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

Privát felhők szolgáltatásbiztonságra optimalizálása és modell-vezérelt terítése

TDK-dolgozat

Készítette:

Zilahi Dávid

Konzulens:

Kocsis Imre

2015.

Tartalomjegyzék

Kivonat

A több szolgáltatásból álló elosztott rendszerek tervezése és telepítése jelentős kihívások elé állítja a rendszermérnököket. Ez a probléma hangsúlyosan jelentkezik a privát felhők világában, különösen a sokoldalú és népszerű privát felhő keretrendszer, az OpenStack esetében. A nagyszámú szolgáltatás, konfigurációs lehetőség és a változatos terítési elrendezések mellett nehéz megfelelő tervezési döntéseket hozni. A szituációt tovább bonyolítja, hogy a szolgáltatások egymásra épülnek, ezért a beállításaiknak minden esetben konzisztensnek kell lenniük egymással. Ezt a problémát kiküszöböli az OpenStacket automatikusan telepítő keretrendszerek alkalmazása (pl.: Fuel, TripleO), ezek közül azonban egyik sem támogatja a megvalósítandó konfiguráció modelljének MDD folyamat részeként létrehozását, illetve elemzését.

Az ilyen komplex rendszerekben a rendelkezésreállás megvalósítása jelenti az egyik legnagyobb nehézséget. Meg kell határozni a szolgáltatásbiztonság szempontjából releváns részszolgáltatásokat és az azokhoz megfelelő szolgáltatásbiztonsági mechanizmusokat, majd megtalálni a helyes telepítési elrendezést az elvárt felhő teljesítmény és rendelkezésreállás biztosítására. A konfiguráció funkcionális és hibatűrési teszteléséhez el kell végezni a telepítést, ami a jelenlegi technológiákkal időigényes feladat, főleg ha a terveket iteratívan javítani kell.

Munkám során az OpenStack példáján keresztül bemutatom, hogy a modellvezérelt fejlesztés eszközkészletének néhány alapvető technikáját adaptálva miként lehet megoldani ezeket a problémákat. Központi elemként létrehoztam egy úgynevezett konfigurációs tér metamodellt, mellyel leírhatók az OpenStack mint keretrendszer elemei, ezek lehetséges beállításai, valamint a közöttük húzódó kapcsolatok és függőségek.

Kidolgoztam és implementáltam egy transzformációt, mely egy konfigurációs tér modellből (pl. egy konkrét OpenStack verzió) egy olyan metamodellt generál, mely segítségével konkrét OpenStack telepítéseket lehet leírni és validálni. Műszakilag új eredményként a legfontosabb OpenStack részszolgáltatásokra megvalósítottam egy konkrét konfiguráció modelleket Docker konténerekben automatikusan megvalósító mechanizmust.

A konfigurációs modell által meghatározott, teljesítendő szolgáltatásbiztonsági igények (például aktív-aktív replikációk) és egyéb optimalizációs kritériumok (pl. robusztus particionálás, költség) kielégítésének támogatására felállítottam és implementáltam egy lineáris programozási modellt, mely a konfigurációs modellből kiindulva a részszolgáltatáspéldányokat futtató gépekhez rendel. A hibatűrési mechanizmusok kiválasztására bemutatok egy kezdeti kockázatvezérelt metodológiát.

Abstract

Design and installation of large distributed systems pose challenges for system architects. This is especially true in case of private clouds, including the popular versatile private cloud framework, OpenStack. The large number of services, huge variety of configuration options and many deployment possibilities make design decisions very difficult. The situation is further complicated by services that depend on each other, so the settings must always be consistent between services.

In such complex systems, the realization of resilience is a major challenge. Services with relevance in the system-wide reliability have to be selected, and appropriate redundancy mechanisms have to be chosen. Complex deployment constraints also have to be enforced to reduce the impact of a failure and improve recovery. Testing of a configuration can be extraordinarily tiresome, because one has to install the whole system.

Through the example of OpenStack I present how model driven approach can be used to solve the aforementioned challenges. I created a configuration space metamodel to describe domain specific knowledge regarding OpenStack, including objects of the system, their settings, relations and dependencies. This model can be used to generate a metamodel to describe and validate deployments according to the configuration space. Deployment models can be realized with Docker containers; herewith practically rapid-prototyping the configuration, These prototypes can then be used for testing.

An appropriate deployment can be calculated with multi-aspect optimization by utilizing linear programming. This method also holds the resource constraints, the constraints originated from dependability mechanisms and also the rules of robust partitioning. Optimization aspects can be dependability and resource management parameters.

1. fejezet

Felhő keretrendszerek konfigurációmenedzsmentjének kihívásai

A számítási felhő napjainkban az informatika egyik legdinamikusabban fejlődő ágazata, mely lehetőséget teremt a felhasználók számára, hogy a lehető leggyorsabban és legegyszerűbben vehessenek igénybe informatikai erőforrásokat. Ezek lehetnek alacsony szintű erőforrások, mint számítási- vagy tárolókapacitás, virtuális gépek és hálózatok, de lehetnek magas szintűek is, mint például egy terheléselosztó vagy egy adatbázis.

A publikus felhő szolgáltatások mellett a nyílt forráskódú felhő keretrendszerek egyre érettebbé válásával mindinkább igaz az, hogy egy szervezet belső használatra vagy értékesítés céljára könnyen tud meglevő fizikai eszközein felhő szolgáltatást kialakítani. Szoftvertechnológiai értelemben ezek a keretrendszerek meglehetősen bonyolult felépítésűek, és rengeteg konfigurációs beállítással rendelkeznek. Ennek megfelelően telepítésük és üzemeltetésük ma jellemzően vagy szakértő által, vagy további dedikált szoftvereszközök igénybevételével történik.

Ennek azonban hátránya, hogy költséges a "jellemző" konfigurációktól való eltérés; a felhő tulajdonosának igényei és a rengeteg alacsony szintű beállítás között nincsen formális kapcsolat. Probléma az is, hogy akár szakértő, akár eszköz telepíti illetve kezeli a felhő szolgáltatást megvalósító keretrendszer konfigurációját, annak a minősége nem értékelhető ki könnyen (például szolgáltatásbiztonság szempontjából), mivel a konfigurációról vagy nem létezik nyílt modell, vagy az túlzottan is egy telepítő eszközhöz kötött.

Dolgozatomban a modellvezérelt tervezés (MDD - Model Driven Design) filozófiáját a felhő környezetek konfigurációmenedzsmentjére alkalmazva megmutatom, hogy ezek a hiányosságok orvosolhatóak (1.1. ábra). Specifikusan,

1. Felállítok egy többszintű modellezési megközelítést az OpenStack felhő keretrendszer konfigurációmodellezésére.

2. Bemutatom a modellezési megközelítést támogató transzformációs eszközkészletet.

3. Példaimplementációval bemutatom, hogy miként lehet a modellezési megközelítést OpenStack környezetek automatizált telepítésének támogatására alkalmazni.

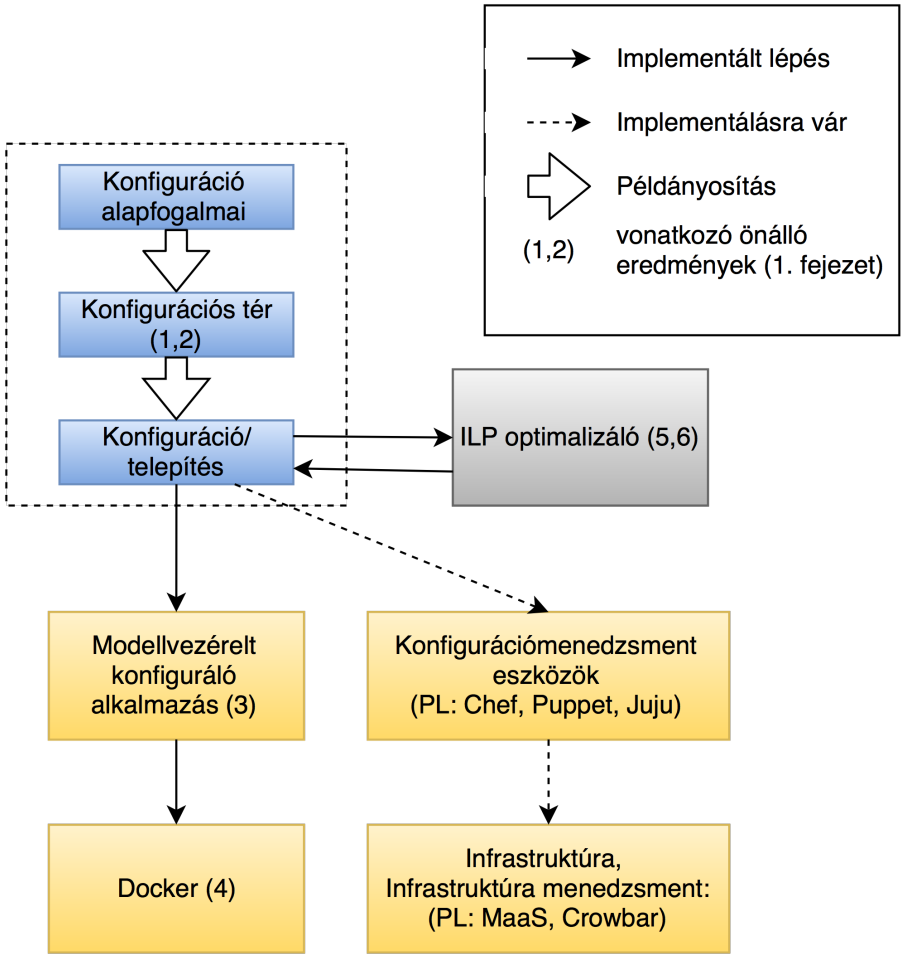
4. A telepítési implementáció Docker konténerekbe helyezi el az OpenStack szolgáltatásokat, ami önmagában egy újszerű műszaki eredmény. Diszkutálom, hogy megközelítésem hogyan lesz képes a jövőben egyéb telepítőeszközök használatát támogatni.

5. Az MDD "hidden formal methods" mintáját követve bemutatom, hogy az OpenStack szolgáltatások fizikai gépekhez rendelésének több aspektusú optimalizálása jól

támogatható egészértékű lineáris programozási problémára (ILP) transzformációval, és az eredmények visszaannotálásával.

6. Az általam kialakított lineáris programozási megközelítés praktikusan támogatja a OpenStack konfigurációk költsége és meghatározó hibatűrési jellemzői (például robusztus particionálás) között az optimum megtalálását.

Dolgozatom bevezetőjében a továbbiakban részletesebben bemutatom a megoldott problémákat, és azok kontextusát. A második fejezet tárgyalja az OpenStack konfigurációk modellezését, és modellvezérelt telepítését. A harmadik fejezet a modell alapján ILP probléma szintetizálását, és megoldását mutatja be. Dolgozatom a számos továbbfejlesztési lehetőség áttekintésével zárul.



1.1. ábra. A dolgozat eredményeit összefoglaló ábra

1.1. Számítási felhők

A dolgozat kontextusát a számítási felhők adják, így jelen alfejezetben röviden összefoglalom hogy milyen szolgáltatásokat hívunk felhő szolgáltatásnak, és megadom ezek egy általánosan használt taxonómiáját.

A számítási felhő definiálásában az Amerikai Egyesült Államok szabványügyi hivatalának (NIST) [28] definícióját használjuk (1.2. ábra). Ez alapján a felhő szolgáltatások főbb jellemzői:

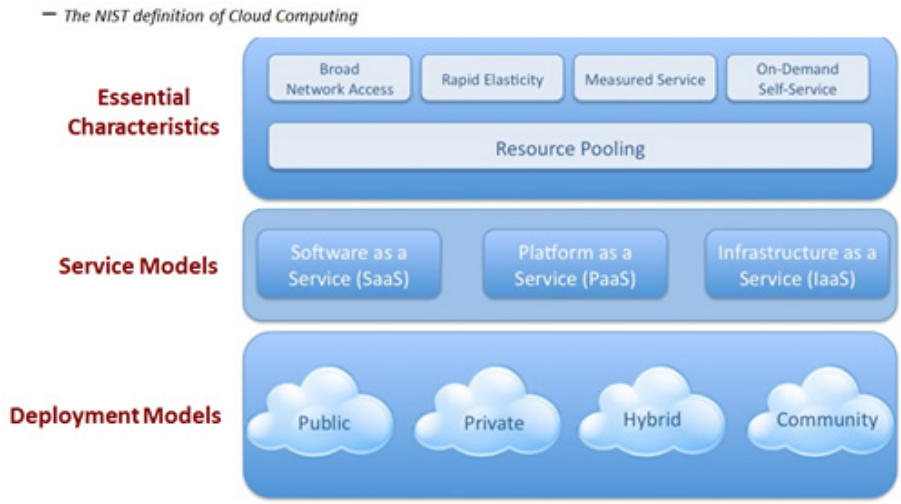
Közvetlen és önkiszolgáló: A fogyasztó saját igényeinek megfelelően foglalhat erőforrásokat, melyek lehetőség szerint azonnal rendelkezésre állnak. Az erőforrások menedzselése programozott, és emberi interakciót nem igényel.

Hálózati hozzáférés: Az erőforrások és azok igénylése hálózaton keresztül, elterjedt és szabványos mechanizmusok alkalmazásával történik.

Erőforrás aggregáció A szolgáltató informatikai erőforrásokat halmoz fel, majd a felhasználói igények változásának megfelelően automatikusan hozzárendeli, visszavonja vagy átcsoportosítja azokat. A felhasználó elől rejtve van az erőforrás valódi helye, és az alatta meghúzódó technológia részletei.

Gyors alkalmazkodás: A felhasználói igények gyorsan változhatnak, hiszen az általuk futtatott alkalmazásokhoz rendelt erőforrások mennyiségét a terheltségnek megfelelően skálázhatják. A számítási felhőknek úgy kell követniük ezen változásokat, hogy a felhasználók számára mindig megfelelő mennyiségű erőforrás álljon rendelkezésre, ugyanakkor az erőforrásokról szabadon le is mondhassanak. A fentiek miatt a legtöbb felhő szolgáltatás esetében úgy látja a felhasználó, mintha a szolgáltató végtelen erőforrással rendelkezne.

Mért szolgáltatás: A felhő rendszerek menedzselik az erőforrások kiosztását, így lehetőség nyílik kihasználtsági metrikák gyűjtésére. Ezen adatok segítségével valódi felhasználás alapú számlázás valósulhat meg, melyben a vásárló nyomon követheti és irányíthatja költségeit.

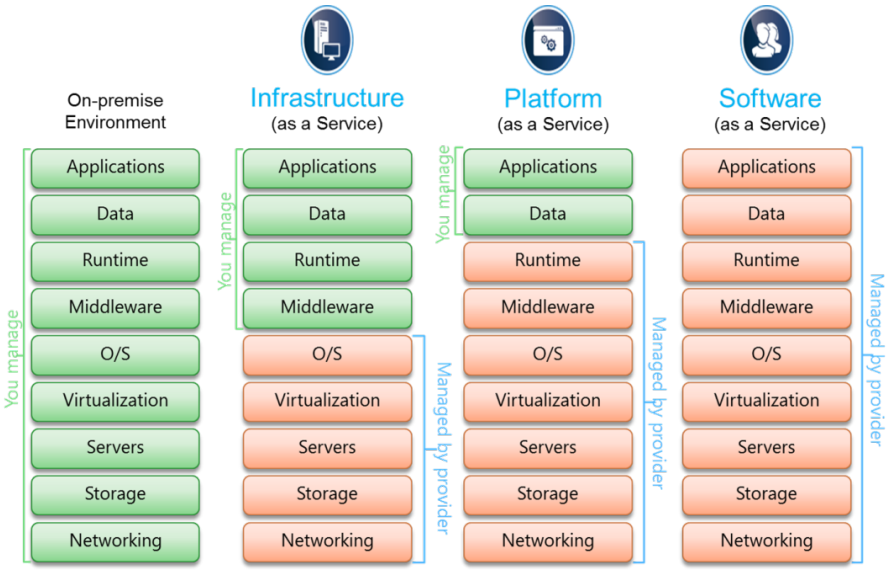


1.2. ábra. A felhő szolgáltatások csoportosítása [6]

Több kiszolgálási modell létezik, melyek azt hivatottak meghatározni, hogy a szolgáltató milyen üzemeltetési feladatokat lát el, és mik azok a feladatok, melyeket a megrendelőnek kell elvégeznie. A feladatokat az 1.3. ábra mutatja be részletesen.

Az infrastruktúra szolgáltatás (IaaS - Infrastructure-as-a-Service) esetében szolgáltató infrastrukturális elemeket nyújt a megrendelőnek, például virtuális gépeket, hálózatokat vagy tárolókat, aki ezeken tetszőleges alkalmazásokat futtathat. Az erőforrásokat létrehozó mechanizmusokhoz nem fér hozzá, de az erőforrásokkal szabadon gazdálkodhat, tehát az üzemeltetési feladatok neki kell elvégeznie.

Amennyiben a szolgáltató egy keretrendszert bocsájt a megrendelő rendelkezésére, melyben alkalmazásait futtathat, azt platformszolgáltatásnak (PaaS - Platform-as-a-Service) nevezzük. Az alkalmazások csak olyan programozási nyelveket, eszközöket és



1.3. ábra. Összefoglaló ábra a szolgáltatási szintekről [38]

szolgáltatásokat használhatnak, melyeket az adott felhő platform támogat. A megrendelő csak az alkalmazás üzemeltetéséért felel, az infrastruktúrát és a keretrendszert a szolgáltató tartja üzemben.

A legmagasabb szolgáltatási szint a szoftverszolgáltatás (SaaS - Software-as-a-Service) melyben a szolgáltató egy interneten elérhető alkalmazás használatára ad lehetőséget. A megrendelőnek korlátozottan van lehetősége az alkalmazás beállításainak módosítására, illetve a felhasználható erőforrások, például tárhely bővítésére, ugyanakkor az üzemeltetést teljes egészében a szolgáltató végzi.

