a szakdolgozatban betöltött szerepét, a harmadikban pedig bemutatom a felhasznált technikákat.

A harmadik fejezetben a bemutatom a feladat megvalósításához felhasznált szoftvereket, azoknak belső működését, valamint azok felhasználásának okát, célkitűzéseiket.

A negyedik fejezet a feladat architektúráját, fizikai megvalósítását, illetve a megvalósításhoz felhasznált komponenseket mutatja be, míg az ötödik fejezet bemutatja a kódszintű implementációt, a fontosabb kódrészleteket kiemelve, azokat elmagyarázva.

2. Felhasznált technikák

A jelen fejezetben bemutatom a dolgozat elkészítése közben megismert, valamint felhasznált technológiákat.

2.1. Algoritmikus hibatűrés (ABFT)

Ahogy napjainkban, úgy a múltban is szükség volt redundanciára az informatika területén, hiszen már régebb óta alkalmazzák az informatika vívmányait olyan területeken, ahol megbízható szolgáltatásokra van szükség. A redundancia költségessége miatti kutatások egy része az algoritmikus hibatűréssel foglalkozott, célként a megbízhatóság szinten tartása mellett a költségek csökkentésének lehetőségét kitűzve. A költségcsökkentés szelleméhez hűen az algoritmikus hibatűrést általánosan [1] egy hibaérzékelő, azt lokalizáló majd javító algoritmusként lehet definiálni, mely redundáns számításokat alkalmaz az algoritmusokon belül a hibák érzékelésére és javítására a normális működés megszakítása nélkül.

Egyes kutatások [2] szerint az algoritmikus hibatűrés alacsony redundancia mellett is erősen megbízható eredményekkel szolgál, különös tekintettel arra, hogy az olyan esetek is felismerhetők hibatűrő algoritmusokkal, amelyeknél a rendszer továbbra is működik, van kimenete, vagy pusztán a kimenet maga nem megbízható. Így nagyobb valószínűséggel fenntarthatják hibatűrő algoritmusok a szolgáltatások, az adott rendszer megbízhatóságát a költségek kordában tartása mellett is. Viszont, mint mindennek, a hibatűrő algoritmusoknak is vannak gyenge pontjai. Ezek ellen azonban a programozó felkészültsége esetén nagyon magas pontossággal lehetséges védekezni. Az ily módon felkészített algoritmusokat robusztus hibatűrő algoritmusoknak nevezik. Ezen algoritmusok kicsit általánosabban kiértékelik a program kimenetét, amelyről nagyon magas valószínűséggel meg tudják állapítani, hogy az helyes-e vagy sem, azaz az elvárt működéshez tartozhat-e. Ez a helyesség a kimeneten lefuttatott elfogadási teszttel állapítható meg.

A hibatűrő algoritmusoknak is vannak gyengeségei, például a lebegőpontos számokkal, ahol kerekítési, illetve a felesleges tizedesjegyek elhagyásából keletkező hibák okozhatnak problémákat az egyes elfogadási tesztek végrehajtása folyamán. Kijelenthető, hogy lebegőpontos számok esetében az elfogadási teszt pontatlan, nem megbízható, ugyanis, ha a toleranciahatáron – ami lebegőpontos számításoknál

általánosan nagyobb, mint egész számok esetében – belül esik a hiba, akkor nem fogjuk észlelni azt.

Jelen alkalmazás szempontjából az előbb bemutatott problémák nem relevánsak, így azokra a megoldásokat nem mutatom be itt.

2.1.1. Alkalmazási területek

Az algoritmikus hibatűrést rengeteg területen lehetséges alkalmazni, a magas teljesítményű számítógépektől – azaz szuperszámítógépektől – egészen az alacsony fogyasztású kihelyezett eszközökig, mint a bemutatásra kerülő KubeEdge peremeszközig. Ezekből a területekből szeretnék párat röviden megismertetni ebben a fejezetben.

2.1.1.1. Nagy teljesítményű számítások

Egyes kutatások [3] kiterjednek nagy számítási teljesítményű informatikai rendszerekben történő alkalmazási lehetőségekre is. Az általam szemlézett kutatás a hibatűrő algoritmusok potenciális mátrixműveletekre történő potenciális adaptációját vizsgálja a párhuzamos végrehajtás igényeinek megfelelően.

2.1.1.2. Megállásos hibák

Vannak kutatások [4] a Fail-stop failures, azaz a megállásos hibák (azon esetekre, ahol a hiba következményeképp a szolgáltatás le is áll) hibatűrő algoritmusokkal történő kezelésére is, mivel ilyen eseteket főleg ellenőrző pontok létrehozásával és üzenet naplózásával kezelték elosztott rendszerekben.

2.1.1.3. ABFT alkalmazása az űrtechnikában, mesterséges intelligenciában

Más kutatások [5] pedig a hibatűrő algoritmusok űrtechnikában történő alkalmazását vizsgálják, ugyanis az elmúlt időszakban egyre jobban elterjedt a kereskedelemben megvásárolható hardverek (commercial-off-the-shelf, továbbiakban COTS) alkalmazása, amelyek a radioaktivitásra nagyon érzékenyek, ellenben az elmúlt évek folyamán jelentősen nagyobb számítási teljesítményre lettek képesek, mint a radioaktivitást tűrő speciális hardver. Manapság már űrtechnikában is alkalmazzák a mesterséges intelligenciát, amely nagymértékben támaszkodik a mátrixszorzás műveletére, így az ABFT alkalmazásának jelentős haszna lehet ezen a területen is.

2.1.2. Triple Modular Redundancy

Jelen dolgozatban a Triple Modular Redundancy, továbbiakban TMR hibatűrő algoritmus alkalmazására esett a választás, egy háromszereplős többségi szavazást megvalósítva. A három szereplő egyike jelen esetben egy virtuálisan generált szenzor lett, ez a szenzor fogja tükrözni a hibás jelet, ezáltal bemutatni a TMR algoritmus toleranciáját. A következő fejezetben ismertetett szenzoradat-fúziót szintén ez a virtuális szenzor valósítja meg, amely a többségi szavazás eredményeképpen szolgáltatott értéket tükrözi, így kiküszöbölve az egyes szenzorok esetleges hibáit.

2.2. Szenzoradat-fúzió

A fejezet a [6]-os forrás feldolgozása alapján készült.

A szenzoradat-fúzió lehetősége, mint sok minden, az élőlények működése nyomán merült fel az emberekben. Ehhez nem is kellett saját magunknál, az emberi lényeknél tovább mennünk, hiszen a saját szervezetünk is alkalmazza a szenzoradat-fúzió elvét, tekintsünk csak a saját érzékszerveinkre, van látásunk, szaglásunk, hallásunk, sőt még ízlelésünk és tapintásunk is, és mindezen érzékszervek – ahogy angolul mondják senses – visszajelzéseit dolgozza fel az agyunk egyszerre, ezen érzékelések összességéből alakítjuk ki a környezetünkről a képet. Ennek köszönhetően, még ha ki is esik egy vagy több érzékszervünk, hisz, az emberek megvakulhatnak, megsüketülhetnek, sőt a mostani világjárvánnyal tarkított időszak egyik velejárója, hogy egyesek ideiglenesen elveszíthetik ízlelésüket és szaglásukat, ennek ellenére is tudunk tájékozódni – még ha nehezebben is – a világban a többi „redundáns” érzékszervünknek köszönhetően.

Sajnos szenzoradat-fúzióra több néven is szoktak hivatkozni világszerte, egyesek szenzor fúzióként, adatfúzióként, sőt akár információ fúzióként is hivatkozhatnak rá, ami nem is feltétlenül meglepő, hiszen ezt a technológiát alkalmazzák a robotikától kezdve a mesterséges intelligencián át – gondoljunk csak a Tesla cég elektromos személyautók Autopilot funkciójára – egészen a stratégiai hírszerzésig. Hogy a többféle elnevezés által előfordulható esetleges félreértéseket elkerüljük, az International Society of Information Fusion, továbbiak ISIF, külön definiálta az információfúziót, valamint a szenzorfúziót, avagy szenzoradat-fúziót is.

Az információfúziót definícióját a következőképpen határozta meg az ISIF: az információfúzió olyan elméletet, technikákat és eszközöket foglal magában, amelyeket a több forrásból (szenzor, adatbázisok, ember által gyűjtött információ stb.) szerzett információ szinergiájának kiaknázására terveztek és alkalmaznak úgy, hogy az ebből eredő döntés vagy cselekvés bizonyos értelemben jobb legyen (minőségileg vagy mennyiségileg, a pontosság, robusztusság stb. tekintetében), mint ami lehetséges lenne, ha e források bármelyikét külön-külön használnák a szinergia kiaknázása nélkül.

A szenzoradat-fúzió az információfúzió egy speciális esete, így annak a definícióját a következőképpen határozta meg az ISIF: a szenzoradat-fúzió az érzékszervi adatok vagy az érzékszervi adatokból származó adatok kombinálása oly módon, hogy a

kapott információ bizonyos értelemben jobb, mint az lenne, ha ezeket a forrásokat külön-külön használnák.

2.2.1. Motiváció

A szenzoradat-fúzió használatának több területen is hasznát lehet venni. Mint említettem így van ez manapság az önvezető rendszerekkel, ahol fontos, hogy a jármű megbízhatóan tájékozódjon a környezetéről a biztonságos szolgáltatás érdekében, de így van ez a robotikánál is, ahol az algoritmusok, robotok pontosságához többféle bemenetre, pontos tájékozódásra van szükség, és így lehet ez a különböző magas pontossággal rendelkező méréseknél is, ahol szükség lehet több szenzort elhelyezni az egyes példányok meghibásodásait kivédendő.

2.2.1.1. A szenzoradat-fúzió alkalmazásának fő motiváció

- Szenzor meghibásodás – azon esetek amikor egy már meglévő szenzorcsoportból néhány elem meghibásodik, elromlik
- Korlátozott lefedettség – amikor az alkalmazott szenzorpéldányok nem képesek tájékozódni megfelelő mértékben a térben. Ilyenkor több akár azonos típusú szenzor használata különböző nézetekben kiküszöbölheti az ilyesfajta limitációkat
- Mintavételezésbeli limitációk – azon eset amikor egy szenzor mintavételi frekvenciája, adatszolgáltatási képessége nem megfelelően magas, túl ritka. Ezt a problémát is lehetséges több szenzor alkalmazásával és megfelelő beállításával kiküszöbölni, minimalizálni
- Pontatlanság – amikor az egyes szenzorpéldányok pontossága nem kellően magas, ezt a magabiztosságot több szenzor alkalmazásával tipikus jelentősen meg lehet növelni
- Bizonytalanság – a pontatlansággal szemben a bizonytalanság megfigyelt eszközön múlik mintsem a megfigyelő eszközön, a szenzoron múlna, azt az esetet takarja, amikor nem mérhető minden számunkra fontos tulajdonsága a megfigyelt tárgynak vagy a mérések nem egyértelműek

A fentiekre egy megfelelő példa a személygépjármű tolatóradarja, amely csak az előtte elhelyezkedő tárgyakat látja, a mellette lévőket nem, tehát korlátozott a lefedettsége a radarnak, ha az autó sarkait is le szeretnénk fedni már több szenzorra van szükség.

Ehhez hasonlóan a mai modern, már valamiféle vezetési segédlettel rendelkező járművekben is szükséges többféle szenzort alkalmazni. Vegyük a radaros tempomat példáját, ahol a radar tökéletesen elégséges az előttünk közlekedő gépjárműtől való távolság tartására, ámde nem lehet vele felkészülni egy esetlegesen elénk besoroló járműre, ami potenciális biztonsági kockázatot hordoz magában, mivel ilyenkor a sofőr nem kerül értesítésre időben a biztonsági berendezés által.

2.2.2. Az adatfúzió megvalósításának néhány technikája

Jelen alfejezetben bemutatok néhány adatfúziós technológiát [7] két különböző forrásból származó információk összefűzésére, az információk gazdagítása érdekében. Az alábbi technikák kifejezetten általánosak, széles körben alkalmazhatók.

- Adatasszociáció – olyan adatfúziós módszer, mely az adatok hasonlósága alapján párosítja össze a különböző forrásokból származó adatokat
- Állapotbecslés – ez a technika jelzi, hogy több adatforrás használata esetén magasabb becslési pontosság érhető-e el
- Döntésfúzió – a döntésfúzió egy olyan technika melyet a belső komponensek döntéseinek egyesítésére, fúziójára használnak egy bizonyos általános cél elérése érdekében

2.2.3. A szenzoradat-fúzió néhány alkalmazási lehetősége

Egy szokványos alkalmazása a szenzoradat-fúziónak, hogy egy hibatűrő egységet építünk fúzióval úgy, hogy fizikai szinten legalább három egységet vetünk be, ebben az esetben a többségi szavazás megoldás lehet. Ilyenkor az egyes fizikai elemeknek ugyanazt a területet kell lefedniük, ugyanazt kell mintavételezniük, máskülönben nem lehetne összevetni az érzékelt adatokat.

Egy hasonló több azonos vagy különböző szenzorból álló rendszer előnyei:

- Robusztusság és megbízhatóság – nyilvánvalóan egy több szenzorból álló rendszer egyik velejárója a redundáns működés, azaz, ha a rendszer egyes elemei – feltéve, ha nem mind – ki is esnek, az még mindig képes – még ha alacsonyabb bizonyossággal is – méréseket biztosítani
- Kiterjesztett lefedettség és gyakoribb mintavételezés lehetősége – a rendszer különböző elemei a tér más-más területeit tudják lefedni vagy más ütemben mintavételezhetnek, növelve ezzel a lefedettséget

- Nagyobb magabiztosság – ha a rendszer egy elemének a mérését megerősíti a többi elem mérése is, akkor jelentősen nagyobb bizonyossággal állítható, hogy a mérés helyes
- Kevésbé kétes, kevésbé bizonytalan mérések – a különböző mérésekből egyesített információhalmaz egyértelműbbé teheti a kiértékelést
- Alacsonyabb érzékenység áthallásokra – a rendszer sérülékenysége a dimenziók térbeli növelésével csökkenhet
- Megnövekedett felbontás – amikor ugyanazon tulajdonság több mérését összefűzzük, fuzionáljuk, annak végeredménye eredendően magasabb felbontással fog rendelkezni

Régebben volt kutatás arra kiterjedően, hogy a szenzoradat-fúzió vagy egy szenzor mérésének a teljesítménye-e a magasabb. Ez a kutatás úgy találta, hogy lehetséges olyan optimális fúziós eljárást tervezni – amennyiben az adott szenzor hibázási karakterisztikái ismertek, hogy a fúziós eljárás legalább olyan hatékony, mint egy adott szenzor méréseinek a feldolgozása.

3. Felhasznált szoftverkomponensek

Ebben a fejezetben bemutatom azon szoftvertechnológiákat, melyeket megismertem és felhasználtam a szakdolgozat elkészítése folyamán.

