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

Kiber-fizikai rendszerek szemantikusan támogatott futásidejű újrakonfigurációja

TDK dolgozat

Készítette:

Szalontai Jenő

Konzulens:

Dr. Kocsis Imre
Dr. Suskovics Péter

2019

Tartalomjegyzék

Kivonat i

Abstract ii

1. Edge alkalmazások és üzemeltetésük kihívásai 1
1.1. Az edge computing mint architekturális minta . . . 2
1.2. Tipikus edge alkalmazáskategóriák . . . 3
1.3. Edge szolgáltatásbiztonság és ellenállóképesség . . . 5
1.3.1. Szolgáltatásbiztonság . . . 5
1.3.2. Ellenállóképesség (resilience) . . . 6

2. Szolgáltatások megvalósítása konténerizált környezetben 7
2.1. Mikroszolgáltatás alapú alkalmazás tervezése . . . 7
2.2. Stateful-Stateless alkalmazások . . . 8
2.3. Kubernetes szükségessége . . . 8
2.4. Kubernetes architektúrája . . . 9
2.5. Kubernetes platformszintű futtatás menedzsment . . . 10
2.5.1. Deployment vezérlése . . . 10
2.5.2. Restart stratégiák . . . 10
2.6. Erőforrás menedzsment . . . 11
2.6.1. Scheduling . . . 11
2.7. Cloud computing integrálása . . . 12

3. Rendszerleírás megvalósítása szemantikus reprezentációval 13
3.1. Sensor, Observation, Sample, and Actuator (SOSA) . . . 13
3.2. Semantic Sensor Network . . . 14

4. Kiber-fizikai rendszerek szemantikusan támogatott futásidejű újrakonfigurációja 15
4.1. Célprobléma: szenzoradat-feldolgozó kiber-fizikai rendszerek . . . 15
4.1.1. Kiber-fizikai rendszerek . . . 15
4.1.2. Alkalmazáscsalád: szenzoradatok adatfolyamai . . . 16
4.1.3. Platform: edge Kubernetes . . . 17
4.1.4. Kommunikációs köztesréteg technológia: DDS . . . 17
4.1.5. Újrakonfigurációs megkötések: kommunikációs szkeleton . . . 18
4.2. Megvalósítási architektúra . . . 19
4.2.1. Modellezési vetületek . . . 19
4.3. Mintaalkalmazás és minta szemantikus újrakonfiguráció . . . 20
4.3.1. Mintaalkalmazás fizikai architektúrája . . . 21
4.3.2. Újrakonfigurációs példa . . . 21
4.4. A megvalósított szemantikus adatbázis . . . 23

4.4.1. SSN ontológia használata . . . 23
4.4.2. Kubernetes ontológia . . . 24
4.4.3. DDS ontológia . . . 24
4.5. Helyettesítő megvalósítások felderítése fogalomelemzéssel . . . 25
4.5.1. Az ismert részfeladat: deployment . . . 25
4.5.2. Helyettesítő megoldások keresése . . . 25
4.5.3. Formal Concept Analysis (FCA)[1] . . . 26
4.5.4. FCA alkalmazása . . . 26
4.6. Futásidejű újrakonfiguráció megvalósítása . . . 30
4.6.1. Monitor . . . 30
4.6.2. Analyze . . . 30
4.6.3. Plan . . . 31
4.6.4. Execute . . . 31

5. Összefoglaló **32**
5.1. Az általam elért, a dolgozatban ismertetett eredmények . . . 32
5.2. További kutatási lehetőségek . . . 32

Irodalomjegyzék **33**

Függelék **35**
F.1. Az elkészült ontológia . . . 35

[1] A szakasz szorosan követi [1] FCA leírását (20.-21. oldal).

Kivonat

Az *edge computing* a felhő számítástechnika következő fejlődési lépését adja abban az értelemben, hogy a ”terepi”, illetve a terep és az internet *határán* (*edge*) telepített számítástechnikai erőforrások, szenzorok és beavatkozók hozzáférés-kezelését és feladatkiosztását a felhő számítástechnika eszközeivel végzi. Dolgozatomban az edge megoldások szemantikusan támogatott modellezésének, monitorozásának és futásidőben tervezett újrakonfigurációjának lehetőségeit vizsgálom, illetve javaslok ennek egy részproblémájára újszerű megoldást.

Az edge computing lehetővé teszi a kiber-fizikai (Cyber-Physical Systems - CPS) alkalmazások dinamikus újrakonfigurálását azáltal, hogy az alkalmazás funkcionális elemeit a környezeti feltételeknek, a bekövetkező hibáknak és a változó igényeknek megfelelően könnyedén újrarendezhetjük az elérhető terepi eszközök, "terep-széli" számítási egységek és a felhő között.

Ennek a dinamikus rekonfigurációnak azonban nagy kihívása, hogy a lehetőségek teljes eléréséhez szemantikus jellegű tervezésre van szükség, például helyettesítő szenzorelrendezések keresésénél. Másrészről a tipikus *edge* alkalmazások kiesésmentes szolgáltatást követelnek meg, ami az újrakonfiguráció tervezést és végrehajtást is lényegesen nehezebb problémává teszi.

Dolgozatomban platformtechnológiaként Kubernetes alapú Docker konténerizációt alkalmazok, az alkalmazáskomponensek közötti megbízható kommunikációt megvalósító köztesrétegként pedig az OMG által szabványosított DDS-t (Data Distribution Service).

Megoldást adok az ilyen környezetek újrakonfiguráció szempontjából meghatározó jellemzőinek közös, szemantikus adatbázisban futásidejű követésére. (Ide értendő az elérhető szenzorok szemantikus – W3C Semantic Sensor Network alapú – reprezentációja, a Kubernetes platform konfigurációja, aktuális feladatallokációja és az alkalmazási szinten a alkalmazáskomponensek közötti DDS alapú kommunikációs kapcsolatok.)

Megmutatom, hogy a szemantikus reprezentáció hogy támogatja egy adott alkalmazásminta esetén a helyettesítő szolgáltatások keresését és az új feladatkiosztás (*deployment*) tervezését, például a formális fogalomelemzés (*formal concept analysis* – FCA) alkalmazásával.

Abstract

Edge Computing is the newest generation of Cloud Computing, located near the field, – or near the field and between the internet – provides access-, and deployment-management for computing resources, sensors, actuators. This work focuses on edge-based solutions supported by semantic modelling, monitoring and real-time reconfiguration capabilities, and provides a novel solution to the problem.

Edge computing enables CPS-based (Cyber-Physical System) application dynamic reconfiguration, through reallocating edge resources by the application functional component regarding to environmental, faults and changing needs.

The challenge for dynamic reconfiguration is to enable the full capabilities of the system. This requires a semantic structure, for example, searching a new sensor replacing a faulty one. On the other side, most of the applications require high availability, that makes dynamic reconfiguration more challenging.

Kubernetes and Docker enable this type of flexibility. A suitable communication middleware type is specified by OMG DDS (Data Distribution Service). In my work, I provide a semantic reconfigurable solution, for these type of systems using semantic databases for real-time reconfiguration. (Taking into account the available sensors through they semantic description – provided by W3C Sensor Semantic Network – Kubernetes configuration, current processing allocation and application-level DDS communication links.)

This work shows the capabilities of semantic representation for a pre-specified application pattern, searching for replacement processing units and performing automatic deploying planning using FCA (*formal concept analysis*).

1. fejezet

Edge alkalmazások és üzemeltetésük kihívásai

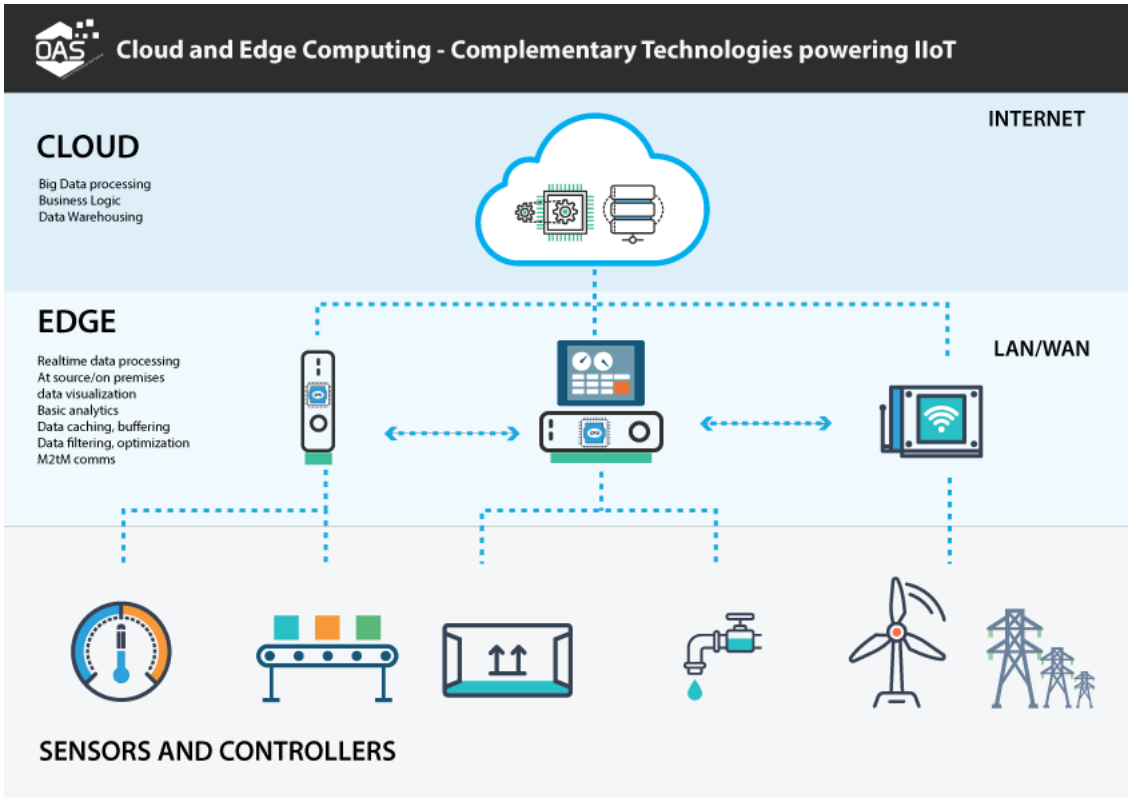
A *cloud computing* lehetővé teszi, hogy a számítástechnikai alkalmazások szinte tetszőleges mennyiségű tárolási és feldolgozási erőforrást skálázható módon, jellemzően használat alapú számlázassal vegyenek igénybe dedikált, de az ügyfelek között megosztottan üzemeltetett szolgáltatói adatközpontokban[2]. A cloud computing számos szolgáltatási modellt alkalmaz, ezek közül a dolgozatom szempontjából releváns, az úgynevezett IaaS (*Infrastructure as a Service*) mely modellben az igény alapon "bérelt" erőforrások a fizikai számítástechnikai erőforrások valamilyen formában virtualizált változatai (például virtuális gép, virtuális hálózati interface vagy virtuális blokkos tárolóeszköz). Az IaaS szolgáltatásokban a virtualizáció klasszikusan hipervizor alapú virtualizációt jelent, de napjainkban rohamosan növekszik a konténerizáció alapú megoldások szerepe.

Az *Internet of Things* (IoT) elsőleg 1999-ben lett bemutatva mint az ellátásilánc-menedzsment RFID-val támogatott úttörője [3]. Manapság a hangsúly az automatikus információgyűjtésen, feldolgozáson és beavatkozáson van. Várhatóan 2021-re a teljes *IoT* eszközök által gyűjtött adatmennyiség 847 Zettabyte lesz évente (2016-ban 218 ZB), viszont az adatközpontok IP forgalma csak 20.6 Zettabyte, a Cisco Global Cloud Index szerint [4]. A nagy mennyiségű adathoz a számítási kapacitást, így "le" kell költöztetni a terepre, ahol is lokálisan tud zajlani a feldolgozás tekintettel az alacsony válaszidőre, a privát adatokra és a hálózat teljesítményére.

Az *edge computing* (terepi számítás) előtérbe került az IoT elterjedésével, ami magával hozza a megnövekedett igényt a lokális feldolgozásra, olyan helyen, ahol az adattovábbítás szűk keresztmetszet. Például egy önvezető autó, ami másodpercenként 1 GB adatot állít elő, valósidejű adatfeldolgozást kíván a helyes, pontos beavatkozásokhoz [5]. A nem valósidejű, későbbi statisztikák számára készülő adatokat a cloudban dolgozzák fel.

Dolgozatomban megvizsgálom, hogyan lehet az edge alkalmazásokat támogatni szolgáltatásbiztonság szempontjából egy szemantikus leíró segítésével. Ehhez a szükséges háttérismeretek bevezetésével kezdem (1.-2. fejezet), ami magában foglalja az edge architektúra ismertetését, az edge computing kihívásait és a jelenleg elterjedt platformtechnológiát, a Kubernetest. A rendszerleírás egy szemantikus reprezentációját ismertetem (3. fejezet), majd egy megoldást adok a problémára (4. fejezet), amiben az újrakonfiguráció jelenik meg az ellenállóképesség növelésére. Egy összefoglalással, további kutatási irányokkal (5. fejezet) zárom a dolgozatom.

1.1. Az edge computing mint architekturális minta



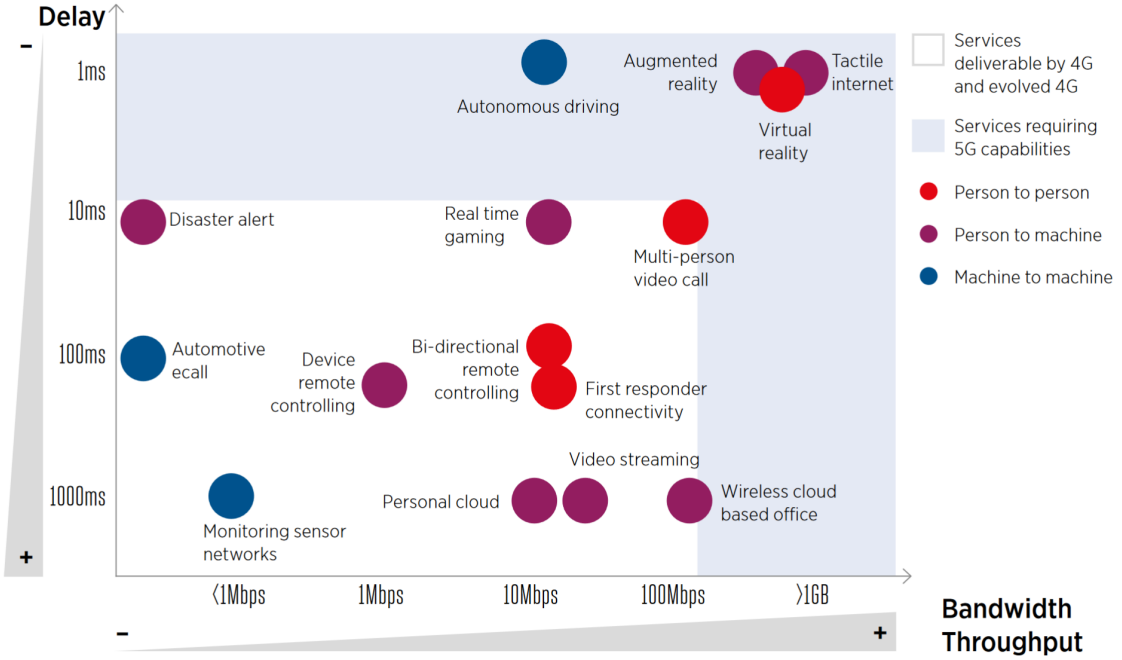
1.1. ábra. Az edge computing tipikus elemei és kapcsolatai[1].