Csoportosíthatjuk a felhőket aszerint, hogy a szolgáltatásokat ki érheti el. A privát felhők csak egy szervezet és annak alszervezetei számára elérhetők. A tulajdonos és az üzembentartó is lehet harmadik személy, és nem feltétlenül kell a szervezet telephelyén elhelyezkednie. Közösségi felhőről akkor beszélhetünk, ha több szervezet közösen használ egy felhőt, mert hasonló célok vagy biztonsági követelmények kedvezővé teszik. A publikus felhők szolgáltatásai széles publikum által igénybe vehetők. A felhő tulajdonosa legtöbbször egy erre specializálódott vállalat, ugyanakkor lehet akadémiai vagy állami is. A hibrid felhő megoldás több különálló felhő együttműködését jelenti, melyeket olyan technológiák kötik össze, melyek lehetővé teszik az átjárhatóságot az alkalmazások és adataik számára. Tipikus esete, amikor egy vállalat a privát felhője mellett egy publikus felhő szolgáltatásait is igénybe veszi.

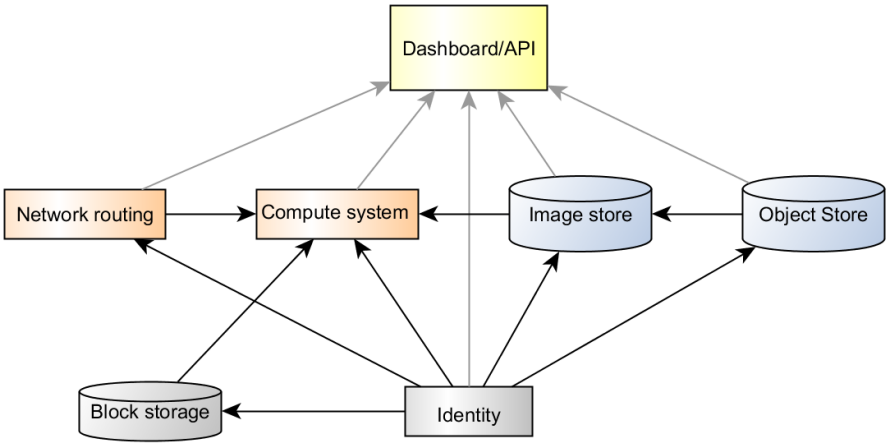
1.2. Egy meghatározó felhő keretrendszer: OpenStack

Az OpenStack [33] egy népszerű nyílt forráskódú felhő keretrendszer. Lazán csatolt komponensekből áll, melyek egy-egy funkciót valósítanak meg, mint például a számítási kapacitás vagy a hálózati kapcsolatok felügyelete. A komponensek részfeladatokat megvalósító szolgáltatásokból állnak. Ezek a tényleges szoftverelemek, melyek együttműködve hozzák létre a komponens funkcionalitását. Az OpenStack elsődleges célja az infrastruktúra szolgáltatáshoz (IaaS) szükséges funkciók megvalósítása, de a platform jellegű komponensek száma is növekszik.

A komponensek többsége egy-egy erőforrás menedzseléséért felel. A szolgáltatások REST API [18] felületen keresztül érhetők el a többi komponens és a felhasználók számá-

ra. Az API-k kezeléséhez használhatók specifikus parancssoros eszközök és egy összevont webes felület is. A komponensek egymás közötti kommunikációra AMQP [45] szabványú üzenetsort is alkalmaznak, többnyire eseményekhez kapcsolódó értesítésekhez. Az 1.4. ábrán látható egy példa a komponensek együttműködésére.

Az OpenStack komponensei többnyire nem oldanak meg önállóan technológiai feladatokat, hanem már ismert és bevált eszközöket használnak fel. A komponensek általában több különböző háttér-technológia felhasználására is képesek konfigurációtól függően.



1.4. ábra. Az OpenStack komponensek, és a közöttük zajló kommunikáció egy virtuális gép elindításakor

Tekintettel az openstack komponenseinek és az azokat alkotó szolgáltatások nagy számára, dolgozatomban csak a komponensek egy részét modellezem. Ez a részhalmaz egy működőképes felhő szolgáltatás megvalósításához szükséges minimális funkciókészletet alkot, reprezentatív azonban abban az értelemben, hogy mind a modellezés, mind az automatizált telepítés, mind pedig a konfiguráció optimalizálás tekintetében a további komponensek bevonása nem mutat túl az itt bemutatott minták alkalmazásánál.

Jelen munkában tekintetbe vett komponensek

Keystone: Ez a komponens egy speciális katalógus szolgáltatást valósít meg, mely egyben felel a felhasználók és szolgáltatások hitelesítéséért és a hozzáférési szabályok érvényesítéséért is. Nemcsak a felhasználók és csoportok adatait tárolja, hanem a hozzáférési szabályokat és a szolgáltatások elérhetőségeit is. A tárolás általában MySQL adatbázisban történik, de képes egyéb technológiák, például LDAP [51] használatára is.

Nova: A Nova a számítási erőforrások menedzseléséért felelős. Bár a számítógép példányok egy közös interfészen érhetők el, a háttérben számos virtualizációs technológia lehet, például Xen [50], KVM, VMWare ESX [47] és Microsoft Hyper-V [46] is. Léteznek olyan megoldások is, melyekkel virtuális gépek helyett valódi fizikai számítógépeket közvetít a komponens.

Neutron: Ez a komponens felelős a számítási egységek közötti hálózati kapcsolatok kialakításáért. A felhasználók számára lehetővé teszi virtuális hálózatok, útválasztók, tűzfalak és terheléselosztók dinamikus létrehozását számítási egységeik számára.

Glance: Amikor a Nova elindít egy számítási egységet, például egy virtuális gépet, szükség van egy előre telepített környezetre, a rendszerképre, ami tartalmazza az ope-

rációs rendszert és a szükséges programokat. A Glance komponens egy rendszerkép könyvtárat alkot, melyből kiválaszthatják a felhasználók, hogy melyik képből indítson a Nova új számítógépet. A komponens a felhasználók felé, is szolgáltat, hiszen egyedi rendszerképeket is feltölthetnek. A használt tárolási technológiák

További komponensek

Az OpenStack keretrendszer természetesen a fentieken kívül számos más komponenssel rendelkezik, melyek az infrastrukturális képességeken platform szintűeket is megvalósítanak.

A Cinder komponens blokk alapú tárolást valósít meg a felhőben, míg a Swift objektum tárat implementál. A Horizon nevű szoftver egy webes felhasználói felület, mely segítségével a felhasználók kényelmesen elérhetik a parancssorból körülményesebben használható funkciókat.

A szolgáltatás méréséhez szükség van egy adatgyűjtő szolgáltatásra, ezt a Ceilometer valósítja meg, ezen felül a felhő belső diagnosztikában és az alkalmazások automatikus skálázásában is szerepe van. A felhőben futó alkalmazások konfigurációmenedzsmentjéért a Heat nevű komponens felel. A Trove már a platformfelhő irányába mutat azzal, hogy adatbázis szolgáltatást kínál a felhőben. A fentieken kívül még számos komponens telepíthető az OpenStack keretrendszerbe, melyek specifikus feladatokat oldanak meg az infrastruktúra- és platformszolgáltatás területén.

Jellemző architektúrális minták openstack telepítésekben

A komponensek különböző szoftverekből épülnek fel, melyek együttműködve végzik feladatukat. A rendszer tervezésekor figyelembe kell venni, hogy az egy komponensbe tartozó szoftvereknek is lehetnek eltérő beállításai, illetve hogy bizonyos szoftvereknek ugyanazon a gépen kell futniuk technikai okokból.

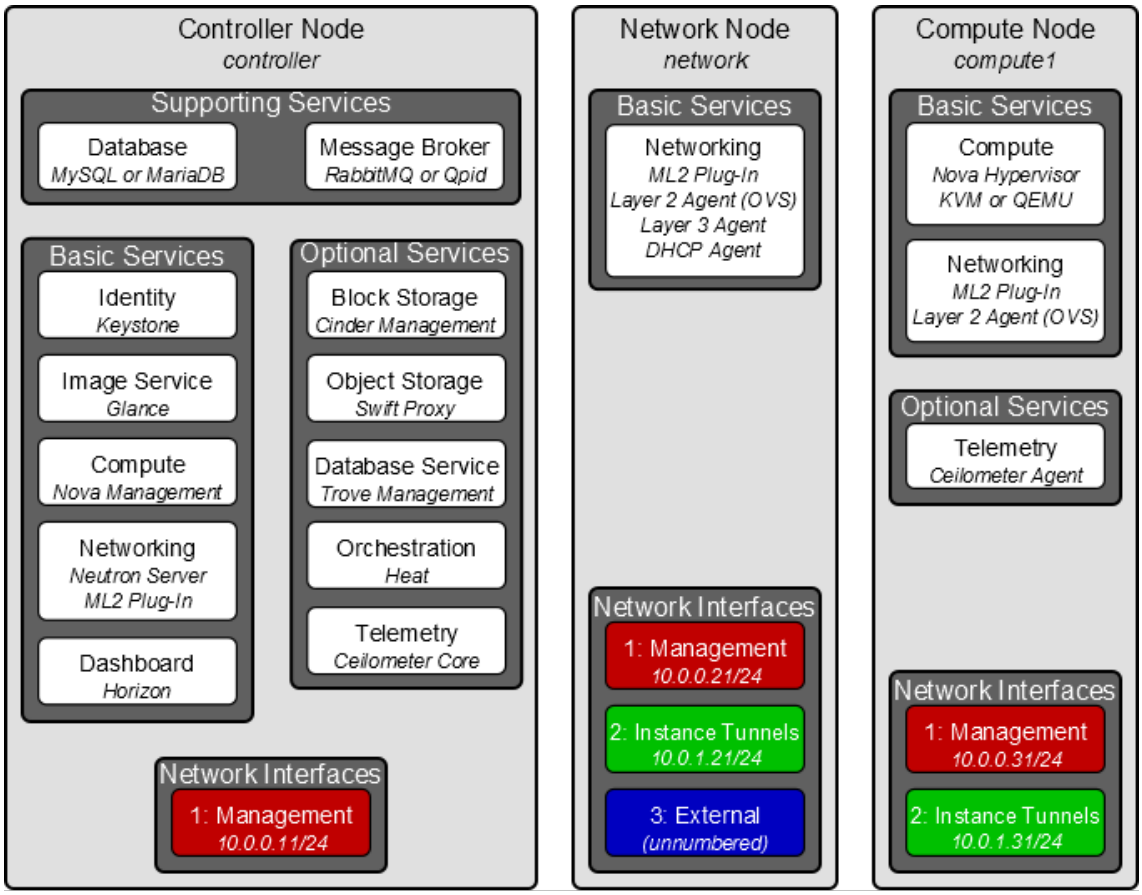
A tervezés megkönnyítése érdekében az OpenStack telepítési útmutatójában [32] szerepel egy alapvető architektúra minta, ami három főbb szerepet definiál a felhőt felépítő számítógépek számára (1.5. ábra). A Controller node szerepű gépeken futnak azok a szoftverek, melyek a felhő menedzsmentjét végzik, például a komponensek központi vezérlő szoftverei, vagy az API szolgáltatások. Ezekből az egységekből több is létezhet egy rendszerben a redundancia és a teljesítményé növelése érdekében, de a feladatátvétel vagy a terheléselosztás megoldására külön szoftverek szükségesek.

A virtuális gépeket a Compute node szerepű gépek futtatják. Ehhez két szoftverre van szükség, egyik a Nova ágense a virtualizációs technológia irányításához, a másik pedig egy Neutron ágense, mely a hálózati beállítások kezeléséhez szükséges. Az utolsó szerep a Network node, ami a hálózatok harmadik rétegbeli működtetésében játszik szerepet DHCP szerverként és útválasztóként. Mivel a kifelé irányuló forgalom ezen megy keresztül, a Compute node esetében nem szükséges közvetlenül összekötni a világhálóval.

Az összetettebb architekturális mintákat az OpenStack architektúra útmutató [30] részletezi.

1.3. A konfigurációmenedzsment kihívásai OpenStack-ben

A több szolgáltatásból álló elosztott rendszerek tervezése és telepítése jelentős kihívások elé állítja a rendszermérnököket. Ez a probléma hangsúlyosan jelentkezik a privát felhők világában, különösen az OpenStack esetében. A nagyszámú szolgáltatás, konfigurációs lehetőség és a változatos terítési elrendezések mellett nehéz megfelelő tervezési döntéseket hozni. A szituációt tovább bonyolítja, hogy a szolgáltatások egymásra épülnek, ezért a



1.5. ábra. Az OpenStack három elemű minta-architektúrája [4]

beállításaiknak minden esetben konzisztensnek kell lenniük egymással. Ezt a problémát kiküszöböli az OpenStack-et automatikusan telepítő keretrendszerek alkalmazása (2.1 fejezet), ezek közül azonban egyik sem támogatja a megvalósítandó konfiguráció modelljének MDD folyamat részeként létrehozását, illetve elemzését.

Az ilyen komplex rendszerekben a rendelkezésreállás megvalósítása jelenti az egyik legnagyobb nehézséget. Meg kell határozni a szolgáltatásbiztonság szempontjából releváns részszolgáltatásokat és az azokhoz megfelelő szolgáltatásbiztonsági mechanizmusokat, majd megtalálni a helyes telepítési elrendezést az elvárt felhő teljesítmény és rendelkezésreállás biztosítására. A konfiguráció funkcionális és hibatűrési teszteléséhez el kell végezni a telepítést, ami a jelenlegi technológiákkal időigényes feladat, főleg ha a terveket iteratívan javítani kell.

2. fejezet

Felhő konfigurációk modellvezérelt menedzsmentje

A szoftverfejlesztésben a modell alapú megközelítések és folyamatok (melyekre összefoglalóan mint Model Driven Design – MDD – szoktunk hivatkozni) alkalmazása ma már bevett gyakorlatnak számít. Ezeknek közös jellemzője, hogy a rendszerfejlesztés folyamata során precíz, metamodellel rendelkező modellek egy sorozata készül el, a folyamat végén legalább részleges kód és konfiguráció generálással.

Hasonló megközelítések az IT rendszerek telepítése és menedzsmentje területén is ismertek. Az IT konfigurációmenedzsmentben ráadásul a futásidejű modelltámogatás is meglehetősen elterjedt. Ennek példája a modern Windows platformok Windows Management Instrumentation (WMI) funkcionalitása, mely szabványos, objektumorientált modellekbe[1] képezi le a platform aktuális futási és konfigurációs állapotát. CIM alapú menedzselhetőséget biztosítanak a VMWare bizonyos termékei, és nagyvállalati adattároló megoldások is (Lásd SMI [40])

Az ITIL (IT Infrastructure Library) által javasolt CMDB (Configuration Management Database) szolgáltatás célja is az, hogy egy központi, modell alapú, federált képet tartson karban egy menedzselt infrastruktúráról, és annak szolgáltatásairól. (CMDB szoftvert több nagy szoftvergyártó is nyújt, de ezek alkalmazásának bonyolultsága miatt CMDB rendszereket jellemzően csak nagyvállalati infrastruktúrákban találunk)

Ezen modell alapú konfigurációleíró és menedzsment megközelítések példáján triviálisan adná magát az ötlet, hogy felhő környezetek konfigurációját is modellvezérelten tervezzük és írjuk le, valamint hogy igényeinket egy modellben határozzuk meg (A végrehajtást egy intelligens telepítőmegoldásra bízva). Irodalomkutatásom azonban azt mutatta meg, hogy az OpenStack-hez ma még nem létezik általános konfiguráció és telepítés leíró modell. Léteznek ugyan OpenStack telepítő eszközök, melyek bemenete egyfajta szakterület specifikus konfigurációleírásként felfogható, ugyanakkor korlátozott, nem nyílt, vagy legalább is nem közismert modelleket alkalmaznak, így felhasználhatóságuk megkérdőjelezhető.

Jelen fejezet áttekinti az OpenStack telepítésére és menedzsmentjére alkalmazható eszközöket, majd bemutatja dolgozatom egyik fő eredményét, egy nyílt és általános modellezés keretrendszert OpenStack konfigurációkhoz és telepítésekhez, melyet végül felhasználunk OpenStack automatizált telepítésére Docker környezetben.

[1] Lásd DMTF CIM szabvány családja: `http://www.dmtf.org/standards/cim`

2.1. Az OpenStack telepítés eszközei

A rendkívül nagy számú konfigurációs lehetőség, különböző Linux disztribúciók és hardverkörnyezetek, továbbá a felhő rendszerek mérete miatt szükség van telepítést végző szoftverekre. Az OpenStack telepítésére szánt szoftverekkel szemben olyan igények merülnek fel, melyek túlmutatnak az általános eljárásokon.

A felhő rendszerek általában rendkívül nagy számú számítógépből állnak, ezért a telepítő szoftvernek rendelkeznie kell olyan funkcionalitással, mely a gépek leltárazását és teljes feltelepítését elvégzi az operációs rendszert beleértve. A szükséges szoftverek feltelepítése után a konfigurációjukat is el kell végezni, méghozzá úgy, hogy az megfeleljen a tervezett OpenStack architektúrának. Ehhez a telepítőnek ismernie kell a gépek és szolgáltatásaik közötti összefüggéseket.

A telepítést segítő programok többsége a teljes spektrum egy bizonyos részét fedik csak le, hardver és Linux disztribúció megszorításokkal, illetve csak néhány konfigurációs lehetőség támogatásával.

Általános konfigurációmenedzsment eszközök

Az általános konfigurációmenedzsment eszközök esetében, mint például a Chef [5] vagy Puppet [36], a saját formátumukban lehet megadni a telepítési eljárás leírását és a paramétereket. A legtöbb ilyen eszközhöz léteznek implementált OpenStack telepítési eljárások, de az OpenStack számos konfigurációs és architekturális döntési lehetősége miatt használatuk nehézségekbe ütközik.

A megoldások egy része kevés beállítási lehetőséggel rendelkezik, hogy a felhasználók dolgát egyszerűbbé tegye, ez azonban a lehetőségek beszűkülésével is jár. Azok az eljárások, amik az OpenStack lehető legszélesebb körű konfigurálhatóságát helyezik előtérbe, rendkívül bonyolulttá válnak. A rengeteg választási lehetőség miatt szükség lenne arra, hogy a konfigurációmenedzsment rendszer valamilyen módon kiszűrje a hibás konfigurációkat, ezzel segítse a felhasználókat.

