

TDK-DOLGOZAT

LIGETFALVI BENCE

Budapest, 2022.

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

**FELHŐ-ORKESTRÁCIÓS
FOLYAMATOK HIBAKERESÉSÉNEK
TÁMOGATÁSA AKTÍV VEZÉRLÉSEL**

Szerző(k):

Ligetfalvi Bence

mérnökinformatikus MSc. szak, II. évf.

Konzulens(ek):

Dr. habil Lovas Róbert

egyetemi docens

Dr. Kovács József

SZTAKI tudományos főmunkatárs

Budapest, 2022.

Tartalomjegyzék

1.	Bevezetés	1
1.1.	Motiváció	1
2.	Kapcsolódó kutatások és fejlesztések	2
3.	A makrólépés-alapú hibakeresés.....	3
3.1.	Lokális töréspontok, kollektív töréspontok és makrólépések.....	4
3.2.	Végrehajtási fa és meta-töréspontok.....	4
3.3.	Makrólépés-alapú hibakeresés felhőkörnyezetben	5
4.	A kísérleti rendszer	7
4.1.	Komponensek és főbb funkcióik	7
5.	A hibakeresés vezérlésének menete.....	9
5.1.	Manuális vezérlés	10
5.2.	Visszajátszás	11
5.3.	Aktív vezérlés, automatikus bejárás	11
6.	A kísérleti rendszer tesztelése	12
6.1.	Automatikus bejárással kapcsolatos mérések.....	14
6.2.	Szabadfutásokkal kapcsolatos mérések	16
7.	Összegzés, továbbfejlesztési lehetőségek	19
8.	Köszönetnyilvánítás	19
	Hivatkozások	20

Ábrajegyzék

3.3.1. ábra: Makrólépések és lokális töréspontok felhőkörnyezetben [1].	6
3.3.2. ábra: A kísérleti rendszer fő komponensei és azok interakciói.	7
4.1.1. ábra: A Makrólépés vezérlő belső felépítését szemléltető ábra.	9
5.3.1. ábra: A három darab virtuális gépet tartalmazó tesztinfrastruktúra kontextualizációs folyamatai.	13
5.3.2. ábra: A három darab virtuális gépet tartalmazó infrastruktúrához tartozó teljes végrehajtási fa, a lehetséges 12 darab végrehajtási útvonallal.	13
6.1.1. ábra: Teljes bejárás ideje az egyes tesztinfrastruktúrák esetében, különböző orkesztrátorok használatakor.	14
6.1.2. ábra: Átlagos infrastruktúra képülési időtartamok Occopus használata esetén.	15
6.1.3. ábra: Átlagos infrastruktúra képülési időtartamok Terraform használata esetén.	16
6.2.1. ábra: Bejárt útvonalak száma három egymás utáni futást tekintve, Occopus esetében.	17
6.2.2. ábra: Bejárt útvonalak száma három egymás utáni futást tekintve, Terraform esetében.	17
6.2.3. ábra: Bejárt útvonalak száma 12 egymás utáni futást tekintve, Occopus esetében.	18
6.2.4. ábra: Bejárt útvonalak száma 12 egymás utáni futást tekintve, Terraform esetében.	18

Rövidítésjegyzék

AWS	<i>Amazon Web Services</i>
EC2	<i>Elastic Compute Cloud</i>
IaaS	<i>Infrastructure-as-a-Service</i>
MPI	<i>Message Passing Interface</i>
MTA	Magyar Tudományos Akadémia
NKFIH	Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal
OTKA	Országos Tudományos Kutatási Alapprogramok
P-GRADE	<i>Parallel Grid Run-Time and Application Development Environment</i>
PaaS	<i>Platform-as-a-Service</i>
PVM	<i>Parallel Virtual Machine</i>
REST API	<i>Representational State Transfer Application Programming Interface</i>
SLA	<i>Service Level Agreement</i>
SQL	<i>Structured Query Language</i>
SZTAKI	Számítástechnikai és Automatizálási Kutatóintézet
ÚNKP	Új Nemzeti Kiválóság Program
VM	<i>Virtual Machine</i>

1. Bevezetés

A számítástechnika fejlődése az elmúlt húsz évben jelentős innovációkat hozott annak sok területén. Ezen új technológiák egyike volt az úgynevezett felhő-infrastruktúrák megjelenése, majd későbbi elterjedése. A felhő-infrastruktúrák egyik csoportja az úgynevezett *IaaS* rendszerek, melyek esetében a felhasználók nagy mértékben személyre szabhatják a tényleges rendszer felépítését, összetételét és a felhasznált szoftvereket. Ezek a rendszerek elsősorban virtuális gép alapú infrastruktúrák, ahol a virtuális gépek valamilyen felhőszolgáltató rendszerén futnak. A felhasználók ezeket a virtuális gépeket tudják személyes igényeiknek megfelelően konfigurálni. Manapság általánosan elterjedtek az olyan eszközök, melyek segítségével az *IaaS* infrastruktúrák kiépítése, felügyelete, skálázása és lebontása automatikusan vagy a felhasználótól csak minimális beavatkozást igényelve véghezvihető. Ezeket az eszközöket általánosan a felhő-orkesztrátor gyűjtőfogalommal illetjük. Ugyan az előbb említett folyamatok (kiépítés, felügyelet stb.) a felhő-orkesztrátorok segítségével könnyen elvégezhetők, de az esetlegesen felmerülő hibák feltárása, azok okainak megtalálása, illetve az általánosan vett hibakeresési tevékenységek terén továbbra sincsenek kiforrott eszközök, eljárások vagy módszerek.

Az *IaaS* rendszerek esetében történő hibakeresést több tényező is hátráltatja, ezek közül a legnehezebben vizsgálhatók az ilyen rendszerek alapvetően nem-determinisztikus viselkedésből fakadó hibák. Az *IaaS* rendszerek kiépülése több fizikai tényezőtől is függ, mint a tényleges erőforrásként szolgáló szervergépek pillanatnyi kihasználtsága, a pillanatnyi hálózati terhelés, memóriakihasználtság stb. Ezenfelül az egyes virtuális gépek közötti függőségek és kapcsolatok is közrejátszanak. A fenti fizikai tényezők nem állandók, nem azonosak minden pillanatban. Ebből adódóan az *IaaS* rendszerek alapvetően nem-determinisztikus viselkedést mutatnak, melynek okán egy képülési vagy skálázási folyamat más-más eredménnyel zárulhat egyik vagy másik időpontban, sőt, akár felhőszolgáltatóktól függően. Az előzőkből fakadóan a hibás folyamatok okának feltárása, annak mikéntje nem magától értetődő.

1.1. Motiváció

A 1990-es években hasonló problémákkal szembesültek a fejlesztők a párhuzamos és elosztott rendszerek területén. A párhuzamosan futó folyamatok egymással kapcsolatban állnak, a folyamatok tényleges futására kihatnak a fizikai tényezők (processzor-, memória-,

hálózat-kihasználtság stb.), így ezekben a programokban sem egyértelmű a nem-determinisztikus viselkedésből fakadó hibák és azok okainak feltárása. Az *IaaS* rendszerekben lévő virtuális gépek kiépítése is többnyire párhuzamosan történik, ezzel gyorsítva a teljes infrastruktúra kiépülését. Az egyes virtuális gépek között kapcsolatok állnak fenn, ezek a gépek egymással kommunikálnak, illetve közrejátszanak a fizikai tényezők is a kiépülésben [1].

Szintén az 1990-es években jelent meg egy olyan hibakeresési eljárás, mely képes ezt a nem-determinisztikus viselkedést kezelni a hagyományos hibakeresés terén használt töréspontok párhuzamos rendszerekre történő kiterjesztésével. Ez a hibakereső eljárás az úgynevezett makrólépés-alapú hibakeresés, mely példaképpen a *P-GRADE* nevezetű rendszerben is helyet kapott [2] [3].

A dolgozatban ezt a később bemutatandó hibakeresési eljárást ültettem át *IaaS* alapú rendszerekre. A dolgozatban bemutatásra kerül maga a hibakeresési eljárás, annak felhőalapú, *IaaS* rendszerekben történő megvalósítása, az általam kifejlesztett kísérleti, hibakeresést vezérlő eszköz, a felhasznált kísérleti rendszer, illetve kapcsolódó mérési eredmények, melyek szemléltetik a makrólépés-alapú hibakeresési eljárás felhő-környezetben történő használatának előnyeit, elsődlegesen az aktív vezérlést illetően.

2. Kapcsolódó kutatások és fejlesztések

A felhőalapú rendszerek elterjedés óta ugyan jött létre kevés számú hibakeresésre használható eszköz, de ezek legtöbbje vagy elsősorban *PaaS* rendszerekben történő hibakeresésre használatos, vagy nem biztosít aktív kontrollt az *IaaS* rendszerben történő hibakeresés során.

A *Microsoft Azure* terméke ugyan rendelkezik beépített eszközzel, mellyel a virtuális gépek állapotát lehet ellenőrizni, de ez kimerül a hagyományos értelemben vett monitorozással, továbbá nincs lehetőség visszajátszásra vagy aktív kontrollra [4].

A *Google Cloud Debugger* nevezetű eszköze segítségével, mely a *Google Cloud Platform* részét képezi, lehetséges *.NET*, *Python*, *Java* vagy egyéb programozási nyelveken íródott felhőalapú alkalmazások tesztelése. Ez a hibakereső eszköz *PaaS* alkalmazások tesztelésére hivatott, tartalmaz *snapshot* és logolási funkciókat [5].