Az *edge computing* lényege, hogy a fizikai közelséget igénylő alkalmazások feldolgozási lépéseit leválasztjuk a cloudból és közelebb helyezzük az adatforráshoz. ”Edge” alatt olyan számítási kapacitást értünk, ami az adatforrás és a cloud között helyezkedik el. A paradigma szerint az edge eszközök lehetnek adatforrások és adatnyelők is. A lokális feldogozás során akár lekérhetőek a cloudból az adatok, amivel számolhatunk is, így képes ”cloud tehermentesítésre”, ami során a feldolgozás egy részét cloudból edge-re mozgatjuk át[6].

[1] `https://www.openautomationsoftware.com/wp-content/uploads/2017/04/edge-v-cloud-computing-graphic.png` Accessed 2019.10.09

1.2. Tipikus edge alkalmazáskategóriák

Durva egyszerűsítéssel, de kijelenthető, hogy az edge megoldások elsődleges előnye az, hogy a különböző terepi alkalmazások számára jóval alacsonyabb és kiszámíthatóbb késleltetéssel elérhető cloud jellegű erőforrásokat szolgáltatnak mint a ”nagy” cloud (bár a paradigma még kialakulóban van, a telekommunikációs ipar például azon dolgozik, hogy közvetlenül 5G bázisállomások mellé telepített kis méretű szerverparkokkal valósítson meg edge computingot). Ennek az ára az, hogy ezen erőforrások sem költséghatékonyságban, sem extra-funkcionális tulajdonságaikban (megbízhatóság, rendelkezésre állás, adatbiztonság, stb...) nem vehetik fel a versenyt a klasszikus felhő számítástechnikával. Így érdemes áttekintenünk, hogy mik azok az alkalmazások ahol az edge által nyújtott előnyök valóban meghatározóak lehetnek. Az 1.2 ábra néhány ilyen szolgáltatást szemléltet, melyek egyben meghatározó 5G használati esetek is (várhatóan az 5G telekommunikációs szolgáltatásokban az edge computing meghatározó szerepet fog kapni). Az ábra egyben azt is szemlélteti, hogy mik azok az alkalmazások, melyek nem feltétlenül igényelnek edge computingot, de az ”backhaul” hálózati költségek csökkentésével hatékonyan profitálhatnak abból.



1.2. ábra. Alacsony késleltetés és magas sávszélesség igényű, terepi jellegű szolgáltatások[2].

Érdemes megvizsgálni az alacsony késleltetést és magas sávszélességet igénylő alkalmazásokat.

- **Autonomous driving:** Az önvezetés során keletkező adatok a V2V (vehicle-to-vehicle) kommunikációban alacsony késleltetést igényelnek, mivel az információ jellegéből fakadóan, csak egy adott pillanatban érvényesek (sebesség, pozíció).
- **Augmented reality:** A kiterjesztett valóságban ember-gép közti kommunikáció zajlik. Ebben az esetben a nagy sávszélesség igény a számítógép által készített objektumok átviteléből és az emberi input során létrejövő adatok, illetve kamerakép

[2] https://www.gsma.com/futurenetworks/wp-content/uploads/2016/02/704_GSMA_unlocking_comm_opp_report_v5.pdf Accessed 2019.10.09

átviteléből tevődik össze. Az alacsony késleltetés igénye, az embertől származó mozgás jellegű információk továbbítása miatt fontos, hiszen a pontos objektum pozíció számításához valósidejű adatokra van szükség.

- **Virtual reality:** Ember-ember közti kommunikáció során nagy mennyiségű adat átvitele a kameraadatok átviteléből adódik; az alacsony késleltetés igénye pedig a folyamatos emberi kommunikációt teszi lehetővé.
- **Tactile internet:** Az alacsony késleltetés és a magas sávszélesség biztosítja a valósidejű beavatkozást, például az ipari felhasználás során, illetve lehetővé teszi az "érintés" valósidejű átvitelét is.

Az ábrát kiegészítve még felsorolhatunk [7] alapján, két alkalmazáskategóriát.

- **10ms alatti késleltetésigényű szolgáltatások:**
  - **Shared Haptic Virtual Environments:** több felhasználó által vezérelt robotok finom mozgatással.
  - **Tele-medical** alkalmazások (tele-diagnosis, tele-rehabilitation).
  - **Process automatizálás** (5ms).
  - **Smart grid** (3ms).
- **1ms alatt késleltetésigényű szolgáltatások:**
  - **Távoli vezérlés**.
  - **Valós idejű telepresence**, szinkron haptic feedback-kel.
  - **Ipari mozgató robotok**.
  - **Ipari zárt körű kontroll rendszerek** (pl, 1ms-ként lekérni a szenzorokat, beavatkozókat).
  - **Ipari zárt körű kontroll rendszerek**: Automatikusan egyeztetett kooperatív vezetési manőverek (például, 1ms időnként adat kérése a szenzortól).
  - **Smart grid**: a villamos rendszerirányításban az energiahálózat termelőinek és fogyasztóinak finom felbontású együttes vezérlése ($<$ 1ms).

A fent említett alkalmazás példákat megvizsgálva rájöhetünk, hogy széles körben van szükség edge alkalmazásokra, az alacsony késleltetés pedig egyrészt megköveteli az új hálózati infrastruktúrát, másrészt az edge computingot.

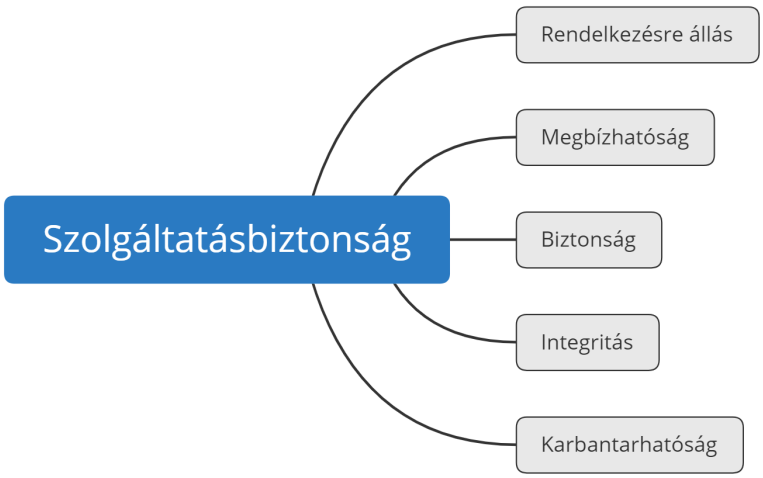
1.3. Edge szolgáltatásbiztonság és ellenállóképesség

Az edge rendszerek extra-funkcionális követelmények szempontjából való megfelelő tervezésének és megvalósításának nehézségét két faktor adja. Az egyik az, hogy mint említettük, az internet és a terep határán elhelyezett szolgáltatási kapacitások nem lehetnek annyira megbízhatóak, mint a felhő adatközpontokban elhelyezettek. A másik oldalon viszont igaz az is, hogy az edge által megcélzott alkalmazási esetek (lásd fent), jellemzően magasabb igényekkel fognak rendelkezni, mint az általános célú számítástechnika. A másik oldalon azonban az edge számítási kapacitások elosztott jellege, lehetővé tesz olyan mechanizmusokat is (terepi rekonfigurációk), melyek a normál felhő számítástechnika kontextusában nem elképzelhetőek. Jelen alpontban röviden áttekintem a szolgáltatásbiztonság és az ellenállóképesség fő fogalmait, mivel a dolgozatomban javasolt megközelítések alapvetően ezen tulajdonságok biztosítását célozzák edge rendszerekben.

1.3.1. Szolgáltatásbiztonság

Definíció szerint [8], a szolgáltatásbiztonság egy képesség, ami biztosítja, hogy a szolgáltatásunk megbízható, amihez egy indokolás is tartozik. A hangsúly az ”indokolt” módon van, ugyan is indoklás nélkül nem létezik szolgáltatásbiztonság. Másképpen a definíció: A rendszer szolgáltatásbiztonsága egy képesség, hogy elkerüljük a szolgáltatás kiesését az elfogadható mértékig. Maga a fogalom több részegységből tevődik össze, mint:

- **Rendelkezésre állás** (Availability): A szolgáltatás megfelelően működik egy adott időpillanatban.
- **Megbízhatóság** (Reliability): A szolgáltatás folyamatosan megfelelően működik.
- **Biztonság** (Safety): A felhasználóra és a környezetére gyakorolt katasztrofális következmények hiánya.
- **Integritás** (Integrity): Illetéktelen adathozzáférés, adatmódosítás megelőzésére.
- **Karbantarthatóság** (Maintainability): A szolgáltatást helyre lehet állítani és módosítani rajta.



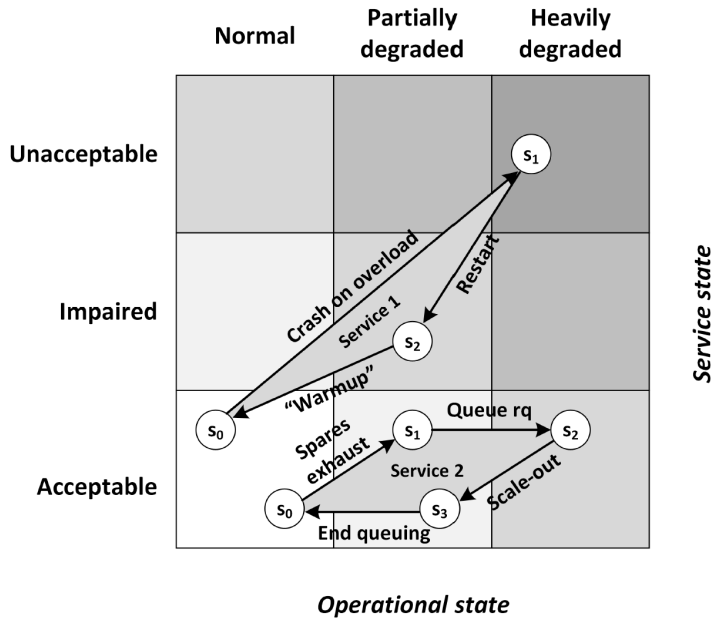
1.3. ábra. Szolgáltatásbiztonság áttekintés[3].

A szolgáltatásbiztos működés tervezésének és fenntartásának irodalma igen széles és már teljes úgynevezett mintanyelvek (pattern languages) is elérhetőek a tipikus megoldásokra [9]. A klasszikus megoldások tipikusan dedikált redundanciákat alkalmaznak, de ma már egyre inkább jellemző az újrakonfiguráció alapú szolgáltatásbiztonság menedzsment (például egy egy szolgáltatás tartalék erőforrásainak adott szinten tartása közösített tartalékból).

1.3.2. Ellenállóképesség (resilience)

Az ellenállóképesség egy olyan szolgáltatás tulajdonság, amely megléte esetén a szolgáltatás általános értelemben megbízható marad még akkor is, ha a szolgáltatást változásoknak tesszük ki. Ilyen változások például az előre nem felételezett típusú hibák, illetve a munkaterhelés és a hibagyakoriságok eltérése az előzetes feltételezésektől [10].

Az ellenállóképesség állapotterekkel való reprezentációja esetén az ellenállóképes viselkedést egy rendszerszintű diszkrét állapottérként tekintjük [11].



1.4. ábra. Kvalitatív modell az ellenállóképesség viselkedéséről állapot terekkel[4].

Az 1.4 ábrán látható, ahogy két különböző service viselkedését modellezzük. *Service 1*-nél látható, hogy összeomlik a szolgáltatás, amikor a terhelés túl magas a képességei számára. A szolgáltatás újraindulásához ilyenkor szükséges egy újraindulás illetve egy "Warmup" állapot is. Ezzel szemben *Service 2* már rendelkezik egy védelemmel, ami a magas "spike" jellegű terheléstől védi. Egy sorral rendelkezik, amivel van elég idő, hogy ha túl magas a terhelés, skálázza a szolgáltatást annyira, hogy el tudja érni a kellő áteresztőképességet. Ennek eredményeképp, ameddig rendszer a "részben degradált" állapotban van, a szolgáltatás állapot ugyan úgy elfogadható. Tehát *Service 2* "ellenállóbb" [1].

Ha ellenálló szolgáltatást szeretnénk tervezni, akkor szükséges valamilyen védelmi mechanizmus beépítése. Az újrakonfiguráció megoldás az ellenállóképesség növelésére.

[3] [8] Fig 2.1
[4] [1] Fig 1.5

2. fejezet

Szolgáltatások megvalósítása konténerizált környezetben

A Kubernetes[1] egy nyílt forráskódú orkesztrációs rendszer amit a konténerizált alkalmazások ”terítése” (deployment) és menedzselésére fejlesztett ki eredetileg a Google, több évtizedes tapasztalattal, amit a skálázható, megbízható rendszerekben szereztek.

A Kubernetes kinőtte a Google specifikusságát és egy általános platformmá vált, amit több nyílt forráskódot támogató szervezet is használ (pl. Cloud Native Computing Foundation[2]) technológiaként, így széles körben elterjedt a *cloud-native* fejlesztők körében kezdve a kis teljesítményű Raspberry Pi-ken át a nagy számítási kapacitású adatközpontokig. Ez nem véletlen, hiszen tartalmazza a szükséges komponenseket a megbízható és skálázható elosztott rendszerek létrehozásához [12].

Az edge computingban ma már szinte magától értetődő, hogy az erőforrások felhasználók és alkalmazások közötti dinamikus megosztására konténerizáció és nem virtualizáció alkalmazandó. Köszönhetően annak, hogy a konténerizáció overheadje a virtualizációhoz képest sokkal kisebb és technológiailag korlátozott futtató platformon is elérhető. Így jelenleg egyértelműnek tűnik, hogy a közeljövő edge megoldásain platformszinten konténerizációt, az orkesztráció szintjén pedig annak legérettebb konténer alapú megoldását a Kubernetest fogják alkalmazni.