Az általános konfigurációmenedzsment rendszerek többnyire nincsenek felkészítve a nagyszámú konfigurációs paraméter felhasználóbarát kezelésére és helyesség-ellenőrzésére, továbbá hardver menedzsment eszközökkel sem rendelkeznek, így önállóan csak korlátozottan alkalmasak felhő rendszerek telepítésére.

Infrastruktúra menedzsment rendszerek

Léteznek olyan megoldások, melyek egy vagy több konfigurációmenedzsment és egy hardverkezelő rendszer együtteséből állnak, így egy magasabb szinten vezérlik az infrastruktúrát. Ilyen például a Foreman [19], a Crowbar [9] továbbá a MaaS [27] és JuJu [48] párosa. Ezek számára már nem okoz gondot a hardverek nagy száma, továbbá a hozzáadott absztrakció miatt felhasználóbarátabbá válhat a konfigurációs felület, azonban továbbra sem képesek átfogóan ellenőrizni a beállítások helyességét, így vagy kevés konfigurációs lehetőség van, vagy hibás konfigurációk állíthatók elő.

OpenStack specifikus telepítők

Az OpenStack telepítésére tervezett szoftverek, mint például a Fuel [34] vagy a Compass [7] jellemzően több lehetséges beállítást és architekturális döntést bíznak a felhasználóra, ugyanakkor több segítséget nyújtanak a konfiguráció tervezésénél. Ezek az eszközök a háttérben többnyire a már korábban ismertetett eszközöket használják, de jelentős szakterület specifikus tudást és kényelmi funkciókat adnak hozzá, például a beállítások különböző variációit alaposan tesztelik.

A TripleO egy olyan megoldás, mely az OpenStack saját eszközeit használja a terítésre. Az Ironic nevű komponens menedzseli a számítógépeket, a Nova készít belőlük felhőbeli számítógép példányt, a Neutron segítségével hozza létre a szükséges hálózati konfigurációt, és a Heat telepíti fel és konfigurálja a szükséges programokat. Ezzel a rendszerrel az OpenStack függetlenné válik a többi konfigurációmenedzsment eszköztől, ráadásul a komponensei fejlődésével a TripleO [43] is egyre fejlettebb lesz.

Speciális feladatra tervezett telepítő szoftverek

Bizonyos telepítők speciális célokat szolgálnak, például a DevStack [11] illetve az Anvil [1] elsődleges célja az OpenStack tesztelése és kipróbálása, illetve szoftvercsomagok készítése. Forráskódból végzik a telepítést, és számos konfigurációt képesek kialakítani.

Az Inception [23] nevű szoftver a telepítést OpenStack felhőben végzi el, ezzel felhőbe ágyazott felhőt létrehozva. Ez tesztelési feladatokat is szolgálhat, de adatbiztonsági vonatkozásai is vannak.

2.2. A modellvezérelt fejlesztés technológiái

A modellvezérelt fejlesztés a szoftverfejlesztésben ma már széles körben alkalmazott paradigma. Legfőbb jellemzője,hogy a megvalósítást precíz modellezési fázis vezeti be, ahol a modellek írják le a szoftver adatszerkezeteit, és azok kapcsolatait. A modellek alapján forráskódot lehet generálni, mely a modell által leírt módon viselkedik. A modellek leírására számos nyelv áll rendelkezésre, többek között az UML, illetve annak egyszerűsített részhalmaza, az EMF Ecore [41].

A megközelítés különösen alkalmas szakterület specifikus nyelvek készítésére a fogalmainak modellekkel történő leírásával.

A modellvezérelt fejlesztési folyamat a legtöbb problémát modellek segítségével oldja meg. Mivel az eltérő feladatokra különböző modellek nyújtanak megoldást, a modell között transzformációkra van szükség. Ehhez különböző transzformáció leíró nyelvek állnak rendelkezésre, például az Atlas ATL [25], vagy a Viatra [10].

A modellek számos felhasználási módja közül az egyik legfontosabb a kódgenerálás, mely lehetővé teszi, hogy a felépített modellekből egy adott logika mentén a modellnek megfelelő programot készítsünk. A generálás leírására számos megoldás használható, az egyik legismertebb nyelv a java alapú Xtend [3], mely egy magas szintű programozási nyelv ami sablon technológiájával megkönnyíti a kódgenerálást, mialatt megtartja a Java kifejezőerejét.

Eclipse Modelling Framework

Az EMF [15] egy Eclipse [14] alapú modellező keretrendszer és kódgenerátor strukturált adatmodell alapú szoftverfejlesztéshez. Különböző fejlesztési és futásidejű eszközöket kínál XMI formátumban leírt modellekhez, melyekkel generálni lehet a modell elmeinek megfelelő Java osztályokat, az ezekhez tartozó adapter osztályokat, melyek a későbbi modellszerkesztést segítik, illetve egy alapvető szerkesztőt. Az EMF irányadó szabvány adatmodellek építéséhez, számos technológia és keretrendszer épül rá.

Az EMF a modellek szerkezetét az Ecore modell mint metamodell segítségével írja le [41]. Az Ecore maga is egy EMF modell, így önmaga metamodellje. Az Ecore az UML szabvány egyszerűsített részhalmaza. Négy olyan osztályt tartalmaz, mely ismerete feltétlenül szükséges a modellezéshez:

EClass: Leírja a modellezni kívánt osztályt. Neve van, 0 vagy több attribútuma és referenciája.

EAttribute: A modellelemek attribútumait írja le. Egy attribútum egy névből és egy típusból áll.

EReference: Az EReference a modellelemek közötti asszociáció egyik oldalát írja le. Rendelkezik egy névvel, az asszociáció másik oldalán található objektum típusával, illetve azzal az információval, hogy csak egy referenciáról van szó, vagy tartalmazási kapcsolatról.

EDataType: Ez az osztály Attribútumok típusainak leírására szolgál. Értéke lehet Java nyelvi primitív elem, mint például az *int*, vagy objektumok, mint a *java.util.Date*.

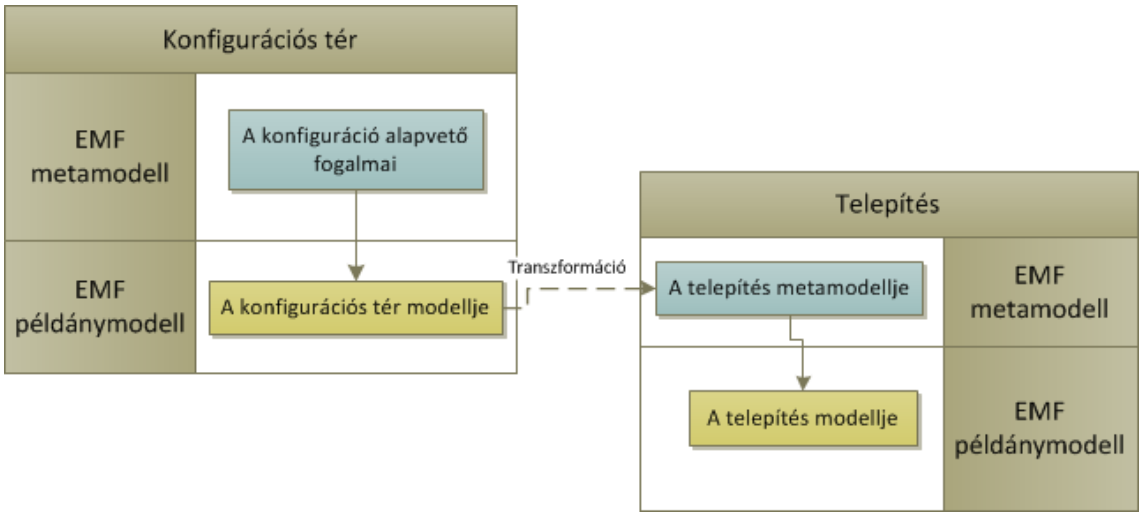
IncQuery

Az IncQuery keretrendszer gráf alapú deklaratív lekérdezéseket tesz lehetővé EMF modellekben. Egy magas szintű lekérdezőnyelvet definiál, így nincs szükség imperatív programozásra a különböző modellezési feladatok megoldásához.[16] Ehhez a lekérdezőnyelv kiegészül olyan elemekkel, melyek segítségével lekérdezésen alapuló modellezési részfeladatokat lehet megoldani. Ilyen a különböző nézetek készítése, adatkötés, validáció, valamit a lekérdezés alapú származtatott értékek. Összetettem feladatok megvalósításánál, mint például modell transzformáció vagy kódgenerálás szintén alapvető építőelemként szolgálnak a lekérdezések.

A lekérdezőnyelven túl az IncQuery másik erőssége a hatékony lekérdezőmotor, mely inkrementális gráf-mintaillesztési módszerével igyekszik leküzdeni a skálázódási problémákat [2].

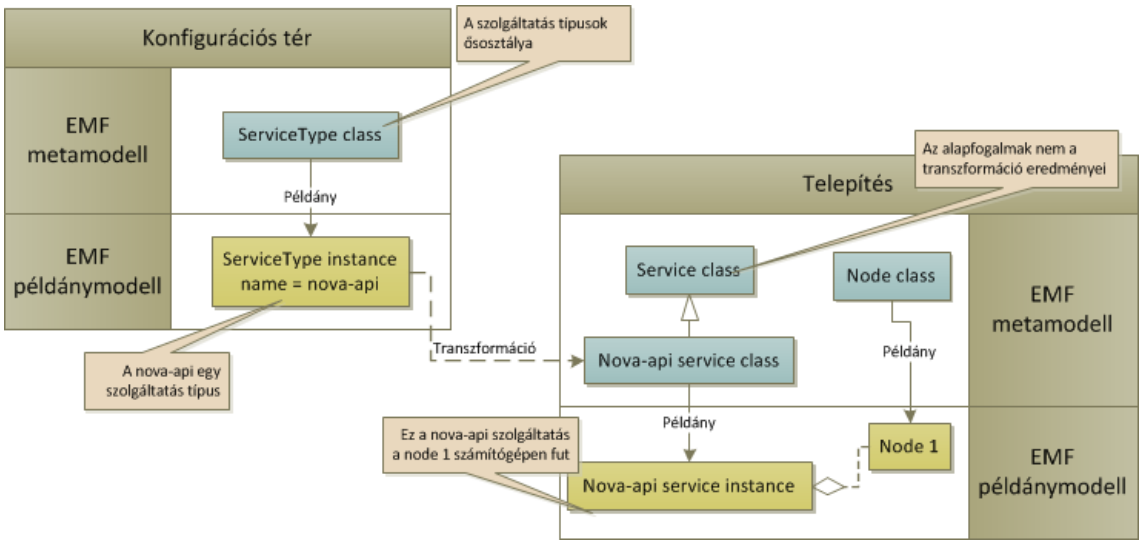
Az IncQuery keretrendszer jól integrálódik az Eclipse-el, az EMF modellező keretrendszerrel, és az ahhoz illeszkedő szoftverekkel [44].

2.3. Modellezési megközelítés



2.1. ábra. A modellezési megközelítés

A konfigurációmodellezés célja egy olyan szakterület specifikus nyelv készítése, mely képes leírni egy OpenStack telepítés architektúráját és konfigurációját. Ezt a modellt a továbbiakban telepítési modellnek nevezzük. Egy ilyen modellen lehet helyesség-ellenőrzést, optimalizálást, teljesítmény és megbízhatósági elemzést, végezni, és fel lehet használni a telepítés és üzemeltetés különböző fázisaiban.



2.2. ábra. A modellezési megközelítés a nova-api szolgáltatás példáján bemutatva

A hatékonyság és használhatóság növelése érdekében a telepítési modellt specifikusan ehhez a feladathoz alakítjuk. Ez úgy valósul meg, hogy a metamodell által definiált osztályok az OpenStack egyes specifikus elemei, az osztályok mezői pedig az adott szoftverelemek beállításai, így az elkészült telepítési modell egyszerűbb és célravezetőbb.

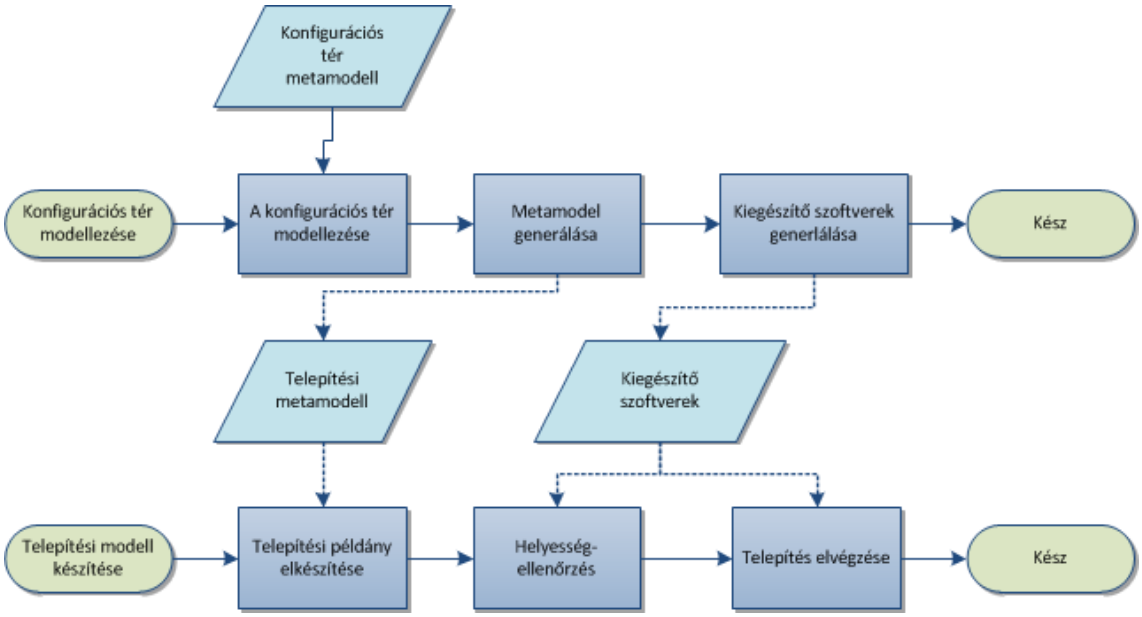
A modellek kényelmes és hatékony használatához hozzátartozik a modellek helyességellenőrzése. Nem csak a beállítások értéktartományait kell figyelembe venni, számos ennél összetettebb szabály van. Ilyenek például a szolgáltatások közötti függőségek, amikor egyik szolgáltatásnak szüksége van egy másik szolgáltatásra. Sokszor ez azzal társul, hogy a függőséget kielégítő szolgáltatásnak egy bizonyos beállítással kell rendelkeznie. Ezek a struktúrán felüli információk sokszor túlmutatnak az EMF Ecore metamodell leíróképességén, így például a függőségi szabályokat a modellhelyesség-ellenőrző szoftverbe kell bevinni.

Mivel az OpenStack rengeteg elemből és szabályból épül fel, nem tűnt hatékonynak manuálisan megépíteni a telepítés metamodelljét és hozzá alakítani a modellt használó programokat, főképp annak tudatában, hogy az OpenStack gyorsan fejlődik és ezzel együtt változik is.

A probléma megoldására létrehoztam a konfigurációs tér modelljét, hogy egységesen le lehessen írni egy adott OpenStack verzió tulajdonságait, ami felhasználható a telepítési modell elkészítésére. A 2.1. ábrán látható ez a megközelítés. A konfigurációs tér metamodelljében olyan osztályok vannak, melyek a szoftverkonfiguráció fogalmait írják le, például szolgáltatás, komponens, beállítás vagy függőség (Példa: 2.2. ábra). A metamodell felhasználásával készült konfigurációs tér modellek így képesek leírni egy OpenStack verzió elemeit, azon beállításait és a közöttük húzódó összefüggéseket, amiből hatékonyan lehet generálni a telepítés metamodelljét és az arra épülő szoftvereket.

2.4. A modellezés folyamata

A metamodellek mentén két részre bonthatjuk a folyamatot. Az első rész a konfigurációs tér elkészítését és telepítési modellhez szükséges elemek generálását tartalmazza. A második rész pedig a telepítési modell megalkotásából és a hozzá kapcsolódó programok használatából áll. A folyamatot a 2.3. ábra mutatja be.



2.3. ábra. A tervezési munkafolyamatok összevont ábrája

A konfigurációs tér modellezése az OpenStack felépítésével és konfigurációjával kapcsolatos információk bevitelét jelenti. A metamodell és a kiegészítő szoftverek generálására modellvezérelt szoftvereket készítettem.

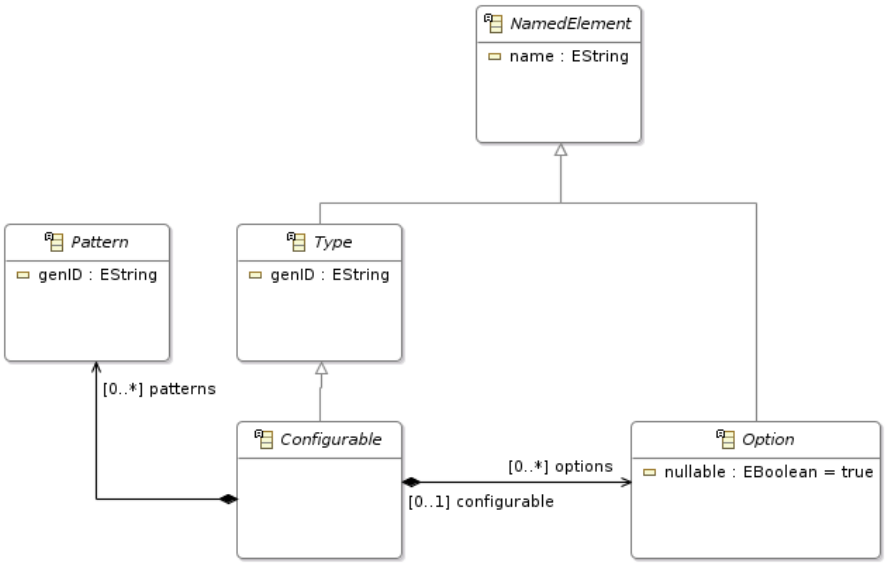
A telepítési modell létrehozása jellemzően úgy történik, hogy a felhasználó egyesével létrehozza és beállítja az elemeket. Léteznek azonban más megoldások is, például a későbbiekben bemutatok egy olyan egyszerű modellgenerátort, ami a lehető legegyszerűbb elosztott telepítést generálja, továbbá bemutatok egy olyan programot, amely lineáris programozással optimalizálja és skálázza a telepítést.

