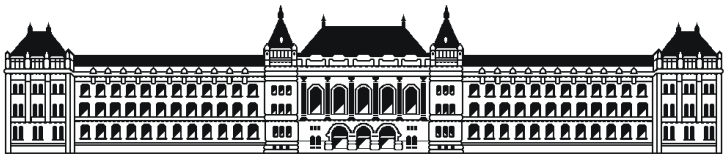


Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Számítógép virtualizációs megoldások szabványos menedzselésének vizsgálata

Diplomaterv

Készítette
Laposa László

Konzulens
Micskei Zoltán

2010. május 14.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

DIPLOMATERV FELADAT

Laposa László
szigorló informatikus hallgató részére
(nappali tagozat informatika szak)

Számítógép virtualizációs megoldások szabványos menedzselésének vizsgálata
(a feladat szövege a mellékletben)

A tervfeladatot összeállította és a tervfeladat tanszéki konzulense:

Micskei Zoltán
tanársegéd

A záróvizsga tárgyai: Valószínűségszámítás
Adatbázisok
Infromatikai rendszerek szolgáltatásbiztonsága

A tervfeladat kiadásának napja:
A tervfeladat beadásának határideje:

dr. Majzik István	*dr. Horváth Gábor*
docens, diplomaterv felelős	docens, tanszékvezető

A tervet bevette:
A terv beadásának dátuma:
A terv bírálója:

MELLÉKLET

Számítógép virtualizációs megoldások szabványos menedzselésének vizsgálata

Az utóbbi évtizedben a számítógép virtualizációs megoldások egyre jobban elterjedtek az informatikában. A virtualizáció segíthet tesztrendszerek kiépítésére, szerverek konszolidációjára, vagy akár a katasztrófa elhárítás részeként hiba esetén másodlagos infrastruktúra biztosítására. A korábban egy gyártó által dominált piac képe mára azonban jelentősen megváltozott. A VMware mellett a Citrix, Microsoft vagy az Oracle/Sun is kínál ilyen termékeket. A zárt megoldások mellett egyre több nyílt forráskódú szoftver is elérhető, a Xen vagy a KVM mára már része a legtöbb Linux disztribúciónak. Manapság nem ritka azonban, hogy egy-egy informatikai rendszerben ezek a megoldások vegyesen jelennek meg, többféle virtualizációs keretrendszer is használatban van. Ennek megfelelően megjelent az igény, hogy az ilyen megoldásokat egységesen lehessen kezelni. Az egyes termékek általában biztosítanak valami távoli menedzsment felületet, azonban ezek megvalósítása és funkciója jelentősen eltérhet. A probléma megoldására kezdenek megjelenni a kapcsolódó ipari szabványok, azonban jelenleg ezeknek a használata még kezdeti stádiumban van. A diplomatervező hallgató feladata az egyes távoli menedzsment felületek és szabványok megismerése, majd egy egyszerű mintaalkalmazáson ezek használhatóságának vizsgálata. A diplomaterv megoldása a következő részfeladatok teljesítését igényli:

1. Mutassa be a számítógép virtualizációt, és tekintsen át a jelenleg elterjedt megoldások közül néhányat.

2. Vizsgálja meg, hogy az egyes megoldások milyen távoli menedzsment felületeket biztosítnak, valamint, hogy milyen szabványos vagy platform-független megoldások léteznek a virtualizációs keretrendszerek menedzsmentjére.

3. Tervezzen meg és valósítson meg egy olyan távoli menedzsment alkalmazást, mely képes többféle virtualizációs megoldást kezelni, és alapvető lekérdezési feladatokat ellátni.

4. Foglalja össze a tapasztalatait a megvizsgált megoldásokkal és technológiákkal kapcsolatban, és értékelje az elkészült alkalmazást.

Micskei Zoltán
tanársegéd

HALLGATÓI NYILATKOZAT

Alulírott *Laposa László*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Tudomásul veszem, hogy az elkészült diplomatervben található eredményeket a Budapesti Műszaki és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaira felhasználhatja.

