

TDK-dolgozat

Polyák Máté

Orkesztrátor és klaszterező algoritmusok kiértékelése IoT-Köd-Felhő szimulátorban

Szerző:

Polyák Máté

*III. évf. programtervező informatikus BSc
szakos hallgató*

Témavezetők:

Dr. Kertész Attila, egyetemi docens

Dr. Márkus András, adjunktus

SZEGEDI TUDOMÁNYEGYETEM

Természettudományi és Informatikai Kar

Szoftverfejlesztés Tanszék

Tartalomjegyzék

Absztrakt.....	4
1. Bevezetés.....	5
1.1. Elosztott orkesztráció.....	6
1.2. Problémafelvetés.....	10
2. DISSECT-CF-Fog.....	11
3. Klaszterező algoritmusok.....	15
3.1. Centralizált ACO algoritmus.....	15
3.2. Elosztott ACO algoritmus.....	20
3.2.1. Feromonmátrix értékeinek meghatározása.....	20
3.2.2. Mátrixok egyesítése.....	21
3.2.3. Klaszter csoportok létrehozása.....	21
3.2.4. Szimuláció előkészítése.....	22
4. Kiértékelés.....	24
4.1. Első scenárió.....	25
4.2. Második scenárió.....	27
5. Konklúzió.....	30
Irodalomjegyzék.....	31

Absztrakt

Az Internet of Things (IoT) fejlődésével egyre több eszköz kapcsolódik dinamikus hálózatokhoz, és ezek a készülékek hatalmas mennyiségű tárolási és feldolgozási kapacitást igényelnek. Az efféle igények kiszolgálása gyakran a köd erőforrásokkal kiegészített felhő adatközpontok feladata, amelyek így rendkívül rövid válaszidőt biztosítanak. A köd rendszerek az erőforrásokat a végfelhasználók közelében helyezik el, aminek következtében magas rendelkezésre állást, minimális késleltetést és adatbiztonságot nyújtanak. Az IoT-Köd-Felhő rendszerek összetettsége miatt a különböző vezérlő algoritmusokat egy szimulátor segítségével értékelik ki, amely lehetővé teszi a rendszerek gyors tervezését és az IoT alkalmazások hatékony kezelését, figyelembe véve az energiahatékonyságot, a költségeket és a végrehajtási időt.

A TDK dolgozat célja a DISSECT-CF-Fog szimulátor kiterjesztése elosztott orkesztráció támogatásával, amelynek a lényege, hogy nem egy központi szervező, hanem több elosztott entitás irányítja a rendszer és a hálózat működését, erőforrásait és az alkalmazások által generált feladatokat. Az orkesztrátorok maguk döntenek a feldolgozandó feladatokról és az erőforrások menedzseléséről. Ez a módszer rugalmasabb és ellenállóbb lehet a hibákra és zavarokra, mivel nincs egyetlen központi csomópont, amely a hálózat működését irányítaná. A munkám során kifejlesztettem kettő, a hangya kolónia algoritmuson alapuló klaszterező megoldást is, amelyek csoportosítják az elérhető erőforrásokat azok fizikai pozíciójuk alapján. Ez lehetővé teszi a decentralizált entitások számára, hogy hatékonyan együttműködjenek anélkül, hogy központi irányításra lenne szükség.

1. fejezet

Bevezetés

Az elmúlt években megjelent chiphiány és a Covid-19 utáni gazdasági recesszió ellenére az Internet-of-Things (IoT) eszközök és rendszerek száma továbbra is jelentős mértékben emelkedik [1], ami megnövekedett adatmennyiséget és számítási igényt is magával hoz. Ezen igények megbízható, gyors és biztonságos kiszolgáltatásának érdekében egyre fontosabbá válik az új, hatékony és gazdaságos módszerek kifejlesztése, amelyek képesek a felhasználoktól érkező igények megfelelő kiszolgáltatására.

Felhő rendszerek esetén tipikusan olyan szolgáltatásokról beszélünk, amelyek képesek számítási feladatok elvégzésére, illetve adattárolási szolgáltatásokat biztosítani, mindezt online kapcsolaton keresztül, a résztvevő eszközök között. A különböző felhő szolgáltatási modellekben a felhasználóknak adottak a lehetőségek, hogy elérjenek online erőforrásokat és applikációkat futtassanak távoli szervereken, amelyek általában egy harmadik-fél által vannak biztosítva. Továbbá, a felhő skálázhatóságot és flexibilitást biztosít, de ezzel egyidőben költséghatékony, mivel a legtöbb esetben a felhasználóknak csak azokért az erőforrásokért kell fizetniük, amiket használtak. Ezt nevezik “pay-per-use” szolgáltatásnak [2].

A felhő szolgáltatásokat és funkcióikat kibővítve a kód rendszer a szolgáltatásokat fizikailag közelebb helyezi el a végfelhasználóhoz, így minimalizálva a késleltetést és kiszámíthatóvá téve a válaszidőt. A központi szerverek helyett az adatok feldolgozását kiszervezi az úgynevezett kód csomópontoknak, amelyek közelebb helyezkednek el az IoT eszközökhöz, csökkentve a hálózati forgalmat és távközlési költségeket. Ezeknek a csomópontoknak a számítási kapacitása gyakran eltérőek. A hálózat szélén elhelyezett nagyobb szervertől a kisebb, korlátozott kapacitású, de szintén virtualizációra képes eszköz felveheti egy kód csomópont szerepét [3]. Utóbbi esetben dedikált feldolgozó egységet hozunk létre akár az eszköz virtualizálásával, így nagyobb biztonsággal tudunk kritikus alkalmazásokat futtatni. A feldolgozott adatok nagy mennyiségben csak ideiglenesek és nem szükséges hozzájuk permanens adattárolást biztosítani [4].

A felhő-kód rendszert használó IoT eszközök, amelyek magában foglalja a különböző szenzorokat és aktuátorokat, adatokat gyűjthetnek vagy valamilyen feladatot hajtanak végre a

végfelhasználók környezetében, vagy akár piaci résztvevők ipari létesítményeiben. Az eszközökön futtatott alkalmazások biztosítják a hálózaton keresztül történő kommunikációt, elősegítve a gyors és hatékony alkalmazkodást a megváltozott környezethez. Teszi mindezt közel valós időben, hiszen a kód rendszereknek köszönhetően ezek az eszközök közelebb helyezkednek el a számításra képes egységekhez, ahol az adatok feldolgozása történik. Mivel IoT rendszerek esetén hatalmas mennyiségű adat generálódik, például szenzorok, kamerák vagy más érzékelők által, ezért ezeket az adatokat fel lehet használni valós idejű statisztika készítéshez vagy mesterséges intelligencia által támogatott azonnali döntéshozatalhoz. Az IoT leggyakoribb alkalmazási területei az önvezető autók, az ipari automatizálás, az okosváros vagy egészségügyi megfigyelés [5].

Sok tekintetben a kód rendszerek kifinomultabb biztonsági funkciókkal rendelkeznek, hiszen a szenzitív adatokat a forráshoz közel, a hálózat szélén tudják feldolgozni, bizonyos esetekben titkosítani, mielőtt továbbküldik azokat. Ez fokozza a biztonságot engedélyezetlen hozzáférésekkel szemben vagy adatszivárgások során, így megerősítve az általános biztonságát a rendszernek. A felhő rendszerekkel szemben kockázatot jelenthet az, ha csak egy központi egység (felhő) végzi a munka nagy részét, sebezhetővé téve azt célzott támadásokkal szemben vagy akár egy nagyobb rendszerleállás miatti kiesés is komoly problémát okozhat egy valós idejű IoT alkalmazás működésében. Ezzel ellentétben a kód modell elosztott tulajdonsága miatt nagyobb redundanciát tud biztosítani, így a leállások vagy hibák nem befolyásolják az egész architektúrát.