Az elkészült telepítési modellen érdemes ellenőrzést futtatni, hogy minden megfelele a konfigurációs tér modellben leírtaknak. A telepítési modelleket fel lehet használni a rendszertervezés és üzemeltetés számos területén. Hogy ezt bemutassam, implementáltam egy olyan programot, mely a telepítési modell által leírt rendszert Docker konténerekben megvalósítja, ezzel tesztelve a konfigurációt.

2.5. A konfigurációs tér metamodellje

A konfigurációs tér modellezését az EMF modellezési környezetben végeztem. A konfigurációs tér modell elemei jellemzően három csoportra oszthatók, mintákra, típusokra és beállításokra. A transzformáció során a konfigurációs tér típusaiból keletkeznek a telepítési tér osztályai, a beállításokból ezek attribútumai, a mintákból pedig azok a különböző szabályszerűségek, melyek nem részei a metamodellnek. A csoportokat absztrakt osztályok jelölik, melyek viszonyát a 2.4. ábra szemlélteti.

A típusok osztályának a nevében szerepel a "Type" kifejezés (Pl.: *ServiceType*), és a *Type* osztály leszármazottai. A típusok rendelkeznek a "name" attribútummal, melynek értéke a típus neve lesz a telepítési térben. Típusok közé tartoznak a szolgáltatások, a komponensek, és a tervezési teret jelentő objektumok is (2.5. ábra). A szolgáltatások lehetnek komponens szolgáltatásai (*ComponentServiceType*), vagy külső szolgáltatások (*ExternalServiceType*). Például a nova-api nevű szolgáltatást egy *ComponentServiceType* példány reprezentálja, melynek a "name" attribútuma a "nova-api" értékre van állítva. Ennek a transzformáció utáni megfelelője a telepítési metamodellben a *NovaapiService* osztály lesz.



2.4. ábra. A főbb ősosztályokat tartalmazó osztálydiagram

A *Configurable* osztály leszármazottai az OpenStack azon elemeit jelölik, melyeket a konfiguráció kialakításában aktívan részt vesznek. Ezek az osztályok tartalmaznak beállításokat és mintákat, továbbá definiálhatnak modulokat és enumerációkat is a beállítások megkönnyítésére.

Az enumeráció választható értékek listáját jelenti. Ezeket az enumerációkat a beállításokban lehet felhasználni érték típusként.

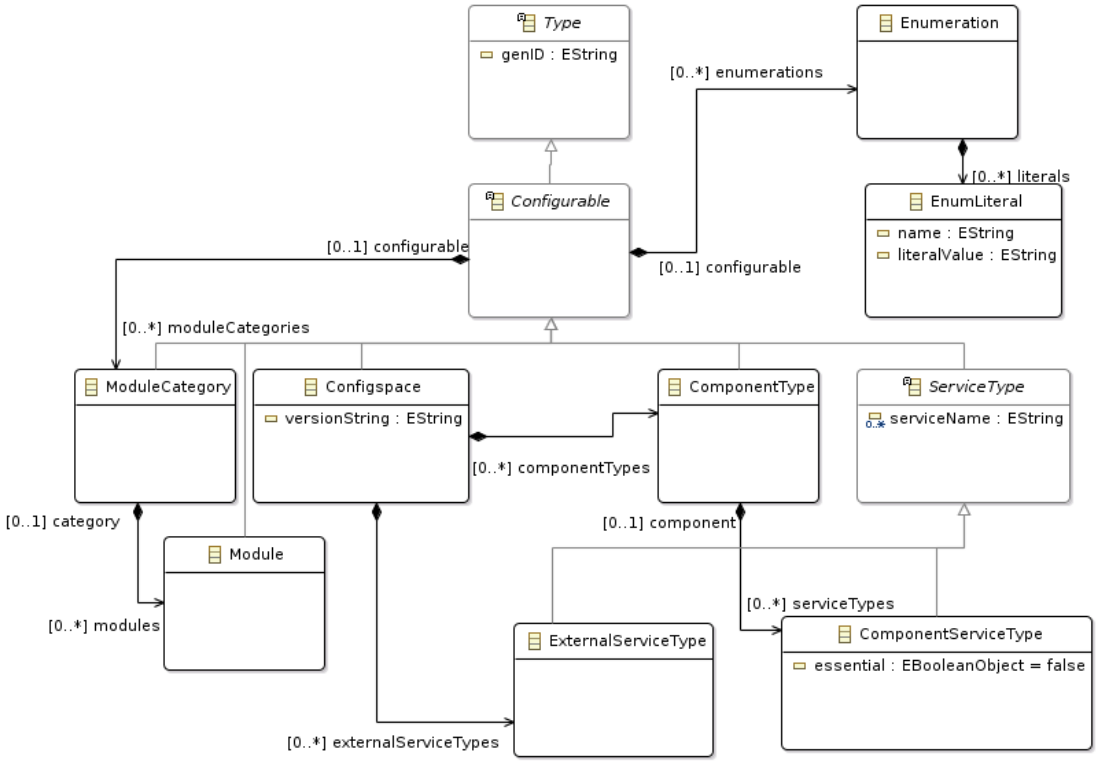
A modulkategóriák és a modulok a *Configurable* osztály leszármazottai. A modulkategória kiválasztható modulok egy halmazát jelenti. Ezzel a két osztállyal lehet újrafelhasználható és kiválasztható konfigurációs blokkokat létrehozni a modellben.

A beállítások az *Option* ősosztályból származnak, és megjelölik a beállítás nevét és az érték típusát, ami lehet többek között szám, szöveg, enumeráció vagy modul. A különböző típusú beállításoknak lehetnek specifikus attribútumai. A beállítások a típusuknak megfelelő attribútumként jelennek meg a telepítési modellben, az esetleges tartalmi kényszereik pedig a helyesség-ellenőrzőben.

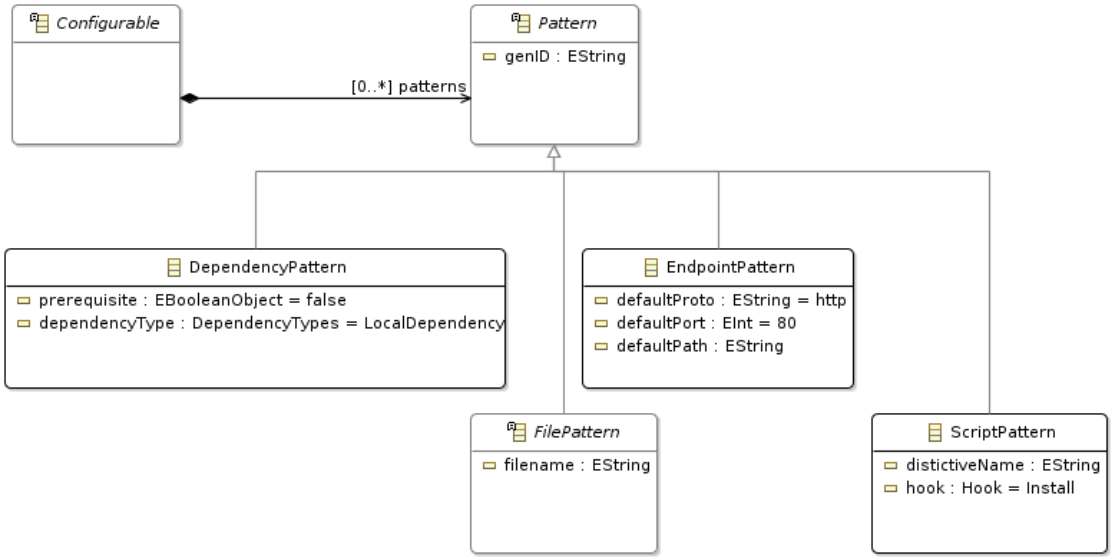
A minták a *Pattern* ősosztályból származnak, és olyan információkat hordoznak, melyek a telepítési modell azon elemein érvényesek, amelyeken a megadott mintaillesztés érvényesül (2.6. ábra).

Az *EndpointPattern* minták akkor érvényesülnek, amikor a modell egyik eleme az adott beállítások mellett egy szolgáltatási végpontot fog üzemeltetni. Ez az információ később felhasználható a modellre épülő szoftverekben.

A *DependencyPattern* minták érvényesülése azt jelzi, hogy az adott elem az aktuális beállításai mellett függ egy másik szolgáltatástól. Két típusa van, a hálózati és a helyi függőség. A hálózati esetben a két elemnek el kell érnie egymást hálózaton keresztül, helyi függés esetén viszont a két szolgáltatásnak ugyanazon a számítógépen kell futnia.



2.5. ábra. Összefoglaló diagram a típusokról



2.6. ábra. A minták átfogó osztálydiagramja

A további minták az alacsony szintű konfiguráció elkészítéséhez szükségesek. A *FilePattern* ősosztály leszármazottai soronként állítják elő a konfigurációs fájlokat mintaillesztéssel. A *ScriptPattern* ősosztály leszármazottai pedig programfuttatások sorozatát definiálják a telepítési folyamat bizonyos állapotaiban, például a szolgáltatás elindítása előtt, amikor elkezd futni a szolgáltatás, továbbá teszteléskor.

2.6. Egy konfigurációs tér példa bemutatása

A konfigurációs tér modellezését egy példán keresztül mutatom be, melyet a 2.7. ábrán láthatnak. Az érthetőség kedvéért nem tartalmazza a teljes telepítést, csak annak egy kis részletét.

A központi elem a *Configspace*, a modell tartalmazási fájának gyökere. Egy adott OpenStack verzió leírását tartalmazza, jelen esetben a Juno verzióét.

Ez a demonstrációs célú modell csak egy komponenst, a Keystone-t tartalmazza. Ez a komponens csak a *keystone-all* szolgáltatásból áll. Ennek nincsenek külön beállítása, tehát a Keystone-nak ebben a modellben csak komponens szintű beállításai vannak. Két karakterlánc típusú beállítás, ami az adminisztrátor felhasználó azonosítási adatait tartalmazza.

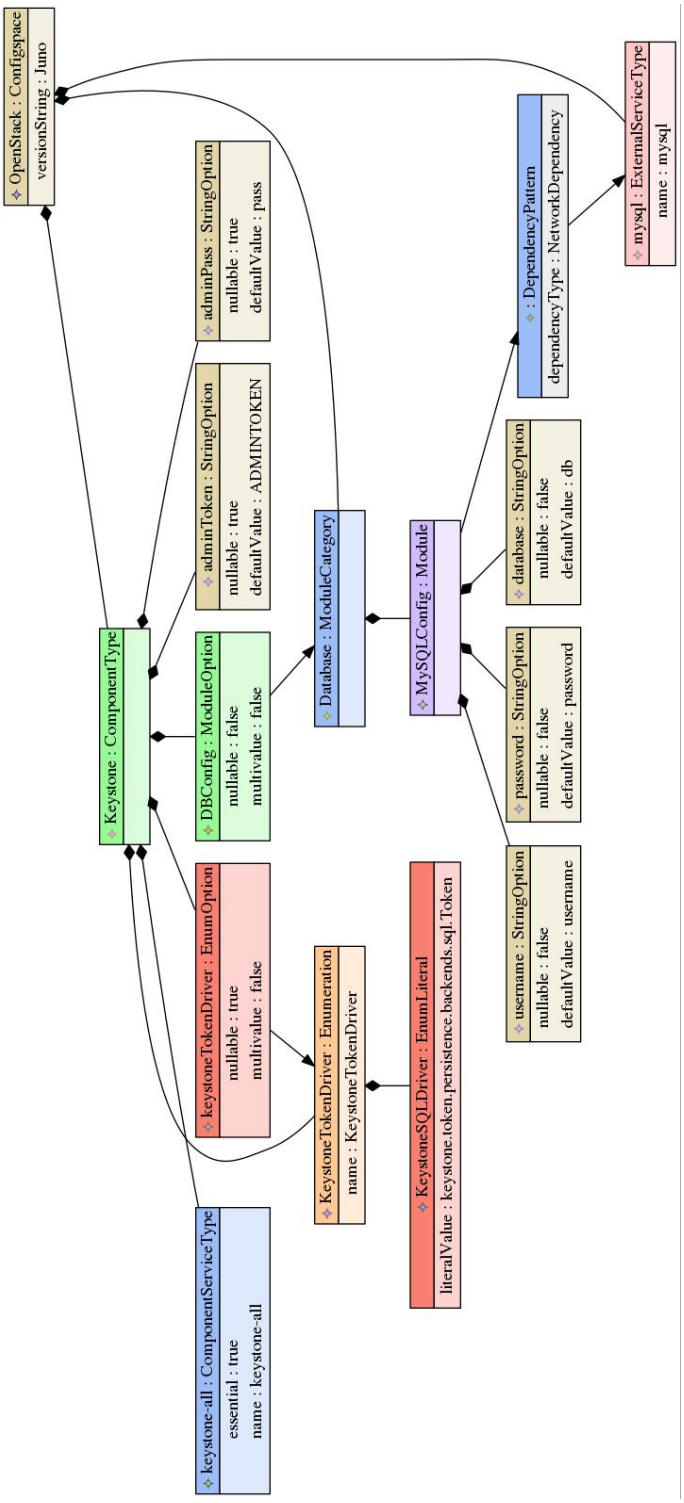
A *keystoneTokenDriver* beállítás felsorolás alapú, a lehetséges értékeit a *KeystoneTokenDriver* enumeráció tartalmazza, ahogy ezt a köztük húzódó referencia is mutatja. A lehetséges elemek az *Enumeration*-hoz kapcsolódó *EnumLiteral* elemek adják.

A *Database* nevű modulkategóriában jelenleg egy modul található, a *MySQLConfig*. Ez felel a szolgáltatások és az adatbázis kapcsolatáért. Ha további adatbázis megoldásokat szeretnénk felvinni a modellbe, a modulkategóriához kellene hozzáadni új modulokat. A MySQL konfigurációja 3 szöveges beállítást tartalmaz az adatbázis hozzáférési adatairól. A Keystone komponens adatbázisát a *DBConfig* beállítással lehet kiválasztani.

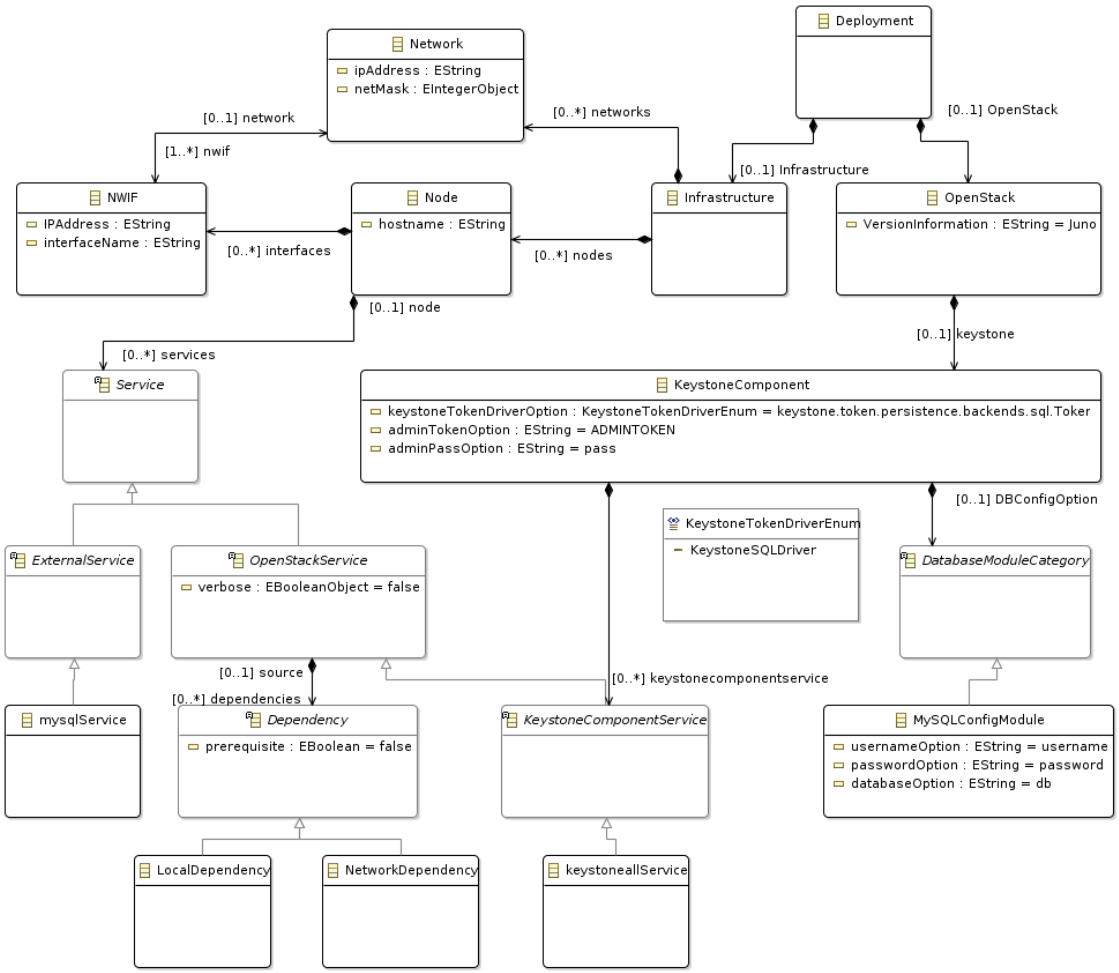
Érdemes megemlíteni, hogy a *KeystoneTokenDriver* tartalmazó eleme a Keystone komponens volt, a "Database"-t viszont a gyökér elem tartalmazza, hiszen a *KeystoneTokenDriver* csak a komponensen belül lesz felhasználva, adatbázis-konfigurációt viszont szinte minden komponens használ, ezáltal a modell rendezettebb és könnyebben érthető lesz.