M. S. S. Quroush és *T. Ovatman* kidolgoztak egy olyan hibakeresésre használható eszközt, melynek segítségével lehetséges felhő-applikációk vizsgálata, illetve az esetleges hibás esetek visszajátszása az úgynevezett „*record & replay*” elvre építve. Ugyan az eszköz

képes visszajátszásra, de nem képes aktív kontrollra, mely az állapottér automatikus bejárásához szükséges lenne [6].

D. Sharma, P. Sharm és S. Chatterjee és egy olyan eszközt dolgoztak ki, melynek segítségével lehetséges ellenőrizni megadott *SLA* feltételek teljesülését. Az eszköz segítségével az egyes infrastruktúra komponenseket lehet ugyan monitorozni, illetve azok korábbi állapotát megtekinteni, de nincs lehetőség aktív kontrollra, továbbá csakis *OpenStack*-alapú felhők esetében alkalmazható [7].

3. A makrólépés-alapú hibakeresés

Korábban már említésre került ez a hibakeresési eljárás, melyet az 1990-es években fejlesztettek ki alapvetően *PVM* és *MPI* alapú rendszerek, programok számára. Ebben az időben az elsődlegesen elterjedt hibakeresési eljárások az úgynevezett „*monitor & replay*” elvre építettek, melyek lényege, hogy a hibakeresési folyamatot két részre osztják [2].

A folyamat első felében információgyűjtés, monitorozás zajlik le, amelyet egy visszajátszási fázis követ. Az információgyűjtés során minden olyan adatot begyűjtünk a párhuzamos program futásáról, ami a hibakeresés szempontjából releváns lehet, legyen szó a program belső állapotáról vagy időzítésekről. Ezeket az adatokat használjuk fel a visszajátszási fázisban a program vezérlésére, újbóli futtatására, ezáltal újra előállítva a kérdéses állapotot, melyet vizsgálni szeretnénk. Ez az elv képes visszajátszást biztosítani, így segítve a fejlesztők munkáját, azonban egy új problémát hoz be a hibakeresésbe. Ez az új probléma a szondahatás, melynek köszönhetően a megfigyelési fázis a párhuzamos program belső időzítéseit befolyásolja. Ennek a hatásnak köszönhetően előfordulhatnak olyan esetek, amikor a nem monitorozott program esetében megfigyelt hibát nem feltétlenül lehet újra előidézni a visszajátszás során [2].

Magát a szondahatást lehet ugyan csökkenteni, de teljesen megszüntetni nem lehet. Ugyan minél kevesebb adatot gyűjtünk a program belső állapotáról, annál kisebb a szondahatás, de a begyűjtött adatmennyiség csökkentése nehezebbé teszi a hibás esetek okainak feltárását [2].

Egy másik hibakeresési elv teljesen elhagyja a monitorozási fázist és olyan teszteseteket használ fel, melyeknek köszönhetően a párhuzamos programban lévő összes belső időzítés letesztelhető. Ez az az úgynevezett „*control & replay*” elv. A visszajátszás során nem a begyűjtött adatokra támaszkodik, hanem ezekre a generált tesztesetekre. Ennek az elvnek

köszönhetően az összes lehetséges programidőzítést le tudjuk tesztelni, akár automatizált módon. A makrólépés-alapú hibakeresés erre az elvre épít [2].

3.1. Lokális töréspontok, kollektív töréspontok és makrólépések

A makrólépés-alapú hibakeresési eljárás alapvetően épít a töréspontokra, az egyes párhuzamosan folyó folyamatokat töréspontok segítségével állítja meg. Ezek az úgynevezett lokális töréspontok a folyamatok közötti kommunikációs pontokhoz vannak elhelyezve. Mikor minden folyamat elér egy lokális töréspontot, akkor az összes folyamat, és így a teljes párhuzamos program futása megállt. Ebben a pillanatban megvizsgálható a program és a folyamatok belső állapota, hasonló módon, mint a hagyományos szekvenciális programoknál.

Egy lokális töréspont halmazt egy kollektív töréspontnak nevezünk. Amennyiben egy kollektív töréspont az összes folyamatot lefedi, tehát az összes folyamatból tartozik bele egy lokális töréspont, akkor *teljesnek* nevezzük, ellenkező esetben *részlegesnek*. Amennyiben minden úgynevezett alternatív útvonalat is lefed egy kollektív töréspont, akkor pedig *erősen teljesnek* nevezzük [2] [3].

Két kollektív töréspont közötti kódterületet, kódhalmazt egy makrólépésnek nevezünk, innen is ered a hibakeresési eljárás elnevezése. A makrólépéseket is több csoportba tudjuk sorolni, függően attól, hogy mennyi kommunikációs utasítást tartalmaznak. Amennyiben többet, mint egy, akkor a makrólépést *vegyesnek* nevezzük. Amennyiben viszont csak egy darabot, akkor *tisztának*. A tiszta makrólépések esetében a kommunikációs utasítások mindig a kódterület legutolsó utasításai [2] [3].

A lokális töréspontok, a kollektív töréspontok, illetve a makrólépések segítségével kiterjeszthető a hagyományos, szekvenciális programoknál használt, töréspontról-töréspontra történő hibakeresés a párhuzamos programok világára is. A különbség csupán annyi, hogy már kollektív töréspontról-kollektív töréspontra, makrólépésenként halad a hibakeresés [2] [3].

3.2. Végrehajtási fa és meta-töréspontok

Egy kollektív töréspont sorozatot egy végrehajtási útnak nevezünk. A végrehajtási fa az összes lehetséges végrehajtási utat tartalmazza. Egy végrehajtási út a párhuzamos program egy belső időzítését, időzítéseit tartalmazza, így a végrehajtási fa a párhuzamos program összes belső időzítését lefedi. A végrehajtási fában a csomópontok a kollektív töréspontok, az élek pedig a makrólépések [2] [3].

A végrehajtási fa minden esetben a gyökércsomóponttal kezdődik, a hibakeresés során pedig folyamatosan bővül. A párhuzamos programok esetében három fajta csomópontot tudunk elkülöníteni. Amennyiben egy csomópont esetében csak egy útvonal lehetséges (egy folyamat léptethető), akkor a csomópont *determinisztikus*. Amennyiben a csomópontnál lehetséges kettő vagy több útvonal, akkor pedig úgynevezett „fork” csomópontról van szó. A hagyományos *MPI* rendszerekben ezek egyik változata volt az úgynevezett *alternatív* töréspontok, melyek a „wildcard” utasításokra épültek. A harmadik csomópontfajta a már korábban említett *gyökércsomópont* [2] [3].

Töréspontokat a végrehajtási fában is elhelyezhetünk, melyeket *meta-töréspontoknak* nevezünk. Ezen töréspontok esetében nem történik más, mint hogy a megfelelő végrehajtási útvonalon haladva elérjük a meta-töréspontot. A program futtatása során ez lényegében egy teljes vagy részleges visszajátszást jelent, függően, hogy a meta-töréspont a fa legaljára vagy egy köztes csomópontra lett helyezve. A meta-töréspontok esetében a programot mindig a megfelelő útvonalra kényszerítjük, mely nem jelent mást, mint hogy az adott csomópontokban a megfelelő folyamatot léptetjük a következő lokális töréspontja felé [2] [3].

Tiszta makrólépések és erősen teljes kollektív töréspontok segítségével elérhető a párhuzamos program teljeskörű tesztelése [2].

3.3. Makrólépés-alapú hibakeresés felhőkörnyezetben

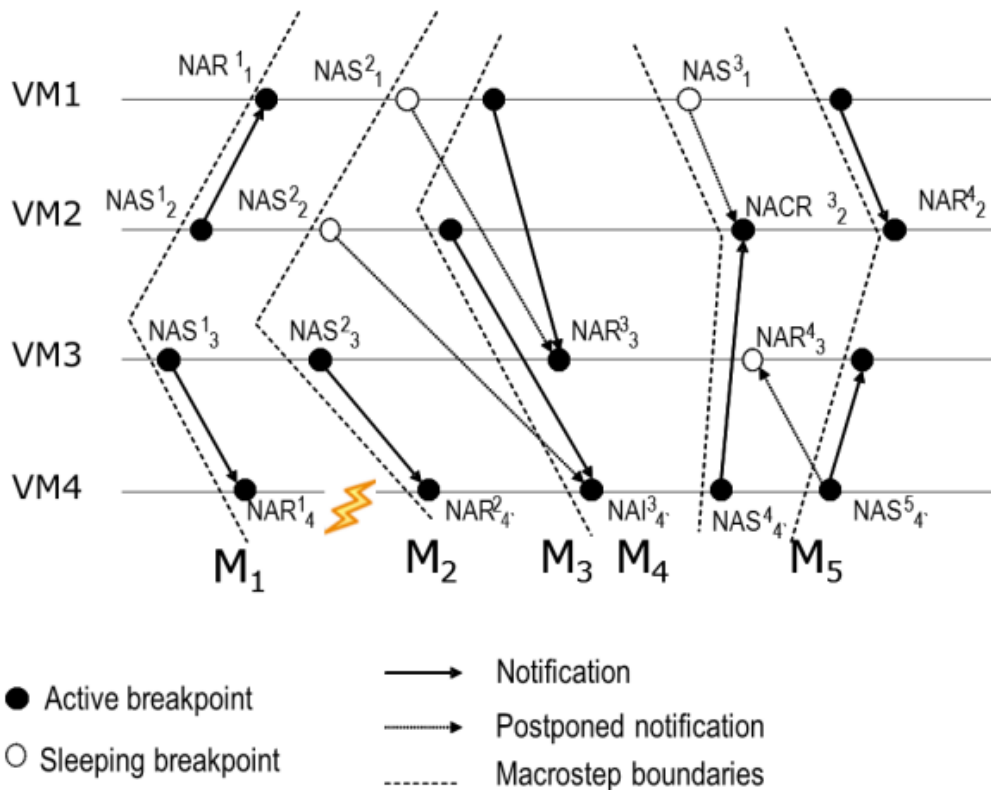
Korábban már említésre kerültek az *IaaS* rendszerek, mint a hagyományos párhuzamos rendszerekkel nagy hasonlóságot mutató infrastruktúrák. Az ilyen rendszerekben akár több tucat virtuális gép is helyet kaphat, ezek kiépítése pedig többnyire párhuzamosan zajlik. Hasonlóan a párhuzamos rendszerekhez, a nem-determinisztikus viselkedésből fakadó hibás futások visszajátszása és vizsgálata egy nehézkes folyamat [1].