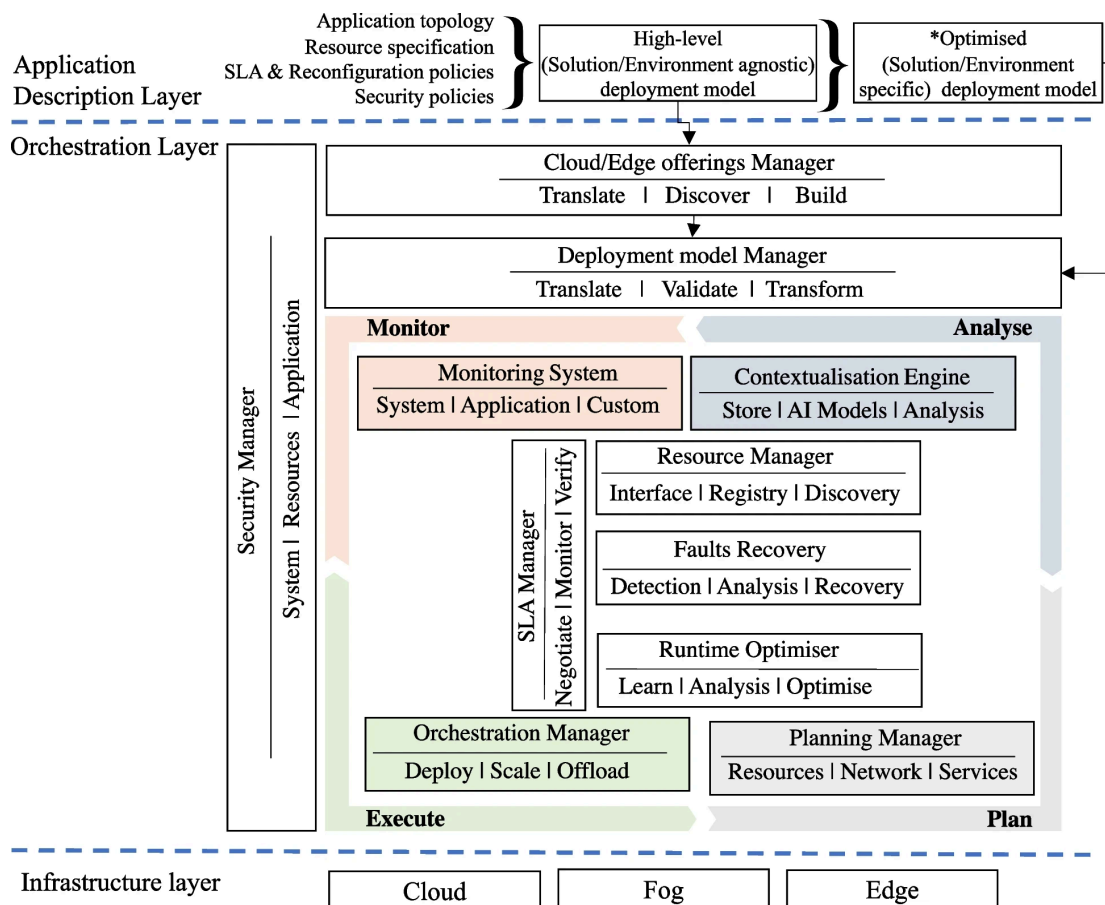
1.1. Elosztott orkesztráció

A dolgozat során az “orkesztrációt” az irányítás, menedzselés és koordináció szinonimájaként használok [6], amelyet az elosztott rendszerben az orkesztrátor csomópontok fognak végrehajtani a lokális csoportokon belül. Az orkesztrátor az architektúrában lévő különböző IoT eszközök, mint például szenzorok és aktuátorok, illetve a számítási erőforrások összehangolt működésének biztosításáért felelős, beleértve így az adatok gyűjtését, feldolgozását és továbbítását. A lokális csoportnak az ilyen eszközökből létrejött klasztereket nevezzük.

Az orkesztrációs megoldásokat három fő kategóriára lehet bontani: statikus, szabály alapú és gépi tanulást alkalmazó megközelítésre [7]. A statikus megközelítés a rendszer mérnökökre helyezi a felelősséget, hogy a tervezés alatt az alkalmazások bizonyos részeit rendeljék a rendszer bizonyos erőforrásaihoz. A szabály alapú megközelítéssel előre

meghatározott, határértékekre épített szabályok segítenek az erőforrások megfelelő kiválasztásában. Az utolsó, a gépi tanulást alkalmazó kategória, ahol a rendszer a begyűjtött adatok alapján hoz informált döntést az erőforrások allokálásáról, menedzseléséről a monitorozott információk alapján.

Elosztott orkesztráció során a számítási környezetben több decentralizált irányító egység van szétszórva, ahol egymástól függetlenül dolgoznak azon, hogy a dedikált applikáció egyes komponenseit menedzselje az orkesztrátor által felügyelt infrastruktúrán és összedolgozzanak, hogy biztosítani tudják a szolgáltatás minőségét (QoS), valamint megfeleljenek a szolgáltató megállapodásnak (SLA) és egyéb szabályoknak [8]. Az orkesztrációs réteg legfontosabb komponenseit az 1.1.1. ábra mutatja be. Ez a dolgozat elsősorban az ábrán szereplő Orchestration Manager által definiált részfeladatok megvalósítását célozza meg.



1.1.1. ábra. Orkesztrációs réteg komponensei [6]

Az orkesztrátor irányító egységeknek képeseknek kell lenniük azonosítani és elemezni a beérkező feladatokat, hogy ezen információk alapján képesek legyenek informált

döntéshozatalra: például, hogy melyik számítási csomópontra allokáálja az egyes feladatokat. Tipikusan ezt egy úgynevezett deployment algoritmus végzi el. Hogy az adott csomóponton belül található (tipikusan fizikai gépeken futtatott virtuális) erőforrások milyen sorrendben dolgozzák fel az egyes feladatokat, azt egy task scheduling, azaz a feladatütemezési algoritmus definiálja. Ezeknél a döntéseknél az orkesztrátornak ismernie kell a vele kapcsolatban álló erőforrásokat, hogy azokat optimálisan tudja kihasználni, valamint hogy egyenletesen terhelje az egész architektúrát. Fontos szerepe van továbbá az ütemezésnek abban is, hogy a válaszidőt minimalizálja. Ennek az eléréséhez akár dinamikusan is tud alkalmazkodni a rendszer, például a már egyszer beütemezett feladatok átütemezésével egy másik, kevésbé leterhelt virtuális erőforrásra.

A válaszidő minimalizálásánál és az erőforrás terheltségének csökkentése érdekében a rendszer kihasználhatja azt, hogy a számításokat az eredetileg allokált csomópont helyett egy másik kód csomópontokon végezteti el. Ezt a módszert hívják offloading-nak, azaz tehermentesítésnek. Ilyenkor több tényezőt is figyelembe kell venni, mint például a rendelkezésre álló sáv szélességet és a hálózati késleltetést, amely befolyásolhatja a szolgáltatás sebességét, hiszen nagy mennyiségű adat átmozgatása a hálózaton közel ugyanannyi időbe telhet, mint megvárni azt, hogy az eredeti helyén elinduljon a feldolgozása.

Ugyanakkor a számítógépség képességei is kritikus paraméterek, nagy erőforrást igénylő probléma esetén nem érdemes egy kisebb, távoli eszközre átmozgatni, ahol nem áll rendelkezésre a megfelelő mennyiségű memória és számítási kapacitás, vagy a feladat elvégzéséhez szükséges további függőségek. Mindennek a velejárója, hogy a nem megfelelően kiválasztott egységek több energiát is használhatnak ugyanannak a feladatnak az elvégzése során, mint a feladat szükségleteihez jobban illeszkedő eszköz. További szempont lehet, ha egy bizonyos területről (geolokációból) gyakran érkezik ugyanolyan számítási feladat, így a rendszer előre felkészülhet a feladatok érkezésére, biztosítva a még gyorsabb kiszolgálást. Mindezen tényezők figyelembevételével lesz képes a rendszer biztosítani a megfelelő QoS-t, valamint képes lesz megfelelnie az SLA-nak is.

A rendelkezésre álló erőforrások kihasználásának maximalizálására egy módszer, ha egy feldolgozási egységen - például egy fizikai gépen - egy feladat elvégzése nem használja ki az összes erőforrást, akkor újabb feladatoknak is helyet biztosítunk ugyanazon a egységen. Ilyenkor alkalmazhatunk horizontális skálázást, amely a fizikai erőforrást virtualizáció segítségével felosztja, és a fizikai erőforrás jobb kihasználtsága érdekében újabb virtuális gépeket inicializálunk. Az elosztott rendszerek képesek a valós idejű adatok alapján egy fizikai egység virtuális kapacitását növelni, amennyiben arra igény van, de képesek

csökkenteni is, amennyiben az elvégzendő feladatok száma csökken. Ezeknek a döntéseknek a meghozatalához lehetnek előre meghatározott határok, amelyek átlépése esetén megfelelő skálázási akciót végez a rendszer. Különböző gépi tanuláson alapuló algoritmusok képesek előrejelezheti a várható terheltséget, így a rendszer felkészülhet, ha az nagyobb igényekre számít bizonyos időpontokban. Ezekkel a tulajdonságokkal tudja biztosítani az orkesztráció azt, hogy minden erőforrás optimálisan legyen kihasználva és a lehető legköltséghatékonyabb működést érje el a szolgáltatás folyamán.

Minden eddig említett folyamat alapjaként szolgál a monitoring funkció, amely folyamatosan adatokat gyűjt a rendszerek működéséről. Ezáltal képes állandóan nyomon követni annak állapotát és teljesítményét. Utóbbinak méréséhez megállapíthatóak kulcsfontosságú teljesítménymutatók, amik lehetnek például a CPU teljesítménymutatói, a memória kihasználtságát jelző számok, a hálózati leterheltséget figyelő mutatók vagy két eszköz közötti válaszidő. Az adatok felhasználásával reagálni képes az orkesztrátor a terhelési változásokra, észlelni és azonosítani tudja a hibákat így segítve azok elhárítását, valamint fontos információkkal szolgál a rendszer további optimalizálásához.

Új orkesztrációs megoldások kutatása szerteágazó, a felhasználási céloktól függően mind centralizált és mind decentralizált megvalósítások is születtek [9].

A dolgozat témájához kapcsolódó kutatások közül elérhetőek még csak koncepcióként bemutatott munkák [10][11][12] vagy már megvalósított kutatási [13][14] és vállalati fejlesztések [15]. Ezekben a munkákban az a közös, hogy mind valamilyen decentralizált architektúrát használ. Az ENORM [10] megoldás az erőforrások optimalizálására fókuszál egy decentralizált, de statikus megoldással. A HYDRA [11] és a Caravela [12] egy peer-to-peer (P2P) rendszert használnak a decentralizálás megoldására és minden eszköz egyszerre számítási egység és orkesztrátor is egyben. A különbség közöttük, hogy az utóbbi piaci alapra épít, ahol külső erőforrások is beléphetnek a hálózatba és ahol a munkájukért kompenzálják őket. Alkalmazásra kész rendszerek között az mF2C rendszer [13] áll a legközelebb a munkámhoz olyan szempontból, hogy előtérbe helyezi a számítási csomópontok elhelyezkedését. A Rainbow [14] rendszer egy absztrakt megoldást használ, ahol a topológiát és egyéb rendszerszintű adatokat egy gráf formájában kapja meg és az alapján optimalizálja és irányítja a rendszert. Utolsóként, az ORMUCO [15] piaci alkalmazás fő célja a növekvő decentralizált igények kiszolgálása vállalatok számára, melyek csak Infrastructur as a Service (IaaS) megoldást keresnek a könnyebb fenntarthatóság és költségek csökkentése érdekében.