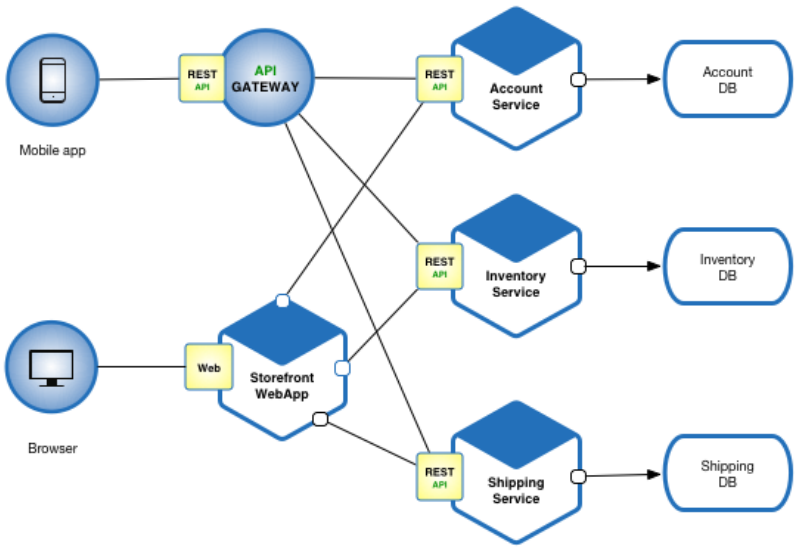
2.1. Mikroszolgáltatás alapú alkalmazás tervezése

Az 1990-es években még a monolitikus alkalmazások voltak elterjedtek, ahol szoros csatolás volt a komponensek között. Ez a 2000-es évekre átalakult szolgáltatásorientált architektúrára (Service-Oriented Architecture), ahol már laza csatolás volt a komponensek között. Majd a 2010-es évektől a mikroszolgáltatás alapú architektúra lett a jellemző, ahol a SOA elveivel, de egy még lazább csatolással alkalmazzák, amit egymással kommunikáló hálózat alkotja. Az egyes mikroszolgáltatások különböző technológiákat, platformfüggetlenül futhatnak, autonóm életciklussal rendelkeznek.

Jellemzője, hogy viszonylag kisméretű szolgáltatásokból állnak, amik a többi szolgáltatással lazán csatolva kommunikálnak. Egy adott szolgáltatás egy magasabb szintű funkciót valósít meg, így a fejlesztése is egyszerű, jellemzően elég egy kis csapat hozzá. Egy mikroszolgáltatás interfészén definiált műveletek jellemzően egy szabványos protokollon keresztül érhetőek el. Saját maga felelős az adatainak és a belső állapotának kezeléséért[13]. Az 2.1 ábrán látható, hogy egy mikroszolgáltatás jellemzően egy jól elkülöníthető konténerből áll.

[1] https://kubernetes.io/

[2] `https://www.cncf.io/`



2.1. ábra. Mikroszolgáltatások architektúra[3].

2.2. Stateful-Stateless alkalmazások

Állapotmentes (stateless) alkalmazásnál elmondható, hogy minden egyes munkamenetet (session) úgy kezeli, mintha az első lenne, nincsen függőség az előző munkamenethez képest. Ez az elosztott architektúránál jelentkezik előnyként, amikor is skálázni szeretnénk az adott funkciót az alkalmazásban és a hibatűrés sokkal könnyebben megvalósítható az általánosan ismert hibaminták alapján, például konténer újraindulással.

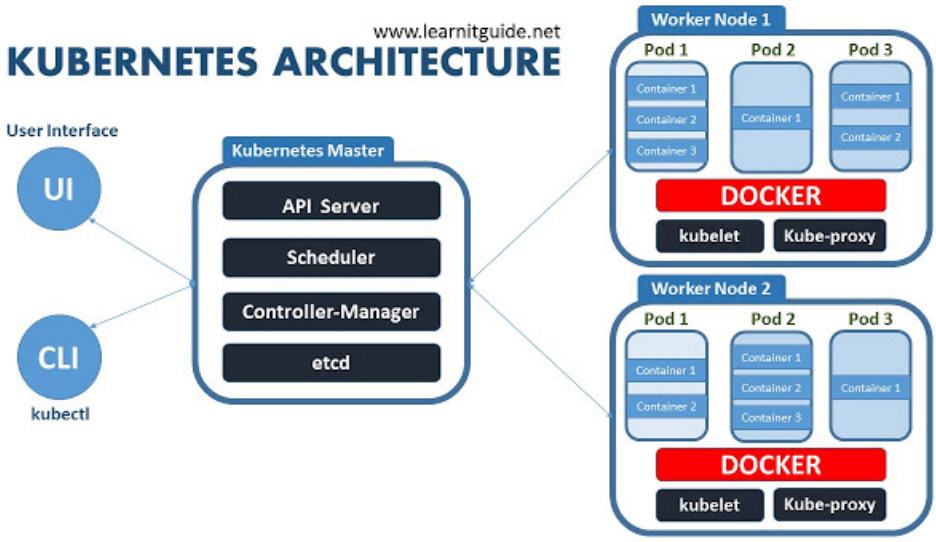
Ezzel szemben az *állapottal rendelkezőnél (stateful)*, számíthatunk arra, hogy szükséges egy szolgáltatástól független tárolóban perzisztálni az állapotinformációkat. Ez megközelítés leginkább biztonságkritikus esetben alkalmazott, amikor is fontos, milyen állapotban voltunk, amikor a szolgáltatásunk leállt. Az igény a *telco* specifikus alkalmazásoknál is megjelenik, egyrészt is a magas rendelkezésre állási követelmény miatt, egy újraindulás során nem engedhető meg, hogy új munkafolyamatot építsen fel, az elmentett állapotból visszatöltve kell folytatnia, hiszen az újraindulási idő így minimalizálható, minimális szolgáltatáskieséssel. Másrészt az adott struktúra komplex információkkal rendelkezik, például a jelenleg eltelt idő a hívásból, ki-kivel lépett kapcsolatba, melyik hálózati szegmenst használtunk, melyik tunelen át folyt a kommunikáció, stb... Az ezekből adódó kontextus szükséges a szolgáltatás helyreállításához hiba esetén.

2.3. Kubernetes szükségessége

A Kubernetes platform célja, hogy létrehozzon, installáljon (deployment) és menedzseljen elosztott alkalmazásokat. Ezek az alkalmazások többféle formában és méretben jönnek, különböző függőségekkel, mint a diverz könyvtárak, különböző fordító függőségek, amiket egy szerverre hosszabb folyamat alapján tudunk csak telepíteni. A hasonló problémákra már találtak megoldást, mégpedig a konténerizációt, amihez szorosan kapcsolódik a mikroszolgáltatás architektúra. A különböző alkalmazás komponensek úgynevezett *konténerekben* futnak, jellemzően több, egymással kommunikáló példánnyal. A Kubernetes képes menedzselni ezeket az alkalmazás konténereket, hogy megfelelően működjön mint a deployment, mint az alkalmazások üzemeltetése, a változatos DevOps szükségletek mellett is.

[3] `https://microservices.io/patterns/microservices.html` Accessed 2019.10.10

2.4. Kubernetes architektúrája



2.2. ábra. Kubernetes architektúra[4].

Az 2.2 ábrán látható, a Kubernetes platform felépítése. A Kubernetes *Master* központi funkciót lát el, az infrastruktúra menedzselése a feladat.

A legkisebb deployolható egység a *pod*. Egy pod tartalmaz egy konténert legalább, ahol a konténerek megosztott erőforrásokat használnak. A *Worker* Node-okon tudunk csak podokat futtatni. A *Node* egy logikai egység, így egy adott fizikai szerver elláthat Master és Worker funkciót is.

Kubernetes komponensei az alábbiak[14]:

- Az **API Server** validálja és beállítja a konfigurációt a komponensek között amit REST útján kap. Egy frontendet biztosít a cluster megosztott erőforrásai számára.
- A **Scheduler** egy erőforrás allokáló szoftver, ami figyeli egy újonnan létrehozott erőforrás kéréshez tartozik-e már lefoglalt erőforrás. Kubernetes terminológiában a *pending* állapotban lévő podokhoz rendel nodekat.
- A **Controller-Manager** egy háttérfolyamat, ami biztosítja a Kubernetes core funkciókat, ”control loop” szerűen. Figyeli a cluster állapotát és a kívánt állapotot igyekszik elérni, így pl a *Replication controller*, ami figyel, hogy az adott podból a beállított példány fusson egyszerre.
- Az **etcd** egy magas rendelkezésre állású Key-Value adatbázis, amiben a Kubernetes állapotinformációi találhatóak.
- A **kubectl**, Command Line Interface-en keresztül lehetőséget biztosít, hogy tudjuk vezérelni a Master működését.
- A Worker Node-on a **kubelet**, ami az elsődleges ”Node Agent”. Az adott Node erőforrásait monitorozza, ő menedzseli az adott node-on lévő podokat.
- A **Kube-proxy** a Kubernetes belső hálózatért felelős, hogy a kiosztott címeken elérhetőek legyenek a szolgáltatások

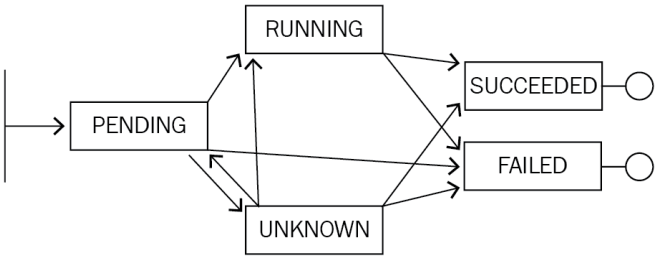
[4] `https://www.learnitguide.net/2018/08/what-is-kubernetes-learn-kubernetes.html` Accessed 2019.10.10

2.5. Kubernetes platformszintű futtatás menedzsment

2.5.1. Deployment vezérlése

Az adott ”terítési” problémára a Kubernetes egy saját fogalmat vezetett be: a *Deployment* egy deklaratív Kubernetes leírás, ami *podok* és a *ReplicaSetek* állapotát írja le. A pod leírásban meghatározzuk az alapvető konténerek szükségeit (milyen imageből álljon, milyen portokon kommunikáljon, stb...), a ReplicaSetek pedig kontrollálják, hogy hány példány fusson az adott podból.

Egy podnak több állapota van [15].



2.3. ábra. Pod lifecycle [5].

- **Pending:** A pod kérést a Kubernetes sikeresen elfogadta, de még egy vagy több konténer nincs kész. Ebben az állapotban töltődnek le a konténer image-ek, illetve ebben az állapotban is marad, ha nem sikerül a Scheduling folyamat.
- **Running:** A podot sikeresen hozzárendelték egy Nodehoz, minden konténer ”elkészült” állapotban van és legalább egy konténer futó állapotban vagy éppen induló vagy újraindulóban.
- **Succedded:** Minden konténer a podban sikerrel lépet ki és nem fog újraindulni.
- **Failed:** Minden konténer a podban kilépett és legalább egy hibával.
- **Unknown:** Valamilyen hiba folyamán nem tudtunk információt gyűjteni a podról, általában kommunikációs hiba miatt.

2.5.2. Restart stratégiák

A *PodSpec*-ben megadhatunk *restartPolicy*-ként 3 fajta stratégiát, amivel a podban található konténereket tudjuk újraindítani.

- **Always:** Mindig amikor a konténer kilépett, újraindítjuk
- **OnFailure:** Csak akkor indítjuk újra, ha hibával lépett ki
- **Never:** Soha nem indítjuk újra

A beépített stratégiáknál van egy beépített védelem, ha a hiba többször, rövid idővel történt, akkor egy egyre növekvő késleltetéssel indítjuk újra az adott konténert (*CrashLoopBackOff*).

3 fajta **Probe** közül választhatunk az üzemeltetés során, amivel a konténerünk állapotát tudjuk monitorozni funkcionalitás szerint, ezzel kiszűrhető a futó, de beragadt konténerek.

[5] `https://subscription.packtpub.com/book/virtualization_and_cloud/9781788834759/5/ch05lvl1sec46/pod-lifecycle` Accessed 2019.10.10

- **livenessProbe:** A konténert poll szerűen lekérdezi, hogy az adott szolgáltatás fut-e. Ha eléri a megadott számú hibás választ egy adott intervallumon belül, akkor leállítja a konténert.
- **readinessProbe:** A szolgáltatásunk elindulásának indikálására használjuk. Ha hibával tér vissza, akkor értesíti az endpoint controllert, hogy erre a podra ne küldjön kérést.
- **startupProbe:** Az szolgáltatás indulásának indikálására használjuk. Ekkor a többi Probe nem használható, ameddig hibával tér vissza. Ha eléri az előre beállított limitet, akkor a konténert terminálja.

2.6. Erőforrás menedzsment

Egy rendszer üzemeltetésében mindig lesznek fix költségek. A konténerizált architektúrával az eddigi Virtual Machinet felváltják a Docker konténerek, ami azért hasznos mert erőforrásokat (RAM, DISK) takaríthatunk meg. A VMekhez szükség van például egy Operációs rendszerre, addig a konténerek a Docker Enginetől függnek, így megspórolva egy OSnyi memóriát és tárhelyet, nem beszélve az indulási idők minimalizálásról. A kihasználás magas fokon tartása is szempont. A kihasználtság definiálható: "jelenlegi áteresztőképesség/teljes áteresztőképesség".

Kubernetes lehetőség nyújt kétféle erőforrás menedzsmentre [12].

- **Resource request** biztosítja, hogy a szükséges erőforrás (RAM, vCPU) rendelkezésre áll, a pod futtatásához.
- **Resource limit** biztosítja, hogy az előre leírt maximális erőforrás értékeket ne lépje túl az adott pod.

2.6.1. Scheduling

Kubernetes terminológiában, a *scheduling*[6] egy Kubernetes Masteren futó folyamat, ami a podokhoz hozzárendeljük az erőforrást (node), ami futtatni fogja.

A *scheduler* figyeli az újonnan létrejövő podokat, amikhez még nincsen node társítva. Minden podhoz, amit a scheduler felfedez, automatikusan felelősséget is vállal, hogy megtalálja a legjobb node-ot, amin a pod futni fog. Ehhez különböző elveket vesz figyelembe, kollektív erőforrás menedzsment, hardver/szoftver megkötések, affinity és anti-affinity specifikációk, adat lokalitás, workload elosztás...

A Scheduling folyamata:

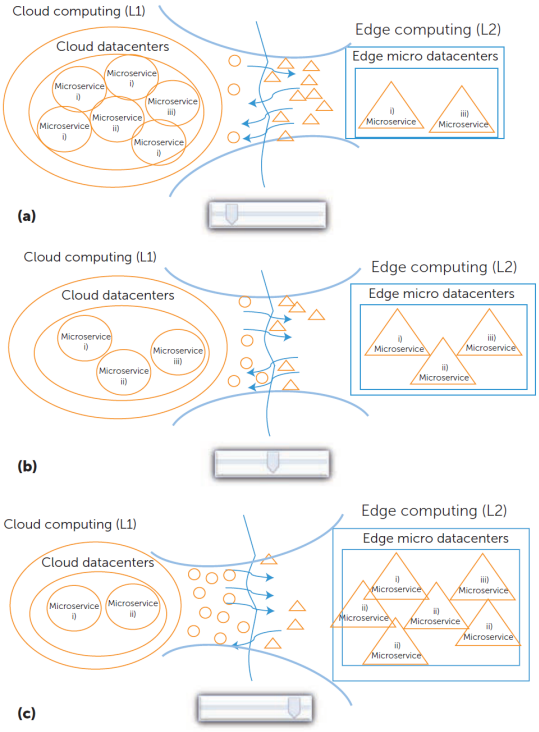
1. **Filtering:** A szűrési folyamat során, az adott podhoz tartozó annotációkat figyelembe véve és a nodeokhoz tartozó képességleírásokat, a scheduler megkeresi azokat a nodeokat, ahova egyáltalán lehetősége van podokat futtatni. Egyfajta node listát készít, amit majd a scoring során felhasznál. Ha a lista üres, akkor a pod *unscheduleable* fázisba kerül.
2. **Scoring:** A szűrés során megmaradt node-ok közül kiválasztja a lehető legmegfelelőbbet, az *activity scoreing rules* alapján.
3. **Assigning:** A scoring alapján kiválasztja a legmegfelelőbb node-ot és értesíti a *kubelet*-et, hogy futtassa a podot. Amennyiben scoring során több nodenak van ugyanannyi "pontja", akkor random választ egyet.