Egy egyszerű példa a szerver-kliens architektúrák esete, ahol elfordulhatnak olyan időzítések, amikor a szervergép rendkívül lassan épül ki, míg a kliens gép(ek) esetében ez gyorsan megtörténik. Ilyen esetekben a szervergép nem feltétlenül tudja kezelni a kliens gép(ek) kéréseit és végsősoron az infrastruktúra hibásan vagy hiányosan épülhet ki.

Az *IaaS* rendszerek kiépülése nem más, mint virtuális gépek kiépülési folyamatai, melyek során a felhasználói igényeknek megfelelő beállítások kerülnek végrehajtásra, ideértve szoftverek telepítése, adatbázisok konfigurálása, felhasználói fiókok konfigurálása stb. Ezeket a folyamatokat egy szóval *kontextualizációnak* szokás nevezni. Ezek a kontextualizációs folyamatok zajlhatnak párhuzamosan egy infrastruktúra kiépítése során.

Az egyes kontextualizációs folyamatokat lokális töréspontok segítségével képesek lehetünk megállítani. A folyamatokat mindaddig gátolni tudjuk, amíg a hibakereső vagy vezérlő úgy nem dönt, hogy azok folytatódhatnak. Hasonlóan a párhuzamos programokhoz, *IaaS* rendszerek esetében is lehetséges kollektív töréspontokba szervezni a lokális töréspontokat, és ennek köszönhetően makrólépéseket is lehetséges alkalmazni. Egy kollektív töréspont elérése nem jelent mást, mint hogy az infrastruktúra kiépülése megáll, egészen addig, amíg az egyik virtuális gép kontextualizációs folyamata nem lett a hibakereső által folytatva [1].

Ebben a környezetben is beszélhetünk végrehajtási fáról, mely az infrastruktúra összes lehetséges végrehajtási útvonalát, az összes lehetséges időzítést tartalmazza. Szintén lehetséges több felé csoportosítani a végrehajtási fa csomópontjait. Amennyiben egy csomópontban csak egy darab kontextualizációs folyamat léptethető, tehát csak egy darab folyamatnak maradt még el nem ért lokális töréspontja, akkor a csomópont determinisztikus, ellenkező esetben „*fork*” (alternatív). A végrehajtási fa itt is a gyökércsomóponttal indul. Szintén elhelyezhetők meta-töréspontok, melyek használatakor az infrastruktúra kiépülése a megadott végrehajtási útvonal szerint történik.

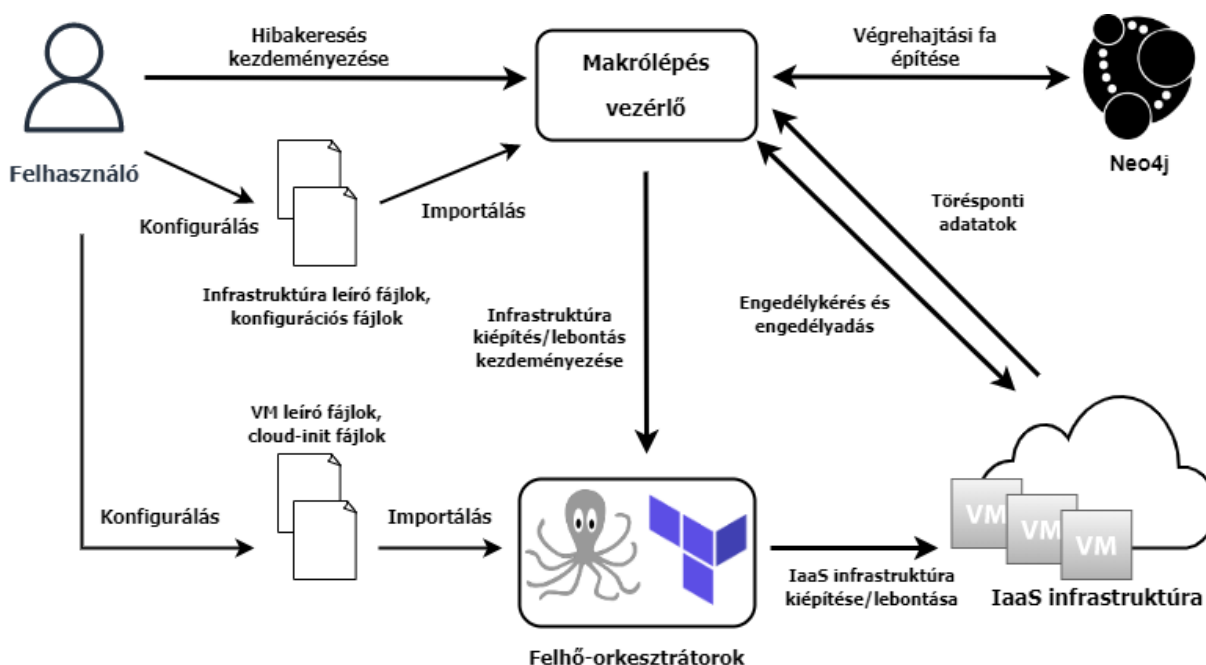


3.3.1. ábra: Makrólépések és lokális töréspontok felhőkörnyezetben [1].

A 3.3.1. ábrán a felhő-alapú rendszerekben történő makrólépés-alapú hibakeresési eljárás alapvető elemei láthatók, benne a lokális töréspontok (például NAR^1_1 , NAS^2_1 , NAS^3_1), a makrólépések (M_1 , M_2 , ..., M_5), illetve a párhuzamos folyamatok ($VM1$, $VM2$, $VM3$, $VM4$) [1].

4. A kísérleti rendszer

Ebben a fejezetben bemutatásra kerül az általam felhasznált kísérleti rendszer, melyek szükségesek egy IaaS rendszerek makrólépés alapú hibakeresésének vezérléséhez. Bemutatásra kerülnek annak komponensei, illetve az azok közötti interakciók. Egy átfogó képet lehet látni a kísérleti rendszerről a 3.3.2. ábrán.



3.3.1. ábra: A kísérleti rendszer fő komponensei és azok interakciói.

4.1. Komponensek és főbb funkcióik

Korábban említésre kerültek a felhő-orkesztrátorok, mint könnyen és egyszerűen használható eszközök a felhőalapú infrastruktúrák kiépítése számára. Ezek közé tartozik a SZTAKI által fejlesztett *Occopus* [8], illetve a *HashiCorp* által fejlesztett *Terraform* [9]. Ezen eszközök végzik az infrastruktúrák tényleges kiépítésének vezérlését a kísérleti rendszerben.

Az *Occopus* egy könnyű, úgynevezett „light-weight” felhő-orkesztrátor, mely alkalmas IaaS rendszerek kiépítésére, felhasználva több népszerű, „kommersz” felhőszolgáltatót, mint például az *Amazon Web Services* [10] vagy a *Microsoft Azure*, de képes privát, például

OpenStack [11] alapú felhők használatára is. Az *Occopus* egy *IaaS* infrastruktúra teljes életciklusát képes végigkísérni, ideértve a kiépítést, a monitorozást, a skálázást és a lebontást. Az infrastruktúrákat és virtuális gépeket infrastruktúra leíró fájlok, csomópont (virtuális gép) leíró fájlok és kontextualizációs fájlok alapján építi ki és tartja karban. Használata egyszerű, tartalmaz parancssori és webes (*REST API*) interfészt is [12].