1.2. Problémafelvetés

Az elosztott architektúrára építő rendszerek fontos tulajdonsága, hogy a különböző csomópontok vagy csoportok önálló működésre és döntéshozatalra is képesek a gyors válaszidő és zavartalan működés biztosítása érdekében. Az elosztott felépítésnek viszont része, hogy lokálisan továbbra is centralizáltak a csomópontok. Ezekenek a csoportoknak az összeállítása nem triviális, hiszen nincsen egy központi szerver, ami ismeri a rendszerben jelen lévő összes eszköz helyzetét és állapotát.

A csoportok kialakításának több szempontja is lehet, például a csomópontok fizikai elhelyezkedése, leterheltségük vagy az egymás közötti hálózati késleltetés. Ezen információkat minden csomópont tárolja magáról, viszont a rendszer elosztott tulajdonsága miatt nincs egy központi döntéshozó, ami összegezni tudna minden adatot, amely alapján kialakításra kerülnek a csoportok. Ezért szükség van egy olyan megoldásra, ahol a csomópontok kommunikálnak egymással a hálózaton keresztül és konszenzus alapján döntenek új csoportok létrehozásáról vagy már meglévőkhöz való csatlakozásról. Valamint további előnyt jelenthet, ha rendszer képes dinamikusan reagálni az architektúra változására, mint például csomópontok kiesésére és új eszközök csatlakozására. Ekkor indokolt lehet a csoportok teljesen újrendezése vagy csak az új eszköz egy már létező csoporthoz való hozzárendelése.

2. fejezet

DISSECT-CF-Fog

A számítási technológiák fejlődésével a felhasználói igények is növekedtek, megváltoztak, amelyek így megkövetelik az újszerű vezérlő algoritmusok implementálását. A megváltozott igények magával hozták a fenntartási költségek növekedését is. Az új, még le nem tesztelt módszerek és algoritmusok valós környezetben való kipróbálása időigényes, sok esetben nem reprodukálható és nem gazdaságos. Ez igaz a kutatási célú felhasználásra is, egyedi vagy speciális rendszereket beüzemelési és fenntartási költsége igen magas.

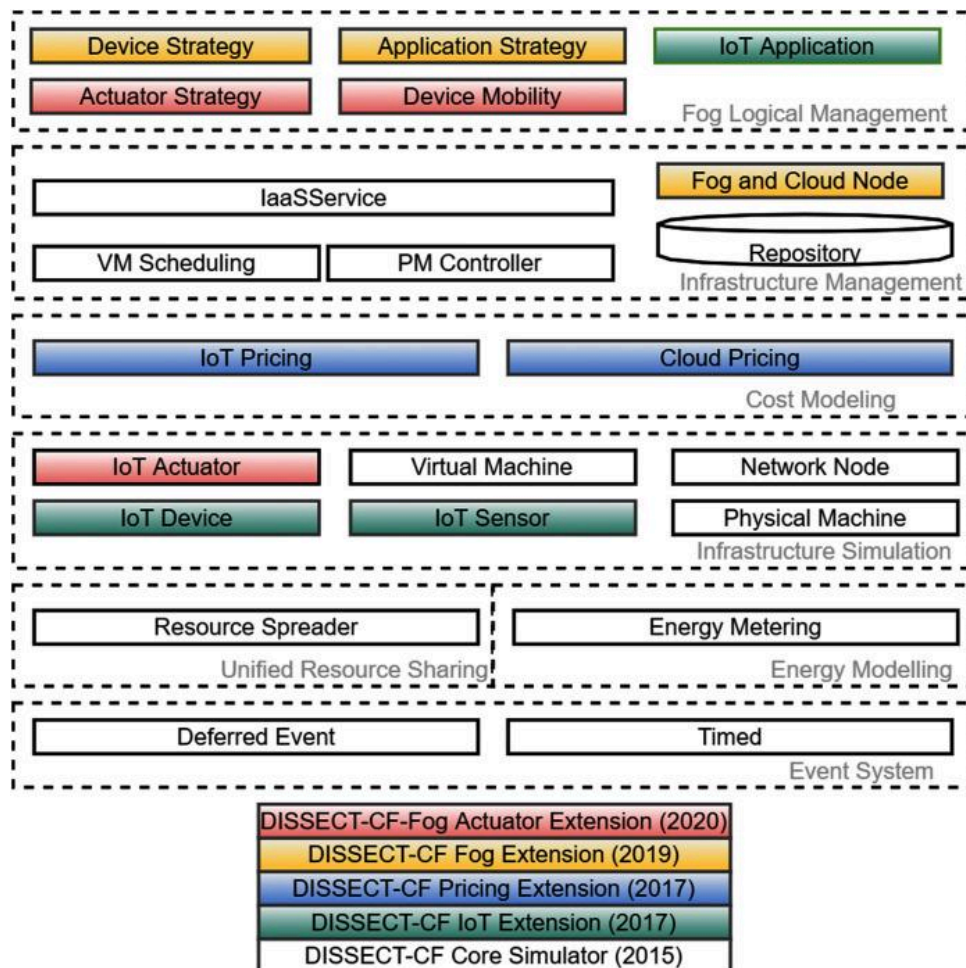
Ennek eredményeképpen a kutatási körökben bevált megoldás a különböző szimulációs eszközök használata, melyek egy költséghatékony módszert tudnak biztosítani új elképzelések tesztelésére. Ezek az eszközök közel realisztikusan képesek modellezni a valós világbeli entitások működését.

A dolgozat keretében a DISSECT-CF-Fog [\[16\]](#) szimulátort használtam, amely egy nyílt forráskódú, eseményvezérelt alkalmazás. A szimulátor alapja a DISSECT-CF [\[17\]](#), amely felhő infrastruktúrák belső elemzésére lett kifejlesztve. Más szimulátorokhoz képest az alkalmazás nagy előnye, hogy tartalmaz egy nagyon részletes IaaS megvalósítást mind fizikai mind virtualizált eszközök modellezésére, valamint modellezi a tárhely- és hálózatkezelés tulajdonságait is.

A DISSECT-CF-Fog ezeket a funkciókat bővíti tovább: egy teljesen új réteget implementál a felhő és az eszközzéteg közé, bevezetve a többretegű IoT-Kód-Felhő rendszerek modellezését. A kód és felhő csomópontok modellezik az IoT alkalmazások viselkedését virtualizált erőforrások segítségével, amelyek a számítások elvégzéséért felelősek. A kibővített szimulátorban megjelenik egy részletes IoT réteg reprezentáció is, ami tartalmazza az aktuátorokat és szenzorokat, továbbá okoseszközök (példuál mobiltelefon) modellezésének lehetőségét is.

A szimulátor IoT munkafolyamat (workflow) szimulációjának vezérlése ezidáig csak egy központosított, magasabb absztrakciós szinten jelenlévő orkesztrátorral működött. A központi felhő szervert és a kód egységeket, illetve a közöttük lévő kapcsolatokat és az általuk alkotott klasztereket előre kellett definiálni a szimulációknál, ami után a rendszer

képes volt például IoT workflow szimulációk futtatására. Ezek a szimulációk alkalmasak valós IoT rendszerek modellezésére: leírhatunk szimulált szenzorok viselkedését, amelyek valamilyen eseményt érzékelnek (pl. mozgásérzékelő szenzorok), majd azokra reagálva valami hatást váltanak ki, például aktuátorok segítségével fel vagy lekapcsolják a lámpákat. Az eddigi fejlesztési szakaszokat, a szimulátor főbb komponenseit a 2.1. ábra mutatja be.



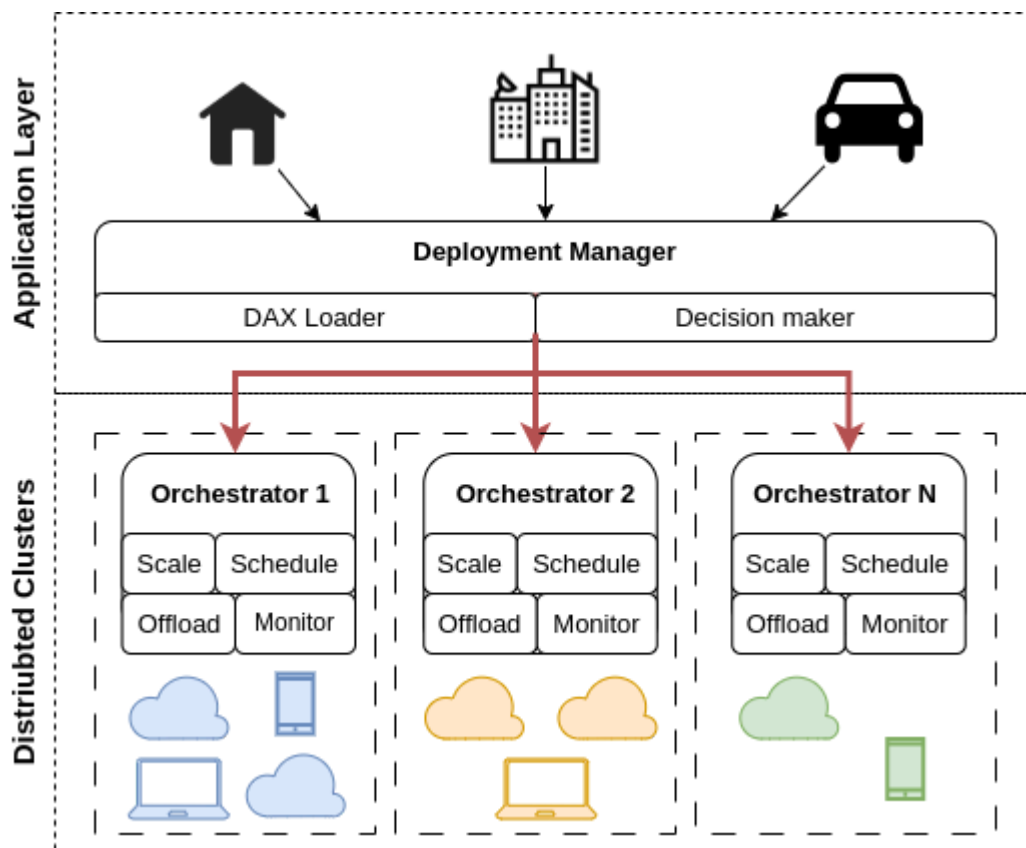
2.1. ábra. A DISSECT-CF-Fog szimulátor legfontosabb komponensei [18]

A szimulátor a feladatok betöltéséhez más workflow szimulátorokhoz hasonlóan a DAX formátumot használja, hiszen ez jól ismert és rendszeresen használt leírás. Ennek ellenére az eredeti DAX formátum valódi workflow modellek futtatására lett kitalálva, ezért szimulációs felhasználás során sok olyan információt is tartalmaz, ami a rendszer számára irreleváns. Például a függőségek betöltése során a szimulátor csak modellezi a fájlok betöltését és nem a konkrét fájlokkal dolgozik. Mivel ez a formátum számítógépek számára lett kitalálva, nehéz lehet a szimulátort használók számára értelmezni és a plusz információk nehezíthetik az olvashatóságot is. Ennek a problémának a megoldására hozták létre a DISSECT-CF-Fog-hoz

egy módosított DAX leírást, amely továbbra is DAX alapokra épít, de egyszerűsíti az olvashatóságot és új, IoT specifikus munkafolyamatok definiálását.