Budapest, 2010. május 14.

_____

Laposa László
hallgató

Tartalomjegyzék

Kivonat

A számítógép virtualizáció egyike az informatika legdinamikusabban fejlődő területeinek. Fejlődésének mozgatói között számos technológiai és gazdasági hatás megtalálható, amelyek egymást erősítik. A költséghatékony rendszer üzemeltetés egyik lehetséges eszköze, amely alapeleme lehet egy hatékony és megbízható infrastruktúrának.

A számítógép virtualizáció felhasználásának terjedésével párhuzamosan újabb és újabb termékek jelentek és jelennek meg, amely a széles választék mellett az igények még pontosabb testre szabhatóságát eredményezi. Azonban az újabb termékek rendszerünkbe való integrálásával, a korábbi egységes architektúra heterogenizálásával elveszítünk valamit, amit a virtualizáció kezdeti bevezetésével nyertünk, az erőforrások egységes kezelésének lehetőségét. Ennek a folyamatnak az oka az egyes termékek által biztosított menedzsment felületek és szolgáltatások eltérésében keresendő.

Ahhoz, hogy az igényeinknek megfelelő testre szabhatóságot és az uniform erőforrás kezelés lehetőségét is megtarthassuk egységes menedzselhetőséget biztosító eszközökre vagy szabványokra van szükség. Jelen dolgozat célja, hogy a kellő háttérismeretek összefoglalását követően bemutassa egy olyan alkalmazás tervezési, implementálási és tesztelési folyamatát, amely funkciókészletével a ma széles körben elterjedt virtualizációs megoldások egységesített menedzsmentjében nyújthat segítséget.

A munkafolyamat eredményeként megszületett alkalmazás lehetőséget biztosít a felhasználónak, hogy egyszerre több és különböző típusú virtualizációs rendszert felügyeljen. A megvalósított funkciók nem csupán a virtuális erőforrásokra korlátozódnak, bizonyos keretek között lehetőséget biztosítanak a kiszolgáló fizikai gép kezelésére is. Az alkalmazás segítségével befolyásolhatjuk az egyes virtuális gépek állapotát és lekérdezhetjük a legfontosabb jellemzőiket, a rendszer adottságainak megfelelően másolatot készíthetünk róluk, vagy megsemmisíthetjük őket.

Mindezen funkciók eléréshez egy egyszerűen használható grafikus kliens alkalmazás nyújt felületet.

Abstract

The computer-virtualization is one of the most dynamically improving part of computer technology. Among the motivating forces can be found many of economical and technological effect, which reinforce each other. Virtualization is a possible tool for cost-effective computer system operation, which can be a key element of an efficient and reliable infrastructure.

In parallel with spreading use of computer virtualization a lot of new product are released, which leads to wider choice and more precise customization of our needs. However with integration of new product into our former system and making that heterogeneous, we lose something, that we won, when we introduced the virtualization, the possibility of unified resource management. It happens because, there are some differences between products, they have different management interfaces and provide different services.

If we want to keep both flexibility and facility of uniform resource handling, we need tools or standards, which ensure uniform management capability. The main goal of this document is to show the reader over the process of design and implementation of an application, which makes available the unified management of wide-spread virtualization products, after we've summarized some needed background knowledge.

The application was born as result of work process ensures opportunity for user to manage multiple, different virtualization system. The implemented functions aren't confined to only virtual resources, some can be used to handle physical resources in limited ways. With help of this application we can influence the states of virtual machines, we can query their vital properties, according to the capabilities of managed system we can copy or destroy them.

All of these functions can be reached by an easy-use graphical surface, which is provided by client application.