A MySQL-t használó szolgáltatásoknak el kell érniük a MySQL szervert. Ezt a modulhoz tartozó függőség szimbolizálja, mely láthatóan hálózati típusú és referenciával rendelkezik az igénybe vett szolgáltatáshoz.

A komponensekhez nem sorolható szolgáltatások, mint például a MySQL szerver a gyökérelemhez tartoznak.



2.7. ábra. A konfigurációs tér modell egy példánya (Részlet, szűrt).



2.8. ábra. A transzformációval készült telepítési metamodell

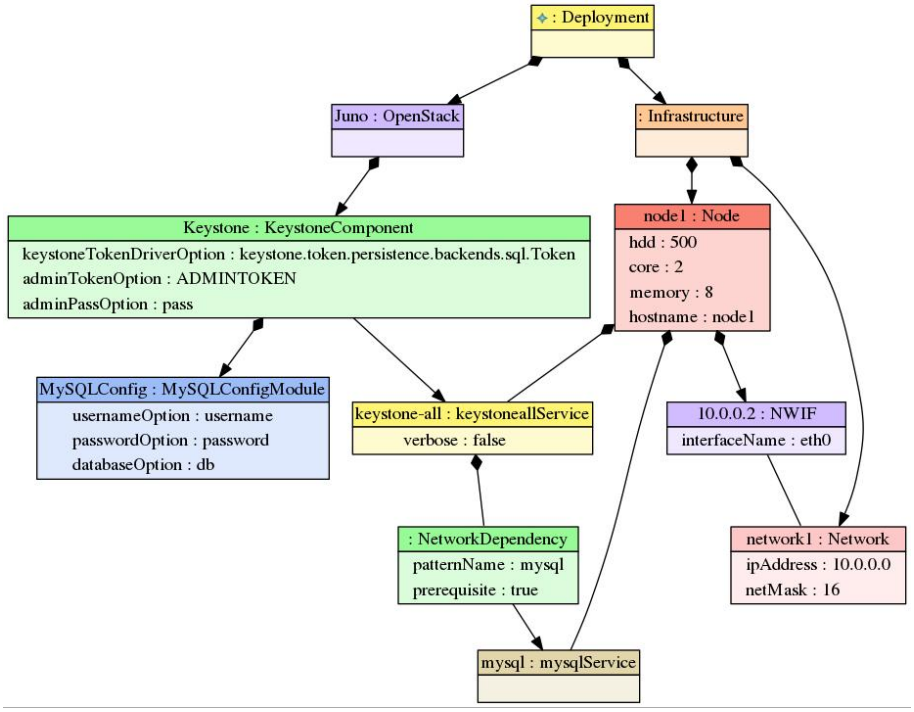
A fent kifejtett konfigurációs tér modelljéből transzformációval telepítési metamodellt állítunk elő, melyet a 2.8. ábrán szemléltetünk. A *Deployment* a gyökér-elem, két fő részre osztja a modellt, infrastruktúrára és az OpenStack-re. Az infrastruktúra számítógépekből, azok hálózati interfészeiből és az ezeket összekötő hálózatokból áll.

A szolgáltatások ősosztályából (*Service*) származnak a külső szolgáltatásokat és az OpenStack szolgáltatásait összefogó absztrakt osztályok. A *mysqlService* természetesen külső szolgáltatás. A *keystoneallService* a *KeystoneComponenService* ősosztálynak leszármazottja, ami a Keystone szolgáltatásainak ősosztálya.

A komponenseket az OpenStack osztály tartalmazza. A *KeystoneComponent* attribútumai a konfigurációs tér beállításainak felelnek meg. A felsorolás alapú beállításból egy *Enum* típus keletkezett, ami a megfelelő attribútum típusa is egyben. A modul beállításból referencia lett a modulkategóriára, ami egy absztrakt osztály, leszármazottja pedig a konkrét modul.

2.7. Egy tervezési modell példa bemutatása

A tervezési tér metamodelljéből képzett modell a 2.9. ábrán látható. A gyökéreleme a "Deployment", az infrastruktúra pedig egy hálózatot és egy számítógépet foglal magában. Az ábrán látható a gépen futó két szolgáltatás, melyek között hálózati függőség van.



2.9. ábra. Egy telepítési modell példány

2.8. Telepítési modell generálása

A konfigurációs tér ismeretében számos módon lehet telepítési modellt generálni. Implementáltam egy megoldást, amely egy egyszerű példán keresztül bemutatja a telepítési modellek generálását. A program célja, hogy egy olyan telepítést generáljon, amely minden beállításnál az alapértékeket próbálja megtalálni, minden szolgáltatásból csak egy példányt telepít ha a függőségek mást nem írnak elő, továbbá a lehető legjobban szétteríti a telepítést. Az így kapott telepítési modell tesztelésével ellenőrizni lehet, hogy a hálózati szolgáltatások címei és az alapbeállítások helyesek-e.

A konfigurációs térben definiálva van egy attribútum, ami azt jelöli, hogy az adott szolgáltatás feltétlenül szükséges e a komponens használatához. Ezeket a program telepíti külön gépekre, majd minden minden szükséges beállítást alaphelyzetbe állít. Az így kialakult rendszert a függőségek mentén kiegészíti a következő iteratív algoritmust használva:

1. Függőségek meghatározása a minták segítségével. Amennyiben nincs kielégítetlen függőség, a program leáll.

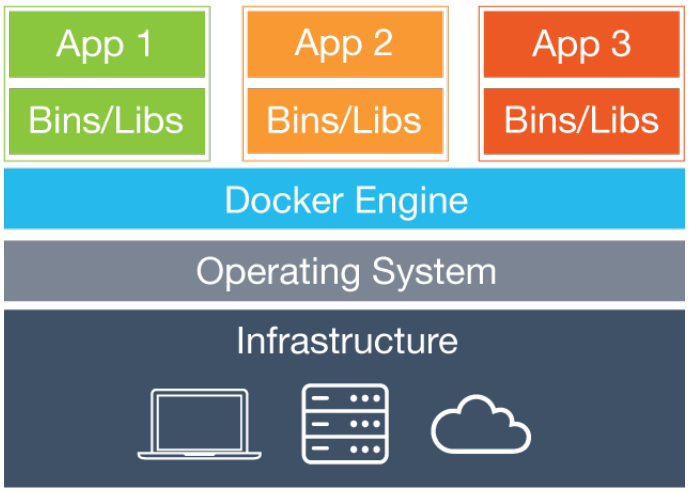
2. Függőségek kielégítésére megfelelő szolgáltatások létrehozása, lokális függésnél ugyanarra a gépre, hálózati függés esetén új számítógépre .

3. Új szolgáltatások beállításainak alaphelyzetbe hozása.

4. Ugrás az első lépésre

2.9. A Docker rendszer

A Docker [49] egy nyílt forrású projekt, melynek fő célja, hogy operációs rendszer szintű virtualizáció felhasználásával javítsa alkalmazások egységbe csomagolhatóságát és hordozhatóságát, így megkönnyítse a fejlesztők és a rendszergazdák dolgát. Az alkalmazásokhoz



2.10. ábra. A konténer alapú virtualizáció sematikus ábrája [12]

rendszerképeket lehet készíteni, melyek tartalmazzák az alkalmazást és függőségeit a kernel kivételével. Az így elkészült szoftvercsomag lényegesen kisebb, mint egy teljes operációs rendszer, azonban az alkalmazás függőségeit tartalmazza, így bármelyik szerveren futtatható függőségek telepítése nélkül (2.10. ábra).

A Docker rendszer legfontosabb eleme a Docker Engine, mely a konténerek létrehozásáért és futtatásáért felelős API-t és eszköztárat nyújts A háttérben több technológia közül választhatunk. Az alapbeállítás a libcontainer nevű vezérlő, amivel operációs rendszer szintű virtualizáció valósítható meg. Ezen kívül még számos virtualizációs technológiát támogat, melyek között megtalálhatók más operációs rendszerek megoldásai, mint például a BSD Jail, de akár teljes virtualizációt is választhatunk a QEMU vezérlővel.

Az operációs rendszer szintű virtualizáció esetében a rendszermag képes izolált felhasználói térben futtatni folyamatok csoportjait. Ezeket a felhasználói tereket nevezik Docker környezetben szoftver konténereknek. A konténerekben futó alkalmazások számára úgy látszik, mintha egy teljes értékű kiszolgálón futnának. Az operációs rendszer szintű virtualizáció költsége rendkívül alacsony [17], hiszen ugyanazt a kernelt használja, mint a natívan futó programok, nem történik hardver emuláció és hardver támogatásra sincs szükség.

A Docker rendszer további összetevői megkönnyítik a rendszer használatát [13], és tovább gyorsítják a fejlesztést és az üzemeltetést. A Docker Hub egy olyan internetes portál, mely a felhasználók által készített rendszerképeket tárolja és teszi elérhetővé. Itt a legtöbb Linux disztribúcióhoz és a fontosabb alkalmazásokhoz, például az Apache webszerverhez is található rendszerkép. A Docker Swarm segítségével gépeinket számítási fürtként kezelhetjük. A Docker Compose a több konténerből álló rendszerek összeállítását teszi lehetővé.

2.10. OpenStack telepítése Docker konténerekbe

Munkám során a Docker rendszert a modellvezérelt konfigurációkezelés tesztelésére, működőképességének bizonyítására használtam. Szerettem volna elkerülni, hogy a megoldásom a Docker rendszertől függjön, ezért nem használtam a magasabb szintű funkciókat.

A rendszerkép

Egy közös rendszerképet használok az konténerek elindításához, melyre az összes szükséges program fel van installálva, hogy ezzel gyorsítsam a tesztelést. Természetesen lehetne

minden konténerhez külön rendszerképet készíteni futásidőben, de ezek a programok viszonylag kis méretűek, ezért nincs szükség erre.

A rendszerkép alapját az Ubuntu 14.10-es verziójának Docker konténerre szánt kiadása alkotta, mivel ez az Ubuntu verzió tartalmazza az általam használt OpenStack IceHouse kiadást. Docker konténerekben nem használatosak az operációs rendszerek saját szolgáltatásvezérlő szoftverét, hiszen azok teljes értékű számítógépekhez vannak tervezve. Emiatt az általam készített rendszerképben a Supervisord [42] nevű szoftver végzi a szolgáltatások menedzselését, ami egy egyszerű lokális szolgáltatásmenedzsment rendszer. Egyszerűsége és távolról vezérelhetősége népszerűvé teszi a Docker világában. A Supervisord számára minden szolgáltatáshoz fel van véve egy konfigurációs fájl, így a nevével hivatkozva el lehet indítani őket.

Az infrastruktúra előkészítése

A művelet egy érvényes telepítési modellből indul. A folyamat első lépése, hogy a program létrehoz minden modellbeli számítógép szimulálására egy konténert, majd annak IP-címét beilleszti a telepítési modellbe, abban végül minden IP-cím a valóságnak megfelelő lesz.

A konténerekbe egyesével feltölti a modellből generált konfigurációs fájlokat, melyek minden gépre különbözőek lehetnek. A konfigurációs fájlok generálása a telepítési modellből történik a (2.5) ban bemutatott elemek alapján.

A szolgáltatások konfigurációja és elindítása

A program a szolgáltatások elindításához az IncQuery lekérdezőnyelvet veszi igénybe. A telepítési modellt kiterjesztjük egy mezővel, ami leírja a szolgáltatás aktuális állapotát. A következő értékeket veheti fel:

none A konfiguráció még nem kezdődött meg

installed A futtatás előtti beállítások készen vannak

started A szolgáltatás fut

running A tesztek lefutottak, így használatba lehet venni a szolgáltatást

Kezdetben minden szolgáltatás a *none* állapotban van. A szoftver ekkor végez egy IncQuery lekérdezést, hogy milyen szolgáltatásokat lehet elindítani. Azokat a szolgáltatásokat lehet elindítani, melyek összes függősége *running* állapotban van. Ebbe kezdetben csak azok a szolgáltatások tartoznak bele, melyeknek nincs függőségük. A program ezektől indulva fokozatosan végighalad a szolgáltatásokon.

Amikor egy szolgáltatást elindít, az kezdetben a *none* állapotban van. A program elvégzi rajta a kezdeti beállításokat, melyeket a konfigurációs térbeli *ScriptPattern* objektumok adnak meg. Ezek után elindítja a szolgáltatást, majd elvégzi rajta az elindítás után szükséges beállításokat.

Végül elvégzi a teszteléshez szükséges parancsokat, majd ha minden rendben volt, akkor beállítja a *running* állapotot. A program ezután újabb elindítható szolgáltatást keres, és ezt addig folytatja, amíg az összes szolgáltatás elindul.

Természetesen előfordulhatnak a modellben körkörös függőségek, amik az ismertetett eljárást idő előtti megállásra késztetnék. A probléma orvoslására bevezettem egy attribútumot a függőségeknél, mely azt határozza meg, hogy az adott függőségre szükség van e a szolgáltatás elindításakor is, vagy már csak a helyes működéshez szükséges. A teljes megoldást egy IncQuery validációs minta jelentené, ami hibát jelez körkörös függőségekkor, így a konfigurációs teret tervező mérnök időben reagálni tud.

Tesztelés és eredmények

A program a 2.8 szekcióban bemutatott modellgenerálás eredményét teljes egészében telepíteni tudja Docker konténerekbe. Nemcsak a tesztek futottak le helyesen, de létre tudtam hozni virtuális gépet, aminek a hálózati funkciói is megfelelően működtek. Ehhez a műveletsorhoz az összes telepített komponensre szükség van, így elmondható, hogy a telepítés sikerült.

3. fejezet

Felhő konfigurációk szolgáltatásbiztonságra optimalizálása

3.1. Motiváció

Az OpenStack komponensek szolgáltatásbiztonsága és hibatűrése kulcsfontosságú aspektusok. hiszen ha a keretrendszer szolgáltatásai nem működnek egy időpillanatban, akkor nincs lehetőség a bérelt erőforrások módosítására, illetve rosszabb esetben azok elérésére sem. Nem véletlen, hogy az OpenStack egyes komponenseihez léteznek olyan hibatűrési mechanizmusok, melyek alkalmazásával a szolgáltatások rendelkezésreállása és hibatűrése nagyban megnövelhető. Ezen mechanizmusok közül a hivatalosan támogatottak az aktív-aktív illetve aktív-passzív replikáció mintáinak valamely változatát valósítják meg.

Látni kell azonban, hogy a komponensek nagy száma és az azok fölött megvalósítandó hibatűrési minták viszonylagos sokfélesége miatt az OpenStack komponensek fizikai gépekhez rendelésének tervezése közel sem triviális feladat, különösen akkor, ha a komponensek terhelésfüggő erőforrásigényeit is figyelembe kívánjuk venni.

Jelen fejezet célja ennek megfelelően az, hogy az OpenStack komponensek fizikai gépekre telepítésének problémájára adjon matematikailag precíz megoldást. A problémára megadok egy lineáris programozási modellt, melyben a fő figyelembe vett kényszerek a következők:

- A szolgáltatások hibatűrési paraméterei.
  - A redundancia megkövetelt szintje.
  - A redundancia kívánt szintje.
- A szolgáltatások erőforrásigényei.
- A szolgáltatás pozitív és negatív szomszédossági kényszerei.

A modell az OpenStack adminisztrációs szolgáltatásait optimalizálja, a fennmaradó gépekből fog állni a felhő számítási erőforráskészlete. Csak teljes számítógépeket veszünk itt figyelembe, hogy a megbízhatóság érdekében ne kerüljön ugyanarra a gépre egy felhasználói alkalmazás és egy rendelkezésreállás szempontjából fontos szolgáltatás. A célfüggvény részben a lefoglalt gépek számát minimalizálja, részben pedig a replikák számát maximalizálja változtatható súlyozással. A fejezet ennek megfelelően a lineáris programozási probléma modellezését mutatja be először, majd néhány kisebb példával azt szemlélteti, hogy értelmes és szükséges a lineáris programozás eszközkészletéhez nyúlni OpenStack terítések tervezésénél is.

3.2. Konfigurációk optimalizálása lineáris programozással

A mérnöki gyakorlat különböző területein nagy hagyománya van a matematikailag precíz konfigurációtervezés és optimalizálás lineáris programozással, illetve (kevert) egészértékű lineáris programozással támogatásának. Ennek jellemző megoldásait és "trükkjeit" itt nem áll módunkban tárgyalni. Kiemelendő azonban, hogy egy felhőben a virtuális gépek felhasználók által meghatározott hibatűrési követelményeknek megfelelő, optimalizált terítés tervezése nagyon hasonlít ahhoz a problémához, amit jelen fejezetben vizsgálunk. A virtuális gép terítési problémára egy hibatűrést is figyelembe vevő újszerű megközelítést adtam szerzőtársaimmal a [26] cikkben; ezt a megközelítést adaptálom jelen munkámban. [26] egyben hivatkozza az általános célú adatközpontok VM terítés optimalizálásának főbb ismert megközelítéseit is.

3.3. Magas rendelkezésreállási minták és azok megvalósítása OpenStack-ben

Az informatikai rendszerek tervezésekor a rendelkezésreállás kérdése megkerülhetetlen. A felhők helyzete különösen bonyolult a szolgáltatások nagy száma és eltérő viselkedése miatt. Ezért egy OpenStack rendszer tervezésénél a különböző szolgáltatásokat érdemes külön tárgyalni.

Az OpenStack ide vonatkozó útmutatója alapján a nagy rendelkezésreállású rendszerek célja az, hogy minimalizáljuk a leállást és az adatvesztést [31]. A hibák bekövetkezése elkerülhetetlen, ezért a rendelkezésreállás növelése érdekében el kell érni, hogy a szolgáltatás hiba esetén is üzemeljen, továbbá a hibákat a lehető leggyorsabban kezelni kell.