A dolgozat keretében a DISSECT-CF-Fog szimulátor kibővítésével megvalósításra kerül egy dedikált orkesztrációs réteg, amely négy, az előző fejezetben bemutatott feladatokra összpontosít: scheduling, offloading, scaling és monitoring.

A szimulátor új, elosztott felépítésben a feladat, hogy a kód csomópontokat valamilyen logika alapján csoportokra, klaszterekre bontsuk. Minden csoportot a hozzá tartozó orkesztrátor fogja irányítani. A dolgozat keretében a csoportokat egy dinamikus hangya-kolónia klaszterezésre alapuló algoritmus alakítja ki. Az orkesztrátor egységek tárolják a klaszterezés eredményét, azaz hogy mely csomópontok tartoznak az adott csoporthoz. Továbbá az elosztott orkesztrátorok adnak helyet és futtatják a scheduling, offloading, scaling és monitoring algoritmusokat. Az új rendszer felépítését és rétegeit a 2.2. ábra mutatja be.



2.2. ábra. Az új decentralizált felépítés

A szimuláció futtatása előtt az IoT alkalmazások működését leíró feladatokat DAX fájlokban átadjuk egy Deployment Manager-nek. Ez egy absztrakt entitás, ami egy stratégia alapján kiosztja a beérkező feladatokat a klaszterek között. A feladatok kiosztása a klaszterek között jelenleg random algoritmus alapján történik, de a létrehozott interfésszel a későbbiekben lehetőség lesz bármilyen deployment algoritmust implementálni, amely figyelembe veheti a klaszterek egyéb metrikáit is hogy informált döntéshozatalra legyen képes. Ezek a metrikák lehetnek például a klaszterek közötti hálózati késleltetések, amelyek lassíthatnak a számítási folyamatokon vagy a futtatandó alkalmazás jellemzője, ami alapján egy specifikus klaszternek lehet küldeni a feladatot.

A konkrét megvalósításban a DAX-ból beolvasott, elvégzendő feladatok listáját a Scheduler osztály példányai tárolják, amely az adott klaszterbe tartozó csomópontok között egyenletesen osztja szét a beérkező feladatok függőségeinek mennyisége alapján. Mindazonáltal itt is rendelkezésre áll egy interfész, hogy a későbbiekben bármely más tulajdonság alapján is képes legyen ütemezni a feladatokat.

A szimuláció során a terheltség folyamatos figyelembevételével a rendszer képes feladatokat átküldeni egyik csomóponttól a másikra, valamint scaling algoritmus alapján folyamatosan alkalmazkodik minden csomópont a beérkező terheléshez. Ehhez a feladathoz egy harmadik interfész is létre lett hozva ami kód eszközökön futó virtuális gépek között biztosít egyenletes eloszlást. Itt a virtuális gépek is rendelkeznek egy listával ami az elvégzendő feladatokat tárolja addig amíg azokat nem végzi. Ez alapján a lista alapján rendezzük az összes virtuális gépet és mindig a legkevesebb feladattal rendelkezőre ütemezünk új feladatot. Ezzel biztosítva hogy minél egyenletesebben legyenek a feladatok szétosztva és hogy ne legyen olyan virtuális gép amely nem végez számításokat.

3. fejezet

Klaszterező algoritmusok

Adatelemzésben, térinformatikában vagy adatbányászatban nagyon elterjedt módszer a klaszterezés, amely során az azonos jellemzőkkel rendelkező elemeket ugyanabba a csoportba vagy klaszterbe helyezünk. A klaszterezési problémák megoldására egy másik megközelítés a tipikus klaszterező algoritmusokkal szemben, hogy az élővilágból megfigyelt rajintelligencia [\[19\]](#) alapján hozzanak létre algoritmusokat, amelyek hasonlóan viselkednek, mint az adott élőlények csoportja a való világban. Ennek az egyik legszemléletesebb példája a hangyakolónia algoritmus, amely a hangyák társadalmi viselkedését modellezve keresi a legjobb megoldást. Az algoritmus a végrehajtás során modellezi, ahogyan a hangyák feromonokat hagynak maguk után, amikor élelmet keresnek. A megtalált élelemtől ugyanezeket a feromonokat használják, hogy visszataláljanak a kolóniához, valamint a többi hangya is ezeket a feromonokat követik. Ennek a viselkedésnek köszönhetően, ahogyan a feromonok párolognak és ahogy új hangyák új feromonokat szórnak le, mindig a hangyák által optimálisnak gondolt úton lesz a legerősebb a feromon szint - de ez nem feltétlen jelenti az optimális útvonalat.

3.1. Centralizált ACO algoritmus

Az első, felhő és kód csomópontokat csoportosító és centralizált döntésen alapuló hangyakolónia algoritmus alapjául a [\[20\]](#) tanulmányban részletezett eljárás szolgál, amely a későbbiekben bemutatott elosztott módon működő hangyakolónia algoritmussal történő összehasonlításban is részt fog venni. A főbb különbség a publikációban tárgyalt eljárás és a dolgozat keretében implementált megvalósítás között, hogy a feromonmátrixot random értékekkel inicializáljuk. A futtatott tesztek során ezzel a módszerrel jobb eredményeket értünk el. Továbbá különbség még az algoritmusban használt heurisztika is.

A centralizált megvalósítás során feltételezzük, hogy a kiindulási architektúrát ismeri egy, a kód eszközkönnél hierarchikusan feljebb lévő központi erőforrás (például egy felhő központ), amely csoportokba tudja osztani a különböző számítási erőforrásokat, hogy azok utána elosztott rendszerként tudjanak tovább működni.

Egy hangyakolóniára épülő algoritmus implementálásának két hangsúlyosabb része van. Az első a hangyák reprezentációja, mivel ezek az entitások fognak a felvetett problémára - jelen esetben a klaszterezési problémára - iterálva megoldást keresni. A második pedig egy feromonmátrix létrehozása - ebben a megvalósításban random értékekkel inicializálva -, amelyet a hangyák iteratív módon, a megoldást keresve módosítanak. A mátrixban lévő értékek kétféleképpen fognak változni: a hangyák "mozgása" növeli mátrix megadott pozíciójában lévő értékeket, illetve minden iteráció végén a mátrixban lévő összes értéket csökkentjük egy bizonyos értékkel, modellezve ezzel a feromonok párolgását.

A centralizált algoritmus indítása előtt több paramétert kell átadnunk. Ez az algoritmus egy előre megadott klaszterszám alapján keres lehetséges megoldást. Az algoritmus esetében a heurisztika - amely alapján az eljárást eldönti, hogy a hangyák által talált megoldás mennyire jó - a klaszterközpont és az adott csomópont közötti távolság lesz. Ezért a klaszterek száma mellett a klaszterek központjaként viselkedő számítási csomópontokat is meg kell jelölni. A későbbiekben ezek a csomópontok fogják az orkesztrációhoz tartozó feladatokat ellátni.

Továbbá szükséges megadni azt is, hogy mennyi hangyát szeretnénk példányosítani, amelyek külön-külön megoldást keresnek, valamint a maximális iterációk számát is rögzíteni kell. Ez az utóbbi két paraméter van a legnagyobb hatással az algoritmus pontosságára és futás idejére, hiszen ha nem végzünk elég sok iterációt vagy kevés hangyát példányosítunk, akkor előfordulhat, hogy a hangyák egy lokális optimumban ragadnak. A másik véglet, hogy az algoritmus eredménye az újabb és újabb iterációk során már nem változik, csupán a futási idő növekszik.

Ezek után meg kell adni azt is, hogy a feromon párolgás milyen mértékben történjen, ez iteránciónként fogja bizonyos százalékos aránnyal csökkenteni a feromonmátrix összes elemét. Szükség van még egy minimális válaszidő megadására is, amelyet az egy csoportba kerülő csomópontok között, hálózati késleltetésként fog alkalmazni a rendszer, valamint egy randomFactor megnevezésű értékre is szükségünk lesz, amely az algoritmus döntéshozatalában fog fontos szerepet kapni.

A feromonmátrix mérete $n \times m$ ahol n a klaszterek száma m pedig a csomópontok száma. A mátrix (x,y) pozíciója azt reprezentálja, hogy milyen valószínűséggel tartozik x eszköz az y klaszterbe. Ennek a mátrixnak az inicializálása után indul az iteratív lépéssorozat, ahol mindegyik iteráció elején új hangyák kerülnek példányosításra három adattaggal. Minden hangya két $1 \times m$ méretű mátrixot is használ az adott lépésben. Az egyik a

solutionMatrix, amely az adott iterációban talált megoldását fogja tárolni, a másik randomMatrix, ami egy (0,1) intervallumba lévő random számot fog tárolni minden csomóponthoz.