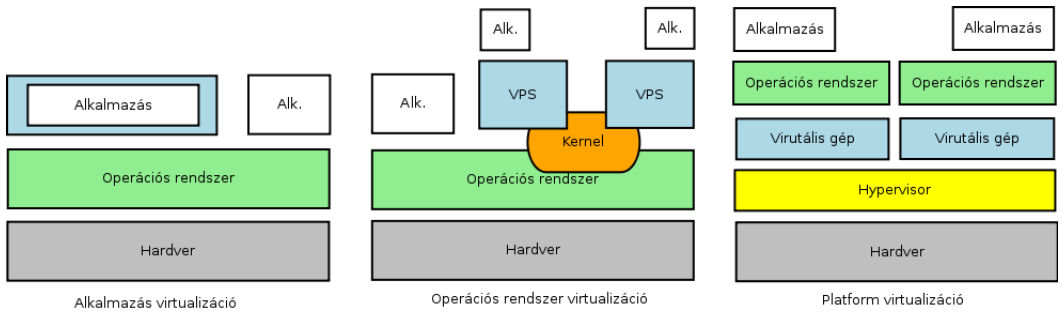
1. fejezet

Bevezetés

A virtualizáció napjaink informatikai világának egyik legdinamikusabban fejlődő területe, amely számos technikai és szolgáltatás jellegű probléma megoldásához biztosít eszközt. A virtualizáció egy gyűjtőfogalom, amely három fő irányzatot foglal magába. Ha egy számítógép architektúrális ábráján lentről indulunk el, akkor az első szint, amely a virtualizáció célterülete lehet a hardver és az operációs rendszer között húzódó réteg. Az erre a területre pozicionálódó alkalmazások *platform szintű virtualizációt* valósítanak meg, amelynek célja, hogy a rá épülő réteg partícionálható legyen. Magyarán ne csak egyetlen operációs rendszer futtatására legyen lehetőség egy időben, egyetlen fizikai gépen. Ezek a megoldások többnyire egy komplett virtuális gépet nyújtanak az egyes operációs rendszerek futtatásához.

A következő szint, ahol a virtualizáció szóba jöhet az operációs rendszer szintje, többnyire ebből fakadóan *operációs rendszer szintű virtualizációnak* is nevezik az ilyen jellegű megoldást. Míg a platform szintű virtualizáció esetében minden virtuális gép saját valós vagy virtuális erőforrással rendelkezik, addig ebben az esetben közös, valós erőforrásokon osztoznak az egyes futó rendszerek. Nem csak a hardver erőforrásokat használják közösen, hanem a legfontosabb szoftver komponenst, a rendszer magot (kernel) is, amely számos előnnyel és hátránnyal is jár.

A virtualizáció harmadik gyakran alkalmazott területe az *alkalmazás virtualizáció*, amely architektúra független alkalmazások fejlesztését teszi lehetővé, burkoló eljárások segítségével.



1.1. ábra. Virtualizációs technikák

Míg az alkalmazás virtualizáció a szoftverfejlesztés költségeinek csökkentésében játszhat fontos szerepet, addig az operációs rendszer és a platform virtualizáció a költség és energia hatékony rendszer üzemeltetésnek lehet fontos eszköze. Az utóbbi kijelentés alátámasztásában fontos szerepet játszik az a tény, hogy a dedikált erőforrások jelentős hányada a működési idő jelentős részében a potenciális teljesítményüknek csak töredékét használják ki. Gyakran e mögött a pazarló magatartás mögött kényszerűség húzódik meg, sok olyan eset és forgatókönyv létezik, amelyben egy szolgáltatás részelemei megkívánják az egymástól való operációs rendszer szintű izolációt. Bár a folyamatok kiszolgálására elegendő lenne egyetlen fizikai erőforrás, ezen kényszerből fakadóan mégis többet kell használnunk, s viselni a járulékos költségeket. A virtualizáció nélkül az ilyen jellegű problémákra nem találnánk más gyors és egyszerű, úgymond polcról levehető megoldást, mint az erőforrás többszörözést, míg a segítségével pusztán logikai erőforrás többszörözésre van csak szükségünk, amelyet a vásárolt vagy alkalmazott termék nagy valószínűséggel automatizáltan végez.