A hibakezelés elengedhetetlen eszköze a szolgáltatások folyamatos felügyelete. Egy ilyen méretű rendszerben nem mindig derül ki, hogy pontosan hol is van a meghibásodás, hiszen egy hibajelenségnek több oka is lehet. A hibaok meghatározásához (hibakorreláció [21]) rendszerszintű diagnosztikára van szükség, mely különböző pontokon szondákkal vizsgálja a szolgáltatásokat. A szondák kimenete és a telepített rendszer és összefüggései ismeretében már behatárolhatjuk a hiba forrását, ami nem csak az operátorok számára fontos, hiszen automatikus hibaelhárítási eljárások is szükségessé válhatnak a helyes működés fenntartása érdekében, például a hibás szolgáltatás leállítása. Bár az OpenStack-hez az általános monitoring szoftverek mellett specifikusabb eszközök is vannak (Pl.:Stacktach [20]), a konfigurációk sokszínűsége miatt az átfogó diagnosztika továbbra is kihívások elé állítja a rendszermérnököket.

A magas rendelkezésreállás elérésére az egyik legfőbb eszköze az egyszeres hibapontok számának csökkentése, amit redundancia bevezetésével érhetünk el. A redundanciának viszont ára van, és ez nem csak a teljesítmény és erőforrásbeli. A többszörözött elemek és az őket összekötő hibatűrési mechanizmusok üzemeltetése önmagában költséget és rizikót hordoz.

Állapotmentes és állapottartó szolgáltatások

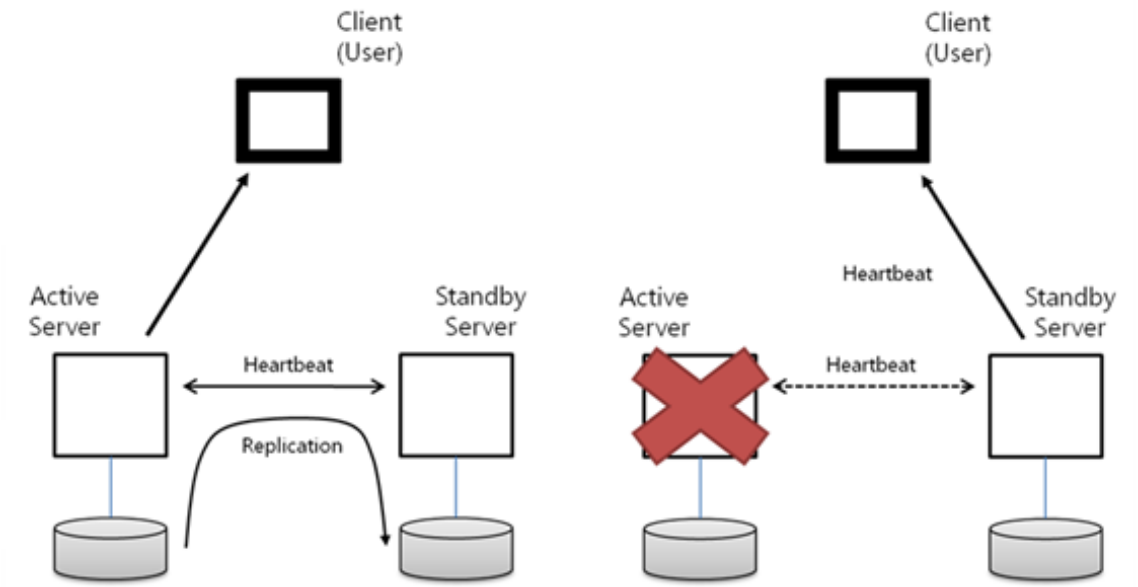
Állapotmentes (stateless) szolgáltatásoknál a redundancia megvalósítása lényegesen egyszerűbb. Ezek olyan szolgáltatások, melyek belső állapota nem változik meg az elvégzett műveletek hatására. Ilyenek például az API szolgáltatások, melyek ugyan az adatbázisban és a többi szolgáltatásban okozhatnak változásokat, de csak összekötői szerepet vállalnak. Mivel nincs belső állapotuk, nem különböznek egymástól a futó példányok a felhasználók szemszögéből, ez megkönnyíti az esetleges feladatátvételt.

Az állapottal rendelkező (stateful) szolgáltatások kérések kiszolgálásakor megváltoztathatják a belső állapotukat. A feladatátvétel megoldása tehát sokkal összetettebb a

probléma, hiszen az állapotaikat szinkronizálni kell, hogy a helyes működést biztosítsuk. A szolgáltatások azonban változatos állapotokkal rendelkezhetnek, melyeket különbözőképpen lehet szinkronizálni. Jó példát adnak a tárolást megvalósító szolgáltatások, melyeknél az adatot replikálni kell, és számolni kell a szinkron replikáció teljesítményköltségével, vagy az aszinkron replikációnál az esetleges adatvesztéssel. A probléma redundáns közös tárolóval vagy replikációt biztosító hálózati fájlrendszerrel megoldható, de külön figyelmet igényel.

Elméletileg lehetőség lenne arra, hogy a futtató platform (fizikai vagy virtuális) szintjén valósítsunk meg állapotszinkronizációt. Ez azonban úgy fizikai hardver, mint virtuális gép esetén extrém költséges, bár lehetséges. úgynevezett lockstep üzemmódban futó processzorok már a mainframe-ekben is léteztek, és ma már virtuális gépekre is elérhető VM lockstep megoldás. Ehelyett az állapotszinkronizációt magasabb szinten köztes rétegben, vagy az alkalmazás szintjén szokás megvalósítani.A magában a felhőben futtatott alkalmazásokra így az ajánlott megoldás önmagában hibatűrő elosztott alkalmazások használata. Mint látni fogjuk, ugyanez a megközelítés alkalmazandó a felhőt megvalósító szoftverkomponensekre is.

A redundancia megvalósítási mintái



3.1. ábra. Egy tipikus aktív-passzív redundancia minta.

Akkor beszélhetünk aktív-aktív redundanciáról, ha a hibamentes működés esetén az összes elem működik, és a terhelés el van osztva közöttük. A biztonságos működéshez természetesen a terhelést egy elemnek is el kell tudni viselnie. Ez a hibatűrési minta biztosítja a hiba utáni leggyorsabb helyreállást, viszont rendkívül költséges hiszen sokkal nagyobb összteljesítményre van szükség, mint azt az hibamentes de redundanciamentes működés megkívánná.

Ha a szolgáltatások állapottartók, az állapotok kölcsönös szinkron replikálására lehet szükség, hiszen itt a redundanciát alkotó elemek mindegyikétől jöhet állapotváltozás. A konzisztencia fenntartása így költségessé válik, és a legtöbb esetben csak alkalmazásszinten megoldható.

Ezzel szemben aktív-passzív (3.1.) mintában egy adott időpontban csak egy példány aktív. Ez a módszer költségkímélőbb, hiszen a passzív szolgáltatást futtató erőforrásokat legalább részben kikapcsolhatjuk vagy másra használhatjuk. Az állapottartó szolgáltatások állapotreplikációja is egyszerűsödik, hiszen csak egy irányból lehet változás, ezért nem kell külön figyelmet fordítani a konzisztenciára. A hátrányai között szerepel az aktív-aktív mintával szemben, hogy itt nem számolhatunk terheléselosztással, illetve probléma esetén a leállás is hosszabb, hiszen a meghibásodás eseményét előbb észlelni kell, majd kezdeményezni a feladatátvételt.

Aktív-passzív redundancia az OpenStack-ben

Az OpenStack keretrendszer nem rendelkezik beépített redundanciával, hanem az ilyen célra általánosan használt szoftverek és módszerek használatosak [31]. Redundancia szempontjából a legegyszerűbb eset az aktív-passzív minta alkalmazása állapotmentes szolgáltatáson. Állapotmentessége miatt a redundanciát biztosító összes szolgáltatás futhat a konzisztencia veszélyeztetése nélkül, és feladatátvételkor csak a szolgáltatás címét kell átállítani. Virtuális IP címek alkalmazásával ez automatikussá és egyszerűvé tehető.

Lényegesen nagyobb kihívás az állapottartó szolgáltatások redundanciája, melyhez az OpenStack a Pacemaker [35] nevű klaszter szoftvert ajánlja, mely előre meghatározott szabályok szerint mozgathatja az erőforrásokat a gépek között. Ebben a kontextusban az erőforrás lehet egy szolgáltatás futtatásának joga, egy virtuális IP cím birtoklásának a joga. Az erőforrásokat úgynevezett erőforrásleírók definiálják, melyek meghatározzák a jellegét és a feladatátvételhez szükséges lépéseket. A rendszer biztosítja, hogy ne fusson egyszerre két ugyanolyan szolgáltatás, illetve probléma esetén a másik gépen el is tudja indítani azt. Az erőforrások átmozgatásakor függőségeket és más paramétereket is figyelembe tud venni.

Az OpenStack-hez használhatók előre elkészített erőforrás-leírók, így a konfigurálás három lépésre rövidül. Az első lépésben fel kell tölteni a leírókat a Pacemaker fürtbe, a második lépésben be kell állítani a virtuális IP címet az adott szolgáltatáson, végül a szolgáltatást használó többi szolgáltatásban is.

A Pacemaker nem valósít meg replikációt, ugyanakkor képes azt vezérelni erőforrásokon keresztül. Amennyiben egy szolgáltatásnak szüksége van replikációra, azt meg lehet valósítani például a DRBD nevű hálózati replikációs protokollal, melyhez ugyancsak létezik erőforrás-leíró.

A Pacemaker a fürt belső kommunikációjára, és ezzel a szolgáltatások felügyeletére a Corosync [8] szoftvert alkalmazza.

Aktív-aktív redundancia az OpenStack-ben

Az aktív-aktív redundancia állapotmentes szolgáltatások esetében a HAProxy [22] nevű szoftverrel lehetséges, ami átjárót képez a szolgáltatást elérni szándékozó kliensek és a kiszolgálók között. Terheléselosztást végez, tehát igyekszik a klienseket egyenlően elosztani, továbbá nagy rendelkezésreállást biztosít, mivel képes észrevenni, ha egy kiszolgáló már nem látja el a feladatát, és kiveszi az aktív kiszolgálók listájából.

A konfiguráció csupán abból áll, hogy megadjuk a HAProxy beállításaiban, hogy melyik porton milyen kéréseket fogadunk, és azokat milyen kiszolgálókhoz lehet továbbítani.

A HAProxy mint szolgáltatás magas rendelkezésreállásához több megoldás is létezik, az OpenStack vonatkozó dokumentuma a Keepalived nevű szoftvert ajánlja virtuális IP-cím átadásra.

Állapottartó szolgáltatásoknál sajnos a kölcsönös szinkron replikáció miatt csak alkalmazásszintű megoldásokat lehet alkalmazni. Az OpenStack útmutatója ajánl megoldást

az adatbázis és az üzenetsor bróker aktív-aktív többszörözésére is. A MySQL szerver beállítására a Galera [29] szoftverrel lehetséges, amely szinkronizációt és terheléselosztást biztosít. Az üzenetsorokat biztosító RabbitMQ [37] beépítve támogat fürtöket és nagy rendelkezésreállású üzenetsorokat.

3.4. A probléma absztrakciója

A korábban ismertetett EMF alapú konfigurációs tér és telepítési modellek nem használhatóak fel közvetlenül optimalizációra, mivel rendkívül összetettek. Könnyen készíthető viszont olyan modellvezérelt alkalmazás, mely a modellekből a szükséges információkat megtartva egyszerűsített modellt készít. Amennyiben ez az alkalmazás úgy készül el, hogy a két modell elemei egymásnak kölcsönösen megfeleltethetők, az optimalizáció eredménye könnyen felhasználható lesz a telepítési modell megalkotásakor.

A lokális függőségek esetében azt az egyszerűsítést alkalmazzuk, hogy az egymástól függő szolgáltatásokat összevonjuk egy csoportba, az erőforrásigényeiket összeadjuk, a hibatűrési paraméterek esetében pedig a megkeressük a maximumot. Ez nem változtat az eredményen, hiszen az egymástól függő szolgáltatások úgyis együtt maradnak az elhelyezés során.

A szolgáltatáscsoportokat hibatűrési paraméterekben jelölt mennyiségben kell a modellhez adni. A megkövetelt redundancia számával megegyező esetben jelölni kell, hogy kötelező telepíteni a szolgáltatást.

A modell a következő információkat tartalmazza:

- Számítógépek listája
  - Erőforrásigényeik
- Szolgáltatások listája
  - Erőforrásigényeik
  - Kötelező szolgáltatás (igaz/hamis)
  - Szolgáltatástípus

Az erőforrásmodell a következő egész szám típusú attribútumokból áll:

- core - A processzormagok száma.
- hdd - Háttértár mérete (GB).
- memory - Memória mérete (GB).

A (3.2. ábra) szemlélteti az optimalizáció folyamatát. Az optimalizációhoz a Gurobi nevű szoftvert használtam [1].

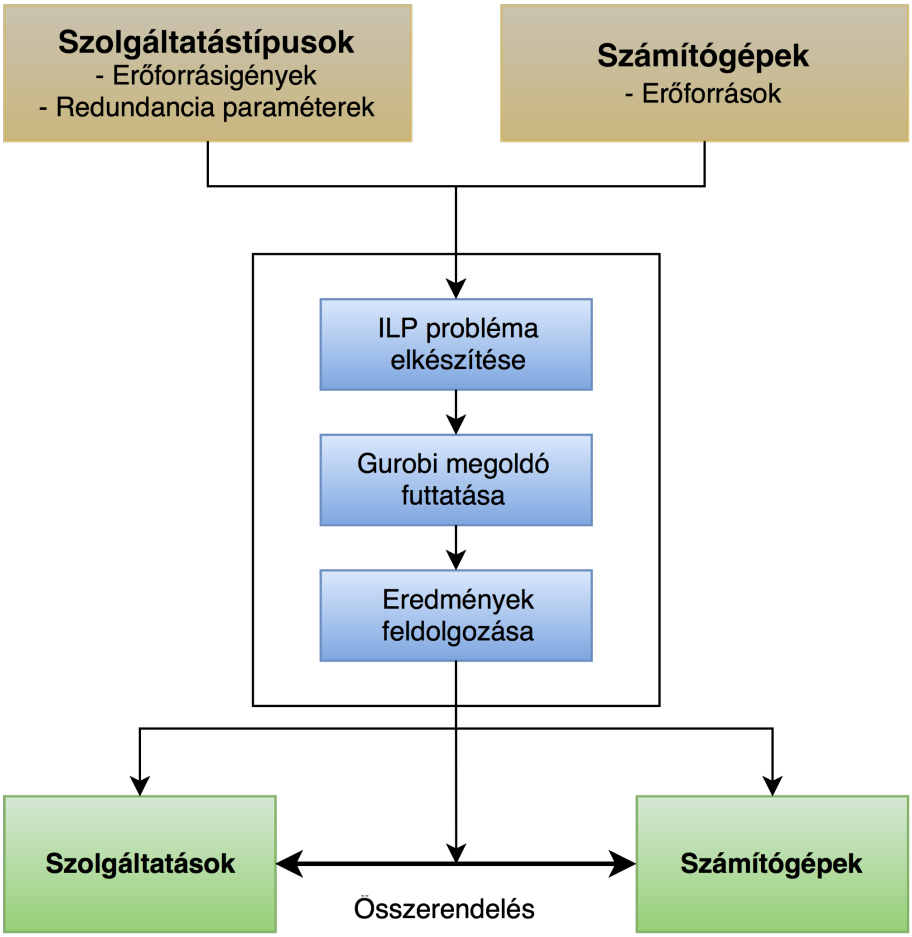
3.5. A probléma formalizálása

A két probléma hasonlósága miatt itt a korábban publikált lineáris modell és ILP megvalósítás [26](3.2 fejezet) egy átdolgozott, az új problémakörre adaptált változatát közlöm.

Az elérhető számítógépek halmazát P jelöli. A számítógépek erőforrásait a $cap(p, r) \in \mathbb{R}^+$ jelöli, ahol $p \in P$ és $r \in R = \{core, hdd, memory\}$.

Az elérhető szolgáltatások halmazát S jelöli. A szolgáltatások erőforrásait a $req(s, r) \in \mathbb{R}^+$ jelöli, ahol $s \in S$ és $r \in R$. Az S halmaz két diszjunkt részhalmazból áll: $S = S_0 \cup S_1$. S_0

[1] Gurobi: `http://www.gurobi.com/`



3.2. ábra. A lineáris programozás alapú optimalizációs megközelítés szemléltető ábrája

a kötelezően telepítendő szolgáltatásokat tartalmazza, amelyeket el kell helyezni, S_1 pedig a nem kötelező szolgáltatásokat tartalmazza, amelyek esetében előnyt jelent, ha sikerül számukra helyet találni.

Célunk, hogy kiszámoljunk egy $pmap : S' \to P$ hozzárendelést, ahol $S_0 \subseteq S' \subseteq S$, tehát minden S_0-beli, és még valamennyi S_1-beli szolgáltatást tartalmaz.

Kényszerek

Egy számítógépen futó programok összes erőforrásigénye nem lehet magasabb, mint az ott rendelkezésre álló erőforrások. Az erőforrásokkal kapcsolatos kényszereket a következőképpen fogalmazhatjuk meg:

$$\forall p \in P, r \in R : \sum_{s \in S : map(s) = p} req(s, r) \leq cap(p, r) \tag{3.1}$$

A redundancia megtartása érdekében szükség van negatív szomszédossági kényszerekre az ugyanolyan típusú szolgáltatások esetében, hiszen ha a redundanciában résztvevő két szolgáltatás ugyanazon a gépen fut, akkor nem jelentenek védelmet hardverhibák ellen. Jelölje a szolgáltatástípusok halmazát T. A $tmap : S' \to T$ hozzárendelés adja meg a szolgáltatások típusát, tehát ha $s1, s2 \in S; tmap(s1) = tmap(s2)$, akkor ugyanolyan típusú szolgáltatások. Így a kényszer:

$$\neg\exists s1, s2 \in S : tmap(s1) = tmap(s2), pmap(s1) = pmap(s2), s1! = s2 \tag{3.2}$$

Minden $t \in T$ szolgáltatástípusra $ST(t) \subseteq S$ jelölje azokat a szolgáltatásokat, amelyek a t típusba tartoznak. Tehát ha $s \in ST(t)$ akkor $tmap(s) = t$.