2	1	3	2	2	3	2	1
---	---	---	---	---	---	---	---

3.1.1. táblázat. Példa egy hangya solutionMatrix adattagjára

Harmadik adattag a fitnessLevel, ami azt reprezentálja hogy a hangya által talált megoldás mennyire pontos. A számítás során a hangya megoldás mátrixon iterálunk, ahol minden értékre kiszámítjuk a heurisztikát: ez az aktuális csomópont és a megoldás mátrixban kiválasztott klaszter orkesztrátor csomópontja közötti távolság. A fitnessLevel értékét ezeknek a heurisztikáknak az összeadásával kapjuk.

$$fitnessLevel = \sum_{i=1}^{|solution|} h(node(i), centerNodes(ant. solution(i)))$$

(3.1.1.)

A folyamatok jobb megértéséhez létrehozok egy példa iterációt, amelyben 8 eszközt 3 darab klaszterbe akarunk osztani 10 hangya használatával. Ennek az iterációnak egy megoldás mátrixát a 3.1.1. táblázat, a hozzá tartozó példa feromonmátrixot a 3.1.2. táblázat mutatja be.

N	K		
	1	2	2
1	0.014756	0.015274	0.009900
2	0.015274	0.009900	0.014756
3	0.015274	0.014756	0.009900
4	0.009900	0.015274	0.014756
5	0.014756	0.015274	0.009900
6	0.009900	0.014756	0.015274
7	0.009900	0.020131	0.009900
8	0.015274	0.014756	0.009900

3.1.2. táblázat. Példa egy feromonmátrixra

0.693241	0.791452	0.986142	0.988432	0.243672	0.096772	0.091432	0.348767
----------	----------	----------	----------	----------	----------	----------	----------

3.1.3. táblázat. Példa a randomMatrix-ra

A megoldás megtalálásához két módszer közül az egyiket fogja alkalmazni minden hangya. Az algoritmus konstruktorában megadott randomFactor - tegyük fel hogy ez a szám most 0.8 - és az algoritmus által generált randomMatrix (lásd 3.1.1. táblázat) minden értékére külön-külön megnézi, hogy az előbbi nagyobb-e az utóbbinál. Ha igen, akkor a feromonmátrixban tárolt értékek közül a legmagasabb kerül kiválasztásra (ez jelzi, hogy az adott csomópont milyen valószínűséggel tartozik az adott csomópontához). Jelen esetünkben az 1, 2, 5, 6, 7 és 8 eszközök csoportját így fogja kiválasztani az algoritmus. Ha viszont nem nagyobb, akkor a feromonmátrixban az ehhez a csomópontához tartozó értékek normalizálásával döntjük el klasztert. Ezt úgy végezzük el, hogy először kiszámoljuk az adott csomópontához tartozó feromonértékek összegét:

$$sum = \sum_{i=1}^{numberOfClusters} pheromonMatrix[i][j] \quad (3.1.2.)$$

Ezután kiszámoljuk milyen valószínűséggel fogja a klasztereket az algoritmus választani és ez alapján meghatározzuk a választott klaszter sorszámát:

$$index = arg \max_j \left\{ \frac{pheromonMatrix[i][j]}{sum} > randomNum \right\} \quad (3.1.3.)$$

N	K		
	1	2	2
1	0.3695	0.3826	0.2479
2	0.3825	0.2479	0.3695
3	0.3825	0.3696	0.2479
4	0.2479	0.3825	0.3695
5	0.3695	0.3825	0.2479
6	0.2479	0.3695	0.3825
7	0.2479	0.5041	0.2479
8	0.3825	0.3695	0.2479

3.1.4. táblázat. Normalizált eloszlás

Ebben a példában megfigyelhetjük a harmadik csomópont értékei alapján, hogy 0.3825 eséllyel kerül az első klaszterbe, 0.3696-al a másodikba és 0.2479-el a harmadikba. Ez az algoritmus végrehajtása során azt jelenti, hogy ha a random generált szám kisebb, mint 0.3695 akkor az elsőt, ha 0.3695 és 0,7521 közé esik akkor a másodikat, ha pedig nagyobb mint 0.7521, akkor az utolsó megoldást (azaz klasztert) választja ki az algoritmus a csomópont számára.

Az első módszert exploitation-nek a másodikat pedig biased exploration-nek nevezik. Ezek a folyamatok minden hangya esetén végrehajtásra kerülnek, azaz minden hangya döntést hoz minden csomópontról. Ezeket a klaszter sorszámokat egy m hosszú egész értékű tömbben tárolja, amely a hangya által talált megoldást reprezentálja. Ezt követően minden iteráció végén az algoritmus kiszámítja minden egyes hangya megoldásának a fitnessLevel értékét a 3.1.1. képlet segítségével, majd ez alapján növekvő sorba rendezzük a hangyákat a heurisztika értéke alapján.

S	N								fitnessLevel
	1	2	3	4	5	6	7	8	
S_1	1	1	1	2	2	3	2	1	1.816471
S_2	2	1	1	2	2	3	2	1	1.842034
S_3	1	1	2	2	2	3	1	1	1.877386
S_4	2	1	1	2	1	3	2	1	1.900668
S_5	2	2	1	2	2	3	2	1	1.982272
S_6	2	1	1	2	3	3	2	1	2.140193
S_7	2	3	1	2	2	3	2	3	2.408086
S_8	1	1	2	2	2	3	3	1	2.474522
S_9	2	1	1	2	2	3	3	1	2.534078
S_{10}	2	1	3	2	2	3	2	1	2.695110

3.1.5. táblázat. Példa a fitnessLevel alapján rendezett megoldásokra

A 3.1.5. táblázaton 10 hangya eredmény tömbje látszik S_x jelöléssel (x a hangyák számát jelöli). Annak érdekében, hogy minél jobb eredményt érjünk el, nem az összes hangya számítási eredményeit veszi figyelembe az algoritmus a feromonmátrix frissítése során, csupán az összes hangya számának bizonyos százalékát. 20%-os arány és a 3.1.5. táblában lévő 10 hangya esetén a legjobb 2 megoldással frissül a feromonmátrix.

3.2. Elosztott ACO algoritmus

Az elosztott algoritmus paramétereinek száma jóval kevesebb, mint a centralizált klaszterező algoritmus esetén, mivel itt több mindent fog az algoritmus automatikusan kiszámítani. A megmaradt szükséges paraméterek megegyeznek a centralizált algoritmus megfelelő paramétereivel. Ezek a paraméterek tehát a maximális iterációk száma, a randomFactor, továbbá meg kell adni a feromonok párolgásának arányát és egy alapértelmezett válaszdőt az egy klaszterbe tartozó csomópontoknak. Az elosztott felépítés miatt itt nincs lehetőség központi feromonmátrix létrehozására, így ebben az algoritmusban minden csomóponthoz saját hangyát hozunk létre és minden hangyának lesz egy saját $1 \times m$ méretű feromonmátrixa, amiben azt fogja tárolni, hogy milyen valószínűséggel alkotna klasztert az x -edik csomóponttal. Így az algoritmus futása során minden hangya, azaz minden csomópont is egy saját hangyakolónián alapuló klaszterező algoritmust futtat.

3.2.1. Feromonmátrix értékeinek meghatározása

Az iterációk során itt is két módszer alapján választ az algoritmus, hasonlóan a centralizált megoldáshoz. A módszerek közötti választás itt is a randomFactor alapján történik, azaz ha a generált random szám kisebb, mint a randomFactor, akkor az algoritmus úgy változtat a feromonmátrix értékén, hogy az evaporáció után visszatérjen az előző iterációban lévő értékéhez, ezzel gyakorlatilag semlegesítve a feromonok párolgását erre az iterációra. Mivel az evaporáció mértékét még a konstruktorban meg kell adni, ezért ez egy számítással megkapható:

$$update = \frac{(1 - evaporationRate) \times pheromoneMatrix[i]}{evaporationRate} \quad (3.2.1.1.)$$

Amennyiben nagyobb lesz a random érték, akkor a feromonmátrixot a heurusztika értékével növeli. Az algoritmus ekkor kiszámolja a két eszköz közötti távolságot kilométerben, majd Gauss kernel segítségével leképezi egy kisebb számra.

$$\begin{aligned} update &= \frac{1}{\sqrt{2\pi}} \times e^{-0.5 \times u^2} \\ u &= \frac{x}{h} \end{aligned} \quad (3.2.1.2.)$$

Itt a h a szórás értékét jelöli. Ez egy változtatható érték a görbe simítására, de a megvalósításban egy előre rögzített $h = 2$ értéket használok, amely egy kielégítő simítást ad arányosan a feromon párolgásához képest.

3.2.2. Mátrixok egyesítése

Miután lefutott az összes iteráció és a hangyák memóriájában létrejöttek a kész feromonmátrixok, az algoritmusnak meg kell állapítani, hogy melyik eszköz melyik klaszterbe fog kerülni. Ehhez viszont előbb létre kell hoznunk a hangyáknak egy egyesített mátrixot. Ez azt fogja jelenteni, hogy minden hangyának lesz egy a feromonmátrixal megegyező méretű mátrixa is. Ebben az x hangya saját feromonmátrixának az y . csomópontához tartozó értékének és az y . csomópont hangyájának feromonmátrixában az x . csomópontához tartozó érték átlagát fogja tárolni.