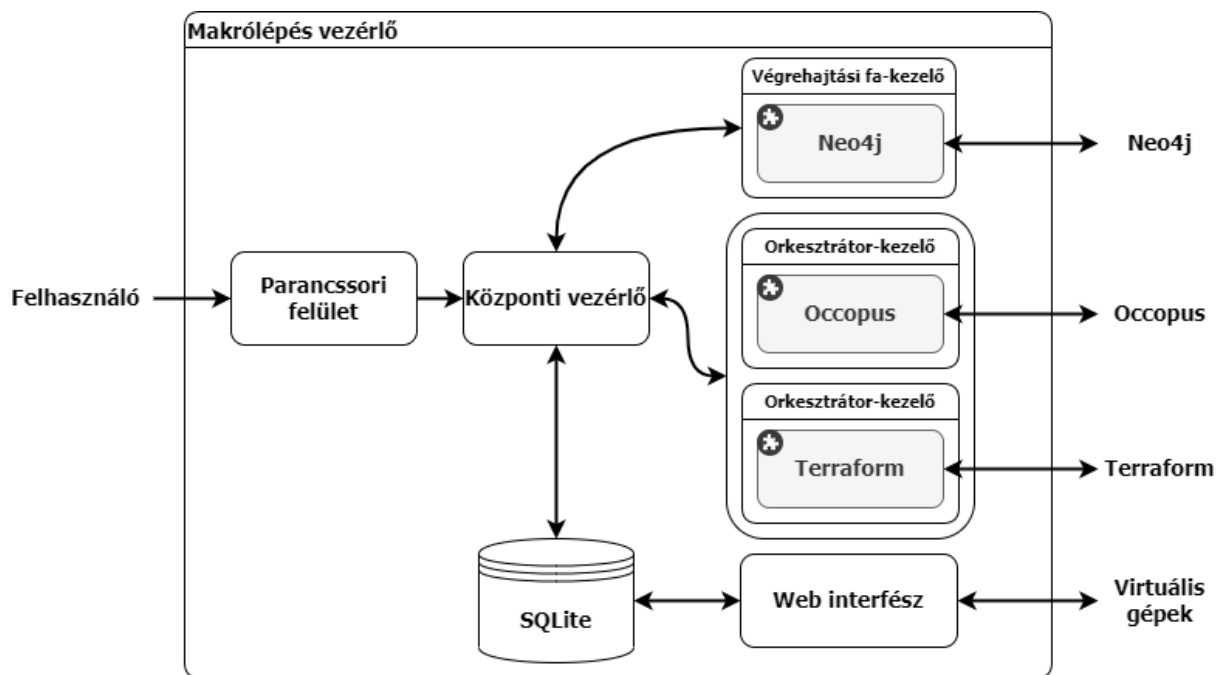
A *Terraform* egy, az *Occopus*-nál robusztusabbnak tekinthető felhő-orkesztrátor, mely nagyobb személyre szabhatóságot biztosít a felhasználók számára. Alapvető funkciói azonosak az *Occopus*-szal: infrastruktúrák kiépítése, felügyelete, skálázása, illetve szükség esetén lebontása. Szintén infrastruktúra leírókat és kontextualizációs fájlokat használ, ezeket kiegészíthetik a felhasználók általuk megadott változókkal, konstansokkal is. Szintén képes a legtöbb publikus felhőszolgáltató, illetve privát felhők használatára. Hátránya, hogy egyelőre csak parancssori interfészt biztosít a felhasználók számára [9].

Mind a *Terraform*, mind az *Occopus* esetében említésre kerültek a kontextualizációs fájlok. Ezek közül a legelterjedtebb a *Cloud-init* nevezetű eszköz, mely manapság már „*de-facto*” standardnak tekinthető az iparban, a legtöbb publikus és privát felhőszolgáltató biztosít használatára lehetőséget. A *Cloud-init* fájlok segítségével a felhasználók könnyen és egyszerűen testre szabhatják a virtuális gépeket, ideértve csomagok, szoftverek automatikus telepítését, felhasználói fiókok konfigurálását, de akár egyedi scriptek futtatását is. A kísérleti rendszerben a *Cloud-init* állományokban kerültek elhelyezésre a makrólépés-alapú hibakeresés alapkövének számító lokális töréspontok [13].

A végrehajtási fa, illetve a párhuzamosan futó kontextualizációs folyamatokból gyűjtött adatok tárolására szükséges volt egy megfelelő eszközt választani, mely képes nagy mennyiségi adat és nagyobb állapotterek, végrehajtási fák tárolására. Erre megfelelő eszköz a *Neo4j* elnevezésű gráf-adatbázis. Használata egyszerű, lekérdezései *SQL*-szerűek, webes interfészt és vizualizációt is biztosít [14].

A felhőkörnyezetben történő makrólépés-alapú hibakeresés tényleges vezérléséért egy általam készített, *Python*-ban írt, kísérleti *Makrólépés vezérlő* a felelős. Ez a vezérlő képes manuális és automatikus működésre, továbbá képes egy korábbi végrehajtási út visszajátszására is. A vezérlő elsődleges feladata a felhő-orkesztrátorok számára a megfelelő parancsok kiadása, például egy infrastruktúra kiépítése vagy lebontása, esetlegesen ezektől adatok lekérése az infrastruktúrák pillanatnyi állapotára vonatkozóan. A virtuális gépektől érkező adatokat egyrészt lokálisan, másrészt a *Neo4j* adatbázisban tárolja, továbbá a vezérlő feladata a végrehajtási fa bővítése, illetve a szükséges folyamatok léptetése visszajátszás és

automatikus bejárás (aktív kontroll) esetében. A Makrólépés vezérlő belső felépítését mutatja a 4.1.1. ábra.



4.1.1. ábra: A *Makrólépés vezérlő* belső felépítését szemléltető ábra.

A *Makrólépés vezérlő* tartalmaz egy web interfészt, mely elsődleges feladata a virtuális gépektől érkező törésponti adatok, illetve engedélykérések fogadása. A beérkező adatok tárolása a beépített *SQLite* [15] adatbázisban történik. Ez egy könnyű és egyszerűen használható *SQL* adatbázis, a világ legelterjedtebben használt adatbázisa. Az adatbázis ezentúl a virtuális gépekre és az infrastruktúrák állapotára vonatkozó adatokat is tárol, mely alapján a *Központi vezérlő* képes a döntéshozatalra. A felhasználó egy parancssori felületen tud a vezérlővel kommunikálni, kérvényezni manuális vagy aktív kontroll alapú vezérlést, esetlegesen visszajátszást vagy szabadfutást. A *Központi vezérlő* úgynevezett kezelőkön keresztül tartja a kapcsolatot a felhő-orkesztrátorokkal, illetve a *Neo4j* adatbázissal, ezeken keresztül történik az infrastruktúrák kiépítésének és lebontásának kezdeményezése, valamint a végrehajtási fa építése.

5. A hibakeresés vezérlésének menete

Ebben az alfejezetben a *Makrólépés vezérlő* manuális üzemmódja, a visszajátszás és az aktív kontroll alapú vezérlés kerül bemutatásra.

A felhasználó kezdetben konfigurálja a virtuális gépek számára szükséges leíró fájlokat, beleértve a Cloud-init fájlokat is, az orkesztrátorok számára szükséges infrastruktúra leíró

fájlokat, valamint a vezérlő számára szükséges konfigurációs állományokat. Ezek a konfigurációs állományok adják meg a felhő-orkesztrátorok helyét, legyen lokális vagy távoli, illetve a Neo4j adatbázis elérési helyét. A konfigurációs állományok továbbá tartalmazznak egy hivatkozást az infrastruktúra állományokra, illetve a felhasználandó felhő-orkesztrátorra. A szükséges fájlok importálása után a felhasználó megkezdheti az infrastruktúra tesztelését.

5.1. Manuális vezérlés

Manuális vezérlés során a felhasználó megadja a kívánt infrastruktúra nevét, majd a *Központi vezérlő* a megfelelő felhő-orkesztrátor-kezelőn keresztül utasítást ad az infrastruktúra kiépítésére. Az orkesztrátorok végzik az infrastruktúrák kiépülésének felügyeletét. A virtuális gépek ezek után a *VM leíró fájlok* alapján (3.3.2. ábra) elindulnak a kijelölt felhő-szolgáltatóknál, megkezdődik azok kiépülése a kontextualizációs állományok alapján, majd a kontextualizációs folyamatok elérik az elhelyezett lokális töréspontokat. A jelenlegi implementációban ezek a lokális töréspontok (*shell*)scriptként lettek megvalósítva.

A törésponti scriptek elérésekor adatgyűjtés történik, majd ezek az adatok továbbításra kerülnek a *Makrólépés vezérlő* webes interfésze felé. Ezen adatok két részre oszlanak, folyamat azonosító adatokra és felhasználói adatokra. A folyamat azonosító adatok szükségesek a virtuális gép (folyamat) azonosítására, míg a felhasználói adatokban a virtuális gép belső állapotára vonatkozó információk találhatók. Ezt a részt a felhasználók személyes igényeik szerint bővíthetik, a beérkező adatokat a vezérlő a belső adatbázisában, majd a Neo4j adatbázisban (is) eltárolja.

Az töréspontok elérése során a virtuális gépek várakozni kényszerülnek mindaddig, amíg egy megfelelő jelet nem kapnak a vezérlőtől a kontextualizáció folytatására. A vezérlő manuális vezérlés esetén (illetve aktív kontroll és visszajátszás során is) addig vár, míg az összes gép el nem éri az első töréspontját. Amint ez megtörtént az összes gép esetében, az infrastruktúra elérte a gyökérállapotot, a végrehajtási fa első csomópontját. Ez az állapot minden esetben az első kollektív töréspont.

Manuális vezérlés esetén a vezérlő mindig a felhasználó által megadott virtuális gép kontextualizációs folyamatát fogja engedélyezni. A léptetett virtuális gép kontextualizációs folyamata folytatódik, majd eléri a következő lokális töréspontot. A törésponti script lejelenti a begyűjtött adatokat a vezérlő számára és engedélyt kér a tovább lépésre. A vezérlő eltárolja a beérkezett törésponti adatokat, majd kérvényezi a felhasználót egy még be nem fejezett folyamat engedélyeztetésére. A többi virtuális gép kontextualizációs folyamata ezen idő alatt

továbbra is várakozik. A fenti folyamatok (felhasználó felé kérés, kontextualizációs folyamat léptetése/engedélyeztetése, törésponti adatok fogadása, várakoztatás) mindaddig folytatódnak, amíg az összes virtuális gép el nem éri az utolsó lokális töréspontját. Minden egyes engedélyeztetés egy darab makrólépést jelent, valamint minden egyes úgynevezett konzisztens globális állapot (amikor az összes virtuális gép várakozik) egy darab kollektív töréspontnak számít. A vezérlő az egyes léptetések után bővíti a végrehajtási fát, létrehozva a még nem létező csomópontokat és éleket, valamint új adatokkal frissítve az esetlegesen már létezőket.