3.1. Kubernetes

A Kubernetes egy jól bővíthető, illetve nyílt forráskódú konténer szervezési és ütemezési rendszer, amely konténerizált elosztott szolgáltatások, alkalmazások szervezésére is alkalmas, így a számítási felhőben is alkalmazható. Mivel konténerszervezési rendszerről beszélünk, annak természetéből származik, hogy az egyes alkalmazások könnyen mozgathatóak.

A Kubernetes a Cloud Native Computing Foundation, továbbiakban CNCF csoport egyik projektje, melynek és a nyílt forráskódnak, több jelentős cég hozzájárulásainak köszönhetően ma már jelentős felhasználói körrel rendelkezik, közéjük több nagy vállalat is tartozik, mint az Ericsson, a Dell, a Canonical, a Huawei és a Hewlett Packard Enterprise is sok más vállalat mellett.

3.1.1. Fejlődéstörténet

A fejezet a [8], [9] és [10]-es források alapján készült

A Kubernetes a Google cég Borg elnevezésű feladatszervezési rendszerének egy kései leszármazottja. A Borgot a Google 2003-2004 környékén kezdte el fejleszteni a saját szolgáltatásainak szervezésére. Ezt a rendszert – a Kubernetesszel egyetemben – klaszter szervezési rendszernek nevezzük, mivel a Google már a kétezres évek közepén is a konténerizált mikroszolgáltatások implementálásán dolgozott. Mint később kiderült, a Kubernetes ezekben az években született meg, még a Borg szoftver kezdetleges formájában.

A Google 2014-ben nyilvánosan is elérhetővé tette a korábban Borg néven ismert rendszerét egy nyilvános forráskódú projektként, amelyhez még az első év folyamán csatlakozott több cég is, köztük a Microsoft, a RedHat, az IBM és a Docker is.

2015-ben kiadásra került az 1.0 verziója a Kubernetesnek, amellyel párhuzamosan a Google a Linux Foundationnel összefogva létrehozta a Cloud Native Computing Foundationt, a CNCF-et. A CNCF célja, hogy egy olyan nonprofit szervezet legyen,

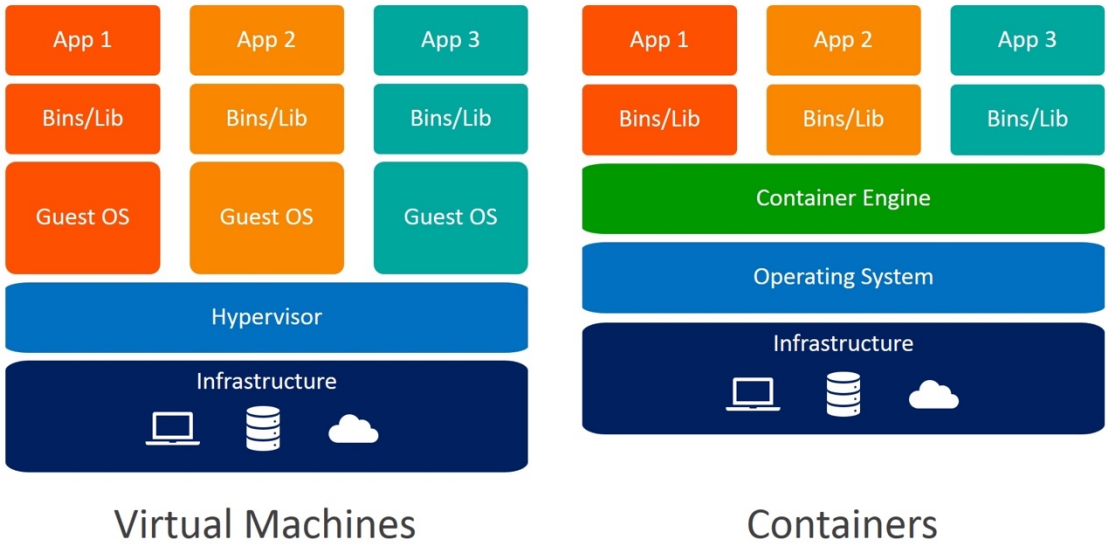
amely összefogja a világ legjobb fejlesztőit, a felhasználókat és a beszállítókat a különböző szoftverkomponensek közös fejlesztésére. Még ebben az évben a közösség tagja lesz az OpenShift, Huawei és a Gondor is, majd 2015 novemberében megszervezik az első KubeCont, az első Kubernetes közösség számára szervezett konferenciát San Franciscoban.

Az ezt követő években a Kubernetes tovább fejlődött, ma már az 1.21.4-es kiadást is ünnepelhetik a fejlesztők, mára az Ericsson is jelentős közreműködője és felhasználója a Kubernetesnek.

3.1.2. Előnyei

Jelen alfejezet a [11] és [12]-es számú forrás feldolgozásával készült.

A 2010-es évek során, ahogy egyre nagyobb szerepet kapott a számítási felhő, egyre nagyobb és fontosabb kihívás lett ezen felhők kihasználtságának magas szinten tartása, optimális hasznosítása. Egy idő után világossá vált a mérnököknek, hogy ez az optimalizálás az alkalmazott virtuális gépeken keresztül nem lehetséges, ugyanis nem elég nagy a kioszthatóerőforrások részletessége, illetve a virtuális gépeknél vannak egyszerűbben, gyorsabban, összességében jelentősen kisebb erőforrásigénnyel rendelkező és mozgatható virtuális platformok. Ezek az utóbb említett platformok mind a Linux konténereken alapulnak, ilyen platformok a Docker, a containerd és cri-o, jelenleg is ezeket támogatja a Kubernetes, bár a Docker támogatás meg fog szűnni hamarosan a Kubernetesben.



3.1. ábra: Virtuális gép kontra konténer architektúra, forrás: [12]

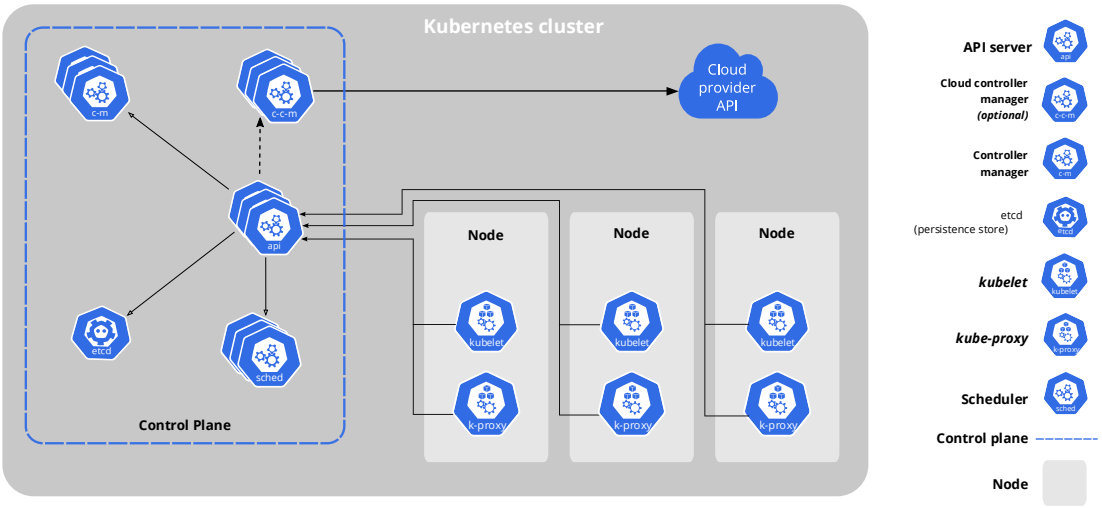
Ahogy az 3.1-es ábra is mutatja, a konténerek egy hatalmas előnye a virtuális gépekkel szemben, hogy nem szükséges az operációs rendszereket több példányban futtatni, az egyes konténerek képesek a gazdarendszer egyes erőforrásait használni, feltéve, ha az a gazda rendszerrel kompatibilis (például Windows rendszeren nem lehetséges Linux alapú konténereket natívan futtatni).

A fenti okoknak köszönhetően a konténer alapú technológiák kezdenek előre törni a számítási felhőben is, hiszen a magasabb kihasználtság érdekében előnyösebb minél

kisebb felbontású erőforrásokat kiosztani, így ugyanis a rendszer egyes felhasználók általi kihasználtságtól függően pontosabban tudják skálázni az erőforrásigényét az adminisztrátorok, az automatizált rendszerek. Ehhez nyilvánvalóan szükség van egy elosztott rendszereken is szétterülő szervezési rendszerre, amellyel meg lehet figyelni az egyes konténerek állapotát, illetve irányítani is lehet őket, és itt mutatott be nagy áttörést a Kubernetes.

A Kubernetes egy olyan konténerszervezési rendszer, amelynek alapjait egy ma már felhőszolgáltatóvá nőtt cég fektette le – ez a cég a Google – és továbbra is számítási felhőket biztosító cégek jelentős közreműködésével, behatásával készül. Ettől fogva a Kubernetes képes több számítógépes rendszeren átívelő csoportokat kezelni az elosztott működés elveihez hűen, ahol az egyes konténerek az állandó és felhasználó által definiálható automatizált megfigyelésnek köszönhetően magas rendelkezésre állással rendelkezhetnek, ugyanis amennyiben a Kubernetes kubelet komponense (a komponenseket később bemutatom) problémát érzékel, újraindíthatja a konténereket, hogy az alkalmazás ismét elérhetővé váljon.

3.1.3. Felépítése



3.2. ábra: a Kubernetes felépítésének magasszintű áttekintése, forrás: Kubernetes hivatalos oldala [13]

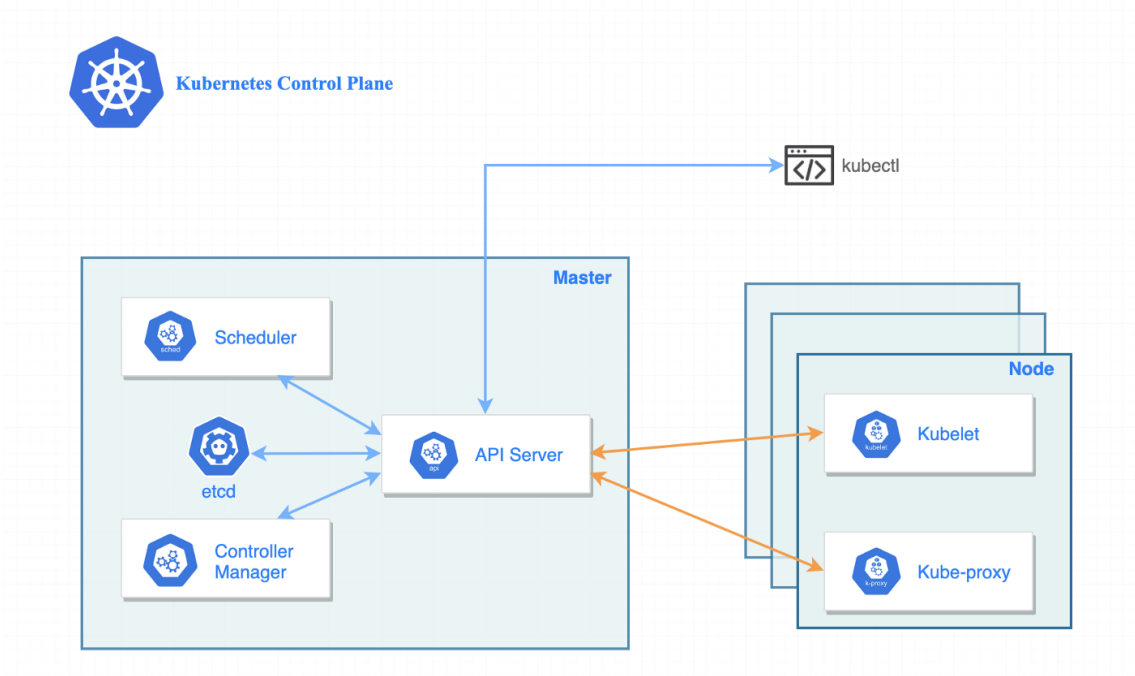
A fejezet a [13]-es forrás, a Kubernetes hivatalos dokumentációja alapján készült.

A Kubernetes egy elosztott működésű rendszert, klasztert hoz létre, ennek megfelelően több munkaállomásból, konténerből és az alkalmazások különböző

komponenseit (amelyek konténerekben futnak) összefogó egységekből, úgynevezett podokból áll. Az előző oldalon látható 3.2-es ábra egy magasszintű áttekintést ad a Kubernetes komponenseiről.

3.1.3.1. Kontrol hálózat

Mint sok hasonló szervezési, vezénylésre alkalmas rendszer, a Kubernetes is rendelkezik egy kontrol – irányítási – hálózattal, amely összekapcsolja az egyes munkaállomásokat a klaszterben, ennek megfelelően a kontrol hálózat észleli és reagál a klaszterben bekövetkezett eseményekre, illetve a kontrol hálózat dönt az alkalmazások munkaállomásra telepítéséről is, ha nincsen specifikálva a futtató állomás. A kontrol hálózat komponensei a klaszter bármely állomásán futhatnak, bár a legtöbb Kubernetest inicializáló eszköz egy és ugyanazon állomásra telepíti az összes komponenst, így egy fő irányítóvá kinevezve az adott állomást.



3.3. ábra: A kontrol hálózat felépítése, forrás: [26]

A kontrol hálózat egy fő komponense a kube-apiserver, amely a Kubernetes Application Programming Interface-ének, továbbiakban API biztosításáért felelős, ez egy tipikusan front end, azaz a felhasználóval kapcsolatot biztosító szolgáltatás. A kube-apiserverből több példány is futhat a klaszteren belül, ami egy rugalmasan skálázható szolgáltatás, amiből a terheléstől függően több példány is futtatható, amelyek között el lehet osztani a forgalmat (load balancing).

A Kubernetes működéséhez szükséges adatokat valahol tárolni is szükséges, erre az etcd szolgáltatást választotta a Kubernetes közösség. Az etcd egy magas rendelkezésre állással rendelkező elosztott kulcs-érték tároló, ami megbízható adattárolást biztosít az elosztott elérésre szoruló rendszereken belül is. Ugyanakkor ez nem jelenti, hogy a felhasználónak semmi teendője nincsen, ugyanis a biztonsági másolatok készítéséről a felhasználónak kell gondoskodnia.

Ahogy korábban említettem, a kontrol hálózat képes dönteni az olyan alkalmazások, podok telepítéséről, amelyeknél nincsen specifikálva a célmunkaállomás. Erről a képességről a kube-scheduler szolgáltatás gondoskodik. Ez egy tipikus ütemező, amely figyelembe veszi az egyes munkaállomások leterheltségét, az adott podok futtatásának feltételeit, és ennek megfelelően helyezi el azokat. Amennyiben nincsen kielégítő célállomás, hibaüzenetet küld a felhasználónak.