A célfüggvény

A rendszer költsége két tényezőből tevődik össze. Az első az aktív számítógépek számát jelöli. Aktív számítógépnek azt tekintjük, amelyikre el van helyezve legalább egy szolgáltatás. A nem használt számítógépeket fel lehet használni a felhő erőforrásaiként, ezért fontos hogy minél kevesebb aktív gép vegyen részt a telepítésben. Az aktív gépeket $A(pmap)$ jelöli:

$$A(pmap) = \big|\, \{p \in P : \exists s \in S\ pmap(s) = p\} \,\big|.$$

A költség kiszámításának második tényezője a nem kötelező szolgáltatások elhelyezését jutalmazza. Ezek számát a $B(pmap) = |S' \setminus S_0|$ adja meg.

$A(pmap)$ értékét minimalizálni, $B(pmap)$ értékét pedig maximalizálni szeretnénk, ráadásul úgy, hogy a két tényező fontosságát súlyozni lehessen. A minimalizálandó célfüggvény tehát $\alpha \cdot A(pmap) - \beta \cdot B(pmap)$, ahol α és β nem-negatív konstansok.

3.6. Egészértékű lineáris programozási modell

Legyen $n = |S|$, $m = |P|$ és $q = |T|$. A szolgáltatásokra, számítógépekre és szolgáltatástípusokra a következőképpen hivatkozhatunk:
S: s_i $(i = 1, \ldots, n)$
P: p_j $(j = 1, \ldots, m)$
T: t_k $(k = 1, \ldots, q)$

Hogy a fenti problémát megfogalmazhassuk egészértékű lineáris programként (ILP), a következő bináris változókra lesz szükségünk:

$$Alloc_{i,j} = \begin{cases} 1 & \text{ha } s_i \text{ a } p_j \text{ gépen van elhelyezve} \\ 0 & \text{egyébként} \end{cases}$$

$$Active_j = \begin{cases} 1 & \text{ha } p_j \text{ aktív} \\ 0 & \text{egyébként} \end{cases}$$

Az egészértékű programot a következőképpen készíthetjük el:

$$\text{minimize} \quad \alpha \cdot \sum_{j=1}^{m} Active_j - \beta \cdot \sum_{s_i \in S_1} \sum_{j=1}^{m} Alloc_{i,j} \tag{3.3}$$

$$\text{subject to} \quad \sum_{j=1}^{m} Alloc_{i,j} = 1, \qquad \forall s_i \in S_0 \tag{3.4}$$

$$\sum_{j=1}^{m} Alloc_{i,j} \leq 1, \qquad \forall s_i \in S_1 \tag{3.5}$$

$$\sum_{s_i \in ST(t_k)} Alloc_{i,j} \leq 1, \qquad \forall t_k \in T, \forall p_j \in P \tag{3.6}$$

$$Alloc_{i,j} \leq Active_j, \qquad \forall i,\ \forall j \tag{3.7}$$

$$\sum_{i=1}^{n} req(s_i, r) \cdot Alloc_{i,j} \leq cap(p_j, r), \qquad \forall j,\ \forall r \in R \tag{3.8}$$

$$Alloc_{i,j} \in \{0, 1\}, Active_j \in \{0, 1\}, \qquad \forall i,\ \forall j \tag{3.9}$$

A célfüggvényt (3.3) a korábban leírtak alapján az aktív számítógépek és az allokált nem kötelező szolgáltatások számából áll. Az első kényszer (3.4) azt biztosítja, hogy a kötelező szolgáltatások pontosan egy gépen szerepeljenek, míg a következő (3.5) azt biztosítja, hogy a nem kötelező szolgáltatások legfeljebb egy gépen szerepeljenek. Az (3.6 kényszer azért felel, hogy egy számítógépen ne fusson többször ugyanaz a szolgáltatás (3.2). A (3.7) kényszer azt írja elő, hogy szolgáltatásokat csak aktív gépen lehet elhelyezni. Az 3.8 egyenlet az erőforrások kényszereit adja meg (3.1), miszerint az egy gépen futó szolgáltatások összes erőforrásigénye nem lehet magasabb a rendelkezésre álló erőforrásoknál. Végül meg kell határozni a bináris változókat (3.9).

3.7. Eredmények bemutatása

Az optimalizációs megközelítést úgy teszteltem, hogy egy valódi telepítéshez hasonló hardverrendszert és szolgáltatásokat reprezentáló adathalmazon különböző súlyozással végeztem méréseket. Az OpenStack alapvető komponenseiből összeállított szolgáltatáscsoportok vesznek részt a tesztelésben. Ezek erőforrásigényei csak az optimalizáció tesztelésére szolgálnak (3.1.), tehát nem mérések eredményei. A valódi erőforrásigények természetesen a rendszer tervezett terhelésétől függnek, így a rendszermérnökök feladata megítélni mértéküket.

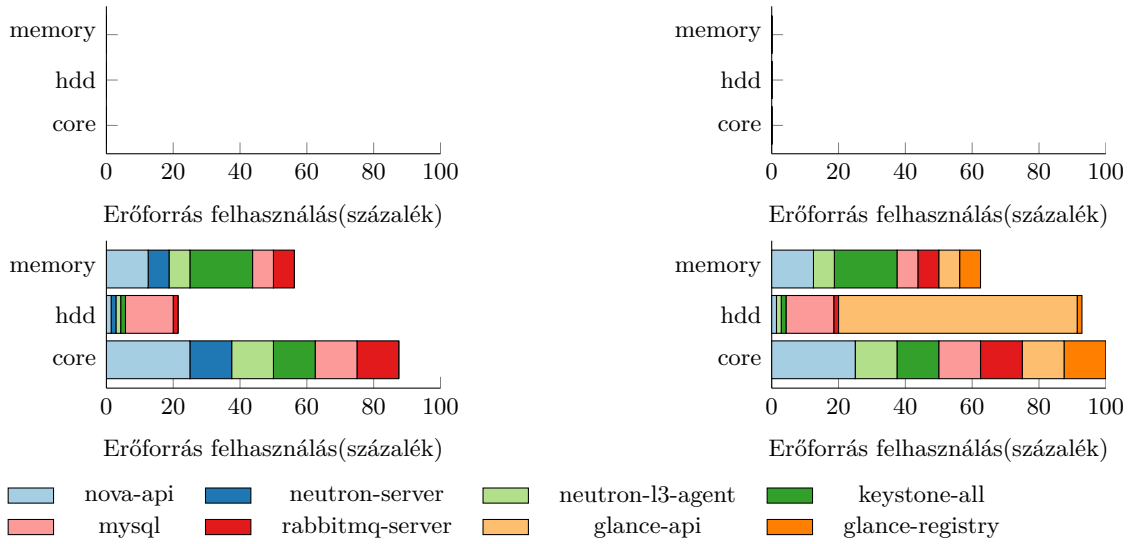
Szolgáltatás	Processzormag	HDD	Memória	Minimum redundancia	Kívánt redundancia
nova-api	2	1	2	2	4
neutron-server	1	1	1	1	4
neutron-l3-agent	1	1	1	2	3
keystone-all	1	1	3	2	3
mysql	1	10	1	2	3
rabbitmq-server	1	1	1	2	3
glance-api	1	50	1	1	3
glance-registry	1	1	1	1	3

3.1. táblázat. A különböző szolgáltatáscsoportok erőforrásigényei

A négy egyforma számítógép áll rendelkezésre, melyek egyenként 8 processzormaggal, 70 GB tárhellyel és 16 GB memóriával rendelkeznek.

Költségminimalizáló eset

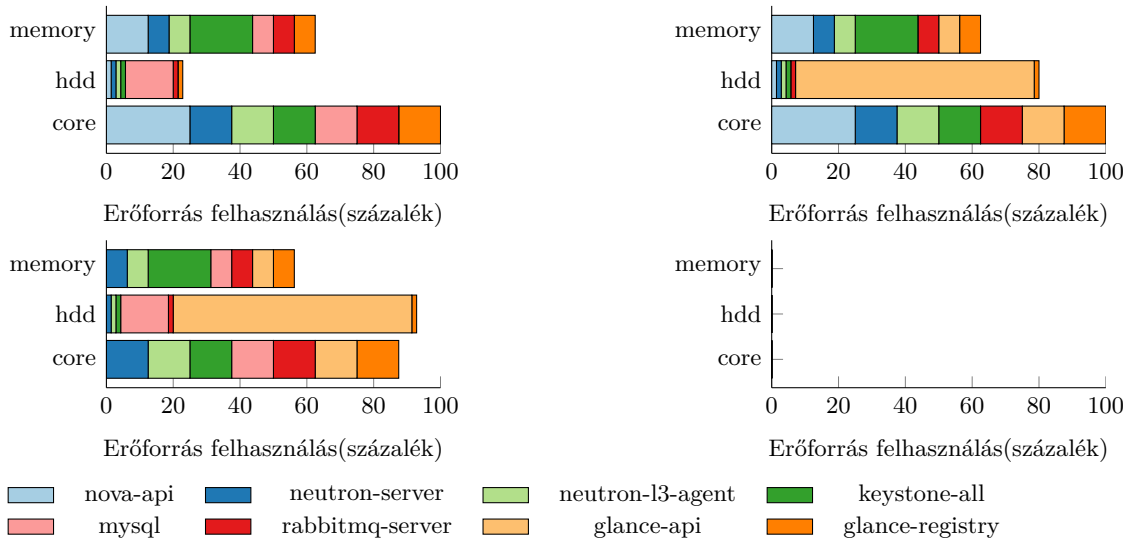
Ebben az esetben az α paraméter értéke 100%, a β pedig 0%, tehát a nem használt gépek számát optimalizáljuk. A diagramon (3.3. ábra) jól látszik, hogy csak két számítógépre van szükség a szolgáltatások telepítéséhez. Ez a megoldás azonban a szűkös erőforrások miatt csak az elvárt legkisebb redundanciát tudja teljesíteni.



3.3. ábra. A szolgáltatások elhelyezkedése a költségminimalizáló esetben

Köztes eset

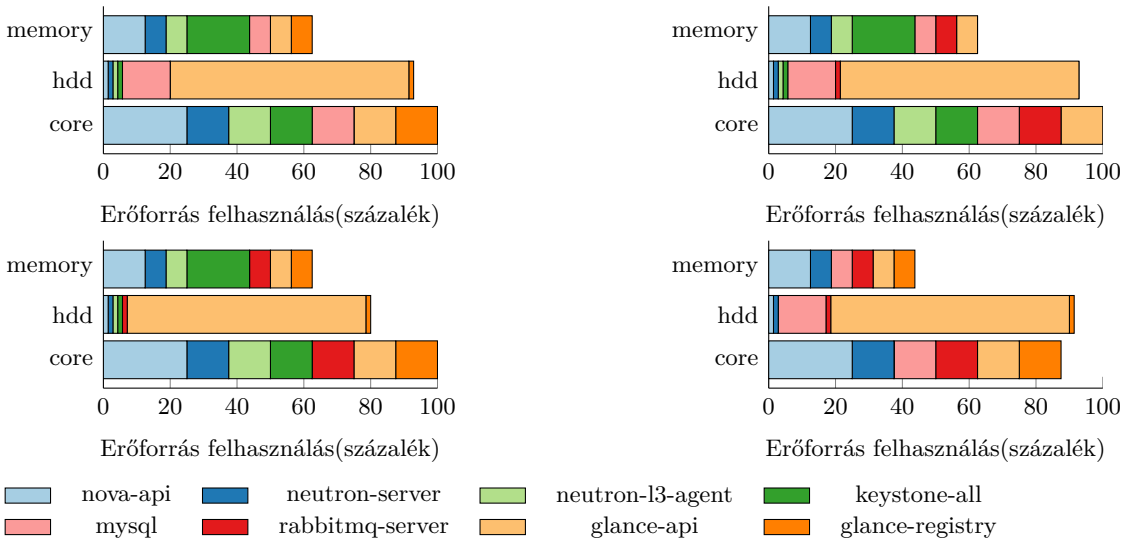
A β értékét 14%-ra növelve kedvezőbbé válik az optimalizációs modell számára a redundancia növelése (3.4. ábra). Itt már három számítógépet használ fel, és nyolc olyan szolgáltatást telepít, amelyet nem kötelező, de javítja a redundanciát.



3.4. ábra. A szolgáltatások elhelyezkedése megnövelt β értékkel

Redundanciamaximalizáló eset

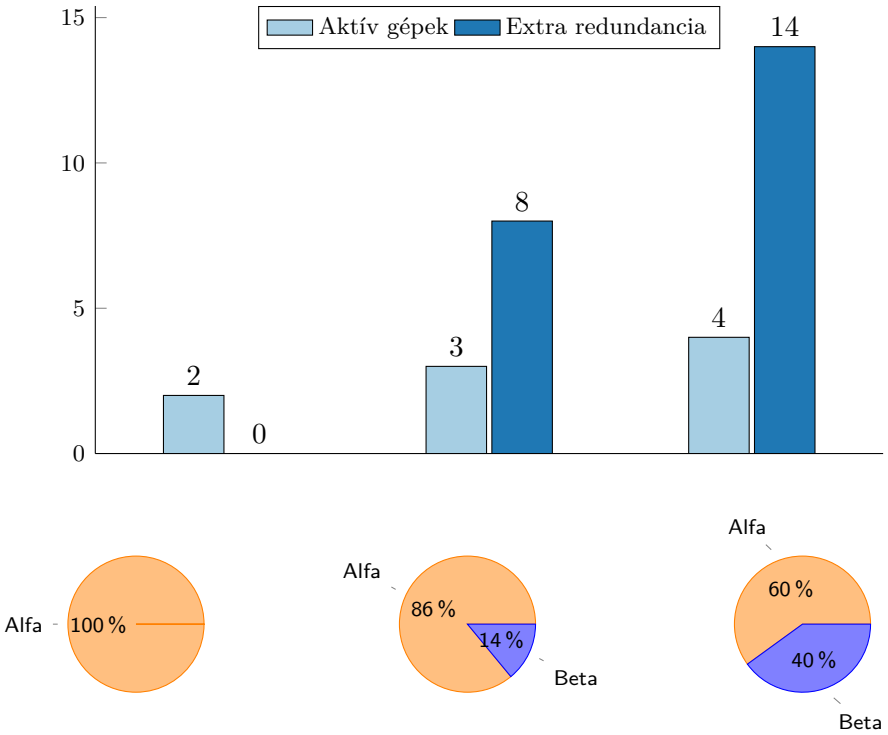
A β értélét 40%-ra növelve már mind a négy gépet felhasználja, és a annyi szolgáltatást telepít a gépekre, amennyit csak lehetséges (3.5. ábra).



3.5. ábra. A szolgáltatások elhelyezkedése a redundancianövelő esetben

Eredmények összehasonlítása

Az eredmények tükrében (3.6. ábra) kimondható, hogy az optimalizációs modell megtartja az erőforrás korlátokat, és a beállított súlyoknak megfelelően változtatja a redundancia szintjét.



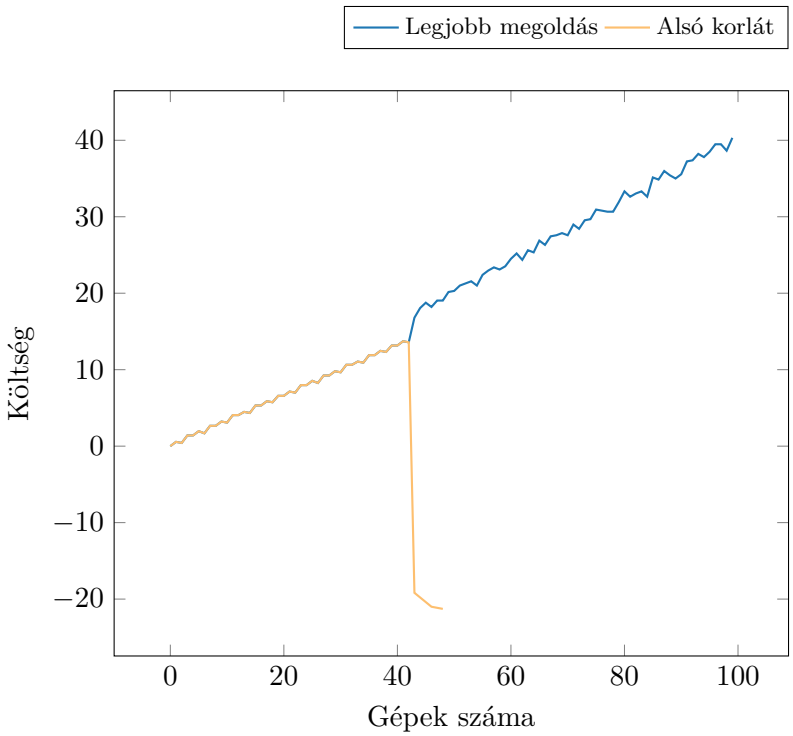
3.6. ábra. A mérési eredmények összehasonlítása

Skálázhatóság

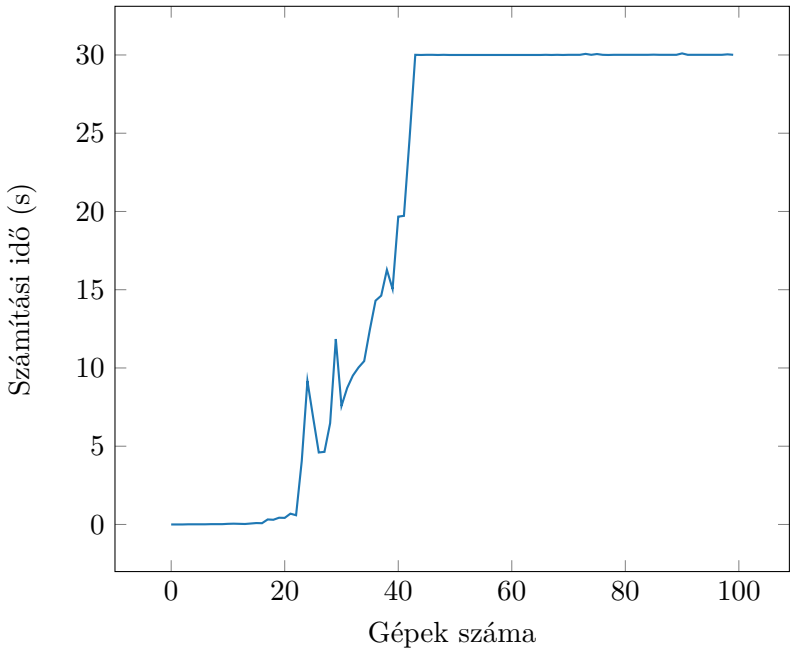
Az optimalizáció skálázhatóságának mérését az optimalizációs modell elemeinek többszörözésével végeztem. A számítógépek számát használtam a modellméret mérőszámának, és a többi elemet aszerint méreteztem, kerekítést alkalmaztam. Az optimalizációt egy asztali számítógépen végeztem, melynek Intel Core i7 870-es processzora van, és 16 GB memóriája. Harminc másodperces időkorláttal végeztem a méréseket, a gépek számát egyesével növelve. Az időkorlát lejártakor az optimalizáció leáll, és az addig talált legjobb megoldást adja eredményül

A mérés eredménye a 3.8. ábrán látható. Az alsó ábrán látható, hogy a a futási idő exponenciálisan nő, és a 43-44-elemet tartalmazó modellek esetén már nem fér bele az időkeretbe. A felső ábrán(3.7. ábra) látható, hogy amint az időkeret miatt a számítások nem tudnak befejeződni, a legjobb megoldás ugyan valamennyivel a megugrik, de alapvetően az optimális megoldásokhoz közeli meredekséget tart, tehát azt mondhatjuk, hogy közelítő eredményt ad. A megtalált alsó korlát értéke láthatóan meredeken letör 43-44 elemszám környékén és nem is tér vissza. Részben ez okozza, hogy a futásidő ennyire meredeken emelkedik. Bár egyenlőre nem tudtuk megoldani ezt a problémát, de a későbbiekben a modell finomításával lehet, hogy orvosolható a probléma.