A virtualizáció és azon belül is a platform virtualizáció másik fontos célterülete lehet az igény szerinti számítás vagy On-demand Computing, amely napjainkban egyre nagyobb teret nyer. Az ezen a területen történő sikeres alkalmazásában kiemelkedő szerepet játszik az a virtualizáció által nyújtott nagymértékű rugalmasság, amellyel más rendszerépítő komponensek esetén nem találkozhatunk. Ez a rugalmasság két fontos adottság találkozásának eredménye, egyrészt a virtualizáció nagyfokú szabadságot biztosít a kívánt rendszer struktúra kialakításához, másrészt pedig mindehhez gyors reagálóképességet biztosít. Mindezek teszik lehetővé, hogy a virtualizáció segítségével olyan rendszereket alkossunk , amelyek alkalmazkodni képesek a környezetből érkező igényekhez, így biztosítva az elvárt szolgáltatás színvonalat.

Jelen dolgozat tárgyterületének a platform virtualizációt tekinti a korábbiakban említett kiemelkedő szerepe miatt, így a továbbiakban annak ismertetésével és benne rejlő lehetőségek kiaknázásával foglalkozik.

A platform virtualizáció mind hardver mind szoftver irányú megközelítésben igen intenzíven fejlődő terület. A két területen elért eredmények ösztönzően hatnak egymásra és részben egy közös fejlődési irányt határoznak meg. Jelentős szerepe van a mai szerver konszolidációs folyamatban, mint költség és energia hatékony eszköznek, és ugyanezen okokból adódóan közkedvelt rendszerépítő elem. A virtualizáció terjedésével számos új cég jelentkezett és jelentkezik saját termékével. Mint ahogy egy hagyományosnak mondható rendszer esetében itt is igény lehet a távoli beavatkozás és menedzselhetőség lehetőségére, így ma már szinte minden megoldás rendelkezik valamiféle távoli elérési ponttal, amelyen keresztül az általa nyújtott funkcionalitások többsége igénybe vehető. Ez a felület többnyire valamiféle saját szabványon alapul, bár megfigyelhető egyfajta uniformizálási folyamat is, amely egyelőre a virtuális gépek leírásának egységesítésére irányul. Ilyen szabvány a DMTF által gondozott Open Virtualization Format [7], amely virtuális gépeken futtatható alkalmazások csomagolására és elosztására optimalizáltak.

A termékek számának növekedéssel a választási lehetőségek skálája egyre bővül, amely lehetővé teszi, hogy igényeinknek legmegfelelőbb megoldást válasszuk. Ez a folyamat

azonban azzal a sajnálatos mellékhatással jár, hogy egyre több és több felülettel kell megismerkednie a felhasználónak, a fejlesztőknek pedig egyre több szabványt és megoldást kell implementálni, amely költséges és időigényes. Ennek kiküszöböléséhez szükség lenne egy olyan szabványra amely nem csak a virtuális gépek egységes ábrázolását rögzíti, hanem egy olyan menedzsment felületet is, amely különböző megoldások egységes kezelését teszi lehetővé, és esetlegesen az egyes megoldások közötti átjárást is biztosítja. Amíg ez nem létezik egyetlen dolgot tehetünk, saját absztrakciós réteget hozunk létre az egyes termékek közötti eltérések elfedésére, így biztosítva az egységes kezelhetőséget.

A dolgozat elsődleges feladatának tekinthető annak megvizsgálása, hogy miként lehet egy olyan, az egyes termékek közötti különbségeket áthidaló alkalmazást fejleszteni, amely segítségével egy időben több virtualizációs alkalmazás is irányítható távolról. Ehhez első lépésben mélyebb ismeretekre van szükség a platform virtualizációval kapcsolatban. Fontos tisztában lenni az alapvető fogalmakkal és a virtualizáció támasztotta követelményekkel. Ahhoz, hogy valamit sikeresen tudjunk ellenőrzésünk alá vonni érdemes azt behatóbban tanulmányozni, ezért fontos az egyes termékek megismerése és lehetőségeik áttekintése. A távoli beavatkozás megvalósításához mindenképpen meg kell ismerni az egyes megoldások által nyújtott távoli menedzsment felületeket, valamint megvizsgálni, hogy ezek a felületek milyen, nem gyártóspecifikus szabványok segítségével érhetőek el, így esetlegesen tovább növelve az egyes komponensek újrahasznosíthatóságát.