5.2. Visszajátszás

Egy visszajátszási folyamat tulajdonképpen egy végrehajtási út teljes vagy részleges bejárását jelenti. A visszajátszási folyamat során a vezérlő kéri a felhasználót egy, a végrehajtási fában lévő csomópont megadására. A visszajátszási folyamat elindulásakor a megadott infrastruktúra egy példánya elindításra kerül a megfelelő felhő-orkesztrátor-kezelőn keresztül, majd a vezérlő mindaddig várakozik, míg az infrastruktúra gyökérállapotba nem kerül.

Ennek elérésekor a virtuális gépek kontextualizációs folyamatai engedélyre várakoznak. A *Központi vezérlő* kiszámítja mely folyamat léptetése szükséges a felhasználó által megadott csomópont eléréséhez és a megfelelő folyamatot fogja engedélyeztetni. Az adott folyamat folytatja a kontextualizációt, majd eléri a következő lokális töréspontot. A vezérlő érzékeli, hogy egy konzisztens globális állapot áll fenn, majd újra kiszámítja, hogy melyik folyamat léptetése szükséges a cél csomópont eléréséhez.

Ez a folyamat mindaddig folytatódik, amíg a cél csomópontot el nem érte az infrastruktúra visszajátszása. A visszajátszási folyamat során egy már korábban bejárt időzítést tudunk újra végrehajtani, mely funkció tulajdonképpen nem más, mint a korábban említett meta-töréspontok megvalósítása.

5.3. Aktív vezérlés, automatikus bejárás

A *Makrólépés vezérlő* képes az állapottér automatikus bejárása köszönhetően a beépített aktív vezérlésnek. Automatikus bejárás során a vezérlő maga választja ki, mely folyamat léptetése szükséges egy még fel nem tárt végrehajtási útvonal bejárásához.

Hasonlóan a manuális vezérléshez és a visszajátszáshoz, itt is az infrastruktúra indításával, majd a gyökércsomópont elérésével kezdődik a folyamat. Annak elérésekor a

vezérlő kiválasztja a névsorrendben, ezen belül pedig azonosító szerint a legelső virtuális gépet (folyamatot), majd, ha az amennyiben még nem fejezte be kontextualizációját (tehát van még el nem ért lokális töréspontja), akkor azt fogja léptetni. A következő kollektív töréspont esetében is megtörténik a kiválasztás, majd egy folyamat léptetésre kerül. Ez mindaddig folytatódik, míg az összes kontextualizációs folyamat be nem fejeződik.

Ebben a pillanatban az infrastruktúra kiépülése befejeződött, a végrehajtási út eltárolásra került, a vezérlő pedig utasítást ad az infrastruktúra lebontására. A vezérlő a végrehajtási fában visszafele haladva megkeresi a legközelebbi alternatív csomópontot és ennek megtalálása után egy visszajátszási folyamat kezdődik meg az adott alternatív csomópontig egy új infrastruktúra(példány) felhasználásával. A visszajátszás végeztekor a vezérlő kiválasztja azt az útvonalat, amely felé még nem történt bejárás és a megfelelő virtuális gépet választja ki léptetésre. A kiválasztás továbbra is a virtuális gépek név- és azonosítósorrendjére alapszik. A fenti visszajátszás, majd új útvonal választás folyamata egészen addig tart, míg az összes lehetséges végrehajtási útvonal bejárásra nem került. A leírtak lényegében nem jelentek mást, mint egy fa mélységi bejárását.

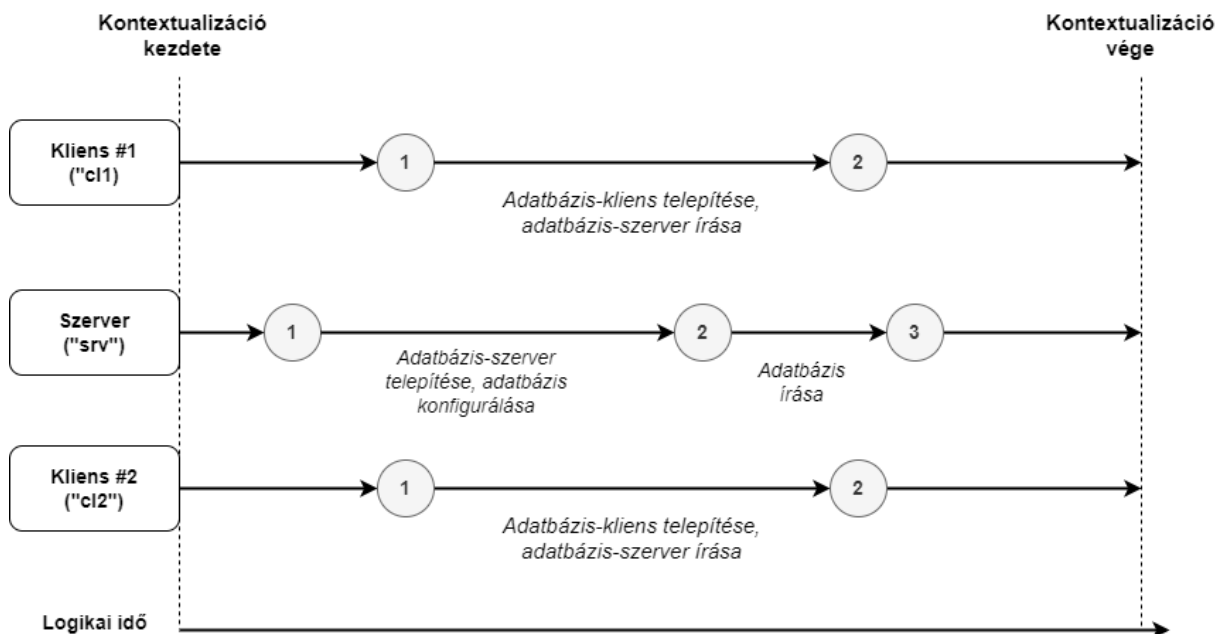
6. A kísérleti rendszer tesztelése

A kísérleti rendszer teszteléséhez egy olyan infrastruktúrát választottam, mely könnyen skálázható, mégis jól szemlélteti a makrólépés-alapú hibakeresés funkcióit. Ehhez egy szerver-kliens architektúrát használtam fel. 3 esetet néztem meg, melyekben rendre 1 szerver és 1 kliens, 1 szerver és 2 kliens, valamint 1 szerver és 3 kliens szerepelt. A kliens(ek) a kontextualizáció során a szervertől függenek, a szerveren található adatbázisba írnak.

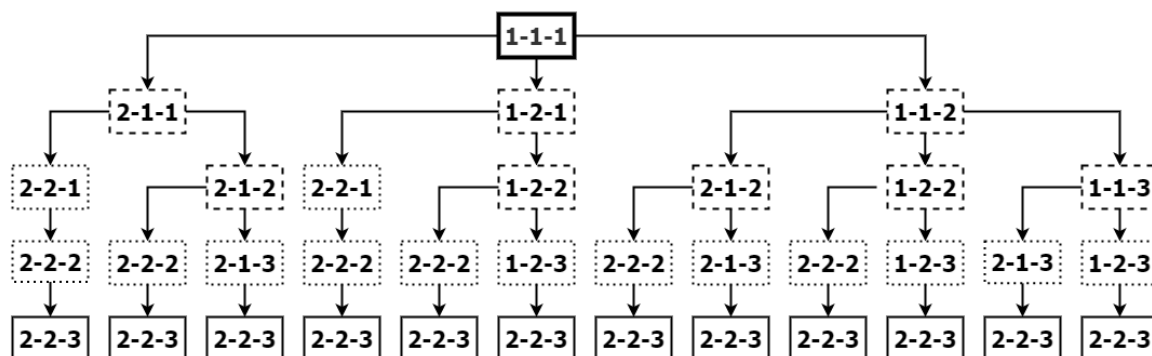
A teszteléshez az *Amazon EC2* [16] szolgáltatását vettem igénybe *t2.micro* példányokkal, a felhasznált felhő-orkesztrátorok az *Occopus 1.10*, valamint a *Terraform 1.1.6*-os verziója voltak. A virtuális gépeken *Ubuntu 18.04* operációs rendszer futott, a kontextualizációt a Cloud-init segítségével végeztem, a lokális töréspontokat megvalósító script-ek is itt kaptak helyet. A szerveren és a klienseken egy *MariaDB 10.1.48*-as [17] verziójú SQL adatbázis (szerver és kliens) került telepítésre és használatra.

Az 5.3.1.-es ábra a végbemenő kontextualizációs folyamatokról ad egy szemléltető képet a három darab virtuális gépet tartalmazó infrastruktúra esetében, míg az 5.3.2.-es ábra a létrejövő végrehajtási fát szemlélteti egy teljes bejárás után, 3 darab virtuális gép használatakor. A tesztinfrastruktúrák esetében a lehetséges végrehajtási útvonalak száma

rendre 3, 12 és 60 volt. A kliensek kontextualizációja kettő, a szerveré három darab lokális töréspontot tartalmazott.



5.3.1. ábra: A három darab virtuális gépet tartalmazó tesztinfrastruktúra kontextualizációs folyamatai.