[6] `https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/`

Lehetőségünk van saját scheduler implementációjára is, amennyiben az infrastrukturális követelmények ezt igénylik [16].

2.7. Cloud computing integrálása

Az edge computing elterjedésével szükségessé válik a meglévő infrastruktúra integrálása az új környezetbe. Ennek a megoldására az úgy nevezett *osmotic computing*-ot ismertetem, ami megoldást nyújthat a problémára [17]. Amint az alkalmazásunk microservice architektúra szerint van implementálva, úgy el lehet kezdeni a deploymentek közötti optimális megtalálását (lásd 2.4 ábra), ami során a microservice-einket a cloud és az edge között "terítjük" ki. Az edgben lévő korlátozott erőforrásokat úgy kell felhasználni, hogy a szükséges időkorlátokat betartva fusson a szolgáltatásunk. Ehhez még hozzájön, hogy az esetlegesen elérhetetlenné vált cloud, kiesése esetén is a szolgáltatásnak működnie kell. Természetesen az adott erőforrásoknál az energiatakarékosság is szempont az edgeben, hiszen pl. nem szeretnénk egy akkumulátorról működő egységet a szükségesnél előbb szervizelni.



2.4. ábra. Microservicek megoszlása a Cloud és az Edge között: (a) Néhány microservice megvalósítása az edgeben, (b) Optimális megoszlás a microservicek között cloud és edgeben (c) Microservicek jelentős része az edgeben [17].

3. fejezet

Rendszerleírás megvalósítása szemantikus reprezentációval

A következő fejezetben általam javasolt rendszerújrakonfigurációs megközelítés nagyban támaszkodik szemantikus leíró technológiákra. Jelen fejezet célja ezek rövid bemutatása.

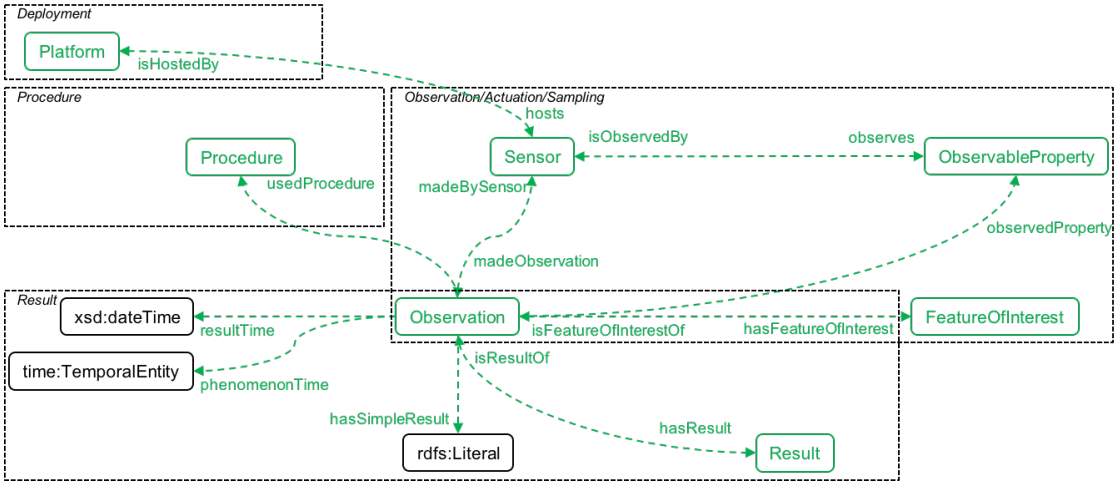
Manapság az elérhető adatforrások közül a szenzoradatok fontos szereppel bírnak. Az adat önmagában számok halmaza, ahhoz hogy ebből információt nyerjünk, szükségünk van egy strukturált leírása, ami megmondja, mit jelentenek a számok számunkra pontosan. Ehhez használhatunk saját magunk által összeállított leírást, de a tervezési szempontok között a hosszútávú és mindenki által elérhető, szabványos leírás a cél. Ebben segítenek minket a W3C Semantic Web[1] technológiák.

3.1. Sensor, Observation, Sample, and Actuator (SOSA)

A *SOSA* egy formális, mégis egyszerű leírást nyújt általános célra történő leírásokra, melyben az alapvető szenzor osztályok és tulajdonságok találhatóak. Ez magában foglalja az interakciókat a *szenzorok*, a *megfigyelések* a *mintavevők* és a *beavatkozók* között. Az 3.1-es ábrán található egy áttekintő ábra, amiből kiemelném a munkám során használt típusokat.

- **Platform:** Egy olyan entitás ami magába foglal több entitást, jellemzően szenzorokat, beavatkozókat, mintavevőket és platformokat, például egy szoba.
- **Sensor:** Egy eszköz, ágens vagy szoftver amivel általában a környezeti változásokat mérjük.
- **Observation:** Egy megfigyelés leírására alkalmazzuk. Leírja, hogy melyik szenzor végezte és hogyan a megfigyelést.
- **ObservableProberty:** Egy megfigyelhető jelenség tulajdonsága, például a szoba hőmérséklete.
- **FeatureOfInterest:** Az a dolog, amire vonatkozóan a megfigyelést végeztük.

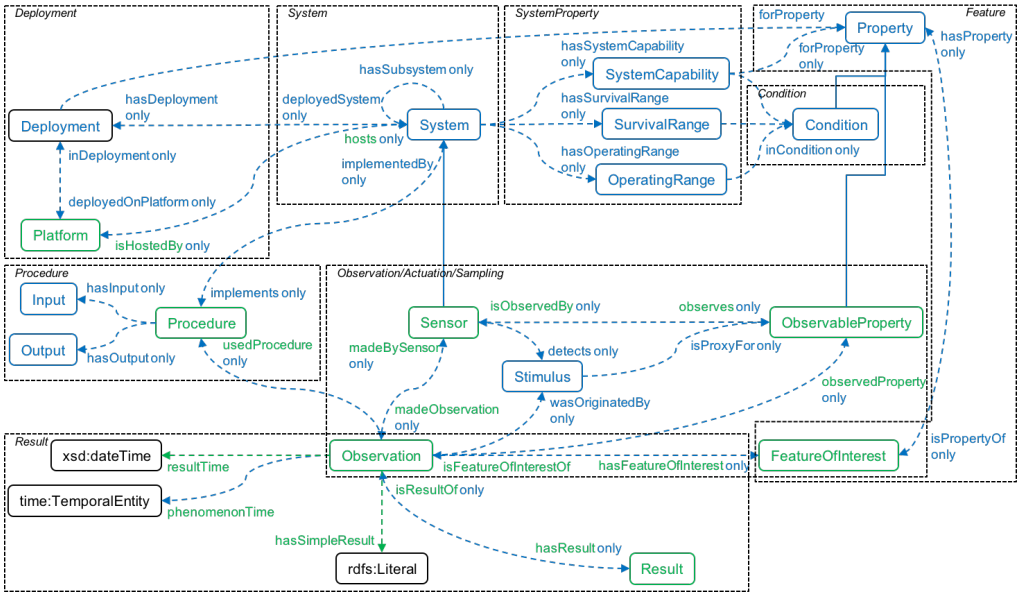
[1] `https://www.w3.org/standards/semanticweb/`



3.1. ábra. A SOSA áttekintő architektúrája[2].

3.2. Semantic Sensor Network

A *Semantic Sensor Network (SSN)*, egy olyan ontológia, ami kifejezetten a szenzorokkal kapcsolatos **folyamatok** leírására lett létrehozva. Leírja a *szenzorokat*, ezek *megfigyeléseit* és a kapcsolódó *folyamatokat*, egy modularizált ontológia szerint. A SOSA beépül a core leírásba, ezt egészíti ki. Megjelenik a *Rendszer* és ahhoz tartozó tulajdonságok, a *Deployment*, a *Property* stb... Így a rendszerünket részletesebben le tudjuk írni, amivel elérhető válik egy kibővített, speciálisabb leírás, amihez kapcsolódóan, az újrakonfigurációs lehetőségeink is nőnek[18].



3.2. ábra. Az SSN áttekintő architektúrája[3].

[2] https://www.w3.org/TR/vocab-ssn/images/SOSA-OntStructure-Observation.png Accessed 2019.10.13.

[3] https://www.w3.org/TR/vocab-ssn/images/SSN-OntStructure-Observation.png Accessed 2019.10.13

4. fejezet

Kiber-fizikai rendszerek szemantikusan támogatott futásidejű újrakonfigurációja

Jelen fejezetben először bemutatom azt a rendszerosztályt, melyet a dolgozatomban behatóan vizsgálok: az olyan kiber-fizikai edge megoldásokat melyek alapvetően szenzoradatok gyűjtését, transzformációját és végeredményben a cloud felé továbbítását végzik. Ezen rendszerosztályon belül definiálom a vizsgált architekturális osztályt is, melynek két fő jellemzője, hogy az alkalmazások fő funkcionális elemeinek megvalósítására Kubernetes podokat használ, illetve, hogy a podok közötti kommunikációt az OMG DDS szabvány megvalósításai segítségével végzi. Egy ezen architekturális mintára illeszkedő általam létrehozott egyszerű rendszerpéldányon bemutatom az ilyen rendszerekben a szemantikus újrakonfiguráció lehetőségeit. Ezután ismertetem az általam megvalósított szemantikus adatbázist, majd megmutatom, hogy annak tartalma hogyan dolgozható fel formális fogalomelemzés segítségével abból a célból, hogy újrakonfigurációk végrehajtásához helyettesítő telepítési elrendezéseket keressünk. Végezetül ismertetem, hogy a szemantikus adatbázis és az ismertetett helyettesítő elrendezéskeresés alkalmazása futásidejű újrakonfigurációra hogyan illeszkedik az autonóm számítástechnika jól ismert ”autonóm menedzsment” felügyeleti megközelítésébe.

4.1. Célprobléma: szenzoradat-feldolgozó kiber-fizikai rendszerek

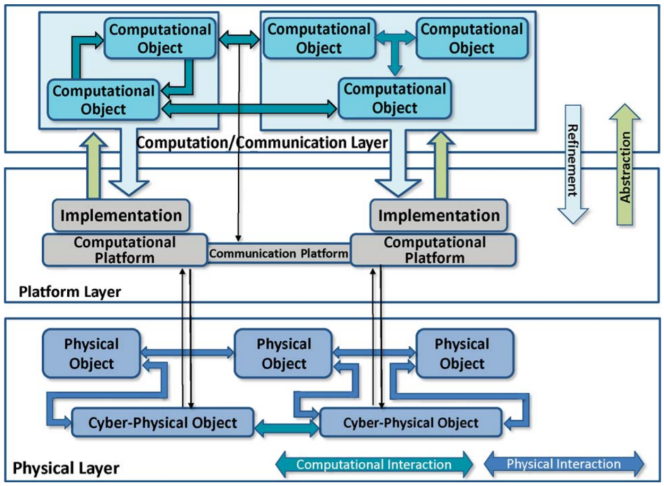
4.1.1. Kiber-fizikai rendszerek

A kiber-fizikai rendszerek egyesítik a számítógépekben rejlő számítási lehetőségeket a valós, fizikai világgal. Beágyazott számítógépek monitorozzák a környezetüket és erre egy fizikai választ adnak. Ezek a rendszerek széles körben használatosak, például magas megbízhatóságú orvosi eszközök, környezetszabályozás, elosztott robotok (telemedicine, telepresence), védelmi rendszerek, assisted living, közlekedésirányítás, ipari rendszerek.

Manapság a legnagyobb kihívás egy ilyen rendszerben a rendszerintegráció, mivel többségében *ad hoc* technikákkal zajlik a komponensek integrálása. Egy kiber-fizikai rendszerben 3 logikailag elválasztott réteg található: fizikai és a két ”kiber” réteg: platform és a szoftver, ami az 4.1 ábrán látható [19].

- **Fizikai réteg:** Fizika komponensek és köztük lévő interakció.

- **Platform réteg:** Egyrészt a hálózati kapcsolatért felelős, másrészt a *Fizikai réteg* irányításért.
- **Szoftver réteg:** A szoftver működésért felelős, különböző komponensei I/O modellen keresztül kommunikálnak.

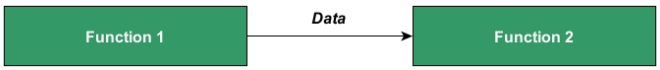


4.1. ábra. Design rétegek a kiber-fizikai rendszerekben [19].

A kutatásom az edge alkalmazásokra koncentrál, ahol jellemzően limitáltak az erőforrások és szükség van egy központi deployment orkesztrátorra. Erre a problémára a Kubernetes-t lehet alkalmazni. Az egyes funkciót megvalósító elemek közti kommunikációra DDS-t alkalmazok, ami biztosítja a kellő flexibilitást egy ilyen környezetben. Az eredeti problémának (Kiber-fizikai rendszerek szemantikusan támogatott futásidejű újrakonfigurációja), ez egy korlátozott részhalmaza, a megoldást viszont szélesebb kontextusra is kiterjeszthető, hasonló jellegű problémákat támogatva [20].

4.1.2. Alkalmazáscsalád: szenzoradatok adatfolyamai

Jellemzően az alkalmazások adatfolyam háló felépítést valósítanak meg, ami során egy adat *Funciton 1*-be érkezik inputként, ott egy feldolgozás történik, majd továbbítja *Function 2*-nek inputként. Ez a megközelítés tovább skálázható. Az ilyen jellegű alkalmazásoknál az egyes funkciók bemeneti és kimeneti adatfolyam interfészeinek definíciója sokszor alkalmasan elvégezhető egyszerű szemantikus leírók segítségével (lásd például az előző fejezetben bemutatott megközelítéseket). Különösen az ilyen adatfolyamok kezdeti lépésére igaz az, hogy a szemantikusan definiált bemenetek és kimenetek teljes mértékben specifikálják a funkciót magát, például egy vagy több finom felbontású bemenet csúszóablakos átlagolása egyszerűen kifejezhető az SSN és a SOSA segítségével. Mindez különösen igaz az adatfolyamhálók legelső adatgyűjtő lépésére. Erre a tulajdonságra a későbbiekben nagyban építeni fogok.



4.2. ábra. Adatfolyam szemléltetése.

4.1.3. Platform: edge Kubernetes

A Kubernetes - Edge alkalmazásokra kiváló, jelenleg egy feltörekvő technológia amit sokan próbálnak integrálni a saját alkalmazásaikba. A terepi komponenseken limitált erőforrásaink vannak, így a Kubernetes erőforrás menedzsmentje reprezentatív példaként szolgál a cloudhoz képest.