Eddig még nem volt szó olyan komponensről a kontrol hálózatban, ami valóban irányítana is, amely az egyes eseményekre reagálva kiadja a szükséges parancsokat a többi komponensnek. Erről a feladatról a kube-controller-manager gondoskodik, ez az a komponens, amely futtatja az egyes irányító folyamatokat. Logikus lenne, hogy minden különböző folyamat más-más példánnyal rendelkezzen, viszont az összes folyamat egy binárisba van összeépítve és egy folyamatként fut. Itt kiemelek néhány főbb irányító szolgáltatást, mint a munkaállomások felügyeletéért felelős szolgáltatás, ezt node controllernek hívják. Ez a szolgáltatás észleli, ha egy munkaállomás elérhetetlenné vagy éppen elérhetővé válik. A munkaállomásokon pedig alkalmazások, szolgáltatások futnak, ezeknek az irányításáért felelős a munkairányító, angolul job controller, ez a szolgáltatás az egyes munkák, feladatok beérkezését figyeli, és azok végrehajtását biztosítja. Ilyen lehet például egy alkalmazás, pod telepítése egy munkaállomásra. Mint már említettem a Kubernetes egy magas rendelkezésre állású elosztott rendszer, ennek megfelelően a Kubernetes által menedzselt alkalmazások is magas rendelkezésre állással rendelkezhetnek amennyiben olyan szolgáltatással is rendelkezik a pod, amely lehetővé teszi számára a szolgáltatások definiálását, amelyeket a végpontirányító, angolul az Endpoints controller regisztrál a Kubernetesen belül, így összekapcsolva a szolgáltatásokat az alkalmazásokkal, podokkal. Ezeken kívül az utolsó irányító folyamat, amelyet megemlítek a Service Account & Token controllers, amely a rendszer üzemszerű működéséhez szükséges fiókokat és API elérési engedélyeket kezeli.

A Kubernetes több felhőszolgáltató cég közreműködésével készül, mivel ők is szeretettel használják vagy tervezik azt használni. Ehhez értelemszerűen szükség van egy olyan komponensre is a kontrol hálózatban, amely a felhőszolgáltató interfészével tartja fent a kapcsolatot, ez pedig a cloud-controller-manager nevet kapta. Amennyiben valaki tanulási célzattal hoz létre egy Kubernetes klaszter példányt, akkor ez a szolgáltatás nem lesz bekapcsolva az adott példányban. A felhőszolgáltatásoktól egyébként más komponensek is függhetnek, mint a Service Controller, ugyanis be lehet kötni az egyes Kubernetes által vezérelt szolgáltatásokhoz a felhőszolgáltató terheléselosztóját is.

3.1.3.2. Munkaállomások felépítése

Ezek a komponensek minden egyes a klaszterbe tartozó munkaállomáson jelen vannak, futnak, karban tartva a futó alkalmazásokat, podokat, illetve ezek biztosítják a Kubernetes végrehajtó környezetét.

A munkaállomásokon az alkalmazások, – mint korábban említettem – podokként futnak, a podok fő építőeleme pedig egy konténer állnak. A kubelet szolgáltatás felelős ezen konténerek életben tartásáért, probléma észlelése esetén bekövetkező újraindításukért.

Az alkalmazások kommunikálni is szoktak, így biztosítva a szolgáltatás elérhetőségét, a megfelelő kommunikációért felelős hálózati szabályokat a kube-proxy szolgáltatás tartja fent, ez gondoskodik arról, hogy az egyes hálózati csomagok a megfelelő irányba legyenek továbbítva.

A fentieken kívül még elengedhetetlen egy konténer szolgáltatás jelenléte a korábban felsorolt lehetőségek közül, hiszen e nélkül nem léteznének konténerek sem.

3.2. KubeEdge

A fejezet a [14]-es és [15]-ös források felhasználásával készült.

A KubeEdge egy eredetileg a Huawei berkeiből származó projekt, amit a Cloud Native Computing Foundation 2019-ben befogadott akkor még, mint kísérleti projektet, majd 2020-ban hivatalosan is inkubátorprojektté nőtte ki magát.

A KubeEdge egy a Kubernetesen alapuló nyílt forráskóddal rendelkező szoftver, amelynek célja a konténeralapú alkalmazásszervezést kiterjeszteni a felhő peremterületeire, a terepre. A KubeEdget három fő probléma leküzdésére tervezték,

mégpedig, hogy áthidalja a hálózati kapcsolat alacsonyabb megbízhatóságát a terep és a felhő között, hogy alacsony erőforrásokkal rendelkező eszközökön is működjön, illetve, hogy skálázható legyen a terepen található erősen szétszórt rendszereken. A projekt jelenleg még inkubációs fázisban áll, azaz jelenleg még a hibák kiküszöbölésén és a nagyobb megbízhatóság elérésén is dolgoznak.

3.2.1. Célkitűzés

Az alfejezet a [16]-os forrás felhasználásával készült el.

Az informatika világa az elmúlt években elért fejlődése nyomán beköltözött az emberek otthonába, életébe az egyre olcsóbban elérhető, kényelmi funkciókat szolgáltató informatikai eszközök révén. Ezt hívjuk a dolgok internetének, angolul Internet of Thingsnek (IoT). Az ilyesfajta végponti eszközöket általában okoseszközöknek nevezzük, ezek segítségével vagyunk képesek környezetünkről, otthonainkról információt nyerni úgy, hogy nem is tartózkodunk otthon. Vegyünk például egy mai tipikus okoseszközt, például egy hőmérőt. Ennek segítségével télen megtudhatjuk, hogy éppen mennyire meleg vagy hideg van az otthonunkban míg dolgozunk, így képesek vagyunk – ismételten egy okoseszköz segítségével – felfűteni otthonunkat mire hazaérkezünk. Ez jelenleg nagy többségben úgy történik, hogy az okoseszközeink a szolgáltatást nyújtó központi felhővel állandó kapcsolatban kell, hogy legyenek. A terepen, azaz otthonunkban található eszközök nem dolgozzák fel az adatokat, pusztán továbbítják azokat. Ugyanez vonatkozik azokra az okoseszközökre, amelyek irányító szerepet játszanak, előző példámból ez a termosztát lenne. Jelenleg az okos termosztátok jelentős többsége is a központi felhőből érkező parancsokat hajtja végre, ritka az, ahol közvetlenül a felhasználótól kapja meg azokat.

A KubeEdge képes függetleníteni a terepen szétszórt okoseszközöket a szolgáltatás központjától, a szerverfarmoktól, erre pedig több oka is lehet. A szerverfarmokkal, szakemberek által üzemeltetett rendszerekkel ellenében az otthonainkban, utcáinkon található eszközök nem feltétlenül ideális körülmények között, többszörös redundanciával működnek. Egy átlagos otthonban nem garantáltan állandó internetszolgáltatás található, így nem adott az állandó összeköttetés az internettel. Ugyanez vonatkozik az áramra is, míg szerverfarmokon vannak másodlagos áramforrások is, otthonainkban, utcáinkon ezek nem találhatóak meg általánosan. Ezen kívül az elmúlt években, különösen napjainkban kiemelt szerephez jutott az adataink

biztonsága, hogy ne kerülhessenek azok illetéktelenek kezébe, megbízható helyen és módon legyenek azok feldolgozva, ne legyenek tárolva. Gondoljunk csak bele, ha egy okos biztonsági kamerát szerelünk fel otthonunkban és az minden képkockát elküld a központi szervereknek, hogy mi is hozzáférhessünk ahhoz. Ez nyilvánvalóan biztonsági kockázatokat rejt magában, és a tudatosabb felhasználókban kételyeket is ébreszthet. Ezeken túl fontos megjegyezni, hogy az informatikai eszközök elmúlt években elért fejlődésének ellenére sem végtelenek a számítási kapacitások, sőt, az ilyen végpontokban elhelyezett eszközökben nagyon is korlátoltak. A KubeEdge pedig ennek megfelelően lett megtervezve, implementálva. A KubeEdge tehát ezekre a problémákra kíván megoldást nyújtani, hiszen az a fő célja, hogy a terepen autonóm működésre is képes eszközöket telepíthessünk, így megoldva a problémát, amikor az internetkapcsolat hiánya megakasztja az adatfeldolgozást. Ugyanakkor, amennyiben nem szükséges minden adatmorzsát visszaküldeni a felhőbe, hiszen helyben feldolgoztuk azokat, kevesebb adatunkkal kapcsolatban merülhetnek fel a biztonsági kockázatok is. Előbbieknek pozitív mellékhatása az is, hogy a helyben történő adatfeldolgozásnál eltűnik az adatok küldésének, illetve fogadásának ideje is, amely az internetkapcsolat minőségétől függően jelentős is lehet.

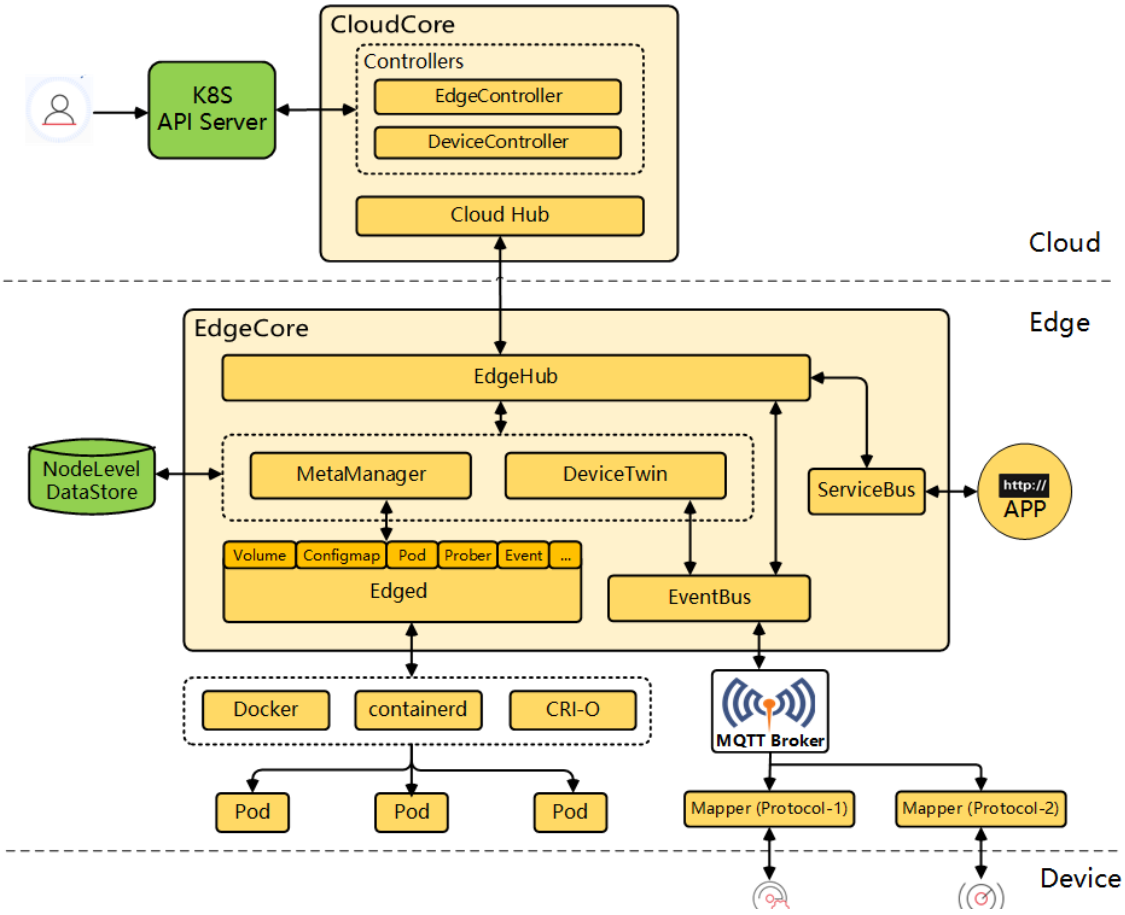
Összességében tehát a KubeEdge lehetővé teszi a felhasználók – legyen az üzleti vagy személyes – számára, hogy az adatok legalább részben a keletkezése helyén feldolgozva kerüljenek továbbításra a felhőben, illetve esetleges hálózati kimaradások esetén autonóm módon működni tudjon a terepen szétszórt rendszer.

3.2.2. Megvalósítás

Az alfejezet a KubeEdge hivatalos dokumentációja alapján készült, [17], [18], [19] és [20]-es forrásokként meg vannak jelölve.

A KubeEdge a Kubernetesen alapul. Ez azt jelenti, hogy működéséhez szükség van egy már létező Kubernetes klaszterre, amely klaszternek viszont nem kötelező rendelkeznie legalább egy hagyományos munkaállomással. Elégséges csak az úgynevezett Master node jelenléte (Master nodenak nevezzük azt a munkaállomást, amelyen helyet foglal az összes a Kubernetes működéséhez elengedhetetlen központi szolgáltatás, azaz a kontrol hálózat komponensei), ugyanis a KubeEdge nem szokványos Kubernetes munkaállomásokat igényel, ez az egyik fő előnye, hogy limitált erőforrásokkal rendelkező eszközökön is képes működni. Ezt a KubeEdge úgy oldja meg,

hogy a Kubernetes felé emulálja egy teljesen működő, hagyományos Kubernetes munkaállomás viselkedését, mindezt egy tömörebb, kisebb erőforrásigényű szolgáltatással kiváltva. Ezt a szolgáltatást EdgeCore-nak hívják és a terepen, az edgen elhelyezkedő eszközökön van jelen. Innen is kapta a nevét (az angol terep vagy perem,



3.4. ábra: A KubeEdge felépítése, forrás: KubeEdge hivatalos dokumentáció [27]

edge kifejezés nyomán). Az EdgeCore magába foglalja az összes mikroszolgáltatást, ő látja el az összes feladatot a terepen, ami a KubeEdge hatáskörébe tartozik (ezt egy kicsit később részletesen is kifejtem). Ahhoz, hogy a Kubernetes ezeket az emulált munkaállomásokat egy hagyományos Kubernetes munkaállomásnak lássa, szükség van egy „fordítóra” is. Ez a „fordító” a Kubernetes Master node-ján fut, a CloudCore szolgáltatás részeként. Ez a CloudCore szolgáltatás tartja a kapcsolatot az EdgeCore szolgáltatások egyes példányaival, így működtetve az egyes munkaállomásokat. Amennyiben a CloudCore szolgáltatással megszakad a kapcsolat, az EdgeCore révén a terepen nyújtott szolgáltatások nem állnak le, ott továbbra is folyhat az adatfeldolgozás,

majd a hálózati kapcsolat helyreálltával szinkronizál a felhő a terepen történő eszközökkel.

Amint a fenti, 3.4-os ábrán látható, a Kubernetesszel ellentétben a KubeEdge esetében a terepen található eszközökön több szolgáltatás fut, ami nem meglepő, hiszen a felhőben a Kubernetes klaszter kötelező jelenléte biztosítja az alapvető szolgáltatások jelenlétét, így a CloudCore-nak pusztán három fő komponense van az EdgeCore összetettségével szemben.