Így minden hangya minden eszközpárra létrehoz egy számot az egyesítés során az összetartozó értékek átlagolásával, hogy pontosítani tudjuk az algoritmus eredményét, mivel két független létrejött eredményt is figyelembe veszünk. Ezt azt is jelenti, hogy minden hangyának az összetartozó értékeik meg fognak egyezni, tehát az x . hangya y . értéke meg fog egyezni az y . hangya x . értékével. Ezek után a hangyák kiválasztják a számukra legideálisabb csomópontot - jelen esetünkben ez a legközelebbi csomópont - és kiemelik annak sorszámát és értékét.

3.2.3. Klaszter csoportok létrehozása

Abban az esetben, ha megvannak az összesített mátrixok és a hangyák kiválasztották az általuk legideálisabbnak ítélt eszközt, elkezdődhet a klaszterek létrehozása. Az algoritmus ezután végig megy minden hangyán és kiválasztja a legmagasabb párosítási értéket. Mivel minden értékhez 2 eszköz tartozik, így ezeket az eszközöket egyesíti egy klaszterbe az algoritmus. Ilyenkor nem csak a klasztersorszámuk fog megegyezni az eszközöknek, de a hangyák feromonmátrixait is egyesítjük, ugyanazzal az átlagolási technikával, mint amivel

létrehoztuk a klaszterezés előtti egyesített mátrixokat. A folyamat során az egymáshoz tartozó értékeket nullára állítjuk, hiszen ezt a két csomópontot már elhelyeztük egy klaszterben. Ezzel a két lépéssel tudjuk garantálni hogy az egy klaszterbe tartozó eszközök egyező értékekkel keressenek tovább, valamint a legmagasabb párosítási érték frissítése után ne jelenjen meg ugyanaz a pár kétszer. Ezek a műveletek fognak megismétlődni minden iterációban az algoritmus futása során addig, amíg minden egyes eszköz be nem került egy klaszterbe.

Annak érdekében, hogy elkerüljük, hogy egyetlen klaszter jöjjön létre, a második iteráció során továbbra is két eddig be nem osztott eszközt fogunk keresni, ezzel garantálva minimum két csoport létrejöttét. A harmadik iteráció után viszont nincs ilyen feltétel, bármely szabad eszköz csatlakozhat bármelyik eddig létrejött klaszterhez vagy amennyiben két eddig be nem osztott csomópontnak lesz a legkiemelkedőbb párosítási értéke, létrejöhet egy új, eddig definiálatlan klaszter is.

Így tehát a centralizált klaszterező algoritmussal összehasonlítva - ahol előre rögzíteni kell a csoportok számát - ebben az esetben a végrehajtás során dinamikusan alakul ki legalább kettő klaszter. Továbbá fontos megemlíteni egy speciális esetet: a rendszer fel van arra is készítve, hogyha csak három csomópontot akarunk klaszterezni, hiszen ez egy olyan eset amit alaphoz az algoritmus nem tudna kezelni. Ebben az esetben az algoritmus 1 iterációt hajt végre, biztosítva, hogy pontosan kettő klaszter jöjjön létre.

3.2.4. Szimuláció előkészítése

Miután lezajlott a klaszterezési folyamat, az eredményeket a szimulátor számára értelmezhető formátumba kell konvertálni, hiszen a megoldás jelenleg még csak a hangyák memóriájában van tárolva. A szimulátor Scheduler osztálya az, amelynek meg kell kapnia ezeket a klasztereket. A szimuláció elosztott módon működik, így klaszterenként kerül a Scheduler osztály példányosítására, hiszen ez az osztály felelős az orkesztrációért (jelenleg ez az absztrakt Orchestration Manager explicit megvalósítása). Ezzel együtt minden Scheduler-hez létrehozunk a klaszterezés eredményeiből egy architektúra reprezentációt, ami csak az adott csoportba tartozó csomópontokat fogja tárolni, így nem lesz konfliktus és a csoportok egymástól függetlenül tudnak működni.

A folyamat három lépésből áll. Először létrehozuk az architektúrákat, amiben tárolni fogjuk az új klaszterek eredményeit. Ezután minden hangyán végigiterálunk, hogy meghatározzuk melyik csomópont melyik klaszterbe tartozik és behelyezzük őket a megfelelő tárolóba. Utolsó lépésként, miután elhelyeztük az összes csomópontot, még be kell állítani a

szomszéd-kapcsolatokat, hogy az egy klaszterbe tartozó eszközök tudjanak egymással kommunikálni. A klasztereken belül a szimuláció során nincsenek hierachikusan megkülönböztetve az eszközök, ezért minden eszköznek ismernie kell az összes szomszédját akivel egy klaszterbe tartozik. Gyakorlatilag a klaszterbe tartozó csomópontokat teljes gráfként reprezentáljuk (minden csúcsa össze van kötve minden más csúccsal duplikált és hurok élek nélkül). Az így elkészült architektúrákat már csak egyszerűen át kell adni a szimulátor workflow végrehajtó osztályának ami befejezi a szimuláció előkészítését és elindítja a scénáriót.

4. fejezet

Kiértékelés

A kifejlesztett orkesztrátor logikát és klaszterező algoritmusokat kettő darab scenárióval fogom tesztelni. Mind a centralizált és mind az elosztott klaszterező algoritmust tesztelni fogom ezeken a scenáriókon és mind a két klaszterező megoldás ugyanazokat az orkesztrációs algoritmusokat fogja használni. Továbbá, az eredmények szemléltetéséhez létrehoztam egy Visualiser osztályt is, ami egy Python script segítségével a számítási csomópontok geolokációja alapján legenerál egy térképet különböző színnel jelölve a különböző klasztereket, azaz a hangyakolónia algoritmusok eredményét.

A szimulációk futtatásához létre kell hozni egy szimulációs osztályt, ami a program belépési pontja lesz. Ennek az osztálynak a futtatásával lehet majd a szimulációt megkezdeni. Ebben az osztályban kell betáplálni a kívánt kód és felhő csomópontok minden tulajdonságát, mint például az elhelyezkedésüket és számítási kapacitásaikat. Annak érdekében, hogy valósághűen tudjuk szimulálni a fizikai számítógépek hálózati viselkedését, CPU kapacitásait, illetve memóriahasználatukat, ezért az MTA SZTAKI LPDS felhő sémáját használjuk a kiértékelés során. A klaszterezés után ebben a szimulációs osztályban fog a Scheduler-ek példányosítása sorra kerülni, valamint a DAX feladatokat is itt olvassuk be. Scheduler-enként külön DAX fájl fogunk beolvasni, így oldva meg a klaszterek között a feladatok szétosztását, de a klaszteren belül már az orkesztrációs folyamatok felelnek a feladatok megfelelő ütemezésére, az erőforrások tehermentesítésére.

A fentebb említett Scheduler osztályok fogják nagyrészen az orkesztrációt végezni a szimuláció során. Első feladatként a beérkező feladatok szétosztása lesz a feladata. A klaszteren belül a csomópontok között egyelőre az orkesztrátor csak random módon osztja szét a feladatokat. Mivel minden beérkező feladat esetén elérhető, hogy mennyi függősége van, így az orkesztrátor a fizikai eszközökön belül a beérkező feladatokat ez a függőségi szám szerint rendezi és a legkisebb függőségű feladatokat időzíti először. Ennek az az oka, hogy a több függőséggel rendelkező feladatok nagyobb eséllyel blokkolhatják a végrehajtást, amely így több időt vesz igénybe. Azt a feladatot, amelyeknek minden függősége feloldásra került, az orkesztrátor hozzárendeli egy virtuális géphez, amely megkezdi a feladat feldolgozását. A

virtuális gépek kiválasztása során az algoritmus a kevésbé leterhelteket fogja előnyben részesíteni a feladatok kiosztásánál, ezzel is biztosítva az egyenletes terhelést mind a fizikai mind a virtuális gépek között.

Ugyanebben az osztályban megvalósításra került egy scaling algoritmus is. Ennek a feladata, hogy nyomkövesse és monitorozza a kód eszközökre osztott feladatokat, valamint az azokon rendelkezésre álló virtuális gépeket. Amennyiben egy kód eszközön lévő elvégzendő feladatoknak a száma nagyságrendekkel meghaladja a rendelkezésre álló virtuális gépek számát, az algoritmus új virtuális gépek létrehozását kéri. Amennyiben erre van fizikai számítási kapacitás, a rendszer új virtuális gépet hoz létre, amivel elkezd a kód eszköz feladatait ellátni.

A szimuláció működéséhez még szükség van a végrehajtó osztály meghívására. Ez az osztály lesz felelős az architektúrákon lévő eszközök, erőforrások kezdeti inicializálásáért, azaz a virtuális gépek, valamint szenzorok és aktuátorok elindításáért. Ez az osztály fogja meghívni és elindítani a Scheduler osztály folyamatait is, valamint a Scheduler által beosztott feladatok szimulációs végrehajtásáért is ez az osztály a felelős.