4.1.4. Kommunikációs köztesréteg technológia: DDS

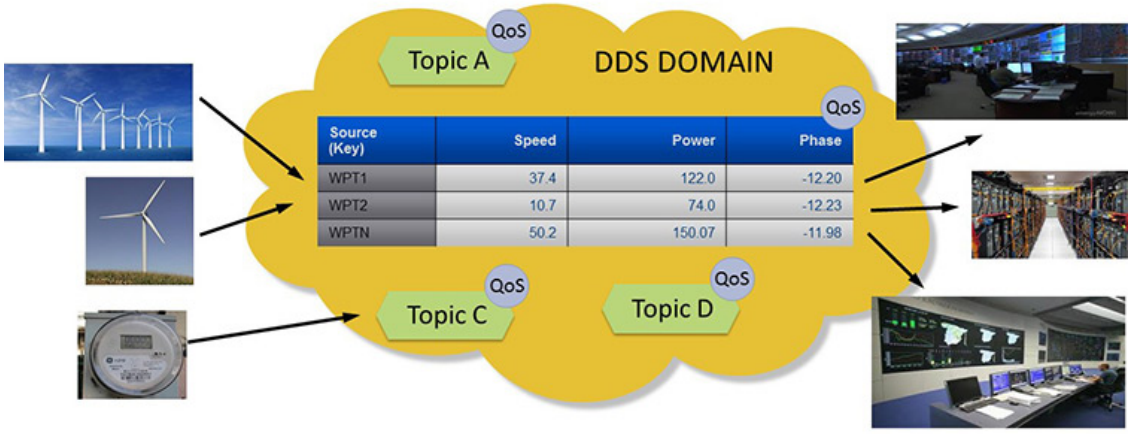
A Data Distribution Service[1] (DDS) egy köztesréteg (az alkalmazás és az operációs rendszer közötti) protokoll és egy API standard, amit az Object Management Group[2] (OMG) fejleszt. A rendszer komponenseinek integrációjára használható, ahol fontos az alacsony késleltetés, magas szintű megbízhatóság és a skálázhatóság. Az elosztott alkalmazásokkal járó kommunikációra, adat megosztásra lehet használni. A DDS egy adatcentrikus megközelítés, ami kedvező az IoT alkalmazások számára. Ez a megközelítés biztosítja, hogy az adathoz kontextus információ is tartozik a küldés során. A DDS számunkra biztosítja ezeket a beállításokat, nekünk alkalmazás szinten nem kell foglalkozni ezzel. Az alkalmazások publish-subscribe modell szerint topicokon keresztül kommunikálnak egymással, amelyik topicokra dinamikusan, futásidőben fel lehet iratkozni egyrészt adatfogadásra, másrészt küldésre. Ez a fajta likviditás megfelelő, hogy futásidőben tudjunk újrakonfigurálni ilyen környezetben és ne kelljen nekünk menedzselni külön, újrakonfigurálás során a kommunikációt. Nincsen egy központi szerver mint a hasonló MQTT[3]-nél, a komponensek közvetlenül tudnak egymással kommunikálni, a dinamikus felderítésnek köszönhetően biztosítható, hogy az alkalmazásunk futásidőben tudjon változtatni a konfigurációján.

Az 4.3 ábrán látható, ahol a különböző szenzoradatok egy névtérben (Domain) tudnak kommunikálni, különböző topicokon, különböző QoS beállításokkal.

Lehetőségünk van Quality of Service (QoS) beállításokra a DDS szintjén. A beállításokat 4 fő csoportban tudjuk elhelyezni, pár fontosabb beállítást emelek ki[4].

- **Real-time Delivery:** Valós idejű szolgáltatások támogatására
  - **Deadline:** A küldőnek mindenképpen küldenie kell egy eredményt egy megadott intervallumon beül.
  - **Lifespan:** Beállítható elévülési idő, így az alkalmazásunk nem fog túl régi adatokat kapni.
- **Persistence**
  - **History:** Az elküldött adatok tárolására alkalmazható.
- **Bandwidth**
- **Redundancy**

[1] `http://www.dds-foundation.org`
[2] `https://www.omg.org`
[3] `http://mqtt.org/`
[4] `https://opendds.org/about/qosusages.html`



4.3. ábra. DDS architektúra példa[5].

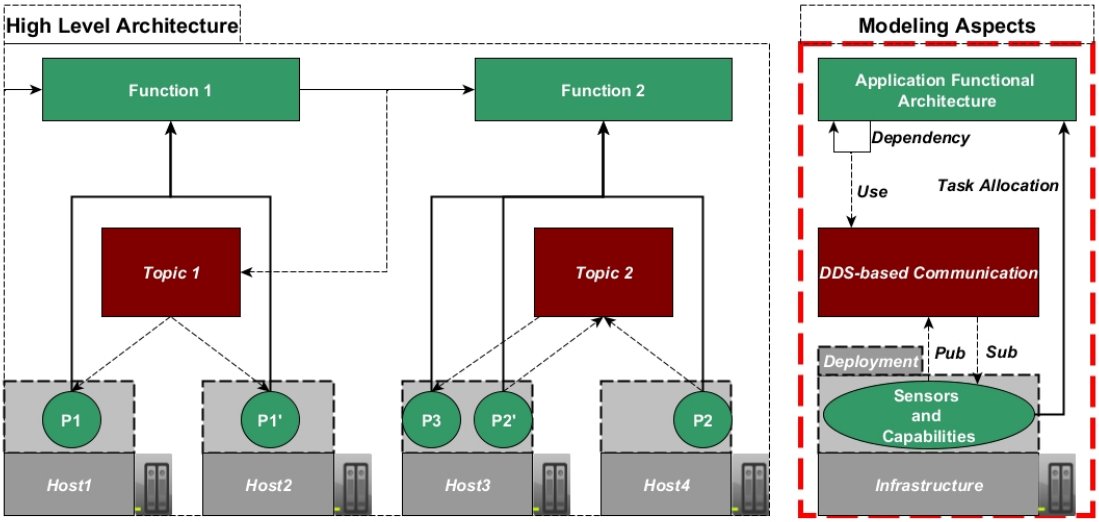
4.1.5. Újrakonfigurációs megkötések: kommunikációs szkeleton

A definiált alkalmazáscsaládban sokféle újrakonfigurációs stratégia elképzelhető. Én azokat az eseteket tekintem, ahol az alkalmazás (adatfolyam jellegű) funkcionális architektúrája és a kommunikációs struktúrája is kötött, azokat az újrakonfiguráció nem érinti. A komponensek közötti egyes adatfolyamokhoz egy-egy dedikált DDS topicot rendelünk, ezt nem akarjuk megváltoztatni az újrakonfigurálás során. Az újrakonfiguráció azt jelenti, hogy az egyes funkcionális komponensek egy alternatív megvalósítását keressük, ahol a megvalósítás lehet ugyanannak a funkciónak egy helyettesítése. DDS alkalmazásával a kommunikációs újrakonfiguráció automatikusan megtörténik, ezzel nem kell külön foglalkozni.

[5] https://www.dds-foundation.org/what-is-dds-3/

4.2. Megvalósítási architektúra

A fent leírt alkalmazáscsaládhoz definiáltam egy architekturális elrendezés-mintát, ami a 4.4 ábrán található. Mivel megközelítésem kulcseleme az alkalmazás és futtatóplatform konfigurációjának és állapotának minél teljesebb és szemantikus futásidejű követése, az ábra az architektúra fő modellezési vetületeit is megadja.



4.4. ábra. Architekturális minta és modellezési vetületei.

4.2.1. Modellezési vetületek

Az általam létrehozott architektúrának alapvetően 4 logikai rétege különböztethető meg (lásd 4.4 ábra *Modeling Aspects*).

- **Fizikai eszközök rétegében** (infrastructure), a Kubernetes konténerizációs környezettel menedzselt olyan fizikai host eszközök találhatóak, melyek amellett, hogy a konténer képek futtatására is képesek, a Kubernetes pod absztrakcióját alkalmazva, egyben szenzorokkal és beavatkozókkal is felszerelhetőek lehetnek (*Sensors and Capabilities*).

- **Deployment réteg**, logikailag a Kubernetes podok hostokra telepítése, tehát az, hogy a különböző konténer képhalmazokat futtató *podok*, hogyan szerepelnek *példányként* az egyes fizikai hostokra telepítve.

- **Kommunikációs réteg**, ami a mintapélda esetében a logikailag létrehozott *DDS topicok*. Ezek a topicok immutábilisak abban az értelemben, hogy az egyes alkalmazásokhoz logikailag ”létrehozott” topicok, logikai nézete akkor is azonos marad, ha az alkalmazás magát egyébként újrakonfigurációnak vetjük alá.

  A topicok és a podok között Publish-Subscribe jellegű kapcsolatok léteznek, melyeket a szemantikus adatbázis követ is. Ismét megjegyzendő, hogy a topicok valójában csak logikai fogalmak DDS használata esetén, legalább is az általános esetben, ellentétben az MQTT-vel, itt nem egy dedikált kiszolgáló valósítja meg a sorokat, a topicokat, hanem a topicok mint logikai fogalmak a DDS könyvtárakat magában foglaló alkalmazások egymást közötti kommunikációját vezérlik, annak egy absztrakcióját adják.

- **Funkcionális réteg:** (Application Functional Architecture) Ez az az architektúra, melyre a korábban ismertetett alkalmazásosztályt képezem le, azaz a szenzor adatfolyamokat kibocsátó és fogadó logikai komponensek mint a funkcionális architektúra elemei *podokba* képződnek le megvalósítási szinten. A funkcionális architektúra elemei közötti összeköttetések (ahol feltételezem, hogy az összeköttetéseken áramló adatok szemantikus leírása elérhető), közvetlenül DDS topicokra képződnek le.

4.3. Mintaalkalmazás és minta szemantikus újrakonfiguráció

A megközelítést demonstrálandó, definiáltam és megvalósítottam egy mintaalkalmazást, mely a 4.4 ábrán *High Level Architecture*-ként látható. Ebben *Function 1* és *Function 2* két alkalmazás funkciót fejez ki, ami fontos, hogy *Function 2* függ *Function 1*-től, így az adat előbb *F1*-ben majd csak utána kerül feldolgozásra *F2*-ben.

A funkciókat *P1-3* valósítja meg, ezek a Kubernetes terminológiában használt podokat rövidítik. A *P1* és *P1'* pedig a replikák jelölésére használatosak. Egy pod tartalmazza a futásidejű implementációját egy funkciónak és jelöli a szenzorokat is, például *P1* tartalmazza a szenzor kiolvasás implementációját és egyben jelöli, hogy *Host1* képes szenzor adatokat szolgáltatni a pod számára.

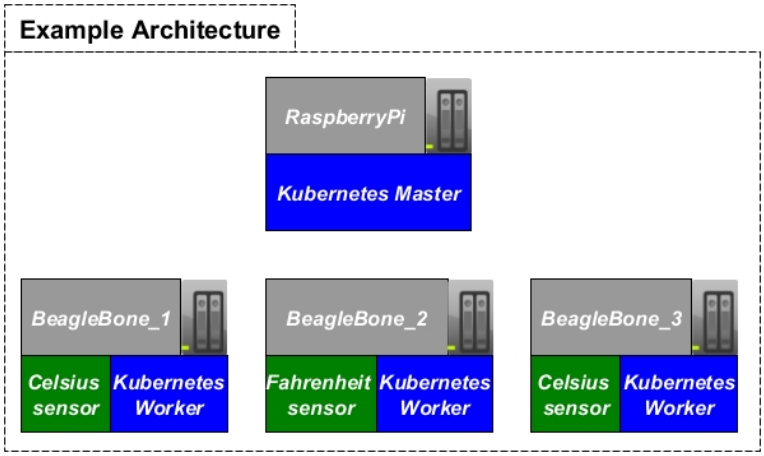
Az alkalmazások kommunikációjára 2 DDS topicot használhatunk. Ez azért fontos, mert az újrakonfiguráció során a topicokat fixnek vesszük és nem módosítjuk őket. Jelen esetben ez egy olyan csatorna, amit tudnak használni az alkalmazások a megfelelő formátumú adat küldésére. Az adott podok különböző *Host1-4*-en futnak, amik fizikailag is elkülönülnek egymástól. Az alkalmazás funkcionális architektúrájának eleme, hogy *F1* és *F2* közötti adatáramlás, szemantikus leírással rendelkezik. Ezt a fajta leírást 4.4.3-ben részletezem.

Az 4.6 ábrán a példában megvalósított alkalmazás 2 funkcionális elemből áll.

- **F1:** Szenzorból származó hőmérséklet adatokat gyűjt, amit továbbít *F2*-nek.
- **F2:** *F1*-től megkapott adatokat feldolgozza (statisztikák számolása), majd továbbítja a következő funkcionális elem felé.

4.3.1. Mintaalkalmazás fizikai architektúrája

A Kubernetes Master szerepében egy Raspberry Pi áll, Ő felelős az alapfunkciók megvalósításáért mint a pod scheduling. A jelenlévő Workerek egy-egy Beaglebone Black Industrialon kapnak helyet. Az adott workerekhez más-más szenzorok tartoznak, amit az 4.5 ábra mutat. A topicok fix kiosztásúak, *Topic 1*-ben Celsius hőmérséklet adatok vannak, míg *Topic 2*-ben szintén Celsius hőmérséklet adatok találhatóak, de már egy aggregált, különböző statisztikák szerinti formátumban.



4.5. ábra. Mintaalkalmazás architektúrája.

4.3.2. Újrakonfigurációs példa

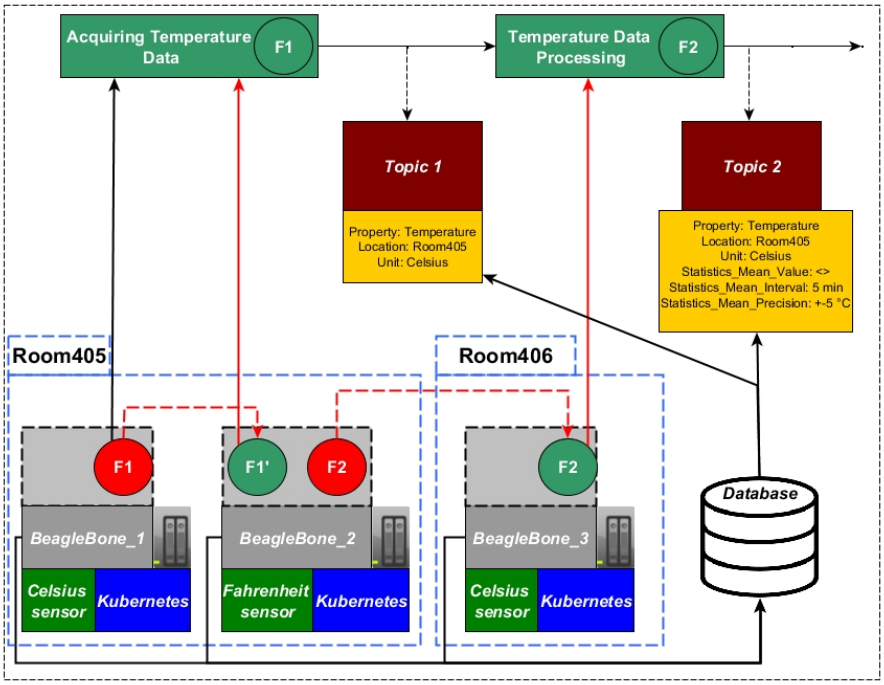
Újrakonfiguráció előtt az alkalmazás megvalósítását *F1* és *F2* Kubernetes podok adják, az 4.6 ábrán a *BeagleBone 1* és *BeagleBone 2*-vel jelölt fizikai hostokon.