A Cloud Hub komponens áll kapcsolatban a terepen található eszközök EdgeCore szolgáltatásainak hasonlóan Edge Hubnak nevezett komponensével, míg a Device Controller és az Edge Controller felelősek a Cloud Hub által szinkronizált adatok feldolgozásáért, illetve ők adatot is küldenek a Cloud Hub komponensen keresztül. A Device Controller – ahogy a neve is sugallja – a Kubernetesben definiált eszközök irányításáért, illetve azokról érkező adatok Kubernetes felé történő továbbításáért felelős. Az eszközök Kubernetes-beli implementációja a Kubernetes Custom Resource Definitions (CRDs) felhasználásával történik, és minden eszköznek létezik egy modellje, amely definiálja az eszköz tulajdonságait, hogy milyen adatokat tárolunk róla, illetve a példányai, amelyek az egyes fizikai eszközöket reprezentálják. Az Edge Controller komponens pedig a terepen található munkaállomásokon bekövetkezett eseményeket dolgozza fel, illetve a Kubernetes felől érkező változásokról értesítést küld az érintett terepen található eszközöknek, hogy az esetleges változtatások érvényre jussanak. Ez a komponens frissíti a Kubernetesben definiált podok, alkalmazások állapotát is, amelyek a hagyományos Kubernetes munkaállomás helyett a KubeEdge terepi munkaállomásain futnak. Az előbb kifejtett Controller komponensek a Kubernetes API szerverén keresztül kommunikálnak a Kubernetes klaszterrel.

Az EdgeCore – ahogy korábban említettem – sokkal több tevékenységért felelős. A peremen is az úgynevezett Hub komponens, az EdgeHub felelős az Kubernetesszel való összeköttetésért, amin keresztül érkeznek a parancsok például a podokkal kapcsolatosan. Azt a komponenst, amely a podok életciklusát kezeli, EdgeD-nek nevezik, ez a komponens felelős a konténerek megfelelő konfigurációiért, azok létrehozásáért, módosításáért és törléséért, felügyeli az egyes podok egészségét a konfigurációban deklarált módszerek szerint, amelynek eredményét továbbítja a Kubernetes klaszter felé, illetve eltakarítja az elhagyatott, munkaállomásra nem tartozó podokat. Ismerjük tehát melyik komponens tartja karban a podokat, avagy konténereket, ezekben a podokban

viszont célszerűen alkalmazások futnak, amelyeknek úgyszintén kommunikálniuk kell a felhővel valamilyen formában. Ennek megvalósításához az EdgeCore telepítésével egyetemben egy Message Queuing Telemetry Transport szoftver, továbbiakban MQTT is telepítésre kerül a munkaállomásra, melyen keresztül kommunikálnak úgynevezett Mapper protokoll segítségével az alkalmazások. Ez azt jelenti, hogy az alkalmazásoknak a KubeEdge által definiált csatornákon egy meghatározott formában szükséges az adatokat publikálniuk, hiszen az EventBus nem képes minden lehetséges csatornát állandó megfigyelés alatt tartani. Az EventBus az MQTT-csatornákon publikált adatokat pedig továbbítja feldolgozásra a többi komponens felé, amelyek – feltételezve az adatok formátumának megfelelőségét – feldolgozzák azokat. Ezen komponensek egyike pedig a DeviceTwin, amely az Kubernetesben definiált eszközöket reprezentálja az EdgeCore-on belül, ő tárolja fizikai eszközről begyűjtött adatokat a terepi munkaállomáson, ez a komponens felelős az eszköz adatainak felhővel történő szinkronizációjáért. A DeviceTwin az általa begyűjtött adatokat egy lokális SQLite adatbázisban tárolja, ahogy a MetaManager komponens is, amely egy üzenetkezelő az EdgeD és az Edge Hub között, és az általa kezelt metaadatokat a DeviceTwinhez hasonlóan a helyi SQLite adatbázisban tárolja.

Összességében látható, hogy a KubeEdge jelenleg elsősorban olyan alkalmazásokra készült, amelyeknél vagy valamilyen eszköz adatait érdemes feldolgozni annak közelében, vagy ahol csak alacsony számítási kapacitások találhatóak meg.

4. Feladat áttekintése

Jelen fejezetben egy magas szintű áttekintést adok a megvalósított feladat motivációjáról, annak mind fizikai mind szoftveres architektúrájáról, illetve röviden bemutatom a megvalósításhoz felhasznált alkalmazásokat.

4.1. Célkitűzés

A feladat a peremi számítástechnika hasznosságát, létjogosultságát óhajtja demonstrálni, bizonyítani, ugyanis a mai modern – és egyszerre akár olcsó – eszközök olyan megoldásokat tesznek lehetővé, amelyek korábban elképzelhetetlenek lettek volna. Ezeknek szellemében a jelen feladat kifejezett célja az olcsó mai modern eszközök peremi számítástechnikán belüli hasznosságának bemutatása, illetve annak demonstrálása, hogy több olcsó szenzorral ki lehet váltani egy drága, „professzionális” szenzor méréseit a költséghatékonyság jegyében a peremi számítástechnika vívmányainak felhasználásával.

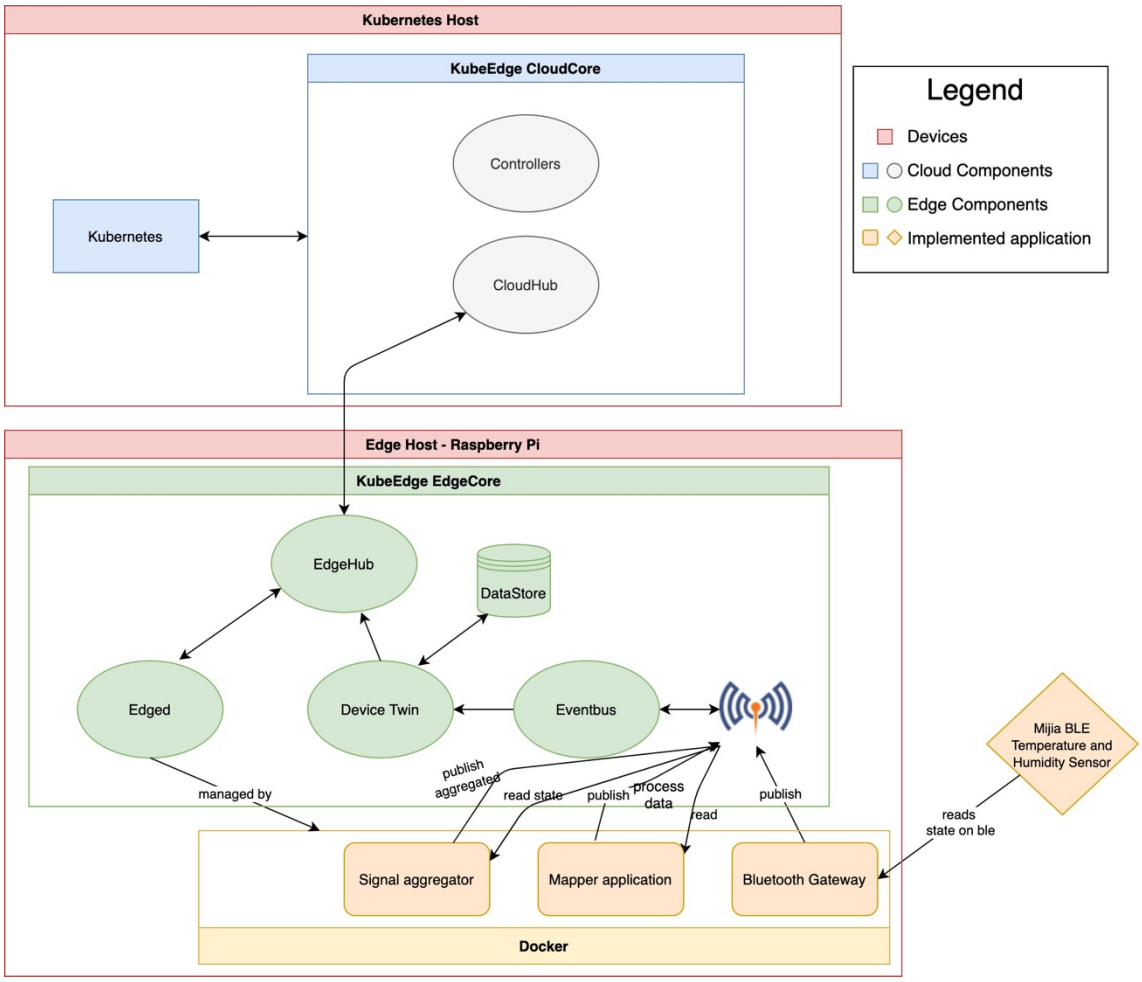
4.2. Architektúra

A 4.1-es ábrán látható architektúrát fogom bemutatni ebben a fejezetben, a képen látható elemek közül a sárgával színezetteket vagy felhasználtam vagy én készítettem teljes egészében.

Ahogy a 4.1-es ábra is mutatja, és ahogy korábban is szó esett róla, a KubeEdge-n alapuló szoftvermegoldások mind konténerizált alkalmazásokat takarnak, így az általam elkészített implementáció jelentős része a Dockerhez kapcsolódik, azon a platformon fut.

Az architektúra legszervesebb eleme egy futó Kubernetes példány, hiszen a KubeEdge a Kubernetes egy kiegészítése, amit a KubeEdge a bemutatásakor ki is fejtettem. A Kubernetes példány elméletileg futhat bárhol, ahol a Kubernetes futtatásához szükséges erőforrások rendelkezésre állnak, és természetesen hálózati kapcsolatot tud teremteni a peremen elhelyezkedő eszközzel, munkaállomással. A Kubernetes példány jelen feladat megvalósításához pusztán egy munkaállomásból is állhat, amit – mivel azon fut a kontrol hálózat minden elengedhetetlen komponense – Master node-nak nevezünk.

Ezen a Kubernetes Master node példányon kerül telepítésre a KubeEdge, melynek a CloudCore komponense fut a Master node-on, ez a szolgáltatás tartja a kapcsolatot a KubeEdge EdgeCore és a Kubernetes Master között. Ezzel az architektúrának tipikus esetben “felhőnek” nevezett része le is van fedve.



4.1. ábra: Az elkészült implementáció funkcionális modellje, készült a KubeEdge architekturális modelljének kiterjesztésével (3.4-es ábra)

A KubeEdge EdgeCore általánosan – a Kuberneteshez és a KubeEdge CloudCore-hoz hasonlóan – a ma elérhető eszközök széles választékán futhat, hiszen úgy tervezték és implementálták, hogy a hardverkövetelményei minimálisak legyenek, melynek köszönhetően lényegében szinte bármilyen korszerű számítástechnikai eszközön képes futni, amely képes kontérek futtatására és természetesen valamilyen Linux disztribúciójú rendszert futtat.

Mint az architektúra is mutatja, a jelen implementáció a Xiaomi Mijia BLE hő- és páratartalom mérő szenzorok által mért adatok feldolgozását valósítja meg. Ezen szenzor adatait egy már létező alkalmazás végzi el, amelyet konténerizált formában futtatunk. Az

alkalmazás, amely Bluetooth gateway (magyarul bluetooth átjáró) néven jelöl a 4.1-es ábra, megfelelő konfiguráció birtokában a felhasználó által konfigurált MQTT-csatornán publikálja a hő- és páratartalomszenzor által mért értékeket. Az előbb említett mért értékeket a következő lépésben egy szintén Docker konténerben futó „mapper” (olyan alkalmazás, ami egy a KubeEdge által definiált formára transzformálja az adatokat) alkalmazás feldolgozza, majd publikálja a KubeEdge által meghatározott MQTT-csatornán az adatokat a feldolgozott, KubeEdge kompatibilis formátumban. Az alkalmazások Go nyelven lettek megírva (hiszen a Kubernetes egy Google projekt alapvetően, a go nyelvet pedig a Google „találta fel”), így szükséges a go nyelv fordítóját is telepíteni valamelyik eszközre annak érdekében, hogy a kódot a felhasználó le tudja fordítani majd futtatni. Ez a bekezdés a feladat megoldásaként implementált szenzoradat-feldolgozást magas szinten be is mutatta.

Az előbb említett MQTT-csatornán a KubeEdge által meghatározott formában publikált adatokat ezután az EventBus komponens továbbítja a DeviceTwin komponens felé, amely eltárolja ezen adatokat a KubeEdge által telepített lokális SQLite adatbázisban, illetve továbbítja azokat a Master node felé.

Végül, de nem utolsósorban pedig az ábrán található még egy Signal aggregator (jelaggregátor) elnevezésű Docker konténer is, ennek a feladata a feladatkiírás szempontjából az algoritmikus hibatűrés és hibainjektálás. Ez az alkalmazás dolgozza fel a csatlakoztatott szenzoroktól érkező adatot, amelyek mellé injektál egy véletlenszerűen generált hőmérsékletet és páratartalmat (amely a két fizikai szenzor átlagától maximum ± 5 fokkal illetve a páratartalom százalékkal térhet el), amelyekből többségi szavazás segítségével próbál előállítani egy aggregált értéket, amely minden időpillanatban megbízható adatokat sugároz, így demonstrálva a több, olcsóbb, alacsonyabb megbízhatóságú eszköz használatának lehetőségét a költséghatékonyság jegyében.

4.3. Fizikai megvalósítás

Az implementáció fizikai megvalósításához a következő eszközöket használtam fel:

- 1 darab x86 architektúrájú számítógép, Ubuntu Server 20.04.3-at futtatva, ez az eszköz futtatja a Kubernetes felhőt, illetve az azt kiegészítő KubeEdge CloudCore komponensét
- 1 darab Raspberry Pi 3 Model B+, ezen eszköz szimulálja a terepen található eszközöket, itt fut a KubeEdge EdgeCore komponense, illetve ehhez csatlakoznak a hő- és páratartalommérő eszközök is és itt futnak az adatfeldolgozást szolgáló konténerek is. Az eszközre az Ubuntu Server 20.04.3-es verziója van telepítve

4.2. ábra: A megvalósításhoz felhasznált konkrét fizikai eszközök

- 2 darab Xiaomi MiJia LYWSDCGQ Hő- és páratartalommérő szenzor

4.4. Implementációhoz felhasznált projektek

Jelen alfejezetben felsorolom az implementációhoz példaként, illetve komponensekként felhasznált alkalmazásokat.

4.4.1. KubeEdge Examples Github projekt

Mivel a Kubernetes, illetve a KubeEdge korábban ismeretlen terület volt számomra, az ismerkedést a termékek magas szintű architektúrájának megismerése és a szoftverek feltelepítése, bekonfigurálása után a KubeEdge mintaprojektjeinek [21] tanulmányozásával folytattam. A KubeEdge projekt közreműködői elérhetővé tettek néhány alkalmazási példát a KubeEdge-re egy Github projekt keretében az érdeklődők számára, ezek közül én is felhasználtam néhány példát. A felhasznált alkalmazásokról a következő két alfejezetben fogok írni.