3 darab virtuális gép

cl1: 2 lokális töréspont
cl2: 2 lokális töréspont
srv: 3 lokális töréspont

(cl1 bp. #) - (cl2 bp. #) - (srv bp. #)



Gyökércsomópont



Alternatív csomópont



Determinisztikus csomópont

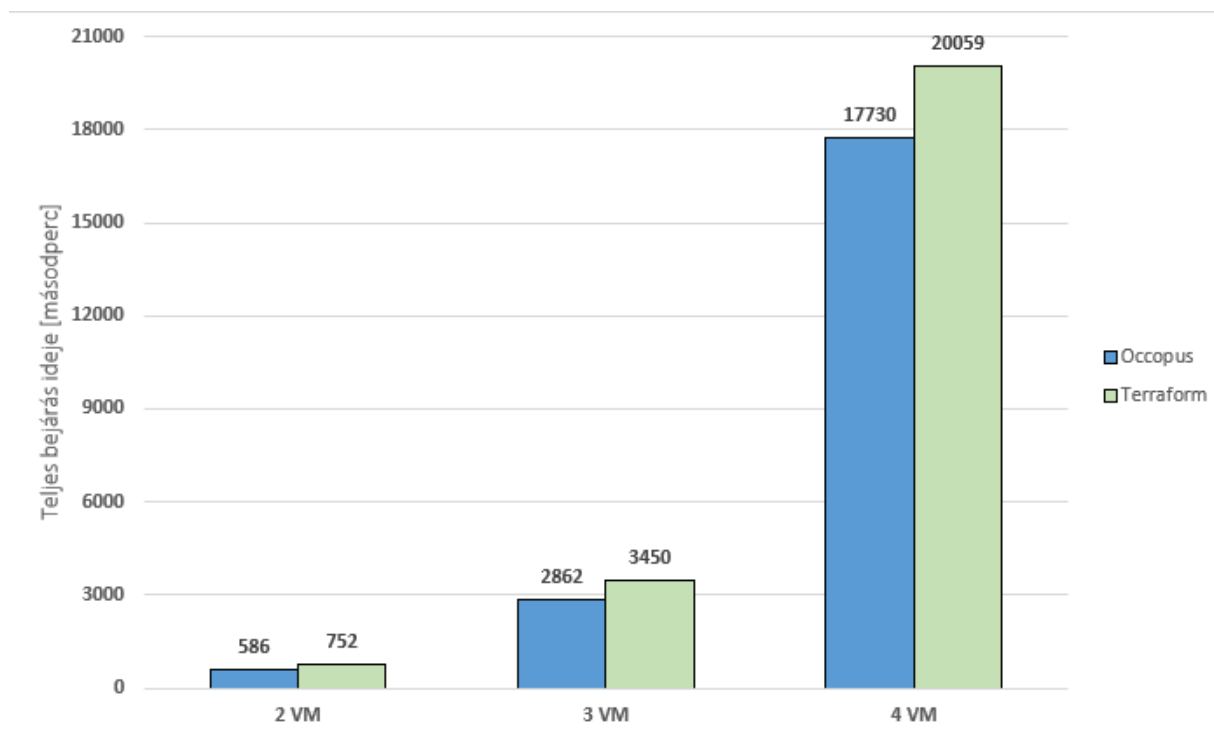


Utolsó/végső csomópont

5.3.2. ábra: A három darab virtuális gépet tartalmazó infrastruktúrához tartozó teljes végrehajtási fa, a lehetséges 12 darab végrehajtási útvonallal.

6.1. Automatikus bejárással kapcsolatos mérések

Az első méréseket a tesztinfrastruktúrák automatikus bejárásával kapcsolatban végeztem, először Occopus, utána pedig Terraform segítségével. A tesztinfrastruktúrák felépítése, a lezajló kontextualizációs folyamatok mindkét felhő-orkesztrátor esetében megegyeztek, pusztán az alkalmazott felhő-orkesztrátorok és a virtuális gépek száma jelentették a különbséget. A 6.1.1. ábrán a végrehajtási fák aktív kontroll alapú, teljes bejárásával kapcsolatos mérési eredmények láthatók.



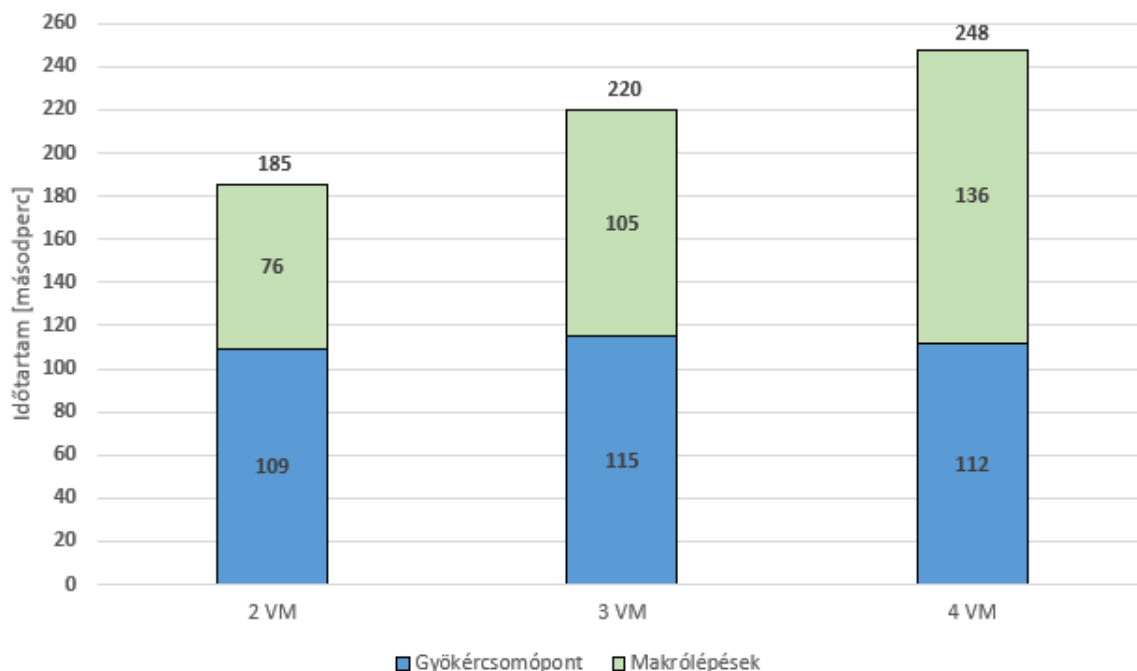
6.1.1. ábra: Teljes bejárás ideje az egyes tesztinfrastruktúrák esetében, különböző orkesztrátorok használatakor.

A 6.1.1. ábrán jól észrevehető a végrehajtási fák bejárásához szükséges idő exponenciális növekedése, mely összhangban van az egyes tesztinfrastruktúrák esetében a lehetséges végrehajtási útvonalak növekvő számával (rendre 3, 12 és 60 darab útvonal).

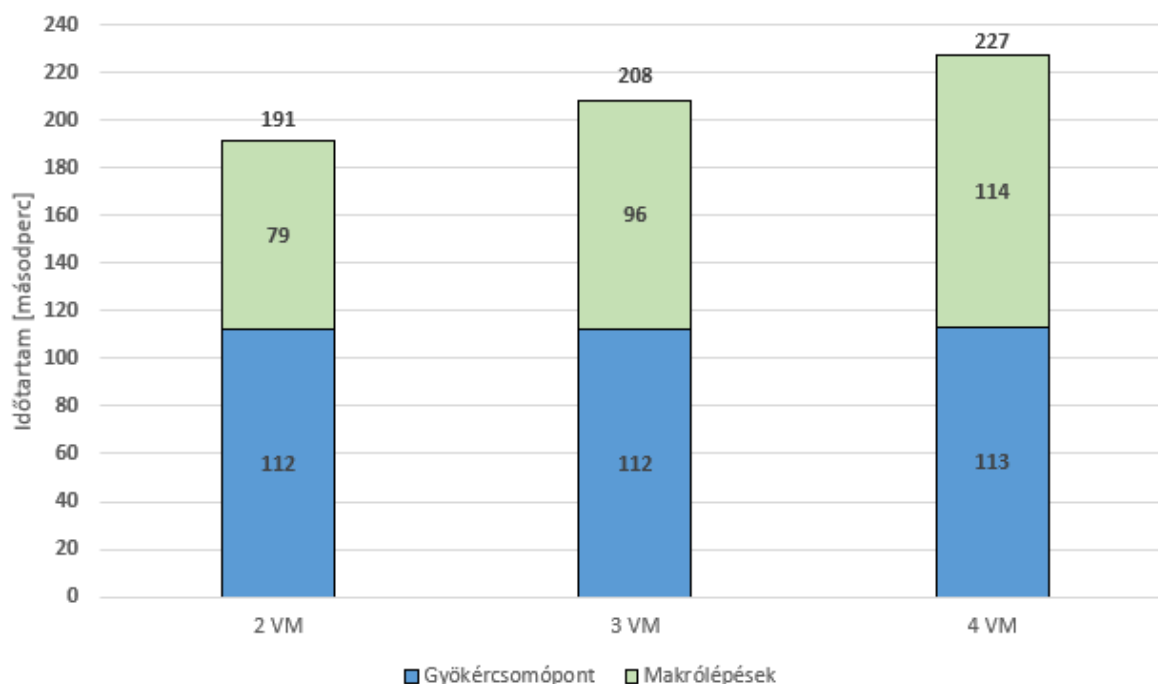
Az ábráról továbbá leolvasható, hogy Terraform alkalmazása esetén rendre lassabban történik meg a teljes állapottér (végrehajtási fa) bejárása, mint Occopus esetében. Szembetűnő lehet, hogy még egy egyszerűbb, mindössze 4 darab virtuális gépet, összesen 9 darab lokális töréspontot tartalmazó infrastruktúra teljes tesztelése is több, akár 5 órát is igénybe vehet.