Az F1 pod megvalósításnak feladata pusztán annyi, hogy a terepi eszközhöz rendelt hőmérséklet érzékelő szenzor segítségével, kiolvassa a hőmérséklet adatokat (Celsiusfokban) és azokat továbbítja F2 pod számára a Topic 1 nevű DDS topic segítségével.

Figyeljük meg, mivel CPS alkalmazási esetről van szó, ezért *BeagleBone 1*, *BeagleBone 2*, *BeagleBone 3* hiába tartoznak ugyan azon Kubernetes clusterbe (4.5 ábra), köztük a taskok nem tetszőlegesen mozgathatóak. Egyrészt a task migrációját korlátozza, az egyes fizikai eszközökhöz kötött szenzor csomagok tulajdonságai, másrészt figyelme be kell venni azt is, hogy az eszközök fizikailag különböző helyekre lehetnek telepítve.

Tehát például hiába áll rendelkezésre *BeagleBone 3*-on egy hőmérséklet szenzor amely celsiusban adja meg a hőmérsékletet, mivel az alkalmazás specifikációjában kifejezetten a room405 szoba hőmérsékletet jelöltük meg. Lásd 4.6 Topic 1-hez kapcsolt szemantikus leíró.

Más részről, a room405 szobán belül *BeagleBone 2*-re nyilvánvalóan át lehet mozgatni a hőmérséklet mérést, ehhez azonban fel kell ismerni azt, hogy *BeagleBone 2*-őn levő hőmérsékletet mérő szenzor, nem pontosan ugyan az mint a *BeagleBone 1*-ben (itt fahrenheitben mér). Ez után biztosnak kell lennünk abban, hogy fahrenheitből celsiusra tudjuk konvertálni a hőmérsékletet. Megfelelő szemantikus támogatással ezen funkciók kezelhetőek.



4.6. ábra. Reconfiguration example: "semantic" failover.

A 4.6 ábrán pontosan egy ilyen lehetséges újrakonfigurációt mutatok be, ahol *BeagleBone 1*-ről a fizikai hardver kiesésének következtében, az F1 funkcionális elem, az F1 poddal való megvalósítását úgy migráljuk át *BeagleBone 2*-re, hogy közben a mérést utófeldolgozásnak (konverziónak) vetjük alá. Eközben az F2-őt migráljuk egy másik számítási kapacitásra, annak érdekében, hogy a terepi eszközt ne terheljük túl telepített funkciókkal. Az utóbbi migráció az általam alkalmazott hardver esetén nem feltétlenül realisztikus, de vannak olyan kis kapacitású terepi eszközök melyek esetén egy ilyen jellegű plusz feldolgozás (konverzió), jelentős processzor időt vesz el, például egy ESP8266 esetében[6].

[6] `https://www.espressif.com/en/products/hardware/esp8266ex/overview`

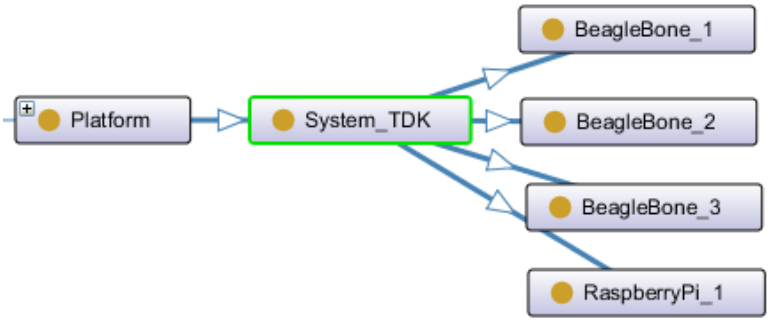
4.4. A megvalósított szemantikus adatbázis

Az adatbázis logikai szinten ontológiák segítségével tárolja az adatokat, fizikai szinten pedig RDF[7] adatbázisban jelennek meg. A szükséges ontológiákról az 4.7 ábra tartalmazza az áttekintést, amit később részletezek is. Az elkészült ontológiát az F.1 függelék tartalmazza.

4.7. ábra. Ontológiák áttekintés

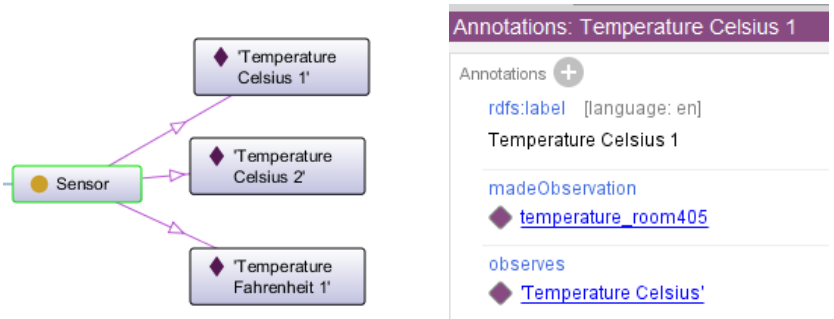
4.4.1. SSN ontológia használata

Az 4.8 ábrán láthatóan, az "sosa:platform" tageket használva építem fel az infrastruktúra elemeit. Itt jelenik meg az infrastruktúra fizikai felépítése.



4.8. ábra. Platform.

Az 4.9 ábrán láthatóan, az "sosa:sensors" tageket használva építem fel a szenzor példányokat, amiket még elláttam külön annotációkkal is 4.9b. Az annotációk szerepe, hogy az adott szenzor példányról el tudjuk dönteni, pontosan milyen attribútumokkal rendelkezik, amit majd fel tudunk később használni az újrakonfiguráció megtervezéséhez (lásd 4.5 fejezetben).

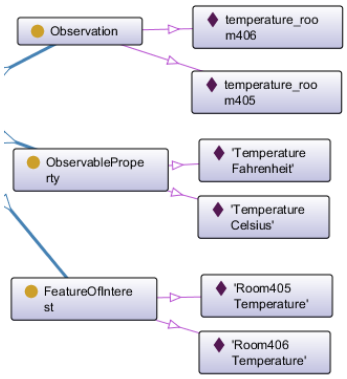


(a) Architektúra **(b)** Temperature Celsius 1 példánya

4.9. ábra. Szenzorok

[7] https://www.w3.org/RDF/

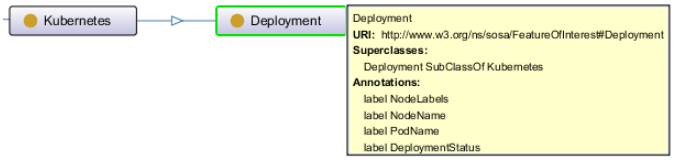
Az eredeti SSN ontológia struktúráját felhasználva, szükséges volt az Oberservation, ObserableProperty és a FeatureOfInterest tagek bevezetése is. Ezeket az annotáció típusokat tudjuk hozzákapcsolni a szenzorok leírásához (4.10 ábra).



4.10. ábra. Observation áttekintés.

4.4.2. Kubernetes ontológia

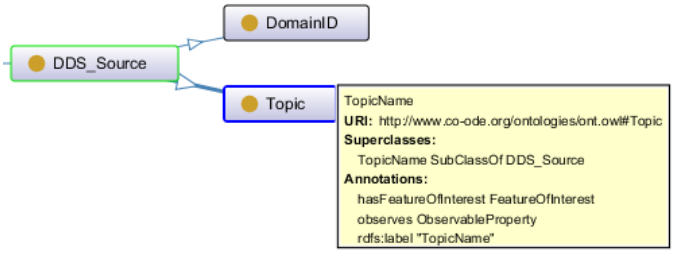
Az ontológia szükséges eleme az adott Pod-Node összerendelés, a nodehoz tartozó *labelekkel*. Itt megjelenik a Kubernetes pod és node szintű információ, amit figyelembe tudunk venni az újrakonfiguráció során.



4.11. ábra. Kubernetes tagek.

4.4.3. DDS ontológia

Szükséges a topic információkat tárolni, erre készült a DDS ontológia. Itt megjelenik a topic specifikus kényszerek, mint a "FeatureOfInterest", ami a mérés helyére utal, illetve az "ObservableProperty" amit a mérés során létrejövő adatok írja le (a mintapéldában Temperature in Celsius).



4.12. ábra. DDS ontológia.

4.5. Helyettesítő megvalósítások felderítése fogalomelemzéssel

A dolgozatomban tárgyalt újrakonfigurációs alkalmazási problémák nehézségét azok *szemantikus* jellege adja. Önmagában a feladatok szenzor, számítási és végrehajtó elemekhez rendelése, mint *pakolási*, illetve *"terítési"* (deployment) probléma nagyon jól ismert. Ugyan úgy igaz mind általános esetben, mind pedig kifejezetten cloud és cloud jellegű környezetek esetén (lásd pl. [21]). Annak algoritmikája azonban, hogy hogyan ismerhetőek fel "közel azonos", illetve *a képességek jelentése* értelmében helyettesítő telepítési lehetőségek egy adott környezetben, még nem teljesen ismert.

Dolgozatomban ezen problémára a következő két fázisú megközelítést javaslom.

4.5.1. Az ismert részfeladat: deployment

A megközelítés *második* fázisa feltételezi, hogy rendelkezésre áll egy olyan mátrix, melynek sorai a fizikai végrehajtóegységek (estünkben: a Kubernetes által felügyelt terepi és edge eszközök), oszlopai pedig a megvalósítandó funkcionális architektúra egyes elemei. A cellákban 1 szerepel, ha az adott funkció megvalósítható az adott egységen, és 0, ha nem. Ezen "telepíthetőségi" kényszereket leíró mátrix alapján egy telepítés keresése jól ismert kombinatorikai probléma, mely során a végrehajtóegységek kapacitásai is figyelembe vehetőek, illetve optimalizációs célok (pl. minimális számú végrehajtó egység alkalmazása) is bevezethetőek.

4.5.2. Helyettesítő megoldások keresése

Az *első* fázis célja, hogy a nem *közvetlenül* adódó telepíthetőségi tények felismerésével támogassa a második fázis által alkalmazott mátrix kitöltését.

Ebből a célból dolgozatomban az ún. formális fogalomelemzés (*formal concept analysis*, FCA) alkalmazását javaslom. Az alapötlet az, hogy ha mind egy megvalósítandó funkció, mind pedig az eszközök képességei szemantikus leírással rendelkeznek, akkor az FCA segítségével szemléletesen strukturálhatóak és sorrendezhetőek az eszközök az alapján, hogy képességeik "mennyire" térnek el a kívántaktól. Dolgozatomban ezt az ötletet szenzorokra demonstrálom: pl. ha egy követelmény egy adott helyiségre és Celsius-fok mérésre vonatkozik, akkor egy szenzor, mely az adott helyiségben található meg, de Fahrenheitben mér, "közelebb" áll a követelmény teljesítéséhez, mint egy olyan, mely emellett még másik helyiségben is (de közben pl. ugyanazon az épület-szinten) található meg. Mindez igaz lesz egy olyan szenzorra is, mely a szomszédos szobában mér, de szintén Celsiusban.

Az FCA abban segít, hogy a "közel azonos" képességeket romló sorrendben fel tudjuk sorolni, hiszen így lehetőségünk van ellenőrizni, hogy rendelkezésünkre áll-e olyan kész algoritmika vagy heurisztika, mely a különbségeket (melyek adott esetben csak reprezentációsak, vagy egyszerű aritmetikaiak) képes áthidalni. Kutatásaim jelenlegi fázisában az ilyen adaptációkat könyvtár-szerűen adottnak veszem (pl. Celsius és Fahrenheit közötti konverzió vagy szomszédos szoba mérései alapján becslés), de ez az aspektus még komoly további kutatásokra ad lehetőséget.

4.5.3. Formal Concept Analysis (FCA)[8]

A *formális fogalomelemzés* (FCA) a matematikának egy olyan ága, amivel a tudásunkat hierarchikusan tudjuk reprezentálni. Egy *fogalom*-nak két fő része van: az *extent* - ami azon objektumokat tartalmazza ami része a fogalomnak; és az *intent*: azok a tulajdonságok, amik olyan objektumokhoz tartoznak amik a fogalom részei. Egy *formális kontextus* egy hármas $\mathbb{K} = (G, M, I)$, ahol

- G az objektumok halmaza (*Gegenstände*),
- M az tulajdonságok halmaza (*Merkmale*), és
- I (*Inzidenz*) egy bináris reláció ezeken a halmazokon, ahol kifejezzük, hogy egy objektumnak "van"-e M tulajdonsága, vagy nincs.

A *formális fogalom* egy olyan *formális kontextus* ami egy (A, B) párból áll ahol $A \subseteq G$ és $B \subseteq M$, ahol az objektumok halmaza pontosan azon objektumokból áll, amiknél közösek a tulajdonságok; és fordítva, a *tulajdonságok halmaza*, azon tuljadonságok halmaza, ami pontosan azon tulajdonságokból áll, amiknél közösek az objektumok.

Létezik egy természetes szubfogalmi-szuperfogalmi reláció az ilyen formális fogalmak között, egy rendezés is:

$$(A_1, B_1) \leq (A_2, B_2) \iff (A_1 \subseteq A_2 \implies B_2 \subseteq B_1) \wedge (B_2 \subseteq B_1 \implies A_1 \subseteq A_2)$$

Vagyis egy fogalom szubfogalmához kisebb számú objektum társul, amelyek maguk között megoszlanak egy szélesebb tulajdonságkészlettel. Az összes formális fogalom halmazának $\mathbb{K}$ fölött a jelölése $\mathfrak{B}(\mathbb{K})$.

Bármely $\mathfrak{B}(\mathbb{K})$ egy teljes háló (lattice), ahol a formális fogalmak minden részhalmazának van legnagyobb alsó és legkisebb felső határa; ami azt jelenti, hogy minden fogalomkészletnek „közös objektum-részhalmaza van a legtöbb megosztott tulajdonsággal" és „közös tulajdonság-részhalmaza a legtöbb megosztott objektummal" formális fogalomként. Vegyük figyelembe, hogy egy fogalom lehet üres – objektum-, és tulajdonság-szinten is.

A fogalmi hálók alapötletét és ún. vonaldiagram alapú vizualizációjuk leolvasásának módját jól ismerteti és magyarázza a következő online tutorial: [22].

4.5.4. FCA alkalmazása

Használjuk az FCA-t a mi mintaalkalmazásunkhoz (4.6 ábra) úgy, hogy felvesszük a fizikai hostok szenzorképességeit, illetve a topic-ok szenzoradat követelményeit mint egy objektum-tulajdonság igazságtábla (lásd 4.13 ábra). Jelen munkában az algoritmusunknak, az FCA-nak nem mondtuk meg, hogy a Topic1-nek és a Topic2-nek kitüntetett szerepe van, hiszen ezek követelmények. Ennek oka, hogy pont a követelmények és eszközképességek egymással teljes, illetve részleges fedéseit keressük. Észrevehető, hogy fogalmilag hierarchikusan építettem fel a mátrix tulajdonság-oszlopait, mivel a kis eltérések (fedések) esetlegesen szoftver könyvtárakkal orvosolhatóak.

Az igazságtáblához tartozó fogalmi háló a 4.14 ábrán látható. (Szemléltetési okokból csak a Topic1-re. A hálót a Concept Explorer eszköz[9] segítségével hoztam létre.) Nézzük végig a fogalmi hálót, látszik, hogy maximális fedése BeagleBone1-nek van; tulajdonságok szempontjából azonos objektumok (ugyanabba a formális fogalomba esnek teljes tulajdonságkészletükkel).