4.4.1.1. KubeEdge Counter Demo

A KubeEdge Counter Demo alkalmazás egy Docker konténerben futó számláló alkalmazás peremeszközre történő telepítését, illetve annak irányítását demonstrálja.

Ebből az alkalmazásból számomra különösen hasznos volt látni egy KubeEdge eszköz definiálását a Kubernetes Custom Resource Definitions segítségével, ez az alkalmazásból ugyanis példázza, hogy egy modellt – amely az eszköz tulajdonságainak leírója – hogyan kell strukturálni és hogyan lehet a modellnek megfelelő eszközpéldányokat specifikálni. A saját implementációmat az általam definiált eszközökről egy későbbi fejezetben fejtem ki.

Az eszközleíráson túl még hasznos információ volt az egyes alkalmazások implementációit a KubeEdge szintjén látni, illetve azt, hogy hogyan teríti az alkalmazásokat a KubeEdge, azaz hogyan választja meg a céleszközöket. Ennek a megvalósítása manuális lépéseket igényel mind a felhőben, mind a felhő peremén elhelyezkedő eszközön a jelenlegi példában, ugyanis az adott mezők csak egy előre meghatározott értéket tartalmaznak hiszen a szerzők nem tudhatják előre, hogy a felhasználó eszközeinek mi az elnevezése.

4.4.1.2. Temperature Demo

A Temperature Demo példának a Temperature Mapper komponense nyújtotta a legtöbb segítséget számomra, talán nem meglepően, hiszen ez az alkalmazás is foglalkozik a hőmérsékletértékek feldolgozásával, ennek ellenére a hőmérséklet-mérésekhez egyáltalán nem kapcsolódó hasznossággal is bírt számomra.

Ahogy az előző mintapélda példázta számomra, úgy ez megerősítette a Kubernetes CRDs (Custom Resource Definitions) használati módját a KubeEdge projekten. Ettől eltekintve ez az alkalmazás hatalmas segítséget nyújtott a go nyelven megírt, mapper elnevezésű komponens implementálásában, ugyanis a Temperature Mapper go alkalmazása is a KubeEdge szabványoknak megfelelő MQTT-csatornákon publikálja az érzékelt adatok feldolgozott és hasznos elemeit. Továbbá, ezen példa futtatásához is szükséges a felhasználónak magának változtatni egyes fájlokon, lefuttatni egyes parancsokat az alkalmazás peremeszközre történő telepítéséhez, mely analógiát én is alkalmaztam.

4.4.2. Xiaomi BLE MQTT project

A Xiaomi hőmérők Raspberry Pi eszközhöz való csatlakoztatásához egy a Githubon megtalálható publikus projektre [22] esett a választásom, ennek a projektnek a pontos neve xiaomi-ble-mqtt. A választás azért erre a projektre esett mivel ez az alkalmazás lehetővé teszi a mérési adatok MQTT-n keresztül történő publikálását, amelyet ezután már egyszerűen fel lehet dolgozni. Ezt az alkalmazást a funkcionális modellben Bluetooth Gateway-nek neveztem el.

Érdemes megjegyeznem, hogy a KubeEdge képes közvetlenül Bluetooth eszközökhöz is csatlakozni, ahhoz viszont rendelkezésre kell álljon az adott eszköz pontos specifikációja. A KubeEdge Bluetooth eszközhöz történő közvetlen csatlakoztatását a KubeEdge mintaprojektjének a bluetooth-CC2650-demo példája mutatja be.

5. A projekt implementációja

Mint már ismeretes, az általam implementált KubeEdge alkalmazás egy Raspberry Pi 3-as eszközre lett tervezve, amely Bluetooth segítségével csatlakozik a két hő- és páratartalommérőhöz. Kettő ilyen eszközzel rendelkeztem már a háztartásomban, amelyeket személyes felhasználásom során is a felhasznált Raspberry eszközhöz csatlakoztatok, így kínálta magát a feladat, hogy készítsek egy olyan KubeEdge megoldást, amely a két hő- és páratartalommérő adatait feldolgozva egy megbízható értéket szolgáltat a felhő, illetve akár a helyi felhasználó számára. Ehhez végül a TMR algoritmusra esett a döntés, amelyhez viszont hiányzott még a harmadik szereplő. Ez a harmadik jel lett egy a valósan mért adatok alapján véletlenszerű hőmérsékletet és páratartalmat generált hibajel a demonstráció céljából.

5.1. Eszközmodell leírás Kubernetesben

Mint minden, a saját alkalmazásom implementációját is az alapoknál volt érdemes elkezdeni, ez pedig az egyes Custom Resource Definitionök (a továbbiakban CRD) implementálása – ugyanis így lehetséges egyes eszközök digitális ikreit specifikálni. Ezek közül is az első lépés a hő- és páratartalommérő szenzor modelljének definiálása volt, mely nem tartogat különösebb meglepetéseket, az alábbi, 5.1-es ábrán látható a modellt definiáló yaml fájl forrása.

```
apiVersion: devices.kubeedge.io/v1alpha2
kind: DeviceModel
metadata:
  name: hudtemp-model
  namespace: default
spec:
  properties:
    - name: temperature
      description: Temperature collected from the edge device
      type:
        string:
          accessMode: ReadOnly
          unit: degree celsius
    - name: humidity
      description: Humidity collected from the edge device
      type:
        string:
          accessMode: ReadOnly
          unit: percentage
```

5.1. ábra: Az eszközmodell KubeEdge-beli leírójának kódja

Mint látható, a CRD-k esetében definiálni szükséges, hogy az adott modell- vagy példányleírók milyen API verziót használnak, ezeket a KubeEdge projekt implementálja és teszi elérhetővé, mint a második sorban is látható, specifikálni kell az adott leíró fájl fajtáját (kind), ez jelen esetben logikusan a DeviceModel, hiszen egy eszközt modellezünk a leíró segítségével. Definiálni szükséges a modell a nevét, illetve ki lehet választani, hogy melyik Kubernetes névtérbe kerüljön beregisztrálásra. Erre én az alapértelmezett (default) névteret választottam, mivel nincs más Kubernetes vagy KubeEdge alkalmazásom telepítve, így nincsen szükség megkülönböztetésre.

A spec szakaszban pedig definiálásra kerülnek az eszköznek a tulajdonságai (properties), mely jelen esetünkben a hőmérsékletet és a páratartalmat jelenti. Ezekhez a tulajdonságokhoz továbbá még mértékegységet is hozzá lehet rendelni, ez a szenzortól is függhet, az én Xiaomi szenzorom Celsius fokban méri a hőmérsékletet és százalékosan a páratartalmat. Valamint az eszköz egyes tulajdonságaihoz való hozzáférést is korlátozni lehet az accessMode tulajdonság révén, ez lehet ReadOnly vagy ReadWrite. Én az előbbit választottam, mivel mi csak mérjük, azaz olvassuk a hőmérsékletet.

Miután megbizonyosodtam, hogy a fenti leírófájl helyes (amelyről a `kubectl create -f hudtemp-model.yaml` parancs Kubernetes Master Node-n történő lefuttatásával győződtem meg, ugyanis ez a parancs regisztrálja a Kubernetesben az eszközmodellt) implementációjáról, létre lehet hozni az egyes eszközök példányleíróit.

5.2. Eszközpéldány leírók a Kubernetesben

Az előbb bemutatott eszközmodell leíróhoz a fizikai eszközök Kubernetesbeli reprezentációjához léteznie kell eszközpéldány leíróknak is, ebből a kettő fizikai és egy virtuális eszközhöz összesen három leíró fájl is született, az első fizikai mérőét mutatom be az alább látható ábrán.

```
apiVersion: devices.kubeedge.io/v1alpha2
kind: Device
metadata:
  name: hudtemp1
  labels:
    manufacturer: 'xiaomi'
spec:
  deviceModelRef:
    name: hudtemp-model
  nodeSelector:
    nodeSelectorTerms:
      - matchExpressions:
          - key: ''
            operator: In
            values:
              - <edge_node>
status:
  twins:
    - propertyName: temperature
      desired:
        metadata:
          type: string
        value: ''
    - propertyName: humidity
      desired:
        metadata:
          type: string
        value: ''
```

5.2. ábra: Az 1-es számú szenzor KubeEdge eszközpéldány-leírója

Mint látható, az eszközmodell leíróhoz alapvetően nagyon hasonlít az eszközpéldány leíró is, itt is specifikálni szükséges az API verziót, amelyet továbbra is a KubeEdge projekt implementál számunkra. Ezen túl értelemszerűen specifikálni szükséges a leíró fájl típusát (kind mező), amely jelen esetben Device, azaz eszköz, mivel egy fizikai eszköz Kubernetesbeli reprezentációját definiálja. Metaadatokat példányleíró esetében is lehetséges specifikálni, itt megadtam az eszköz gyártóját és az eszköz nevét. Továbblépve, a spec szakaszban specifikálni szükséges, hogy az adott eszközpéldány milyen típusú eszköz egy példánya (a deviceModelRef mező specifikálja), ez értelemszerűen megegyezik az imént bemutatott modell-leíró nevével. Mint látható, eddig egész hasonló a modell- és az eszközleíró szerkezete, értelemszerűen itt a spec

szakaszban definiálni szükséges, hogy milyen eszköz példányát írja le a fájl egyáltalán, ezen túl viszont definiálni kell (a nodeSelector tulajdonság használatával lehet ezt specifikálni) a KubeEdgenek, hogy az adott eszköz melyik munkaállomáson létezik, a KubeEdge ennek megfelelően az adott eszközről gyűjti az eszköz adatait. Ez azért szükséges, mert az adott munkaállomáson működő DeviceTwin KubeEdge komponens regisztrálja az eszközt, illetve az a munkaállomás fogja monitorozni az eszközhöz tartozó MQTT-csatornát (ugyanis a KubeEdge definiálja, hogy az egyes eszközökhöz mely MQTT-csatornák tartoznak [23]). A modell-leírótól eltérően, itt létezik még egy status elnevezésű szakasz is, amely azon túl, hogy definiálja a Digital Twin létezését (amelynek az értékét a DeviceTwin komponens frissíti). Ebben a szakaszban specifikálni lehet, hogy az adott eszköztulajdonságoknak mi a desired, azaz kívánt értékük. Ennek akkor van értelme, ha az adott eszközt irányítani próbáljuk, ugyanis a desired értéket a Kubernetes felhőben specifikálni lehet, majd ez az érték továbbításra kerül a KubeEdge eszközt tartalmazó munkaállomáshoz, ahol az adott érték feldolgozásra kerül, és a helyi eszköz felé továbbításra fog kerülni a kívánt állapot. Mivel jelen esetben az adott eszközt mi nem irányítani, csak olvasni akarjuk, a kívánt értéket üresen kell hagyni.

A leíró elkészültével és a Kubernetesbeli példányosításával már mind a Kubernetes, mind a KubeEdge adatbázisában létezik egy hő- és páratartalom mérő eszköz. Mivel a három eszköz ugyanazokkal a tulajdonságokkal rendelkezik, a másik kettő eszköz leírója csupán a nevében különbözik ezért azoknak a leírókódját nem mutatom be itt. Az alábbi, 5.3-as ábrán látható egy eszközpéldány, illetve annak részletes lekérdezése a Kubernetesből közvetlenül a létrehozása után. Mint az ábrán is látható, az eszköz ilyenkor még csak inicializálva van, értékei még nincsenek ugyanis még nem fut az adott mérési értékeket tápláló alkalmazás.

```
zozo@kubemaster:~/kubeedge_thesis_demo$ kubectl get devices
NAME                 AGE
hudtemp-aggregated   19m
zozo@kubemaster:~/kubeedge_thesis_demo$ kubectl describe device hudtemp-
aggregated
Name:         hudtemp-aggregated
Namespace:    default
Labels:         fault_tolerance=Tolerates-the-failure-of-one-of-the-physical-
hudtemp-sensors
              manufacturer=tmr
              type_of_fault_tolerance=Triple-modular-redundancy
Annotations:  <none>
API Version:  devices.kubeedge.io/v1alpha2
Kind:         Device
Metadata:
  Creation Timestamp:  2021-12-08T23:06:59Z
  Generation:          1
  Managed Fields:
    API Version:  devices.kubeedge.io/v1alpha2
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:labels:
          .:
          f:fault_tolerance:
          f:manufacturer:
          f:type_of_fault_tolerance:
      f:spec:
        .:
        f:deviceModelRef:
          .:
          f:name:
        f:nodeSelector:
          .:
          f:nodeSelectorTerms:
      f:status:
        .:
        f:twins:
    Manager:         kubectl-create
    Operation:       Update
    Time:            2021-12-08T23:06:59Z
  Resource Version:  2031673
  UID:               995c2a10-8358-479a-a874-b98be101a923
Spec:
  Device Model Ref:
    Name:  hudtemp-model
  Node Selector:
    Node Selector Terms:
      Match Expressions:
        Key:
        Operator:  In
        Values:
          pi-ubuntu
Status:
  Twins:
    Desired:
      Metadata:
        Type:       string
      Value:
```

```
    Property Name:  temperature
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  humidity
Events:             <none>
```

5.3. ábra: Egy eszköz Kubernetesbeli ábrázolása a létrehozása után

5.3. Hő- és páratartalommérő csatlakoztatása

Az egyik fő kihívás a Xiaomi hő- és páratartalommérő szenzor KubeEdge-hez való csatlakoztatása volt, de bizakodó voltam, ugyanis a személyes felhasználás során egy Home Assistant [24] nevű szoftverhez csatlakoztatva használom, ami Python nyelven íródott. Lehetséges ugyan a KubeEdge-hez közvetlen hozzácsatlakoztatni bluetooth eszközöket, viszont ennek az implementációjához rendelkezni kéne a jelen esetben használt hő- és páratartalommérő pontos technikai specifikációjával, mely viszont nem publikus, bár az eszköz KubeEdge-hez történő közvetlen csatlakoztatása jelentős mértékben meg is nehezítette volna a szenzorok méréseinek helyben történő feldolgozását – ugyanis ez esetben azok az adatok közvetlenül továbbításra kerülnek a KubeEdge CloudCore és a Kubernetes Master node számára –, bár az a felhő oldalon még mindig elvégezhető lett volna.