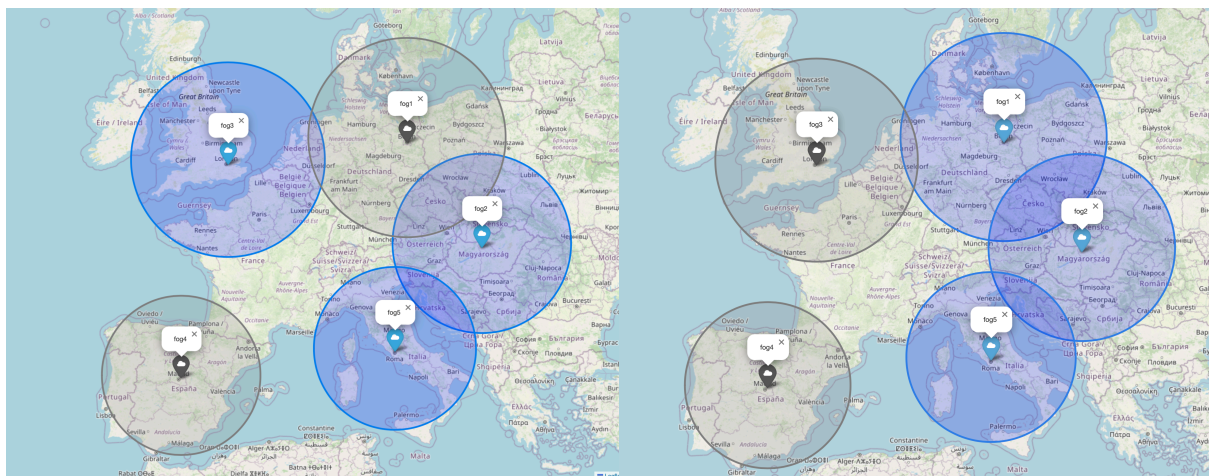
Mind a két scenáriót ugyanazon a számítógépen futtattam, amely egy 2,3 GHz kétmagos Intel Core i5 processzort használ, 8 GB 2133 MHz DDR3 RAM-mal rendelkezik és macOS Ventura operációs rendszert futtat. Mivel a szimulációk során random értékeket is használunk, ezért minden esetben 10-szer futtattam a szimulációkat, melyeknek átlag értékeit mutatom be. A szimuláció során minden kód eszköznek egységes kapacitásai voltak: 4 GB memória, 4 magos CPU, magonként 0.001 instrukció/ms feldolgozási sebességgel.

A szimuláció során beolvasott feladatok egy tudományos modell részét képezik. A Cybershake [\[21\]](#) elnevezésű workflow modell, amely egy szeizmikus veszély előrejelzését szimuláló munkafolyamatot ír le. A workflow-t arra használják, hogy szintetikus szeizmogramokat generálva jellemezzék és elemezzék a földrengési veszélyeket. A DAX fájl, amely ezt a workflow-t írja le 100 db feladatot tartalmaz különböző mennyiségű függőségekkel és futási idővel.

4.1. Első scenárió

Ebben a scenárióban egy kis méretű példát választottam mindösszesen 5 kód eszközzel. Klaszterenként pedig 100-100 feladatot olvastunk be. A kód eszközök viszonylag közel helyezkednek el egymáshoz, így elemezhető az algoritmusok konzisztenciája is ezzel a scenárióval. Az eszközöket Európában található városokba helyeztem el. A klaszterező

algoritmusok megegyező paraméterei ugyanazzal az értékekkel dolgoznak: a maximum iterációk száma 50, a feromon evaporáció 0.1 azaz 10%, valamint a randomFactor 0.3. A scenáriók során a hálózati késleltetést egységesítettem (20 ms-ra állítva). A centralizált algoritmusnál további paraméterekre volt szükség, ahol megadtam a már említett eszközök számát, ami 5 darab, valamint 50 darab hangyát példányosít az algoritmus a megoldás kereséséhez, aminek 2 darab klasztert adtam meg. Ennek az az oka, hogy az elosztott algoritmus futtatása során folyamatosan két klasztert hozott létre az algoritmus, ezért a pontos összehasonlítás érdekében a centralizált algoritmusnál is ezt a számot adtam meg.



(a) Centralizált algoritmus eredménye

(b) Elosztott algoritmus eredménye

4.1.1. ábra. Első scenárió klaszterezés eredménye vizualizálva

A 4.1.1. ábrán látható klaszterezési eredmények vizualizálásához a korábban említett Visualiser osztályt használtam, így jól elkülöníthetőek az algoritmusok által létrehozott, de egymástól különböző klaszterek.

Centralizált algoritmus	
Költségek	Szimulációs idő (perc)
1,00492347	~33,3

Elosztott algoritmus	
Költségek	Szimulációs idő (perc)
0,85910875	~26,2

4.1.1. táblázat. Első scenárió eredményei

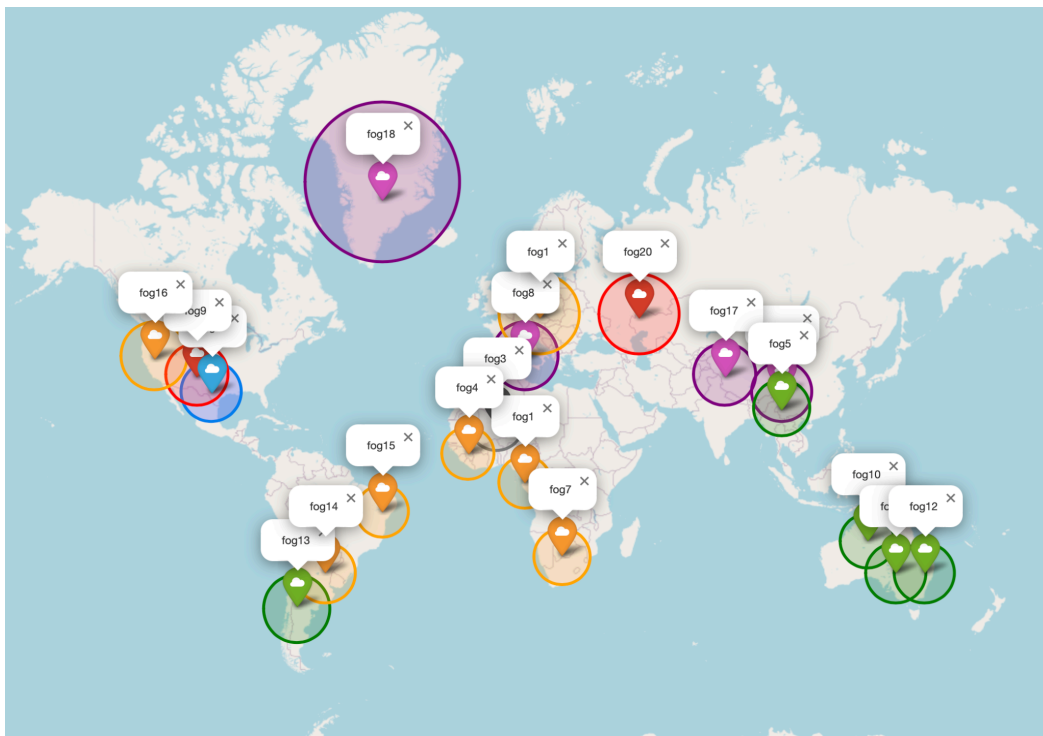
A futtatások átlagolt eredményeit a 4.1.1. táblázat szemlélteti. A klaszterező algoritmusokat két metrika alapján hasonlítottam össze. Mivel a feladatok feldolgozása virtuális gépek segítségével történik, ezért felhasználtam a szimulátor árazási modulját, amely

“pay-per-use” árazás alapján számolja a virtuális gépek költségeit. A másik metrika a szimulációs idő, amely az összes feladat feldolgozásához szükséges időt jelenti.

Érdemes megjegyezni, hogy a centralizált algoritmus ebben a scenárióban több esetben is arra az eredményre jutott, hogy az 5-ből 4 eszközt egy klaszterbe sorolt, 1 eszközt pedig külön hagyott a másik klaszterben. Mivel minden klaszter 100-100 feladatot kapott, ezért az egyedül maradt csomópontnak sokkal több időbe telt a ráosztott feladatokat feldolgozni. Ezen az offloading algoritmus sem segíthetett, mivel minden klaszter csak a saját erőforrásait látja.

4.2. Második scenárió

A második scenárióban egy sokkal nagyobb számú mintát értékeltem ki 20 darab csomóponttal, szétszórva a Föld kontinensein.

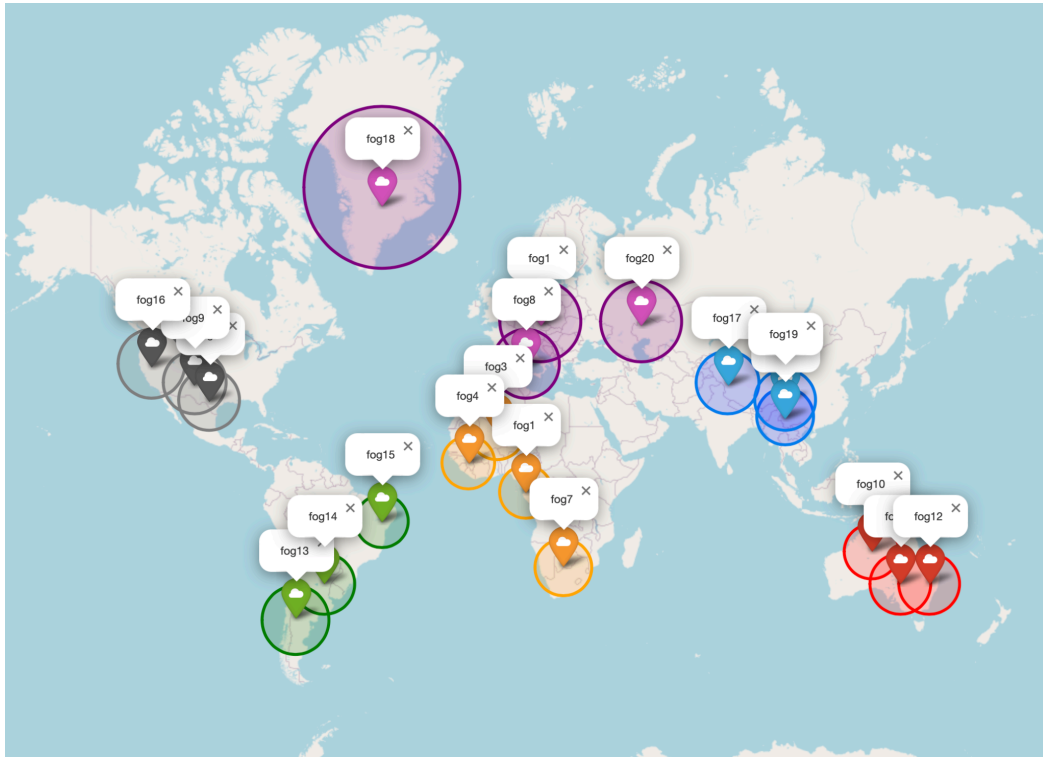


4.2.1 ábra. Centralizált algoritmus eredménye a második scenárión

Az elosztott klaszterező algoritmus által létrehozott csoportok ebben a scenárióban is hatékonyabban dolgozták fel a feladatokat, azaz hasonlóan az első scenárióhoz, az elosztott algoritmus itt is sokkal konzisztensebb eredményt tudott elérni. A hangyakolónia algoritmusok maximális iterációk számát megnöveltem 100-ra, a feromon evaporáció 0.2, a random szint továbbra is 0.3, valamint a klaszteren belüli válaszidő továbbra is 20 ms. A centralizált algoritmus esetében a hangyák számát 60-ra növeltem, így ez nagyságrendben

egyezik az elosztott algoritmus iterációjának számával és annak eredménye alapján a 20 eszközt 6 klaszterbe szeretnék szétosztani.