A 6.1.2. és 6.1.3. ábrákon az Occopus és Terraform esetében a makrólépések összesített idejére vonatkozó mérési eredmények láthatók. Látható, hogy mindkét felhő-orkesztrátor esetében közel hasonló ideig tart egy-egy infrastruktúra példány során a makrólépésenkénti

végrehajtás, kivéve a 4 darab virtuális gépet tartalmazó infrastruktúra esetében. Mindkét orkesztrátor esetén közel azonos ideig tart az infrastruktúra kiépítésének kezdetétől a gyökércsomópont eléréséig eltelő időtartam is. Ez az elvárásoknak megfelelő, hiszen a felhő-orkesztrátorok csakis az infrastruktúra(példányok) kiépítésének megkezdéséért felelősek, a tényleges kontextualizációs folyamatok a Cloud-init fájlokban foglaltak alapján történik, azok pedig mind Occopus, mind pedig Terraform használata esetében azonos lépéseket (és gyakorlatilag azonos kódot) tartalmaznak.



6.1.2. ábra: Átlagos infrastruktúra képülési időtartamok Occopus használata esetén.



6.1.3. ábra: Átlagos infrastruktúra képülési időtartamok Terraform használata esetén.

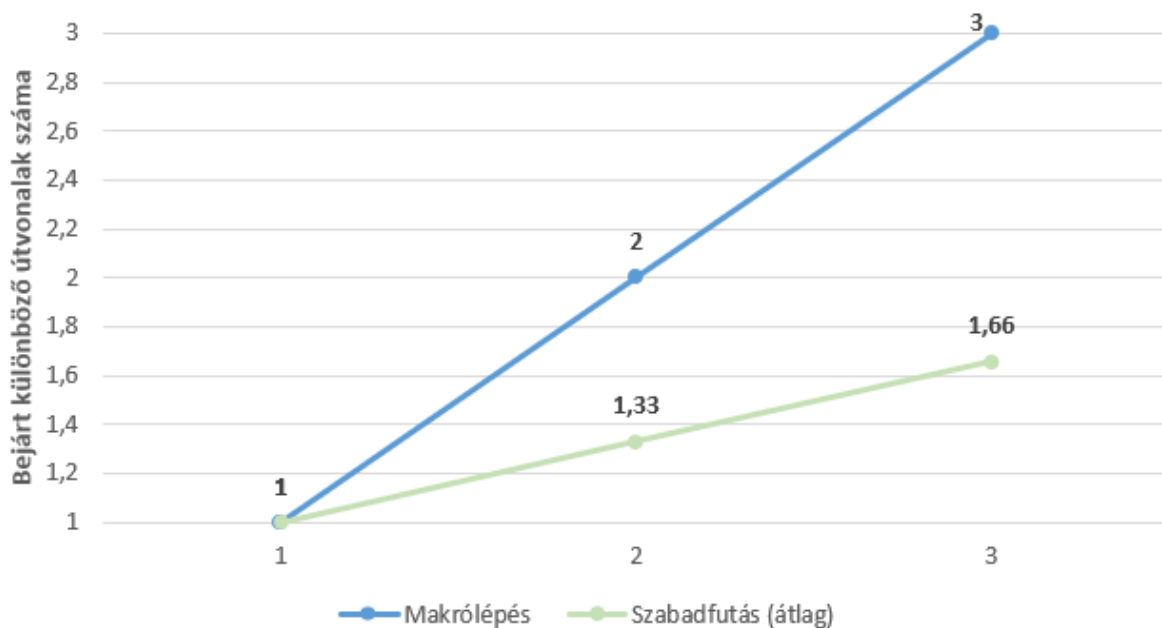
A 6.1.1., a 6.1.2. valamint a 6.1.3. ábra alapján adja magát a kérdés, hogy mi okból lehetséges, hogy mégis közel fél óra különbség van a 4 virtuális gépet tartalmazó infrastruktúra teljes végrehajtási fájának bejárása esetén Occopus és Terraform használatakor. Valószínűsíthetően az egyes infrastruktúra-példányok lebontása során lassabb a Terraform, ez pedig 60 darab végrehajtási útvonal során több mint 30 perc idővesztést eredményezett az Occopus-szal szemben.

6.2. Szabadfutásokkal kapcsolatos mérések

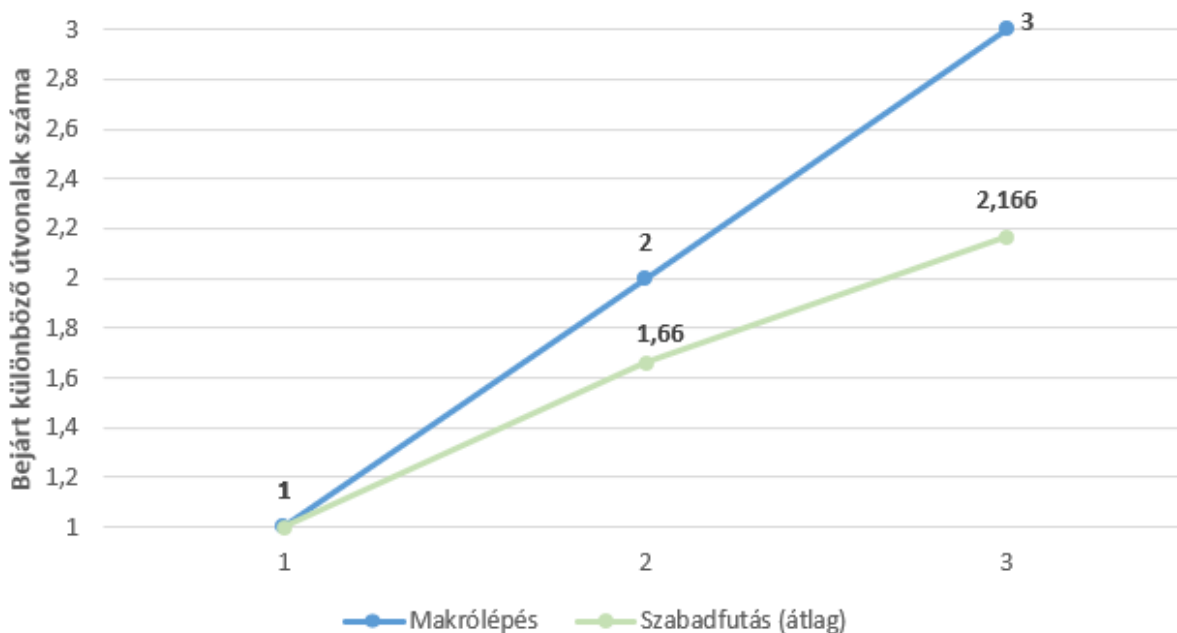
További mérésként szabadfutásokat is végeztem a 2 és 3 virtuális gépet tartalmazó infrastruktúrák esetében úgynevezett „free-run” üzemmódban. Ebben az esetben a virtuális gépek nincsenek gátolva, egy töréspont elérésekor automatikusan folytatódhat a kontextualizációs folyamatuk. Ebben az üzemmódban a vezérlő csupán a töréspontok elérését regisztrálja, majd ebből visszafejthető, hogy az infrastruktúrában szereplő gépek milyen sorrendben (időzítésben) érték el töréspontjaikat. A szabadfutásokat, hasonlóan az előző méréseknél, mind Occopus, mind Terraform használatával is elvégeztem.

A csak kettő darab virtuális gépet tartalmazó infrastruktúra esetében összesen 36 darab szabadfutás történt, 18 Occopus-szal, 18 Terraform-mal. Az egyes szabadfutás sorozatokat

hat-hat egyenlő részre osztottam. Ez azt jelenti, hogy hat darab 3 futást tartalmazó rész esetében néztem a bejárt útvonalak számát, mindkét orkesztrátor esetében.



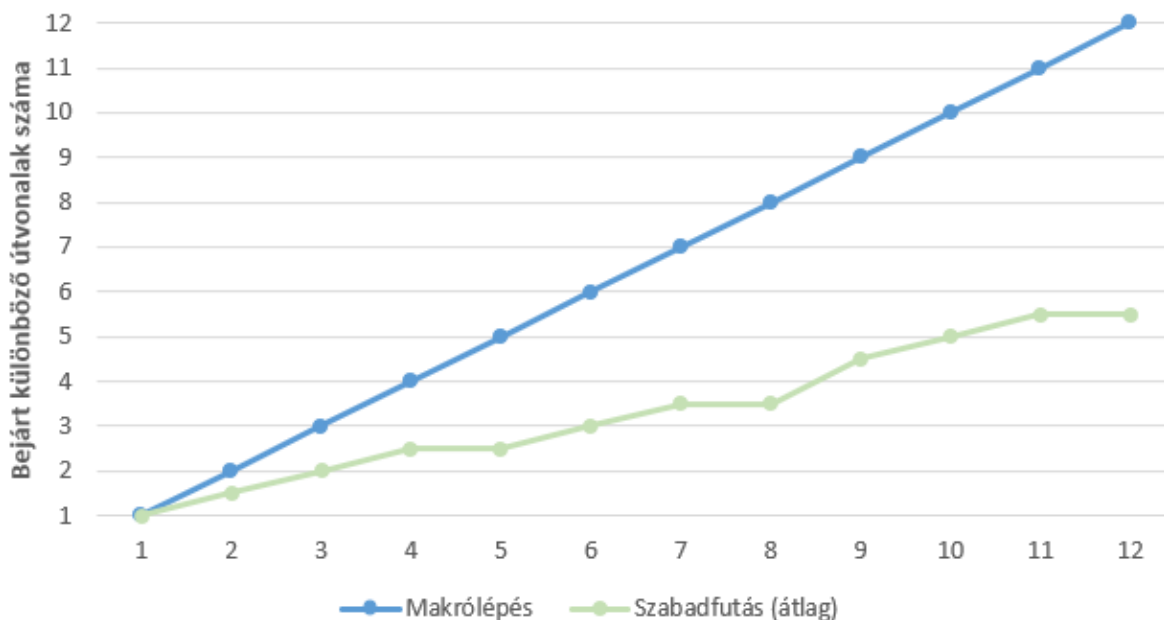
6.2.1. ábra: Bejárt útvonalak száma három egymás utáni futást tekintve, Occopus esetében.