Végül találtam egy, a Githubon elérhető, publikus, xiaomi-ble-mqtt elnevezésű projektet [22] amely lehetővé teszi egy Python script segítségével a mért hő- és páratartalom felhasználó által konfigurálható MQTT-csatornán keresztül. Ezt felhasználva elkészítettem egy Docker lemezképet, amelyből amennyiben a felhasználó kézzel elindít egy konténert, bekonfigurálhatja a Cron segédeszközt, hogy a felhasználónak megfelelő időközönként lefuttassa a Python scriptet, továbbítva az MQTT-n keresztül a mért adatokat. Az alábbi ábrán látható a kezdetleges Docker lemezkép Dockerfile leirata. Azért nevezem ezt a lemezképet kezdetlegesnek, ugyanis ez még nem képes arra, egy ebből elindított konténer automatikusan, előre meghatározott időközönként kiolvassa a szenzorok méréseit.

```
FROM ubuntu:20.04

ENV TZ=Europe/Budapest

RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone

RUN touch /var/log/cron.log

RUN apt update

RUN apt install -y libglib2.0-dev \
    pi-bluetooth python3-pip cron nano

RUN pip3 install bluepy paho-mqtt btlewrap

COPY xiaomi-ble-mqtt /root/xiaomi

CMD cron && tail -f /var/log/cron.log
```

5.4. ábra: A *Dockerfile.inital* fájl tartalma

A fenti Dockerfile csak azt biztosítja, hogy a Python script helyes működéséhez szükséges csomagok jelen legyenek a belőle generált lemezképben, illetve, hogy a cron csomag telepítve legyen. Viszont amennyiben ebből a lemezképből valaki elindítana egy konténert, a Cron nem futtatná a számunkra kritikus Python scriptet a háttérben manuális intervenció nélkül. Ezért én ebből a lemezképből elindítottam egy konténert kézzel, a Docker használatával, amiben aztán konfiguráltam a Cron jobot, hogy az fél percenként meghívja a xiaomi-ble-mqtt projekt által rendelkezésünkre bocsátott `data-read.py` fájlnevű Python scriptet, amely a megadott konfiguráció szerinti eszközből a megadott MQTT-csatornára publikálja a mérési adatokat. Ebből a konténerből a crontab konfigurálása után a `docker commit` parancs segítségével készítettem egy lemezképet, amit fel is töltöttem a Docker Hubra (de újra előállítható is a `Dockerfile.initial` fájlból készített lemezkép konténerbe futtatásával, majd abban a konténerben a cron job `'crontab -e'` parancs lefuttatása segítségével ízlés szerint történő bekonfigurálásával, amely a xiaomi-ble-mqtt Github projekt oldalán a 3-as lépésben le is van írva), amelyet már felhasználva manuális intervenció nélkül meghívja a Cron a Python scriptet fél percenként. A Python script konfigurációjához kettő fájl létezik, ebből csak az egyiket kell érdemi mértékben megváltoztatnia a felhasználónak. A fájlok forráskódja a következő oldalon, az 5.5. illetve az 5.6-os ábrákon láthatók.

```
[broker]
client=xiaomi-ble
host=127.0.0.1
port=1883
username=
password=
```

5.5. ábra: Az mqtt.ini.sample fájl tartalma

Az előkészítés folyamán a fájlban a felhasználónak pusztán ki kell törölnie az utolsó két sort, ugyanis a KubeEdge nem állít be biztonsági védelmet az MQTT telepítésekor.

```
[bedroom]
device_mac =
topic = sensors/bedroom
availability_topic = sensors/bedroom/availability
average = 3
timeout = 5

[kidsroom]
device_mac =
topic = sensors/kidsroom
availability_topic = sensors/kidsroom/availability
average = 5
timeout = 5
```

5.6. ábra: A devices.ini.sample fájl tartalma

Az 5.6-os ábrán látható, `devices.ini` fájlban lehet definiálni a szobák neveit, amelyekben az egyes eszközök találhatóak, valamint itt szükséges definiálni az egyes Xiaomi hő- és páratartalomszenzorok MAC címeit, az MQTT-csatorna nevét, a többi mezőn nem szükséges változtatni.

A saját implementációmban az egyes eszközök nincsenek különválasztva, így egy konténer fut a két szenzor adatainak a lekérdezésére. Erre azért volt szükség ugyanis a felhasznált Raspberry Pi 3 munkaállomás véges erőforrásokkal rendelkezik, így célszerű azokkal spórolni.

A fenti két fájl kitöltése és átnevezése (azaz ’.sample’ tag levágása a fájlok végéről) után az alábbi, 5.7-es ábrán látható Dockerfile felhasználásával elkészíthető a végső Docker lemezkép, amelyet a KubeEdge alkalmazásleírója meg fog hivatkozni.

```
FROM hzozo/kubeedge-pi-hudtemp:v0.5.2

COPY xiaomi-ble-mqtt/mqtt.ini /root/xiaomi/mqtt.ini
COPY xiaomi-ble-mqtt/devices.ini /root/xiaomi/devices.ini

CMD cron && tail -f /var/log/cron.log
```

5.7. ábra: A felhasználó által is felhasználandó Dockerfile kódja

Ez a Dockerfile az általam előkészített és Docker Hubra feltöltött lemezképet letöltve felülírja a szükséges konfigurációs fájlokat a lemezképben, hogy aztán amikor a KubeEdge elindít egy konténert a lemezképből, akkor az a konténer már azonnal, automatikusan publikálja a mért adatokat a felhasználó által meghatározott MQTT-csatornákon. Ahhoz viszont, hogy a konténer létrejöjjön a peremi munkaállomáson, szükséges azt definiálni valahogy a Kubernetesben, ezt a következő alfejezetben mutatom be.

5.3.1. A Bluetooth átjáró KubeEdge telepítési leírója

A fent leírt konténeralkalmazás KubeEdge általi szervezéséhez, elindításához valahogy a Kubernetes és ezáltal a KubeEdge tudtára kell hozni a konténerindítási szándékunkat. Ezt a szándékot egy KubeEdge alkalmazás definiálásával tehetjük meg, amely az alábbi, 5.8-as ábrán látható leíró segítségével történik.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: kubeedge-sensor1-mapper
  name: kubeedge-sensor1-mapper
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: kubeedge-sensor1-mapper
  template:
    metadata:
      labels:
        k8s-app: kubeedge-sensor1-mapper
    spec:
      nodeName: <edge_node>
      hostNetwork: true
      containers:
      - name: kubeedge-sensor-mapper1
        image: kubeedge-sensor1-mapper
        imagePullPolicy: IfNotPresent
      restartPolicy: Always
```

5.8. ábra: A kubeedge-hudtemp-mapper1.yaml fájl forráskódja

A fent látható leíró felépítésében hasonlít a korábban bemutatott eszközleíró fájlokhoz. A fő különbség az, hogy ez nem egy eszközt, hanem egy alkalmazást ír le, amelyet szeretnénk, ha a KubeEdge vezényelne. Ennek a leírófájlnak az általam készített alkalmazásban létezik egy párja is, amelyet nem `kubeedge-hudtemp-mapper1.yaml`-nek hanem `kubeedge-hudtemp-mapper2.yaml`-nek neveztem el, ez – mint az előző oldalon kifejtettem – a második szenzorért felelős Python scriptet hívja meg, míg a fent látható fájl az első szenzor méréseinek lekérdezéséért és publikálásáért felelős.

Mint látható, a fenti leíró fajtája Deployment, azaz magyarul talán telepítési leírónak lehetne nevezni, amely azt takarja, hogy a spec szakaszban specifikált konténert fogja elindítani az ott specifikált lemezképből, amelyet csak akkor fog letölteni, ha az nincs jelen. Fontos megjegyezni még, hogy a restartPolicy tulajdonság segítségével definiálni lehet mi történjen amennyiben a futtatott konténer összeomlik, illetve meg lehet adni a hostNetwork segítségével azt is, hogy a gazda számítógép hálózati hozzáférését

használja-e a konténer. Hasonlóan fontos továbbá a nodeName tulajdonság, amellyel specifikálni lehet azon munkaállomás nevét, ahol a konténert futtatni szeretnénk.

Az következő oldalon, az 5.9-es ábrán látható egy a KubeEdge által definiált alkalmazás Kubernetesbeli leírása. Fel szeretném hívni a figyelmet az ábrán a Kubernetes általi eseményekre, ezek az események a Raspberry Pi újraindítása következtében generálódtak. Fontos viszont megjegyeznem hogy viszonylag könnyű túlterhelni a Raspberry Pi operációs rendszerét, az viszonylag gyorsan ki tud futni a memóriából, így az alábbi nézet nagyon hasznos lehet hibakeresés során, úgy ahogy a ’`kubectl logs pod <name_of_pod>`’ parancs kimenete is.

```
zozo@kubemaster:~$ kubectl describe pod kubeedge-bluetooth-gateway-78fffb4d-
5zbdn
Name:         kubeedge-bluetooth-gateway-78fffb4d-5zbdn
Namespace:    default
Priority:     0
Node:         pi-ubuntu/192.168.1.28
Start Time:   Wed, 08 Dec 2021 23:30:59 +0000
Labels:       k8s-app=kubeedge-bl-gw
              pod-template-hash=78fffb4d
Annotations:  <none>
Status:       Running
IP:           192.168.1.28
IPs:
  IP:           192.168.1.28
Controlled By:  ReplicaSet/kubeedge-bluetooth-gateway-78fffb4d
Containers:
  kubeedge-bl-gw:
    Container                                                              ID:
docker://10b28efd65e39f3348db6423ba6c72dfa97fae36ed81881816d9672cb23a0814
    Image:          kubeedge-bl-gw:v1.0
    Image                                                                  ID:
docker://sha256:68018bfb59c76d8d298d89d9aceb63676cae781605be679cb123586166e0b
1a5
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Thu, 09 Dec 2021 10:04:08 +0000
    Last State:     Terminated
      Reason:       Error
      Exit Code:    255
      Started:      Thu, 09 Dec 2021 04:34:54 +0000
      Finished:     Thu, 09 Dec 2021 10:02:29 +0000
    Ready:          True
    Restart Count:  2
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount   from   kube-api-access-
lvnc7 (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  kube-api-access-lvnc7:
    Type:                     Projected (a volume that contains injected data
from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists
for 300s
                             node.kubernetes.io/unreachable:NoExecute
op=Exists for 300s
Events:
  Type     Reason        Age                 From             Message
```

```
----     ------        ----                ----             -------
Warning  NodeNotReady  33m (x50 over 10h)  node-controller  Node is not ready
Warning  NodeNotReady  18m                 node-controller  Node is not ready
```

5.9. ábra: A Bluetooth átjáró Kubernetesbeli reprezentációja

5.4. A mapper program bemutatása

A KubeEdge példaalkalmazások nyomán én is készítettem egy mapper alkalmazást, ugyanis a Bluetooth Gateway, azaz Bluetooth átjáró nem a KubeEdge által elfogadott formátumban publikálja az adatokat az MQTT-n keresztül. Ez a mapper alkalmazás alapvetően azt a feladatot látja el, hogy folyamatosan monitorozza a Bluetooth átjáró által használt MQTT-csatornát (amelyet a felhasználó specifikál), hogy aztán az ott publikált adatokat átalakítsa KubeEdge számára feldolgozható formába majd publikálja azokat a KubeEdge által specifikált MQTT-csatornán.

A mapper program Go nyelven épült, így ugyanis fel tudtam használni a KubeEdge Temperature Mapper példaalkalmazásából a mapper implementáció egy részét, de ami a legfontosabb, fel tudtam használni a KubeEdge szabványainak megfelelő MQTT adatátviteli formátumot.

Az imént említett KubeEdge kompatibilis formátumot a `createActualUpdateMessage` függvény állítja elő, ez az 5.10-es, alábbi ábrán látható.

```
func createActualUpdateMessage(tempValue string, hudValue string)
DeviceTwinUpdate {
    var deviceTwinUpdateMessage DeviceTwinUpdate
    actualMap := map[string]*MsgTwin{"temperature": {Actual: &TwinValue{Value:
&tempValue}, Metadata: &TypeMetadata{Type: "Updated"}}, "humidity": {Actual:
&TwinValue{Value: &hudValue}, Metadata: &TypeMetadata{Type: "Updated"}}}
    deviceTwinUpdateMessage.Twin = actualMap
    return deviceTwinUpdateMessage
}
```

5.10. ábra: A publikált üzenetet előállító függvény kódja

Mint látható, az üzenethez a lényegi adatokon kívül még egy időbélyeget is hozzá kell fűzni, illetve az adatokat szöveges formátumban továbbítja.

A fenti üzenetet a `publishToMQTT` függvény publikálja a megfelelő csatornán, az alábbi, 5.11-es ábrán látható ez a függvény.

```
func publishToMqtt(temp float32, hud float32) {
    updateMessage   :=   createActualUpdateMessage(fmt.Sprintf("%.2f",   temp),
fmt.Sprintf("%.2f", hud))
    twinUpdateBody, _ := json.Marshal(updateMessage)

    token := publishingClient.Publish(edgeTopic, 0, false, twinUpdateBody)

    if token.Wait() && token.Error() != nil {
        fmt.Println(token.Error())
    }
}
```

5.11. ábra: Az MQTT-csatornára publikáló függvény forráskódja

A lebegőpontos szám szöveggé konvertálása is ebben a függvényben történik, a felhasználó kímélése érdekében jelen esetben viszont 2 tizedesjegyre korlátoztam azt.

Mivel az MQTT-csatornát és a felhasználó határozza meg a Bluetooth átjáróban, így azt a konstansok között szükséges specifikálnia a felhasználónak.

Az `edgeTopic` változó tartalmazza a KubeEdge által meghatározott csatorna nevét, ezt figyeli ugyanis az EventBus komponens. Ebben az esetben a `'<your_device>'` stringet kell a felhasználónak felülírnia az általa definiált Kubernetesbeli eszköz nevével (mely `'kubectl get devices'` paranccsal ellenőrizhető). A topic_device változó kell, hogy tartalmazza a Bluetooth átjáróban bekonfigurált MQTT-csatorna nevét.

Megemlítendő még a `subscribe` azaz a feliratkozás függvénye a programnak, ez ugyanis a fő mozgatórugója az egész folyamatnak, ez a függvény az alábbi, 5.12-es ábrán látható.

```
func subscribe(client mqtt.Client) {
    token := client.Subscribe(topic_device, 1, func(client mqtt.Client, msg
mqtt.Message) {
        if string(topic_payload) != string(msg.Payload()) {
            topic_payload = msg.Payload()
            var sensorData SensorData
            err := json.Unmarshal([]byte(topic_payload), &sensorData)
            fmt.Println("error", err)
            publishToMqtt(sensorData.Temperature, sensorData.Humidity)
        }
    })
    token.Wait()
}
```

5.12. ábra: Az MQTT-csatornára feliratkozó függvény forráskódja

Mint látható, hogy ez a függvény figyeli a felhasználó által a Bluetooth átjáróban és a `deviceTopic` változóban specifikált MQTT-csatornát. Ha a kettő érték a Bluetooth átjáró konfigurációjában és a `deviceTopic`-ban nem egyezik meg, akkor a két alkalmazás el fog beszélni egymás mellett és nem fognak frissülni a mért adatok.

A szenzorok mérési adatainak átmeneti tárolásához szükséges volt létrehozni egy struktúrát, mely a `SensorData` nevet kapta, ez a struktúra az 5.13-as ábrán látható.