Ezeket a lépéseket sorra véve eljuthatunk arra a pontra, amikor átfogó képpel és kellő ismeretanyaggal rendelkezünk ahhoz, hogy nekiálljunk egy menedzsment felület és azt megvalósító alkalmazás megtervezéséhez, majd ezek alapján történő implementációjához. A megvalósítandó alkalmazásnak nem az a célja, hogy meglévő, több virtualizációs terméket támogató szoftver másolata legyen, hanem az, hogy a munka során megismerjük az egyes termékeket, interfészeiket és azok használatát, így nyerve egy teljesebb képet róluk. A megvalósítandó alkalmazással szembeni követelmények közül mindenképpen mellőzni kell a teljességet, hiszen egy annak megfelelő menedzsment termék fejlesztése meghaladja a dolgozat kereteit, viszont a minél szélesebb körű alkalmazhatóságot követendő irányelvnek tekinthetjük. Mindezek alapján az elkészítendő alkalmazás nyelvéül a Java fog szolgálni, márt csak azért is mert ez a nyelv kellően gazdag szabványtámogatással rendelkezik, lehetővé téve, hogy a feladat specifikus részekre koncentráljunk és ne vesszünk el az alkalmazott kommunikációs protokollok részleteiben.

2. fejezet

A számítógép virtualizáció alapjai

2.1. A virtualizáció fogalomköre

2.1.1. Virtualizáció

A virtualizáció fogalmát sok helyen és sokféle, de többnyire hasonló értelemben használják. Jelen esetben számítógép virtualizációról beszélünk, amelyet egy absztrakciós folyamatként képzelhetünk el, ahol a különböző célok elérése érdekében fizikailag nem vagy csak részben létező erőforrás állományt hozunk létre, melyet operációs rendszerek azonos fizikai eszközön történő, párhuzamos működtetéséhez használunk fel.

Tehát az absztrakciós folyamat egyrészt nem létező erőforrások létrehozását, másrészt létező, valós fizikai eszközök partíciónálását jelenti. A virtualizáció alkalmazásának számtalan oka és célja lehet, kezdve a költséghatékony rendszerüzemeltetéstől egészen a biztonságkritikus rendszerek kialakításáig.

A témakör alapvetéseit a hetvenes években alapozták meg, talán a legismertebb és leghíresebb munka a Gerald J.Popek és Robert P.Goldberg nevéhez fűződő Formal Requirements for Virtualizable Third Generation Architectures című publikáció [17], amelyben formalizált alakban fogalmazzák meg egy architektúra virtualizálhatóságának követelményeit. A virtualizációval szemben támasztott követelményeiket az alábbi három feltételben foglalhatjuk össze.

- **Ekvivalencia:** egy virtuális gépen futó alkalmazás számára olyan környezetet kell biztosítani, mintha az közvetlenül egy vele megegyező fizikai gépen futna.
- **Erőforrás felügyelet:** a virtualizációs folyamatok vezérlőjének teljes felügyelettel kell bírnia a virtualizált erőforrások felett.
- **Hatékonyság:** a virtuális gépen futó kód jelentős hányadának végrehajtása a vezérlő beavatkozása nélkül kell hogy történjen.

Tanulmányukban a virtualizálhatóság megállapításának egyik fő szempontjaként a processzor utasításkészlet részhalmazait jelölték meg, az ő besorolásuk szerint egy processzor utasítás az alábbi három halmazba sorolható be.

- **Privilegizált utasítások:** olyan utasítások, amelyeket csak rendszer módban hajthat végre a processzor.
- **Irányítás-érzékeny utasítások:** olyan utasítások, amelyek megváltoztatják az erőforrások konfigurációját.
- **Viselkedés-érzékeny utasítások:** ebbe a csoportba azok az utasítások sorolhatók amelyek eredménye az erőforrások konfigurációjától függ.