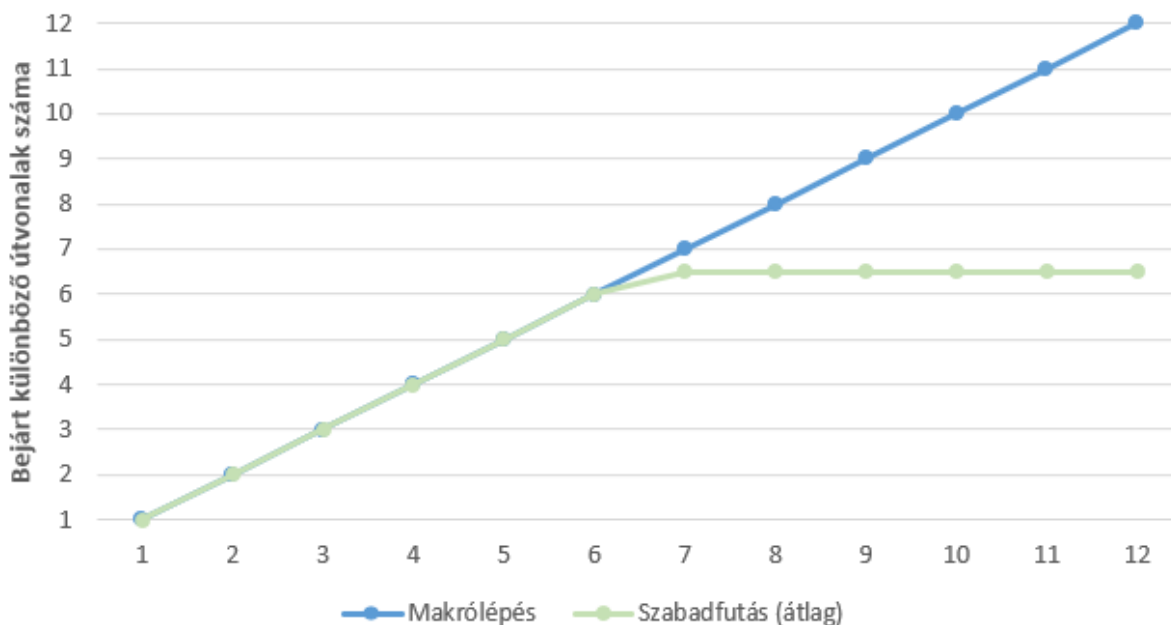
6.2.2. ábra: Bejárt útvonalak száma három egymás utáni futást tekintve, Terraform esetében.

A 6.2.1. és 6.2.2. ábrák alapján látható, hogy a *Makrólépés* vezérlő minden egyes új infrastruktúrapéldány esetében egy új útvonalat járt be, szemben a szabadfutásokkal. A makrólépés-alapú hibakeresés hatékonyságát talán még jobban szemlélteti a 6.2.3 és 6.2.4

ábrák, melyek a 3 darab virtuális gépet tartalmazó infrastruktúra esetében történt szabadfutásokat hasonlítják össze az aktív vezérléssel. Ebben a konfigurációban összesen 48 darab szabadfutás történt, 24 Occopus, 24 Terraform segítségével.



6.2.3. ábra: Bejárt útvonalak száma 12 egymás utáni futást tekintve, Occopus esetében.



6.2.4. ábra: Bejárt útvonalak száma 12 egymás utáni futást tekintve, Terraform esetében.

A 6.2.3. és 6.2.4. ábrák alapján újra jól látható a makrólépés-alapú hibakeresés esetén használt aktív kontroll elsődleges előnye, vagyis, hogy a vezérlő 12 egymás utáni infrastruktúra példány esetében mindig új útvonalat volt képes bejárni, ezzel a teljes

végrehajtási fát lefedni. Szabadfutások esetében ez Occopus használatakor 12 futás után átlagosan csak 5,5, míg Terraform esetében átlagosan 6,5 útvonal került bejárásra.

7. Összegzés, továbbfejlesztési lehetőségek

A dolgozatban bemutatásra került egy korábban már bevált hibakereső eljárás, valamint annak felhő-infrastruktúrák esetében alkalmazható kísérleti prototípusa, annak komponensei és működése. Bemutatásra került a manuális és aktív kontroll menete, a visszajátszások, illetve mérések során bemutatásra került az aktív kontroll-alapú hibakeresés legfontosabb előnye.

A kísérleti rendszer viszont rendelkezik hiányosságokkal is, ezek közül az egyik, hogy a lokális töréspontok elhelyezése jelenleg kézzel történik. Ez a nagyobb, akár ezer vagy több ezer soros Cloud-init fájlok esetében kényelmetlenné teszi a felhasználók munkáját. Ebből fakadóan minél hamarabb szükséges lenne egy automatikus eljárás a töréspontok generálására a Cloud-init fájlokban.

A továbbiakban lehetőség adódhat a rendszer oly módú továbbfejlesztésére is, melynek köszönhetően a hibás állapotokat vagy a vezérlő, vagy egy megfelelő eszköz jelzi a felhasználó számára. Lehetőség adódhat még gépi tanulás alapú útválasztásra, mely segítségével el is lehet kerülni a hibás esetek, időzítések egy részét. Ez jelenleg egy OTKA kutatási projekt keretében vizsgálat alatt áll [18].

8. Köszönetnyilvánítás

Szeretnék köszönetet mondani Dr. habil. Lovas Róbert egyetemi docensnek, illetve Dr. Kovács Józsefnek, a SZTAKI tudományos főmunkatársának. A bemutatott eredmények részben az MTA Bolyai János Kutatási Ösztöndíjának, részben a Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal (NKFIH), OTKA K 132838 számú szerződésének, valamint az Innovációs és Technológiai Minisztérium ÚNKP-20-5 kódszámú Új Nemzeti Kiválósági Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.

Hivatkozások

- [1] **R. Lovas:** "Orchestrated service deployment, maintenance, and debugging in IaaS clouds for crowd computing", International Conference BOINC:FAST 2015 Proceedings of the Second International Conference BOINC-based High Performance Computing: Fundamental Research and Development (BOINC:FAST 2015), **2015**, pp. 70-80.
- [2] **P. Kacsuk, J. Kovács, R. Lovas:** "Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps", Euro-Par 1999. Lecture Notes in Computer Science, **1999**, pp. 90-97.
- [3] **P. Kacsuk, J. Kovács:** "The DIWIDE distributed debugger", Parallel and Distributed Computing Practices, Vol. 4. (4. szám), **2001**, pp. 331-347.
- [4] **Microsoft Azure:** <https://azure.microsoft.com/>, utoljára megtekintve: 2022.04.17.
- [5] **Google:** "Cloud Debugger", <https://cloud.google.com/debugger>, utoljára megtekintve: 2022.04.17.
- [6] **M. S. Quroush, T. Ovatman:** „A Record/Replay Debugger for Service Development on the Cloud”, *Cloud Computing and Services Science*, pp. 64-76., **2019**
- [7] **P. Sharma, S. Chatterjee, D. Sharma:** „CloudView: "Enabling tenants to monitor and control their cloud instantiations", Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Gent, Belgium, **2013**, pp. 443–449.
- [8] **Occopus:** <https://occopus.readthedocs.io/en/latest/>, utoljára megtekintve: 2022.04.16.
- [9] **HashiCorp:** „Terraform”, <https://www.terraform.io/>, utoljára megtekintve: 2022.04.16.
- [10] **Amazon:** „Amazon Web Services”, <https://aws.amazon.com/>, utoljára megtekintve: 2022.04.16.
- [11] **OpenStack:** <https://www.openstack.org/>, utoljára megtekintve: 2022.04.16.
- [12] **P. Kacsuk, J. Kovács:** "Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures", *Journal of Grid Computing*, Vol. 16. (1. szám), **2018**, pp. 19-37.
- [13] **Cloud-Init:** „The Standard for Customising Cloud Instance”, <https://cloud-init.io/>, utoljára megtekintve: 2022.04.17.
- [14] **Neo4j:** „The Neo4j Graph Platform – The #1 Platform for Connected Data”, <https://neo4j.com/>, utoljára megtekintve: 2022.04.17.
- [15] **M. Owens:** "The Definitive Guide to SQLite", Apress: Berkeley, CA, USA, **2006**
- [16] **Amazon:** „Amazon EC2”, <https://aws.amazon.com/ec2/>, utoljára megtekintve: 2022.04.15.
- [17] **MariaDB Foundation:** „MariaDB Server: The open source relational database”, <https://mariadb.org/>, utoljára megtekintve: 2022.04.05.
- [18] **B. Ligetfalvi, M. Emódi, J. Kovács, R. Lovas:** „Fundamentals of a Novel Debugging Mechanism for Orchestrated Cloud Infrastructures with Macrosteps and Active Control”, *Electronics*, 10. kötet, 24. szám, **2021**