```
type SensorData struct {
    Temperature float32 `json:"temperature"`
    Humidity    float32 `json:"humidity"`
    Battery     int     `json:"battery"`
    Average     int     `json:"average"`
}
```

5.13. ábra: A szenzoradatokat tároló struktúra leírása

Fontos még megjegyeznem, hogy a felhasználónak – amennyiben több mapper alkalmazást is tervez futtatni – kötelező legalább a második mapper alkalmazás lefordítása előtt átnevezni a már fent említett átnevezendő változókon túl a lenti, 5.14-es ábrán látható MQTT kliens azonosítóját, ezért a kód egy generikus, `sed` paranccsal egyszerűen lecserélhető szöveget tartalmaz.

```
    subscriptionClient = connectToMqtt("subsciption_<device_number>")
    publishingClient = connectToMqtt("publishing_<device_number>")
```

5.14. ábra: A generikus, könnyen módosítható MQTT kliensek

Az alkalmazás többi része nagyrészt a KubeEdge mintapéldái közül a Temperature Mapper példaalkalmazásból származik és a feladat szempontjából különleges funkcionalitással nem bír, így nem emeltem ki azokat a részleteket.

Tehát, amint a mapper alkalmazás telepítésre kerül – feltéve, hogy a bluetooth átjáró is példányosítva lett addigra, helyes konfigurációval -, amint az publikálja az első mérési adatokat a megfelelő MQTT-csatornán, a KubeEdge elkezdi feldolgozni azokat, és a következő, 5.15-ös ábrán látható módon frissítésre kerülnek a Kubernetesben az eszköz mérési értékei.

```
zozo@kubemaster:~$ kubectl describe device hudtemp1
Name:         hudtemp1
Namespace:    default
Labels:       manufacturer=xiaomi
Annotations:  <none>
API Version:  devices.kubeedge.io/v1alpha2
Kind:         Device
Metadata:
  Creation Timestamp:  2021-12-08T23:06:46Z
  Generation:          14194
  Managed Fields:
    API Version:  devices.kubeedge.io/v1alpha2
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:labels:
          .:
          f:manufacturer:
      f:spec:
        .:
        f:deviceModelRef:
          .:
          f:name:
        f:nodeSelector:
          .:
          f:nodeSelectorTerms:
      f:status:
    Manager:      kubectl-create
    Operation:    Update
    Time:         2021-12-08T23:06:46Z
    API Version:  devices.kubeedge.io/v1alpha2
    Fields Type:  FieldsV1
    fieldsV1:
      f:status:
        f:twins:
    Manager:         cloudcore
    Operation:       Update
    Time:            2021-12-08T23:40:26Z
  Resource Version:  2150663
  UID:               0859e03c-2ace-45f5-851d-9058d9f882b3
Spec:
  Device Model Ref:
    Name:  hudtemp-model
  Node Selector:
    Node Selector Terms:
      Match Expressions:
        Key:
        Operator:  In
        Values:
          pi-ubuntu
Status:
  Twins:
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  temperature
    Reported:
      Metadata:
```

```
        Timestamp:  1639045146490
        Type:       string
      Value:        23.90
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  humidity
    Reported:
      Metadata:
        Timestamp:  1639045146490
        Type:       string
      Value:        25.70
Events:             <none>
```

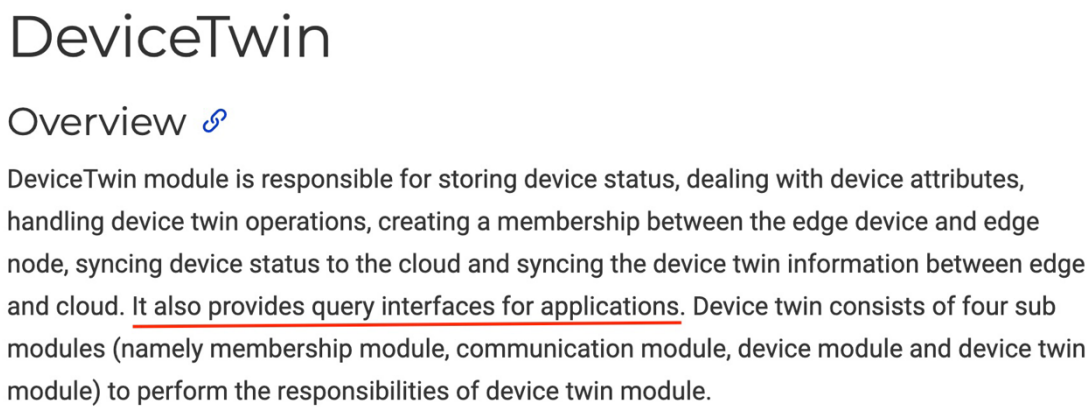
5.15. ábra: Az eszköz Kubernetesbeli reprezentációja, immár frissített mérési adatokkal

5.5. A jelaggregátor komponens bemutatása

A mapper alkalmazás pusztán csak a fizikai szenzorok méréseit táplálja be a KubeEdge-be és ezáltal a Kubernetesbe, azonban az nem valósít meg semmilyen logikát, adatfeldolgozást, így szükség volt még legalább egy szoftverkomponensre. Ez a komponens a Signal aggregator, azaz jelaggregátor nevet kapta.

A jelaggregátor feladata viszonylag összetett, ugyanis a Raspberry Pi limitált erőforrásaira tekintettel ez a komponens többszörös feladatkört valósít meg. Ezeket a feladatköröket alapvetően három területre lehet felosztani: a fizikai szenzorok mérési eredményeinek begyűjtése, a harmadik, virtuálisan generált hibás jel előállítása, valamint az így rendelkezésünkre álló három mérési eredmény – amelyekből az egyik virtuálisan lett előállítva – feldolgozása.

Meg szeretném jegyezni, hogy a jelaggregátor program készítése közben felfedeztem a KubeEdge egy sajnálatos limitációját, amely a digitális ikrekhez kapcsolódik. A limitáció az hogy egyelőre a KubeEdge DeviceTwin modulja nem bocsát rendelkezésünkre egy egyértelmű, jól dokumentált interfészt az egyes eszközök közvetlen, SQL alapú lekérdezésére. Szintén megjegyzendő viszont, hogy a KubeEdge projekt egyelőre még az életének nagyon korai szakaszában van, így egyelőre ebből a hiányosságból még nem érdemes messzemenő konklúziókat leszűrni.


DeviceTwin

Overview

DeviceTwin module is responsible for storing device status, dealing with device attributes, handling device twin operations, creating a membership between the edge device and edge node, syncing device status to the cloud and syncing the device twin information between edge and cloud. It also provides query interfaces for applications. Device twin consists of four sub modules (namely membership module, communication module, device module and device twin module) to perform the responsibilities of device twin module.

5.16. ábra: a DeviceTwin hivatalos dokumentációjának első bekezdése [28]

Az előbb leírtaknak megfelelően így a jelaggregátor program a Bluetooth átjáró által publikált adatokat olvassa MQTT-n keresztül, nem az adatbázisból.

5.5.1. Fizikai szenzorok méréseinek lekérdezése

A következő, 5.17-es ábrán látható a `subscribe` függvény forráskódja, mely a fizikai szenzorok mérési adatait a megfelelő MQTT-csatornákról olvassa ki és menti el azokat globális változókba.

```
func subscribe(client1 mqtt.Client, client2 mqtt.Client) {
    var sensorData1 SensorData
    var sensorData2 SensorData
    token1 := client1.Subscribe(firstDeviceTopic, 1, func(client1 mqtt.Client,
msg mqtt.Message) {
        fmt.Println(string(msg.Payload()))
        if string(firstTopicPayload) != string(msg.Payload()) {
            firstTopicPayload = msg.Payload()
            err := json.Unmarshal([]byte(firstTopicPayload), &sensorData1)
            fmt.Println("error", err)
            aggregateHudTemp(sensorData1,                          sensorData2,
generateThirdSignal(sensorData1, sensorData2))
        }
    })
    token1.Wait()
    token2    :=    client2.Subscribe(secondDeviceTopic,    1,    func(client2
mqtt.Client, msg mqtt.Message) {
        fmt.Println(string(msg.Payload()))
        if string(secondTopicPayload) != string(msg.Payload()) {
            secondTopicPayload = msg.Payload()
            err := json.Unmarshal([]byte(secondTopicPayload), &sensorData2)
            fmt.Println("error", err)
            aggregateHudTemp(sensorData1,                          sensorData2,
generateThirdSignal(sensorData1, sensorData2))
        }
    })
    token2.Wait()
}
```

5.17. ábra: A jelaggregátor *subscribe* függvényének forráskódja

Mint látható ez a függvény szinte azonos a mapper alkalmazás `subscribe` függvényével, azzal a különbséggel, hogy ez a függvény két MQTT-csatornát is figyel, az egyik csatorna az egyik fizikai szenzor méréseit segít kézbesíteni, míg a másik csatorna a másik fizikai szenzorét.

5.5.2. A harmadik, hibás mérési érték generálása

Ezen a ponton érdemes bemutatnom a harmadik virtuális szenzor mérési adatait generáló logikát, az ezt implementáló a függvénynek a `generateThirdSignal` nevet adtam. Ebben a függvényben tulajdonképpen átlagolom a két fizikai szenzor méréseinek hő- és páratartalommérésének értékeit és annak a ± 5 fokos, illetve százalékos tartományából véletlenszerűen generálok egy hőmérséklet és páratartalomértéket, mely a hibás jelet tükrözi, így megvalósítva a hibainjektálást. Az alábbi 5.18-as ábrán látható a `generateThirdSignal` függvény forráskódja.

```
func generateThirdSignal(sensorData1 SensorData, sensorData2 SensorData)
SensorData {
    // in this section, we produce the 3rd signal required for TMR
    var sensorData3 SensorData
    rand.Seed(time.Now().UnixNano())
    avg_temp := (sensorData1.Temperature + sensorData2.Temperature) / 2
    avg_hud := (sensorData1.Humidity + sensorData2.Humidity) / 2
    min_temp := avg_temp - 5
    min_hud := avg_hud - 5
    max_temp := min_temp + 10
    max_hud := min_hud + 10
    sensorData3.Temperature = rand.Float32()*(max_temp-min_temp) + min_temp
    sensorData3.Humidity = rand.Float32()*(max_hud-min_hud) + min_hud
    return sensorData3
}
```

5.18. ábra: A *generateThirdSignal* függvény forráskódja

5.5.3. Az aggregált jel előállítása

Ezen a ponton rendelkezésre állnak a szenzorok mérési adatai, végre fel lehet dolgozni azokat, ezt az adatfeldolgozást az `aggregateHudTemp` nevű függvényben implementáltam le. A logikát azon feltételezés mentén készítettem, hogy a három mérésből kettő mindig megbízható ámbár nem feltétlen teljesen pontos – tekintettel arra, hogy olcsóbb szenzorokat használunk.

Az aggregált jelet a három rendelkezésünkre álló jelből többségi szavazás segítségével valósítottam meg, melynek az első lépése az, hogy minden két szenzor méréseinek előállítom az átlagát. Azért tartottam ezt fontosnak, mert a pontosságot – pontosabban a pontatlanságot – így valamennyire ki lehet átlagolni. Az átlagok ismeretében már meg lehet valósítani a többségi szavazást, a fő érdekesség itt az, hogy jelen esetben nem az egyes mérési értékeket szavaztatom meg hanem kettő mérés átlagát. A többségi szavazás tárgyai így tulajdonképpen az egyes átlagértékek lettek. A szavazás úgy történik, hogy minden egyes átlagértéket kivonok minden egyes mérési értékből és az összes így kapható különbséget összehasonlítom egymással. Végül mindegyik átlagérték kapni fog egy szavazatot, amely azt fogja tükrözni, hogy melyik szenzor mérési értékéhez állt a legközelebb. Ha ezek a szavazatok már rendelkezésre állnak, pusztán csak azt kell már megállapítani, hogy mely átlagérték áll legközelebb kettő szenzor mérési adatához is, és az lesz a végleges, aggregált mérési adatunk, ezt az értéket fogja tükrözni a „hudtemp-aggregated” elnevezésű digitális iker, Kubernetesben definiált eszköz.

Az `aggregateHudTemp` és a `getVote` függvény forráskódja a következő oldalon található 5.19-es ábrán olvasható.

```
func getVote(sensorReading float32, average [3]float32) int {
    var vote int = 0
    for i := 0; i < 3; i++ {
        if            math.Abs(float64(sensorReading-average[i]))           <
math.Abs(float64(sensorReading-average[vote])) {
            vote = i
        }
    }
    return vote
}

func   aggregateHudTemp(sensorData1   SensorData,   sensorData2   SensorData,
sensorData3 SensorData) {
    // now comes the implementation of the TMR
    var finalTemp float32
    var finalHud float32
    var tempAverages [3]float32
    var hudAverages [3]float32
    tempAverages[0] = (sensorData1.Temperature + sensorData2.Temperature) / 2
    tempAverages[1] = (sensorData1.Temperature + sensorData3.Temperature) / 2
    tempAverages[2] = (sensorData2.Temperature + sensorData3.Temperature) / 2
    hudAverages[0] = (sensorData1.Humidity + sensorData2.Humidity) / 2
    hudAverages[1] = (sensorData1.Humidity + sensorData3.Humidity) / 2
    hudAverages[2] = (sensorData2.Humidity + sensorData3.Humidity) / 2

    var temp_votes [3]int
    temp_votes[0] = getVote(sensorData1.Temperature, tempAverages)
    temp_votes[1] = getVote(sensorData2.Temperature, tempAverages)
    temp_votes[2] = getVote(sensorData3.Temperature, tempAverages)

    var hud_votes [3]int
    hud_votes[0] = getVote(sensorData1.Humidity, hudAverages)
    hud_votes[1] = getVote(sensorData2.Humidity, hudAverages)
    hud_votes[2] = getVote(sensorData3.Humidity, hudAverages)

    if temp_votes[0] == temp_votes[1] {
        finalTemp = tempAverages[temp_votes[0]]
    } else if temp_votes[0] == temp_votes[2] {
        finalTemp = tempAverages[temp_votes[0]]
    } else if temp_votes[1] == temp_votes[2] {
        finalTemp = tempAverages[temp_votes[1]]
    } else {
        return
    }

    if hud_votes[0] == hud_votes[1] {
        finalHud = hudAverages[hud_votes[0]]
    } else if hud_votes[0] == hud_votes[2] {
        finalHud = hudAverages[hud_votes[0]]
    } else if hud_votes[1] == hud_votes[2] {
        finalHud = tempAverages[hud_votes[1]]
    } else {
        return
    }

    publishToMqtt(finalTemp, finalHud)
}
```

5.19. ábra: Az *aggregateHudTemp* és a *getVote* függvény forráskódja

A jelaggregátor alkalmazás telepítése után az elkezdi előállítani az aggregált és megbízható virtuális eszköz méréseit, ennek a Kubernetesbeli reprezentációja az alábbi, 5.20-as ábrán látható. Mint említettem, a harmadik, hibás jel erőforráskímélési okokból csak a Go programkódon belül létezik, így azt nem tudjuk lekérdezni sajnos.