[8] A szakasz szorosan követi [1] FCA leírását (20.-21. oldal).

[9] `http://conexp.sourceforge.net/` Accessed 2019.10.22.

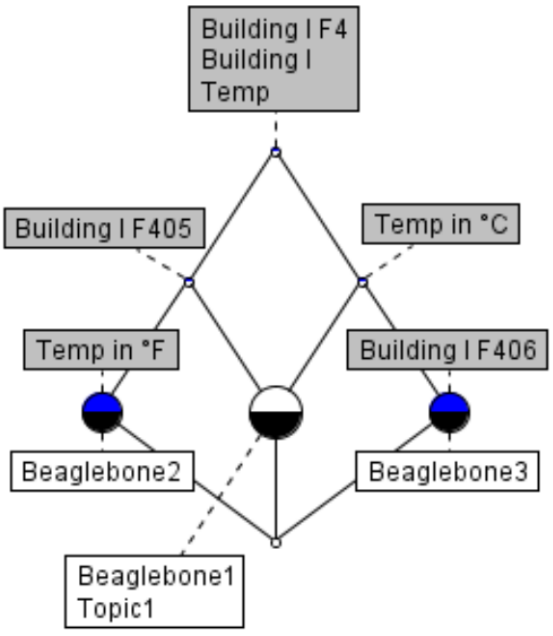
Viszont, amint BeagleBone1 eltűnik, (hiba hatására), akkor szükséges lesz egy másik hostot választani, mivel ide nem tudjuk elhelyezni az adott feladatot. Ekkor észrevesszük, hogy a *következő* legjobb fedése BeagleBone2-nek és BeagleBone3-nak van (az ábrán "balra fel", vagy illetve "jobbra fel" lépve elhagyunk tulajdonság-közösséget, ezzel biztosítva a követelmények teljesítését; melynek köszönhetően az adódó formális fogalomba ismét bekerül egy-egy fizikai eszköz). Itt meg kell néznünk, hogy milyen szoftver könyvtárakkal rendelkezünk, amikkel a hiányzó fedést ki tudjuk pótolni. Tehát BeagleBone2-nél a Temp in Celsiust, BeagleBone3-nál a Building I F405-öt. (A követelmények teljesülésének részlegessége egyszerűen ellenőrizhető, mint halmazkülönbség.)

A jelenlegi könyvtárban elérhető egy Fahrenheit to Celsius konverter, amit fel tudunk használni a BeagleBone2 esetén. BeagleBone3 esetén, viszont nincsen Building I F406 –> Building I F405 konverzió (hiszen fizikailag is más helyen vannak, itt meg lehet jegyezni, hogy fizikai adottságokat feltételezve, mint, hogy a két szoba egymás mellett van és esetleg egybe van nyitva egy ajtóval, epszilon közelítést lehet adni a F405-re).

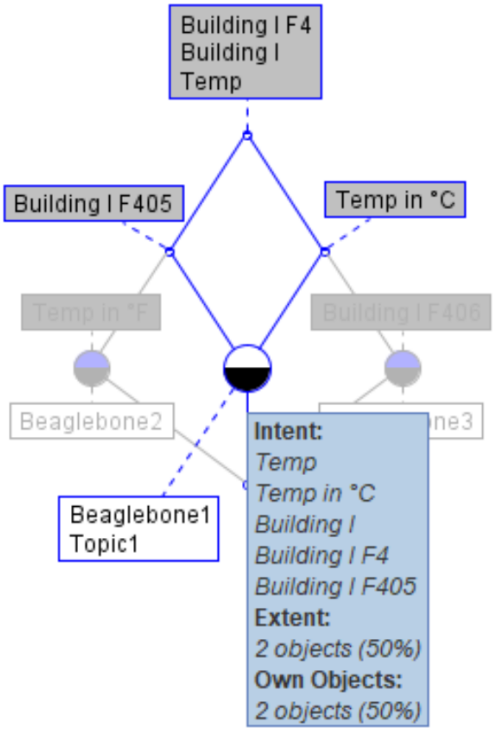
A megközelítés attól lesz szemantikus, hogy a "hőmérséklet" és a "hely" a fogalmi hierarchiáját befoglaltam a tulajdonságmodellbe. Ez csak egy szemantikus adatbázisból és ontológiával támogatott adatbázisból realisztikusan elképzelhető, hogy ezt a finomítási hierarchiát ki lehet generálni. (A finomítási hierarchiában az adott Building rétegződése (egyre pontosabb helymeghatározása) általánosan szerepel a mintapéldában szemléltetve a hierarchikus felbontást, amint felveszünk egy Q épületben lévő szenzort, információértékkel fog rendelkezni a mostani Building I és Building I F4 is).

A	B	C	D	E	F	G	H
	Temp	Temp in °C	Temp in °F	Building I	Building I F4	Building I F405	Building I F406
Topic1	X	X		X	X	X	
Beaglebone1	X	X		X	X	X	
Beaglebone2	X		X	X	X	X	
Beaglebone3	X	X		X	X		X
Topic2	X	X		X	X	X	

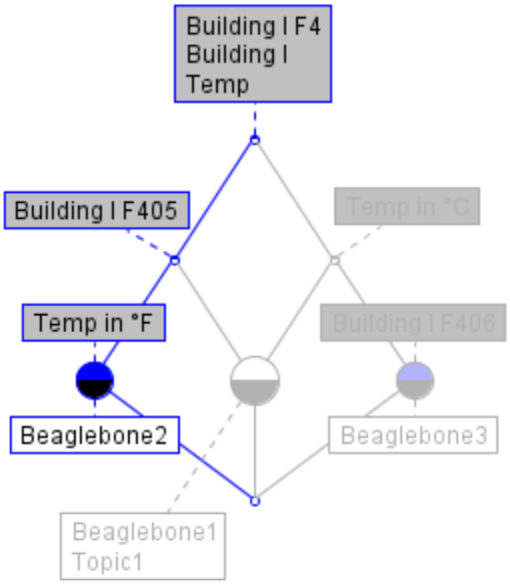
4.13. ábra. FCA mintapéldára.



4.14. ábra. FCA alkalmazása a mintapéldára.

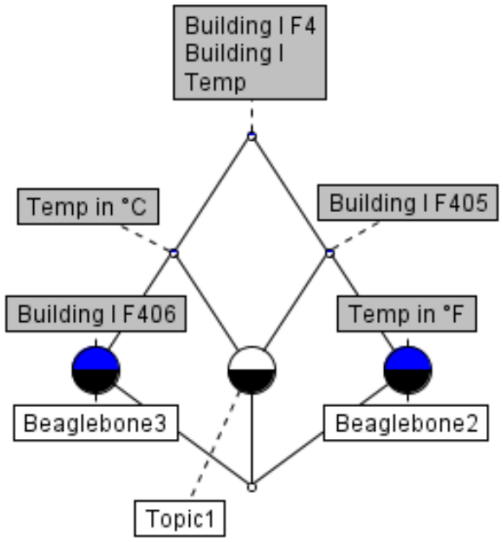


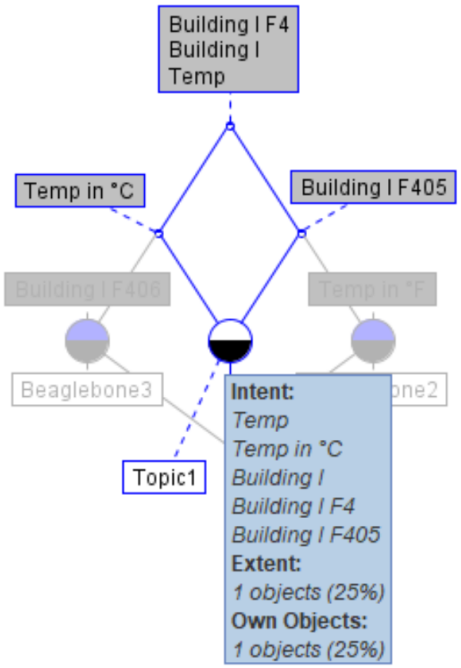
4.15. ábra. Topic 1 és BB1 intentjei



4.16. ábra. Beaglebone2 intentjei

Az 4.15 ábrán látható, ahogy Topic 1 és Beaglebone1 közötti intentek megegyeznek, ami azt jelenti, hogy Topic 1 és Beaglebone1 attribútumai megegyeznek, tehát Beaglebone1 rendelkezik minden olyan követelménnyel, ami Topic 1-hez szükséges. Ez a diagrammról leolvasható a 4.15 ábra nélkül is, hiszen ha kiválasztjuk a Topic 1 ”pontját” és elindulunk felfele, akkor a csúcspontig minden attributum szerepel, ugyan úgy mint Beaglebone 1-nél. Ezzel szemben (4.16 ábrán) ha mondjuk Beaglebone 2 ből indulunk, akkor nem tudunk eljutni a "Temp in C" pontra, így tudjuk, hogy Beaglebone 2 nem teljesíti ezt a követelményt.





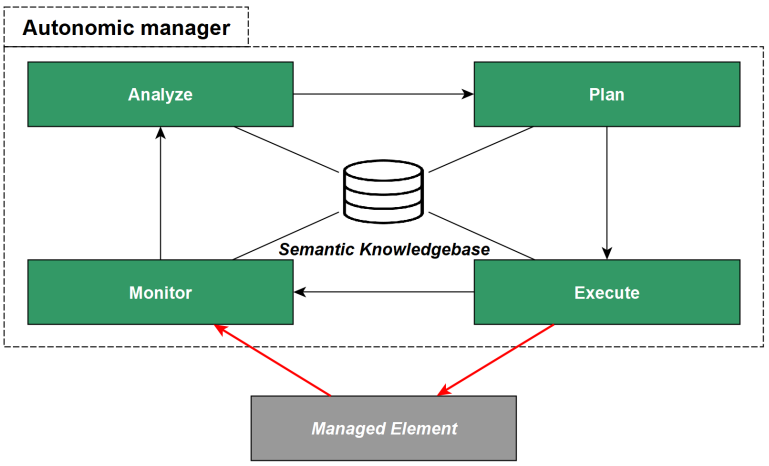
4.17. ábra. Beaglebone1 nélkül **4.18. ábra.** Beaglebone1 nélkül - Topic 1 intentjei

Majd ”szimuláltam”, hogy mi történik, ha BeagleBone1 kiesik. Jól látható, hogy Topic 1-nek most már nincsen teljes fedése az intentjeinek a többi objektummal. Elmondható, hogy ahhoz, hogy Beaglebone2-őt vagy Beaglebone3-at felkészítsük Topic 1 követelményeihez, egy külső könyvtárral a szükséges funkciót helyettesíteni kell. Tehát Beaglebone3-nál át kell hidalni hogy egy másik szobában van (akár valamilyen szobák közötti átlagolás segítségével), Beaglebone2-nél pedig, hogy a hőmérséklet fahrenheitben van.

4.6. Futásidejű újrakonfiguráció megvalósítása

Az előző fejezetekben bevezettem a szemantikus újrakonfiguráció alapjait, egy megoldást javasoltam a szemantikus adattárolás, újrakonfiguráció problémájára. Jelen fejezet az implementáció és az újrakonfiguráció üzemeltetési kérdéseit taglalja.

A logikai megvalósítást érdemes több részegységre felbontani. Egy részegység legyen az *autonomic manager*, ami felelős a hozzá kapcsolt komponens vezérléséért. Az autonomic computing világában [23], több autonomic manager kommunikál egymással, hogy a rendszerhez tartozó szolgáltatás önmenedzselő módon működjön. A mikroszolgáltatások elterjedésével ez a fajta logikai felbontás konténerizált szinten is jelentkezik, így egy adott *Managed Element* lehet egy konténer implementáció is, de akár nagyobb logikai egységet is nézhetünk. Az 4.19 ábra tartalmazza egy ilyen autonomic manager felépítését ami "control loop" elven működik, amivel a folyamatos megfigyelési és beavatkozási ciklusok modellezhetőek. A hozzákapcsolt szemantikus adatbázis biztosítja a szükséges futásidejű információk strukturált tárolását. Ez a minta az autonóm számítástechnikában a MAPE-K szabályozási kör nevet viseli (Monitor, Analyze, Plan, Execute with Knowledge); végeredményben ezen ismert minta alkalmazását javaslom a dolgozatban vizsgált problémakörre.



4.19. ábra. Autonóm szemantikus újrakonfiguráció az autonóm számítástechnika MAPE-K mintája segítségével.

4.6.1. Monitor

A managed element futása során létrejövő állapotinformációk monitorozása, a szemantikus leíróból vett kényszerek mellett történik. Ebből a fázisból hibadetekció esetén továbblépünk és elindul a loop előröl.

4.6.2. Analyze

Az első lépésben az *autonomic manager* feltérképezi az adott funkció környezetét. Mivel egy előre modellezett architektúra elemeiből építkezünk (lásd 4.4 ábra), ezért a fő funkciók lekérése, az adott DDS Topic kiosztás, a szenzorok és az infrastruktúra megállapítása történik, a szemantikus adatbázis segítségével, így ezek az információk szabványos struktúraként érhetőek el. A "loop" ismételt futtatásával, ideálisan csak a változtatásokat kéri le, amik a rendszer új állapotát írják le.

4.6.3. Plan

Hibadetekció esetén egy újrakonfigurációs stratégiát állítunk össze, amiben az adott hiba elhárítása a cél. Mivel egy szemantikus adatbázis is rendelkezésére áll, így az ebből származó információk felhasználásával és az FCA korábban ismertettet lehetőségeivel, egy alternatív konfigurációt keresünk a szolgáltatás helyreállítására.

4.6.4. Execute

Az alternatív újrakonfiguráció megtalálása után újra kell konfigurálnunk az adott funkciók implementációját futtató infrastrukturális elemeket. Ezt a konfigurációt a Kubernetes felé kell továbbítani, ahol is egy új pod-node összerendelés történik, illetve lehetőség szerint az új terhelés miatt egy terheléselosztási algoritmus segítségével a meglévő podok a nekik és a nodeok számára megfelelőbb eloszlást vesznek fel.

5. fejezet

Összefoglaló

5.1. Az általam elért, a dolgozatban ismertetett eredmények

Megvizsgáltam az edge computing lehetőségeit, kihívásait és az ehhez kapcsolódó platformtechnológiát a Kubernetest. A rendszer leírásárát támogató ontológiákat kerestem (SOSA,SSN) és létre is hoztam a probléma támogatására egy saját ontológia javaslatot, majd egy szemantikusan támogatott újrakonfigurációt támogató modellezési sémát vezettem be, aminek a működését egy mintapéldával szemléltettem. A helyettesítő megoldások keresésénél az FCA-t újraszerűen használtam az újrakonfigurációs lehetőségek keresésére. A futásidejú újrakonfiguráció megvalósítására egy magas szintű tervet adtam.

5.2. További kutatási lehetőségek

További kutatási lehetőségként adódik a munkámban szemléltetett architektúra kiterjesztése egy még általánosabb felépítésű rendszerekre. Illetve a szemantikus adatbázis kiterjesztése egy futásidejú konfigurálási példával, Plug and play CPS eszközökkel [20]. Majd a kidolgozott mintapélda megvalosítása egy demonstrátor rendszerben.