Bár az optimalizációs modellen még sokat lehet javítani, a megoldások azonban az 50 gépből álló modellek után sem állnak messze a vélt optimumtól, ráadásul a fél perces időkorlát helyett egy valós életbeli feladatnál ennek sokszorosát is érdemes lehet rászánni. A kísérlet alapján ez a technológia alkalmas lehet OpenStack telepítések optimalizálására.



3.7. ábra. Az optimalizáció eredménye a gépek számának függvényében (30 másodperces időkorláttal)



3.8. ábra. Az optimalizáció számítási ideje a gépek számának függvényében (30 másodperces időkorláttal)

3.8. Robusztus particionálás kényszerei

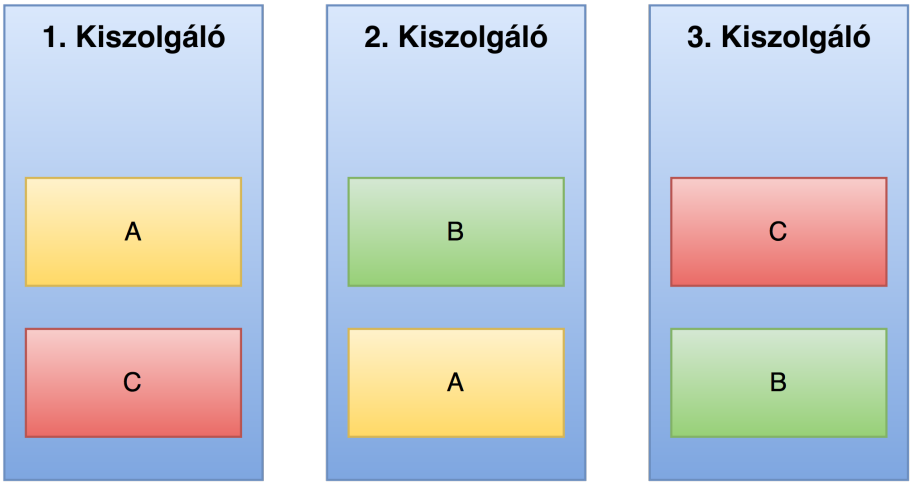
A robusztus particionálás [24][39] az elhelyezni kívánt elemek olyan partíciókba sorolását jelenti, amelyek garantálják, hogy az egy bizonyos feladatot ellátó szoftverek nem kerülnek mindig ugyanazokkal a szoftverekkel közös hardverre. Ez jelentősen csökkenti a hardverterhelés alapú hibaterjedést. Bár ez a módszer alapvetően a beágyazott rendszerek világában használatos, az ötletet alkalmazhatjuk az OpenStack szolgáltatásainak elhelyezésénél is. A hibaterjedés esélyeinek csökkentésén kívül itt biztonsági előnyei is lehetnek, hiszen ha az egyik szolgáltatástípus sérülékeny, akkor az adatbiztonság (security) szempontjából jobb, ha minden szolgáltatástípusnak van olyan példánya, amelyik nincs egy gépen a sérülékeny ("megtörhető") szolgáltatástípussal.

A mechanizmust a 3.9. ábra szemlélteti. Három duplikált szolgáltatásra az ábrán bemutatott módon teljesül a robusztus particionálás követelménye: hiába van adott esetben A-nak olyan közös módusú hibája, mely minden példányát egyszerre érinti, és ami a példányokat futtató kiszolgálón futó többi szolgáltatást is negatívan érinti, a másik két szolgáltatásnak továbbra is van legalább egy olyan példánya, mely nem fut közös kiszolgálón egy A példánnyal sem. Így a fennmaradó két szolgáltatás aktív marad, még ha csökkent összkapacitással is. Ugyanez igaz a B és C típusú szolgáltatások példányainak ilyen közös módusú hibáira is. Jelen kontextusunkban ilyen hiba lehet egy adott típusú kérés hatására például memóriaszivárgás, mellyel szemben az ugyanazon a kiszolgálón futó többi szolgáltatás a gyakorlatban jellemzően nem védett.

Ezt az igényt a 3.4 fejezetben ismertetett formális problémához a következő kényszerrel lehet hozzáilleszteni:

$$\forall t1, t2 \in T, t1 \neq t2 :$$
$$\exists s1 \in ST(t2), \forall s2 \in ST(t2) : pmap(s1) \neq pmap(s2)$$

Tehát bármelyik $t1, t2$ típuspárra igaz, hogy létezik olyan $t1$ szolgáltatás, amelyik egyik $t2$-beli szolgáltatással sincs közös gépen.



3.9. ábra. Az A-tól C-ig jelölt szolgáltatások robusztus particionálása két szintű redundancia mellett

Ennek lineáris programozás megoldásához vezessünk be egy segédváltozó halmazt (gyakorlatilag egy három dimenziós tömböt): $Z_{1,1,1}, \ldots, Z_{m,q,q}$, ahol m a gépek száma, q pedig a szolgáltatástípusok száma. A robusztus particionálás a következő korlátok teljesülése esetén valósul meg:

$$\sum_{s_i \in ST(t_a)} Alloc_{i,j} \geq Z_{j,a,b}, \qquad \forall t_a, t_b \in T, \forall p_j \in P \tag{3.10}$$

$$1 - \sum_{s_i \in ST(t_b)} Alloc_{i,j} \geq Z_{j,a,b}, \qquad \forall t_a, t_b \in T, k \neq g, \forall p_j \in P \tag{3.11}$$

$$\sum_{j=1}^{m} Z_{j,a,b} \geq 1, \qquad \forall t_a, t_b \in T \tag{3.12}$$

A fenti kényszerek megértéséhez egy adott $t_a, t_b \in T$ szolgáltatástípus párra nézzük meg a kényszerek hatását, amik azt hivatottak elérni, hogy létezzen legalább egy olyan számítógép, amelyiken fut t_a, de nem fut t_b. A (3.10) és (3.11) korlátok azt érik el, hogy Z változó értéke csak akkor lesz pozitív, ha az adott gépen fut t_a, és nem fut t_b beli szolgáltatás. Ebben az esetben a $0 \leq Z \leq 1$, Minden más esetben $Z \leq 0$. A (3.12) kényszer miatt lesz legalább egy olyan Z érték, ami pozitív, tehát lesz legalább egy olyan gép, ami megfelel a kívánalmainknak.

A módszer tesztelésekor a már ismertetett rendszert optimalizáltam az $\alpha = 100\%$ értékével. Ez a beállítás korábban két aktív számítógépre telepítette a rendszert és nem telepített olyan szolgáltatásokat, amiket nem volt kötelező. A robusztus particionálás kényszereit két gépen természetesen nem lehetett kielégíteni. Az optimalizálás végeredménye egy olyan telepítés lett, amely öt gépen van szétosztva, és tizenegy darab szolgáltatást telepít a kötelezőeken felül, mert a kötelezőekből nem tudta volna kielégíteni a robusztus particionálás kényszereit.

4. fejezet

Összefoglalás

Munkám során az OpenStack példáján keresztül bemutatom, hogy a modellvezérelt fejlesztés eszközkészletének néhány alapvető technikáját adaptálva miként lehet megoldani ezeket a problémákat. Központi elemként létrehoztam egy úgynevezett konfigurációs tér metamodellt, mellyel leírhatók az OpenStack mint keretrendszer elemei, ezek lehetséges beállításai, valamint a közöttük húzódó kapcsolatok és függőségek.

Kidolgoztam és implementáltam egy transzformációt, mely egy konfigurációs tér modellből (pl. egy konkrét OpenStack verzió) egy olyan metamodellt generál, mely segítségével konkrét OpenStack telepítéseket lehet leírni és validálni. Műszakilag új eredményként a legfontosabb OpenStack részszolgáltatásokra megvalósítottam egy konkrét konfiguráció modelleket Docker konténerekben automatikusan megvalósító mechanizmust.

A konfigurációs modell által meghatározott, teljesítendő szolgáltatásbiztonsági igények (például aktív-aktív replikációk) és egyéb optimalizációs kritériumok (pl. robusztus particionálás, költség) kielégítésének támogatására felállítottam és implementáltam egy lineáris programozási modellt, mely a konfigurációs modellből kiindulva a részszolgáltatáspéldányokat futtató gépekhez rendel. A hibatűrési mechanizmusok kiválasztására bemutatok egy kezdeti kockázat-vezérelt metodológiát.

4.1. További lehetőségek

A munkám továbbfejlesztéseként szeretném a későbbiekben megvalósítani, hogy az OpenStack telepítését több eszközzel és több célplatformra támogassa. Az OpenStack jellegét tekintve moduláris és kiterjeszthető, ezért szükség lehet egy olyan mechanizmusra, amely a modell kiterjesztését teszi lehetővé, hogy az a folyton változó OpenStack-et követni tudja.

A Docker alapú gyors prototípizáció lehetővé teszi a konfigurációk gyors tesztelését. Ez számos módon használható, például a jövőben szeretném a konfigurációs tér modellek helyességét funkcionális teszteléssel automatikusan ellenőrizni. További érdekes lehetőség, hogy egy ilyen megoldás akár az OpenStack integrációs tesztelésében is hatékony segítséget nyújthat.

Az optimalizációs eljárást számos módon szeretném bővíteni, hogy a valós életbeli problémákhoz jobban megfeleljen. Ilyen például az a megoldás, hogy a hibatűrési paramétereket egy absztrakciós szinttel magasabbra emelve olyan klaszterszintű hibatűrési tulajdonságokat is hozzávegyünk, mint például a meghibásodás nélkül eltűrt maximális fizikai hibák száma.

Az itt bemutatott megközelítés egy önmagában zárt, modellben megfogalmazott igényektől a konkrét telepítésig ívelő munkafolyamat prototípusa. Tágabb kontextusban azonban érdekes és mindenképp további vizsgálatokat igénylő kérdés az, hogy adható-e lehetőség szerint kockázatvezérelt metodológia arra, hogy az OpenStack szolgáltatások replikáció

ciós stílusát és replikáció szintjét ne az alkalmazó mérnöknek kelljen deklarálnia, hanem ezekre is matematikailag precíz módszertant adjunk.

Köszönetnyilvánítás

Mindenek előtt szeretném megköszönni a segítséget konzulensemnek, Kocsis Imrének, hogy vezette és segítette munkámat. Továbbá szeretném megköszönni Dr. Mann Zoltánnak a szakmai segítségét a lineáris programozás témakörét illetően.

Végül szeretnék köszönetet mondani a Pro Progressio Alapítványnak, a támogatásuk nélkül nem jöhetett volna létre ez a dolgozat.

Irodalomjegyzék

[1] ANVIL Documentation - ANVIL 2015-dev documentation.
URL `https://anvil.readthedocs.org/en/latest/`.

[2] Gábor Bergmann – Ákos Horváth – István Ráth – Dániel Varró – András Balogh – Zoltán Balogh – András Ökrös: Incremental evaluation of model queries over emf models. In *Model Driven Engineering Languages and Systems*. 2010, Springer, 76–90. p.

[3] Lorenzo Bettini: *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013, Packt Publishing Ltd.

[4] Chapter 1. architecture - openstack installation guide for red hat enterprise linux, centos, and fedora - icehouse. URL `http://docs.openstack.org/icehouse/install-guide/install/yum/content/ch_overview.html`.

[5] Chef | IT automation for speed and awesomeness.
URL `https://www.chef.io/chef/`.

[6] Coalfire: Spotlight on cloud computing: An overview, 2015.
URL `http://www.coalfire.com/Resources/Spotlight-Compliance`. [Online; accessed 10-October-2015].

[7] Compass. URL `http://www.syscompass.org/`.

[8] Corosync. URL `https://corosync.github.io/corosync/`.

[9] Crowbar. URL `http://http://crowbar.github.io/`.

[10] György Csertán – Gábor Huszerl – István Majzik – Zsigmond Pap – András Pataricza – Dániel Varró: Viatra-visual automated transformations for formal verification and validation of uml models. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on* (konferenciaanyag). 2002, IEEE, 267–270. p.

[11] DevStack - an OpenStack Community Production - DevStack.
URL `http://docs.openstack.org/developer/devstack/`.

[12] Docker: Docker engine, 2015. URL `https://www.docker.com/docker-engine`. [Online; accessed 11-October-2015].

[13] Docker Docs. URL `https://docs.docker.com/`.

[14] Eclipse. URL `https://eclipse.org/home/index.php`.

[15] Eclipse Modeling Project. URL `http://www.eclipse.org/modeling/emf/`.

[16] Emf-incquery. URL `https://www.eclipse.org/incquery/`.

[17] Wes Felter – Alexandre Ferreira – Ram Rajamony – Juan Rubio: An updated performance comparison of virtual machines and linux containers. *technology*, 28. évf. (2014), 32. p.

[18] Roy Thomas Fielding: *Architectural styles and the design of network-based software architectures*. PhD értekezés (University of California, Irvine). 2000.

[19] Foreman. URL `http://theforeman.org/`.

[20] Foreman. URL `stacktach.readthedocs.org`.

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

[22] Haproxy. URL `http://www.haproxy.org/`.

[23] Inception - OpenStack. URL `https://wiki.openstack.org/wiki/Inception`.

[24] Shariful Islam – Neeraj Suri – András Balogh – György Csertán – András Pataricza: An optimization based design for integrated dependable real-time embedded systems. *Design Automation for Embedded Systems*, 13. évf. (2009) 4. sz., 245–285. p.

[25] Frédéric Jouault – Freddy Allilaire – Jean Bézivin – Ivan Kurtev – Patrick Valduriez: Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (konferenciaanyag). 2006, ACM, 719–720. p.

[26] Imre Kocsis – Zoltán Adám Mann – Dávid Zilahi: Optimal deployment for critical applications in infrastructure as a service.

[27] MAAS. URL `https://maas.ubuntu.com/`.

[28] Peter M. Mell – Timothy Grance: Sp 800-145. the nist definition of cloud computing. Jelentés, Gaithersburg, MD, United States, 2011.

[29] Mysql - galera. URL `http://galeracluster.com/products/`.

[30] OpenStack Architecture Design Guide - current.
URL `http://docs.openstack.org/arch-design/content/`.

[31] OpenStack High Availability Guide - current. URL `http://docs.openstack.org/high-availability-guide/content/index.html`.

[32] OpenStack Installation Guide for Ubuntu 14.04 - juno. URL `http://docs.openstack.org/juno/install-guide/install/apt/content/`.

[33] OpenStack Open Source Cloud Computing Software.
URL `http://www.openstack.org/`.

[34] OpenStack Software | Distributions | Mirantis.
URL `https://www.mirantis.com/products/mirantis-openstack-software/`.

[35] Pacemaker. URL `http://clusterlabs.org/`.

[36] Puppet Labs: IT Automation Software for System Administrators.
URL `https://puppetlabs.com/`.

[37] Rabbitmq-ha. URL `https://www.rabbitmq.com/ha.html`.

[38] Redgate: A comprehensive introduction to cloud computing, 2015. URL https://www.simple-talk.com/cloud/development/a-comprehensive-introduction-to-cloud-computing/. [Online; accessed 10-October-2015].

[39] John Rushby: Partitioning for safety and security: Requirements, mechanisms, and assurance. CR-1999-209347. NASA Contractor Report, 1999. június, NASA Langley Research Center. Also to be issued by the FAA.

[40] Smi.
URL http://www.snia.org/tech_activities/standards/curr_standards/smi.

[41] Dave Steinberg – Frank Budinsky – Ed Merks – Marcelo Paternostro: *EMF: eclipse modeling framework*. 2008, Pearson Education.

[42] Supervisord. URL http://supervisord.org/.

[43] TripleO - OpenStack. URL https://wiki.openstack.org/wiki/TripleO.

[44] Zoltán Ujhelyi – Gábor Bergmann – Ábel Hegedüs – Ákos Horváth – Benedek Izsó – István Ráth – Zoltán Szatmári – Dániel Varró: Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98. évf. (2015), 80–99. p.

[45] Steve Vinoski: Advanced message queuing protocol. *IEEE Internet Computing*, 2006. 6. sz., 87–89. p.

[46] Virtualization for your modern datacenter and hybrid cloud | Microsoft. URL http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[47] vSphere ESXi Bare-Metal Hypervisor | United States.
URL http://www.vmware.com/products/esxi-and-esx/overview.

[48] Welcome to the Juju charm browser | Juju. URL https://jujucharms.com/.

[49] What is Docker. URL https://www.docker.com/whatisdocker.

[50] The Xen Project, the powerful open source industry standard for virtualization. URL http://www.xenproject.org/.

[51] K. Zeilenga: Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. RFC 4510 (Proposed Standard), 2006. június.
URL http://www.ietf.org/rfc/rfc4510.txt.