```
zozo@kubemaster:~$ kubectl describe device hudtemp2
Name:         hudtemp2
…
  Twins:
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  temperature
    Reported:
      Metadata:
        Timestamp:  1639044615712
        Type:       string
      Value:        23.80
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  humidity
    Reported:
      Metadata:
        Timestamp:  1639044615711
        Type:       string
      Value:        26.70
Events:             <none>
zozo@kubemaster:~$ kubectl describe device hudtemp-aggregated
Name:         hudtemp-aggregated
Namespace:    default
Labels:         fault_tolerance=Tolerates-the-failure-of-one-of-the-physical-
hudtemp-sensors
              manufacturer=tmr
              type_of_fault_tolerance=Triple-modular-redundancy
Annotations:  <none>
API Version:  devices.kubeedge.io/v1alpha2
Kind:         Device
Metadata:
  Creation Timestamp:  2021-12-08T23:06:59Z
  Generation:          19606
  Managed Fields:
    API Version:  devices.kubeedge.io/v1alpha2
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:labels:
          .:
          f:fault_tolerance:
          f:manufacturer:
          f:type_of_fault_tolerance:
      f:spec:
        .:
        f:deviceModelRef:
          .:
          f:name:
        f:nodeSelector:
          .:
          f:nodeSelectorTerms:
      f:status:
    Manager:      kubectl-create
    Operation:    Update
```

```
    Time:         2021-12-08T23:06:59Z
    API Version:  devices.kubeedge.io/v1alpha2
    Fields Type:  FieldsV1
    fieldsV1:
      f:status:
        f:twins:
    Manager:         cloudcore
    Operation:       Update
    Time:            2021-12-08T23:58:22Z
  Resource Version:  2150664
  UID:               995c2a10-8358-479a-a874-b98be101a923
Spec:
  Device Model Ref:
    Name:  hudtemp-model
  Node Selector:
    Node Selector Terms:
      Match Expressions:
        Key:
        Operator:  In
        Values:
          pi-ubuntu
Status:
  Twins:
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  temperature
    Reported:
      Metadata:
        Timestamp:  1639044562805
        Type:       string
      Value:        23.85
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  humidity
    Reported:
      Metadata:
        Timestamp:  1639044562805
        Type:       string
      Value:        26.40
Events:             <none>
zozo@kubemaster:~$ kubectl describe device hudtemp1
Name:         hudtemp1
…
Status:
  Twins:
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  temperature
    Reported:
      Metadata:
        Timestamp:  1639045146490
        Type:       string
      Value:        23.90
```

```
    Desired:
      Metadata:
        Type:       string
      Value:
    Property Name:  humidity
    Reported:
      Metadata:
        Timestamp:  1639045146490
        Type:       string
      Value:        25.70
Events:             <none>
```

5.20. ábra: Az aggregált és a két fizikai jel mérési értékei a Kubernetesben

5.6. A projekt fájlszintű felépítése

A feladat megoldása elérhető a Githubon publikus repositoryként [25], ebben az alfejezetben egy áttekintést nyújtok annak a tartalmára.

A projektben használt fájlokat alapvetően négy típusra lehet felosztani, ezek a típusok pedig a következők:

- a KubeEdge leírófájlok, angolul CRD-k
- a Go nyelven megírt programok, melyen konténerizált környezetben futnak
- a Docker konténer-lemezkép leírók melyeket a felhasználók az egyes lemezképek létrehozására használhatnak
- dokumentumok, ezek a funkcionális modell (a ‘`drawio`’ kiterjesztésű fájl), annak exportált képe, illetve a README, telepítéshez tartozó instrukciókat tartalmazó fájl
- a xiaomi-ble-mqtt mappa, mely a Git repository egy submodule-ja, avagy egy abba beágyazott Git repository, ezt a projektet hasznosítottam a hőmérőkkel való kapcsolattartásra
- minden egyéb fájl, ezek a go programkód lefordításához lehetnek szükségesek vagy a Githez tartozó adatállományok

Az alábbi, 5.21-es ábra listázza a projektben található fájlok összességét. Az itt megvalósított KubeEdge alkalmazás feltelepítéséhez a pontos lépések a README fájlban találhatók, angol nyelven leírva.

```
zozo@Zozos-MacBook-Pro:~/kube/kubeedge_thesis_demo (master)$ tree .
.
├── Dockerfile
├── Dockerfile.initial
├── README.md
├── crds
│   ├── hudtemp-aggregated.yaml
│   ├── hudtemp-instance1.yaml
│   ├── hudtemp-instance2.yaml
│   ├── hudtemp-model.yaml
│   ├── kubeedge-bl-gw.yaml
│   ├── kubeedge-hudtemp-aggregated-signal.yaml
│   ├── kubeedge-hudtemp-mapper1.yaml
│   └── kubeedge-hudtemp-mapper2.yaml
├── function-level_model.drawio
├── go.mod
├── go.sum
├── images
│   └── function-level_model.jpg
├── mapper
│   ├── Dockerfile
│   └── main.go
├── signal_aggregator
│   ├── Dockerfile
│   └── main.go
└── xiaomi-ble-mqtt
    ├── README.md
    ├── contrib
    │   ├── Dockerfile
    │   ├── Dockerfile.arm32v7
    │   ├── README.md
    │   └── multi-arch-manifest.yaml
    ├── data-read.py
    ├── devices.ini.sample
    ├── mitemp
    │   └── mitemp_bt
    │       ├── __init__.py
    │       ├── __pycache__
    │       │   ├── __init__.cpython-35.pyc
    │       │   └── mitemp_bt_poller.cpython-35.pyc
    │       └── mitemp_bt_poller.py
    ├── mqtt.ini.sample
    └── run.sh

9 directories, 32 files
```

5.21. ábra: A projektben található fájlok összessége fa nézetben

5.7. Tapasztalatok, tanulságok

A feladat elkészítése alatt felfedeztem pár, a KubeEdge szoftver éretlenségéből adódó problémákat, így fel szeretném hívni a potenciális jövőbeli felhasználók figyelmét, hogy különös figyelemmel kövessék a telepítés folyamatát, ugyanis nem feltétlen van a hivatalos dokumentáció minden pontja folyamatosan frissítve, illetve jelenleg még szükségesek olyan manuális konfigurációs lépések, amelyekre a felhasználó nem feltétlen számít, így átsiklik felette. Különösen érdemes a felhasználónak ellenőrizni minden lépés után, hogy az adott lépés érvényre jutott, a megfelelő hatást okozta a rendszeren. Erre két példát hoznék: az első a kube-proxy Kubernetes komponens eltávolítása a KubeEdge Edge node-ról, tapasztalatom szerint, az általam használt verzióban (v.1.8.2) legalábbis nem működött a hivatalos leírás szerinti parancs, míg a másik fontos elem hogy a CloudCore szolgáltatás nem kerül a telepítés közben a systemd-be beregisztrálásra – legalábbis nem az általam használt Ubuntu Server 20.04.3-as verzión, így erre is érdemes figyelni.

Az előbb említett kellemetlenségektől eltekintve viszont a tapasztalatok szerint lehetséges így egy megbízható, magas rendelkezésre állású végpontot létrehozni így, bár érdemes figyelni arra, hogy a peremi számítógép, ez esetben a Raspberry Pi memóriája ne teljen meg. A KubeEdge szoftver magas harmóniában van a Kubernetesszel, tapasztalatom szerint a kettő szoftver közötti együttműködés kifejezetten harmonikus, így a potenciál ebben és ehhez hasonló alkalmazásokban hatalmas a jövőre nézve.

6. Összefoglalás

Mint ismertettem, a feladat megoldása során megismerkedtem a Kubernetes érdekes világával, amelynek újfent ismeretét biztosan kamatoztatni fogom a jövőben, hiszen még a jelenleg is nagy, monolitikus szolgáltatásokat alkalmazó vállalatok is dolgoznak már a kisebb, könnyebben mozgatható, skálázható mikroszolgáltatásokra való jövőbeli átálláson.

A Kubernetes szoftver személyében egy jól összeszedett, érett, bár hibáktól nem mentes szoftvertermékkel találkoztam. Ugyan még az egyedi rendszereken nehézkes lehet egy frissen telepített Kubernetes példány bekonfigurálása, a hatalmas közreműködői társadalomnak köszönhetően bárki viszonylag gyorsan találhat segítséget.

A KubeEdge megismerése a Kuberneteshez képes egy jelentős minőségbeli változást tükrözött. Mint ismeretes, a KubeEdge project jelenleg a CNCF-nek még mindig az inkubátor fázisában tartózkodik, sajnos ez tükröződik is a használat közben. A potenciál viszont meglátásom szerint megtalálható benne, ugyanis az, hogy egy Raspberry Pi 3-ason lehetővé teszi a konténerszervezés mellett a lokális feldolgozáson túl a lokális adattárolást autonóm működéssel egyetemben, hatalmas potenciált rejt magában a jövőre tekintve. Sajnos az olyan hibák, mint a DeviceTwin nem lekérdezhetősége vagy a tény, hogy a KubeEdge CloudCore szolgáltatása nem kerül a systemd-be beregisztrálásra automatikusan, egyelőre lehetséges, hogy néhány új felhasználót, potenciális közreműködőt eltántorítanak, de a szoftver folyamatos fejlesztés alatt áll, így valószínűsítem, hogy a közeljövőben el fog érni egy kellően magas érettségi szintet a szoftver.

Az általam megvalósított projekt pedig véleményem szerint kitűnően példázza, hogy manapság már számolni lehet – és kell is – az olcsóbb, alacsony kapacitású eszközökkel, ugyanis azokkal is létre lehet hozni megbízható – még ha egyelőre nem is feltétlen üzemkritikus – rendszereket, amelyek az olcsóságuk ellenére is – megfelelő specifikáció, konfiguráció ellenében – képesek autonóm működésre, folyamatos, megbízható adatszolgáltatásra.

Irodalomjegyzék

[1] M. Vijay and R. Mittal, “Algorithm-based fault tolerance: a review,” *Microprocessors and Microsystems,* vol. 21, no. 3, pp. 151-161, 1997.

[2] J. G. Silva, P. Prata, M. Rela and H. Madeira, “Practical issues in the use of ABFT and a new failure model,” *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing,* pp. 26-35, 1998.

[3] G. Bosilca, R. Delmas, J. Dongarra and J. Langou, “Algorithm-based fault tolerance applied to high performance computing,” *Journal of Parallel and Distributed Computing,* vol. 69, no. 4, pp. 410-416, 2009.

[4] Z. Chen and J. Dongarra, “Algorithm-Based Fault Tolerance for Fail-Stop Failures,” *EEE Transactions on Parallel and Distributed Systems,* vol. 19, no. 12, pp. 1628-1641, 2008.

[5] S. Roffe and A. D. George, “Evaluation of Algorithm-Based Fault Tolerance for Machine Learning and Computer Vision under Neutron Radiation,” *2020 IEEE Aerospace Conference,* pp. 1-9, 2020.

[6] E. Wilfried, “An Introduction to Sensor Fusion,” Vienna University of Technology, Austria, 2002.

[7] B. P. L. Lau, S. H. Marakkalage, Y. Zhou, N. U. Hassan, C. Yuen, M. Zhang and U.-X. Tan, “A survey of data fusion in smart city applications,” *Information Fusion,* vol. 52, pp. 357-374, 2019.

[8] B. Burns, “The History of Kubernetes & the Community Behind It,” 2018. [Online]. Available: https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/. [Accessed 9 12 2021].

[9] C. McLuckie, “From Google to the world: The Kubernetes origin story,” Google, 22 7 2016. [Online]. Available: https://cloud.google.com/blog/products/containers-kubernetes/from-google-to-the-world-the-kubernetes-origin-story. [Accessed 9 12 2021].

[10] RisingStack Engineering, “The History of Kubernetes on a Timeline,” 11 10 2021. [Online]. Available: https://blog.risingstack.com/the-history-of-kubernetes/. [Accessed 9 12 2021].

[11] “What is Kubernetes?,” [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 9 12 2021].

[12] M. Moravcik, P. Segec, M. Kontsek, J. Uramova and J. Papan, “Comparison of LXC and Docker Technologies,” *2020 18th International Conference on Emerging eLearning Technologies and Applications,* pp. 481-486, 2020.

[13] “Kubernetes Components,” [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed 9 12 2021].

[14] “Introducing KubeEdge,” [Online]. Available: https://kubeedge.io/en/slides/introducing-kubeedge/#/2. [Accessed 9 12 2021].

[15] CNCF Staff Blog Post, “TOC Approves KubeEdge as Incubating Project,” 16 9 2020. [Online]. Available: https://www.cncf.io/blog/2020/09/16/toc-approves-kubeedge-as-incubating-project/. [Accessed 9 12 2021].

[16] “IoT and Edge computing with Kubernetes,” [Online]. Available: https://docs.google.com/document/d/1We-pRDV9LDFo-vd9DURCPC5-Bum2FvjHUGZ1tacGmk8/edit#. [Accessed 9 12 2021].

[17] “Cloud Components of KubeEdge,” [Online]. Available: https://kubeedge.io/en/docs/architecture/cloud/. [Accessed 9 12 2021].

[18] “Edge Components of KubeEdge,” [Online]. Available: https://kubeedge.io/en/docs/architecture/edge/. [Accessed 9 12 2021].

[19] “Device Manager,” [Online]. Available: https://kubeedge.io/en/docs/developer/device_crd/. [Accessed 9 12 2021].

[20] KubeEdge, “Control home appliances from cloud,” 17 11 2020. [Online]. Available: https://kubeedge.io/en/blog/control-home-appliances-via-cloud/. [Accessed 9 12 2021].

[21] “Examples for KubeEdge,” [Online]. Available: https://github.com/kubeedge/examples. [Accessed 9 12 2021].

[22] “Xiaomi BLE Temperature and Humidity Sensor Bluetooth To MQTT gateway,” [Online]. Available: https://github.com/algirdasc/xiaomi-ble-mqtt. [Accessed 9 12 2021].

[23] “EventBus,” [Online]. Available: https://kubeedge.io/en/docs/architecture/edge/eventbus/. [Accessed 9 12 2021].

[24] “Home Assistant,” [Online]. Available: https://www.home-assistant.io. [Accessed 9 12 2021].

[25] Z. Herjavecz, “KubeEdge Thesis Demo,” [Online]. Available: https://github.com/hzozo/kubeedge_thesis_demo. [Accessed 9 12 2021].

[26] KubeSphere, “Monitoring Kubernetes Control Plane using KubeSphere,” 29 3 2020. [Online]. Available: https://itnext.io/monitoring-kubernetes-control-plane-using-kubesphere-7885461f40b1. [Accessed 9 12 2021].

[27] “Why KubeEdge,” [Online]. Available: https://kubeedge.io/en/docs/kubeedge/. [Accessed 9 12 2021].

[28] [Online]. Available: https://kubeedge.io/en/docs/architecture/edge/devicetwin/. [Accessed 9 12 2021].