Az eszközök számával a terhelést is megemeltém az architektúrán. Továbbra is a Cybershake modellt használta a szimulátor, azonban a feladatok számát felskáláztam, így most minden klaszterre 5000 feladat jut, tehát összesen 30.000 darab feladatot dolgoznak fel a csomópontok.



4.2.2. ábra. Elosztott algoritmus eredménye a második scenárión

Centralizált algoritmus	
Költségek	Szimulációs idő (perc)
4072,053558	~1393,2

Elosztott algoritmus	
Költségek	Szimulációs idő(perc)
2964,9832	~574,6

4.1.2. táblázat. Második scenárió eredményei

Az eredmények azt mutatják, hogy a szimulációs idő, illetve a költségek közötti különbség tovább nőtt a két algoritmus között. Az elosztott algoritmus konzisztens értékeket és csoportokat tudott kialakítani, a centralizált viszont nem teljesített jól: az elosztott algoritmus közel 2.5-ször gyorsabban dolgozta fel a felskálázott feladatokat, amellyel megközelítőleg

1108 USD-t spórolt a költségeken. Az eredményeket elemezve viszont mindenképpen meg kell említeni, hogy a legjobb eredmény a centralizált algoritmus érte el egy 560 perces szimulációs idővel az elosztott 562 perces legjobb idejével szemben. Átlagban viszont a centralizált nagymértékben le van maradva az elosztott algoritmussal szemben.

5. fejezet

Konklúzió

A munkámban az elosztott felépítésű kód-felhő megoldások alapvető klaszterezési problémájára kerestem megoldást azzal, hogy hangyakolónián alapuló új klaszterező algoritmust hoztam létre, amely nem igényel előzetes információkat, mint például a létrehozandó klaszterek vagy az architektúrában jelen lévő eszközök számát. Ezt az új algoritmust különböző skálájú scenáriók segítségével értékeltem ki. Továbbá, összehasonlítást végeztem egy már korábban publikált hangyakolóniára épülő algoritmussal, amely alapján azt a következtetést tudtam levonni, hogy az új algoritmus sokkal konzisztensebb eredményeket tud generálni, mint a korábbi változata. Előnyei közé tartozik még, hogy az új algoritmus támogatja az elosztott klaszterezést is.

Ezekén felül a DISSECT-CF-Fog szimulátorban az elosztott felhő-kód rendszerek működéséhez szükséges orkesztrátor modult fejlesztettem ki, amelyben megvalósítottam a scheduling, offloading és scaling funkciókat. Az eddig működő monitoring funkciókat pedig kibővítettem, hogy képesek legyenek az elosztott rendszerek minden eszközéről összegyűjteni a szimuláció futtatása közben és után létrejött adatokat.

Az általam használt scenáriókban mind a heurisztikák mind a válaszdíők megegyeztek, azonban a heurisztikák jövőbeli változtatására helyet biztosítottam egy dedikált módszer segítségével, amellyel a klaszterező algoritmusok változtatása nélkül ki lehet próbálni más heurisztikákat is. További feladat lehet, hogy a jelenlegi egységesített válaszdíőket lecseréljük egy dinamikusabb megvalósításra.

A szimulátor nyílt forráskódú, az általam fejlesztett új modulok megtalálhatóak GitHub-on:

<https://github.com/sed-inf-u-szeged/DISSECT-CF-Fog>

Irodalomjegyzék

[1] State of IoT 2023 online (megtekintve: 2024.04.15.):

<https://iot-analytics.com/number-connected-iot-devices/>

[2] Lowe, Devesh & Galhotra, Bhavna. An Overview of Pricing Models for Using Cloud Services with analysis on Pay-Per-Use Model. International Journal of Engineering & Technology. 7. 248. (2018). DOI: 10.14419/ijet.v7i3.12.16035.

[3] Bermbach, David & Pallas, Frank & Pérez, David & Plebani, Pierluigi & Anderson, Maya & Kat, Ronen & Tai, Stefan. A Research Perspective on Fog Computing. (2018). DOI: 10.1007/978-3-319-91764-1_16.

[4] Luan TH, Gao L, Li Z, et al. Fog computing: focusing on mobile users at the edge. ; 2015. DOI: 10.48550/arXiv.1502.01815

[5] Atzori, Luigi & Iera, Antonio & Morabito, Giacomo. The Internet of Things: A Survey. Computer Networks. 2787-2805. (2010). DOI: 10.1016/j.comnet.2010.05.010.

[6] Breno Costa, Joao Bachiega Jr., Leonardo Rebouças de Carvalho, and Aleteia P. F. Araujo. Orchestration in Fog Computing: A Comprehensive Survey. ACM Comput. Surv. 55, 2, Article 29 (January 2022) DOI: 10.1145/3486221

[7] Kiss, T. et al. Swarmchestrator: Towards a Fully Decentralised Framework for Orchestrating Applications in the Cloud-to-Edge Continuum. In: Barolli, L. (eds) Advanced Information Networking and Applications. AINA Lecture Notes on Data Engineering and Communications Technologies, vol 203. Springer, Cham. (2024). DOI: 10.1007/978-3-031-57931-8_9

[8] U. C. Özyar and A. Yurdakul, "A Decentralized Framework with Dynamic and Event-Driven Container Orchestration at the Edge," 2022 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics), Espoo, Finland, pp. 33-40, (2022), DOI:10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics55523.2022.00017.

[9] Ullah, A., Kiss, T., Kovács, J. et al. Orchestration in the Cloud-to-Things compute continuum: taxonomy, survey and future directions. J Cloud Comp 12, 135 (2023). DOI:10.1186/s13677-023-00516-5

- [10] N. Wang, B. Varghese, M. Matthaiou and D. S. Nikolopoulos, "ENORM: A Framework For Edge NNode Resource Management," in IEEE Transactions on Services Computing, vol. 13, no. 6, pp. 1086-1099, (2020), DOI: 10.1109/TSC.2017.2753775.
- [11] L. L. Jiménez and O. Schelén, "HYDRA: Decentralized Location-Aware Orchestration of Containerized Applications," in IEEE Transactions on Cloud Computing, vol. 10, no. 4, pp. 2664-2678, (2022), DOI: 10.1109/TCC.2020.3041465.
- [12] Pires, A., Simão, J. & Veiga, L. Distributed and Decentralized Orchestration of Containers on Edge Clouds. J Grid Computing 19, 36 (2021). DOI: 10.1007/s10723-021-09575-x
- [13] Masip-Bruin X, Marín-Tordera E, Sánchez-López S, Garcia J, Jukan A, Juan Ferrer A, Queralt A, Salis A, Bartoli A, Cankar M, et al. Managing the Cloud Continuum: Lessons Learnt from a Real Fog-to-Cloud Deployment. Sensors. 21(9):2974. (2021); DOI: [10.3390/s21092974](https://doi.org/10.3390/s21092974)
- [14] Rainbow project (megtekintve: 2024.04.28): <https://rainbow-h2020.eu/rainbow-platform/>
- [15] ORMUCO Infrastructure as a service (megtekintve: 2024.04.28.): <https://ormuco.com/iaas/>
- [16] A. Markus, A. Al-Haboobi, G. Kecskemeti and Attila Kertesz. Simulating IoT Workflows in DISSECT-CF-Fog. Sensors 23, no. 3: 1294, (2023). DOI: 10.3390/s23031294
- [17] Kecskemeti, Gabor. HÁLÓZATI FORGALOMKEZELÉS SZÁMÍTÁSI FELHŐ SZIMULÁTOROKBAN. Multidiszciplináris tudományok. 4. 147-158. (2014).
- [18] Markus A, Biro M, Kecskemeti G, Kertesz A. Actuator behaviour modelling in IoT-Fog-Cloud simulation. PeerJ Comput Sci. 30;7:e651. (2021). DOI: 10.7717/peerj-cs.651.
- [19] Z. Zhang, K. Long, J. Wang and F. Dressler, "On Swarm Intelligence Inspired Self-Organized Networking: Its Bionic Mechanisms, Designing Principles and Optimization Approaches," in IEEE Communications Surveys & Tutorials, vol. 16, no. 1, pp. 513-537, First Quarter 2014, DOI: 10.1109/SURV.2013.062613.00014.
- [20] P.S Shelokar, V.K Jayaraman, B.D Kulkarni, An ant colony approach for clustering, Analytica Chimica Acta, Volume 509, Issue 2, Pages 187-195, ISSN 0003-2670, (2004), DOI:10.1016/j.aca.2003.12.032.
- [21] Graves, R., Jordan, T.H., Callaghan, S. et al. CyberShake: A Physics-Based Seismic Hazard Model for Southern California. Pure Appl. Geophys. 168, 367–381 (2011). DOI: 10.1007/s00024-010-0161-6
- [22] DISSECT-CF-Fog szimulátor online (megtekintve: 2024.04.28.): <https://github.com/sed-inf-u-szeged/DISSECT-CF-Fog>