Irodalomjegyzék

[1] Imre Kocsis. Qualitative models in resilience assurance. PhD Thesis BME, 2019.

[2] Anthony D Josep, Randy Katz, Andy Konwinski, Lee Gunho, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[3] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[4] Cisco Global Cloud Index. Cisco global cloud index: Forecast and methodology, 2016–2021 white paper, 2018.

[5] Dr. Mark van Rijmenam. Self-driving cars will create 2 petabytes of data, what are the big data opportunities for the car industry?, Accessed: 2019-10-15. `https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172`.

[6] J. Ren, H. Guo, C. Xu, and Y. Zhang. Serving at the edge: A scalable iot architecture based on transparent computing. *IEEE Network*, 31(5):96–105, 2017. DOI: `10.1109/MNET.2017.1700030`.

[7] Simone Mangiante. Fog and edge computing in telecom networks, Accessed: 2019-10-15. `http://pages.di.unipi.it/throughthefog/wp-content/uploads/sites/13/2017/02/mangiante.pdf`.

[8] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[9] Robert S Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[10] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On dependable systems and networks*, pages G8–G9, 2008.

[11] James PG Sterbenz, Egemen K Çetinkaya, Mahmood A Hameed, Abdul Jabbar, Shi Qian, and Justin P Rohrer. Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation. *Telecommunication systems*, 52(2):705–736, 2013.

[12] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[13] Dudás Ákos Benedek Zoltán. Mikroszolgáltatások architektúra bevezető, Belső BME anyag, Accessed: 2019-10-15.

[14] Kubernetes. Kubernetes documentation, Accessed: 2019-10-15. `https://kubernetes.io/docs/`.

[15] Kubernetes. Kubernetes pod lifecycle, Accessed: 2019-10-15. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/.

[16] Anna Reale Peter Suskovics Benedek Kovacs Michael Chima Ogbuachi, Chinmay Gore. Context-aware k8s scheduler for real time distributed 5g edge computing applications, Accessed: 2019-10-15. http://softcom2019.fesb.unist.hr/wp-content/uploads/2019/09/FINAL_PROGRAM_2019_v13.pdf.

[17] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[18] W3C. Semantic sensor network ontology, w3c recommendation 19 october 2017 (link errors corrected 08 december 2017), Accessed: 2019-10-15. https://www.w3.org/TR/vocab-ssn/.

[19] Janos Sztipanovits, Xenofon Koutsoukos, Gabor Karsai, Nicholas Kottenstette, Panos Antsaklis, Vijay Gupta, Bill Goodwine, John Baras, and Shige Wang. Toward a science of cyber–physical system integration. *Proceedings of the IEEE*, 100(1):29–44, 2011.

[20] Vaclav Jirkovsky, Marek Obitko, Petr Kadera, and Vladimir Marik. Toward plug&play cyber-physical system components. *IEEE Transactions on Industrial Informatics*, 14(6):2803–2811, 2018.

[21] Imre Kocsis, Zoltán Ádám Mann, and Dávid Zilahi. Optimal deployment for critical applications in infrastructure as a service. In *3rd International IBM Cloud Academy Conference (ICACON 2015)*, 2015.

[22] Bernhard Ganter. Formal concept analysis: A useful example of modern mathematics, Accessed: 2019-10-15. https://www.youtube.com/watch?v=Xuxm929tIRY.

[23] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, (1):41–50, 2003.

Függelék

F.1. Az elkészült ontológia

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix qudt-1-1: <http://qudt.org/1.1/schema/qudt#> .
@prefix qudt-unit-1-1: <http://qudt.org/1.1/vocab/unit#> .
@base <http://www.w3.org/2002/07/owl#> .

[ rdf:type owl:Ontology
 ] .

#################################################################
#    Annotation properties
#################################################################

###  http://qudt.org/1.1/schema/qudt#unit
qudt-1-1:unit rdf:type owl:AnnotationProperty .


###  http://www.w3.org/ns/sosa/hosts
sosa:hosts rdf:type owl:AnnotationProperty .


###  http://www.w3.org/ns/sosa/madeObservation
sosa:madeObservation rdf:type owl:AnnotationProperty .


###  http://www.w3.org/ns/sosa/observes
sosa:observes rdf:type owl:AnnotationProperty ;
              rdfs:subPropertyOf sosa:observes .


###  http://www.w3.org/ns/sosa/FeatureOfInterest#label
<http://www.w3.org/ns/sosa/FeatureOfInterest#label> rdf:type owl:AnnotationProperty .


###  http://www.w3.org/ns/sosa/hasFeatureOfInterest#hasFeatureOfInterest
<http://www.w3.org/ns/sosa/hasFeatureOfInterest#hasFeatureOfInterest> rdf:type owl:AnnotationProperty
      .


#################################################################
#    Object Properties
#################################################################

###  http://www.co-ode.org/ontologies/ont.owl#DomainID
<http://www.co-ode.org/ontologies/ont.owl#DomainID> rdf:type owl:ObjectProperty .


###  http://www.co-ode.org/ontologies/ont.owl#TopicName
<http://www.co-ode.org/ontologies/ont.owl#TopicName> rdf:type owl:ObjectProperty .
```

```
###  http://www.w3.org/ns/sosa/FeatureOfInterest#DeploymentStatus
<http://www.w3.org/ns/sosa/FeatureOfInterest#DeploymentStatus> rdf:type owl:ObjectProperty .


###  http://www.w3.org/ns/sosa/FeatureOfInterest#FeatureOfInterest
<http://www.w3.org/ns/sosa/FeatureOfInterest#FeatureOfInterest> rdf:type owl:ObjectProperty ;
                                                                rdfs:subPropertyOf owl:
    topObjectProperty .


###  http://www.w3.org/ns/sosa/FeatureOfInterest#NodeLabels
<http://www.w3.org/ns/sosa/FeatureOfInterest#NodeLabels> rdf:type owl:ObjectProperty .


###  http://www.w3.org/ns/sosa/FeatureOfInterest#NodeName
<http://www.w3.org/ns/sosa/FeatureOfInterest#NodeName> rdf:type owl:ObjectProperty .


###  http://www.w3.org/ns/sosa/FeatureOfInterest#PodName
<http://www.w3.org/ns/sosa/FeatureOfInterest#PodName> rdf:type owl:ObjectProperty .


###  http://www.w3.org/ns/sosa/ObservableProperty#ObservableProperty
<http://www.w3.org/ns/sosa/ObservableProperty#ObservableProperty> rdf:type owl:ObjectProperty ;
                                                                  rdfs:subPropertyOf owl:
    topObjectProperty .


#################################################################
#    Classes
#################################################################

###  http://www.co-ode.org/ontologies/ont.owl#BeagleBone_1
<http://www.co-ode.org/ontologies/ont.owl#BeagleBone_1> rdf:type owl:Class ;
                                                        rdfs:subClassOf <http://www.co-ode.org/
    ontologies/ont.owl#System_TDK> .


###  http://www.co-ode.org/ontologies/ont.owl#BeagleBone_2
<http://www.co-ode.org/ontologies/ont.owl#BeagleBone_2> rdf:type owl:Class ;
                                                        rdfs:subClassOf <http://www.co-ode.org/
    ontologies/ont.owl#System_TDK> .


###  http://www.co-ode.org/ontologies/ont.owl#BeagleBone_3
<http://www.co-ode.org/ontologies/ont.owl#BeagleBone_3> rdf:type owl:Class ;
                                                        rdfs:subClassOf <http://www.co-ode.org/
    ontologies/ont.owl#System_TDK> .


###  http://www.co-ode.org/ontologies/ont.owl#DDS_Source
<http://www.co-ode.org/ontologies/ont.owl#DDS_Source> rdf:type owl:Class .


###  http://www.co-ode.org/ontologies/ont.owl#Domain
<http://www.co-ode.org/ontologies/ont.owl#Domain> rdf:type owl:Class ;
                                                  rdfs:subClassOf <http://www.co-ode.org/ontologies/
    ont.owl#DDS_Source> ;
                                                  rdfs:label <http://www.co-ode.org/ontologies/ont.
    owl#DomainID> .


###  http://www.co-ode.org/ontologies/ont.owl#RaspberryPi_1
<http://www.co-ode.org/ontologies/ont.owl#RaspberryPi_1> rdf:type owl:Class ;
                                                         rdfs:subClassOf <http://www.co-ode.org/
    ontologies/ont.owl#System_TDK> .


###  http://www.co-ode.org/ontologies/ont.owl#System_TDK
<http://www.co-ode.org/ontologies/ont.owl#System_TDK> rdf:type owl:Class ;
                                                      rdfs:subClassOf sosa:Platform .
```

```
###  http://www.co-ode.org/ontologies/ont.owl#Topic
<http://www.co-ode.org/ontologies/ont.owl#Topic> rdf:type owl:Class ;
                                                 rdfs:subClassOf <http://www.co-ode.org/ontologies/
    ont.owl#DDS_Source> ;
                                                 rdfs:label "TopicName" ;
                                                 sosa:observes <http://www.w3.org/ns/sosa/
    ObservableProperty#ObservableProperty> ;
                                                 <http://www.w3.org/ns/sosa/hasFeatureOfInterest#
    hasFeatureOfInterest> <http://www.w3.org/ns/sosa/FeatureOfInterest#FeatureOfInterest> .


###  http://www.w3.org/ns/sosa/FeatureOfInterest
sosa:FeatureOfInterest rdf:type owl:Class .


###  http://www.w3.org/ns/sosa/ObservableProperty
sosa:ObservableProperty rdf:type owl:Class .


###  http://www.w3.org/ns/sosa/Observation
sosa:Observation rdf:type owl:Class .


###  http://www.w3.org/ns/sosa/Platform
sosa:Platform rdf:type owl:Class .


###  http://www.w3.org/ns/sosa/Sensor
sosa:Sensor rdf:type owl:Class .


###  http://www.w3.org/ns/sosa/FeatureOfInterest#Deployment
<http://www.w3.org/ns/sosa/FeatureOfInterest#Deployment> rdf:type owl:Class ;
                                                         rdfs:subClassOf <http://www.w3.org/ns/sosa/
    FeatureOfInterest#Kubernetes> ;
                                                         <http://www.w3.org/ns/sosa/FeatureOfInterest
    #label> <http://www.w3.org/ns/sosa/FeatureOfInterest#DeploymentStatus> ,

            <http://www.w3.org/ns/sosa/FeatureOfInterest#NodeLabels> ,

            <http://www.w3.org/ns/sosa/FeatureOfInterest#NodeName> ,

            <http://www.w3.org/ns/sosa/FeatureOfInterest#PodName> .


###  http://www.w3.org/ns/sosa/FeatureOfInterest#Kubernetes
<http://www.w3.org/ns/sosa/FeatureOfInterest#Kubernetes> rdf:type owl:Class .


#################################################################
#    Individuals
#################################################################

###  http://example.org/data/foi/room405_temp
<http://example.org/data/foi/room405_temp> rdf:type owl:NamedIndividual ,
                                                    sosa:FeatureOfInterest ;
                                           rdfs:label "Room405 Temperature"@en .


###  http://example.org/data/foi/room406_temp
<http://example.org/data/foi/room406_temp> rdf:type owl:NamedIndividual ,
                                                    sosa:FeatureOfInterest ;
                                           rdfs:label "Room406 Temperature"@en .


###  http://example.org/data/host/BeagleBone_1
<http://example.org/data/host/BeagleBone_1> rdf:type owl:NamedIndividual ,
                                                     sosa:Platform ;
                                            rdfs:label "BeagleBone_1"@en ;
                                            sosa:hosts <http://example.org/data/sensor/tempC1> .
```

```
###  http://example.org/data/host/BeagleBone_2
<http://example.org/data/host/BeagleBone_2> rdf:type owl:NamedIndividual ,
                                                     sosa:Platform ;
                                            rdfs:label "BeagleBone_2"@en ;
                                            sosa:hosts <http://example.org/data/sensor/tempF1> .


###  http://example.org/data/host/BeagleBone_3
<http://example.org/data/host/BeagleBone_3> rdf:type owl:NamedIndividual ,
                                                     sosa:Platform ;
                                            rdfs:label "BeagleBone_3"@en ;
                                            sosa:hosts <http://example.org/data/sensor/tempC2> .


###  http://example.org/data/host/RaspberryPi_1
<http://example.org/data/host/RaspberryPi_1> rdf:type owl:NamedIndividual ,
                                                      sosa:Platform ;
                                             rdfs:label "RaspberryPi_1"@en .


###  http://example.org/data/host/system_tdk
<http://example.org/data/host/system_tdk> rdf:type owl:NamedIndividual ,
                                                   sosa:Platform ;
                                          rdfs:label "System_TDK"@en ;
                                          sosa:hosts <http://example.org/data/host/BeagleBone_1> ,
                                                     <http://example.org/data/host/BeagleBone_2> ,
                                                     <http://example.org/data/host/BeagleBone_3> ,
                                                     <http://example.org/data/host/RaspberryPi_1> .


###  http://example.org/data/o/temperature_room405
<http://example.org/data/o/temperature_room405> rdf:type owl:NamedIndividual ,
                                                         sosa:Observation .


###  http://example.org/data/o/temperature_room406
<http://example.org/data/o/temperature_room406> rdf:type owl:NamedIndividual ,
                                                         sosa:Observation .


###  http://example.org/data/op/temperature_celsius
<http://example.org/data/op/temperature_celsius> rdf:type owl:NamedIndividual ,
                                                          sosa:ObservableProperty ;
                                                 qudt-1-1:unit qudt-unit-1-1:DegreeCelsius ;
                                                 rdfs:label "Temperature Celsius"@en .


###  http://example.org/data/op/temperature_fahrenheit
<http://example.org/data/op/temperature_fahrenheit> rdf:type owl:NamedIndividual ,
                                                             sosa:ObservableProperty ;
                                                    qudt-1-1:unit qudt-unit-1-1:DegreeFahrenheit ;
                                                    rdfs:label "Temperature Fahrenheit"@en .


###  http://example.org/data/sensor/tempC1
<http://example.org/data/sensor/tempC1> rdf:type owl:NamedIndividual ,
                                                 sosa:Sensor ;
                                        rdfs:label "Temperature Celsius 1"@en ;
                                        sosa:madeObservation <http://example.org/data/o/
    temperature_room405> ;
                                        sosa:observes <http://example.org/data/op/temperature_celsius
    > .


###  http://example.org/data/sensor/tempC2
<http://example.org/data/sensor/tempC2> rdf:type owl:NamedIndividual ,
                                                 sosa:Sensor ;
                                        rdfs:label "Temperature Celsius 2"@en ;
                                        sosa:madeObservation <http://example.org/data/o/
    temperature_room406> ;
```

```
                                        sosa:observes <http://example.org/data/op/temperature_celsius
    > .


###  http://example.org/data/sensor/tempF1
<http://example.org/data/sensor/tempF1> rdf:type owl:NamedIndividual ,
                                                sosa:Sensor ;
                                        rdfs:label "Temperature Fahrenheit 1"@en ;
                                        sosa:madeObservation <http://example.org/data/o/
    temperature_room405> ;
                                        sosa:observes <http://example.org/data/op/
    temperature_fahrenheit> .


###  Generated by the OWL API (version 4.5.9.2019-02-01T07:24:44Z) https://github.com/owlcs/owlapi
```