2.1.2. Hypervisor, Virtual Machine Monitor

A virtualizáció során végrehajtandó feladatokat végző és vezénylő komponenst *hypervisor* vagy *Virtual Machine Monitor* néven szokás emlegetni. A hypervisor szót általában olyan dedikált rendszerek esetén használják, amelyek kizárólag virtualizációs feladatokat látnak el, míg a Virtual Machine Montior-t többnyire általános célra szánt operációs rendszereken futó virtualizációs alkalmazások esetében használják, bár sok esetben a névhasználat nem ennyire konvencionális. Az operációs rendszer szintű virtualizációs alkalmazások esetén ez a komponens egy közebékelődő réteget képez az erőforrások és a virtuális gépek között, részben elrejtve azok elől a valós fizika eszközöket, részben pedig megmutatva azokat. Megvalósítástól függően a hypervisorok többnyire módosított operációs rendszer magok, amelyek a célnak megfelelően lettek átszabva. Ez a technikai megvalósítás többnyire a Linux/Unix alapú termékek esetén jelentős, a másik lehetőség a meglévő kernel érintetlenül hagyását biztosító, azt kiegészítő és használó kernel modul alapú megközelítés. Az alkalmazás alapú megvalósítások esetében mindenképpen számolni kell azzal, hogy a hypervisor vagy VMM nem rendelkezik közvetlen felügyelettel az erőforrások teljes skálája felett, így némi adminisztrációs többletköltségből származó teljesítményveszteségre kell készülnünk.

2.1.3. Vendég

Vendégnek vagy guest-nek hívjuk azokat a számítógépeket, amelyek fizikailag nem képeznek önálló egységet, úgymond csak virtuálisan léteznek a virtualizáció absztrakciós szintjén belül, a működésükhöz használt virtuális erőforrásokat a hypervisor képezi le valós fizikai eszközökre, amelyek paraméterei gyakran eltérnek a logikai eszközétől, ilyen módon olyan erőforrásokat is létre tudunk hozni, amelyekkel igazából nem is rendelkezünk. Mindezeket a tényeket figyelmen kívül hagyva a virtuális gépek a felhasználók és a futtatott operációs programok tekintetében mára már semmiben nem különböznek valós társaiktól, legfeljebb a teljesítményükben találhatunk némi eltérést.

2.1.4. Gazdagép

Gazdagépnek (host) nevezzük azt a fizikai gépet, amely a virtualizáció során szükséges feladatokat ellátja, és fizikai erőforrásokat biztosít a virtuális gépek futtatásához. A virtualizáció szoftver oldali komponenseivel párhuzamosan a fizikai eszközök is jelentős fejlődésen mentek keresztül, a virtualizáció fontosságának felismerését követően számos

olyan újítás jelent meg a hardverekben, amely virtualizáció hatékonyságának növelését célozza meg. Ilyen fejlesztés a processzorok virtualizációs támogatása vagy az eszköz és memória virtualizációt segítő funkcionális egységek.

2.2. A virtualizációs technikák csoportosítása

A virtualizált rendszerek alapját képező technikai megoldásokat sokféle szempont szerint csoportosíthatjuk. Ezek közé a szempontok közé tartozhat magának a terméknek a szerkezeti jellege, rendszeregységként figyelembe vett mérete. Az utóbbi csoportosítás szerint különbséget teszünk hosted és bare metal jellegű virtualizáció között, ezt a csoportosítást használja James Smith és Ravi Nair is virtualizációról szóló cikkében [23]. Míg az első esetben tulajdonképpen egy különleges jogokkal rendelkező alkalmazás oldja meg a virtualizációs feladatokat, addig az utóbbi esetben egy erre a feladatra készített vagy átalakított operációs rendszer látja el ugyanezeket. A csoportosítás egyik legkézenfekvőbb és legtöbbször használt szempontja a processzor virtualizáció típusa szerint történő csoportosítás, ez alapján három nagy csoportot különbözethetünk meg: emuláció, paravirtualizáció, teljes vagy más néven hardveresen támogatott virtualizáció.

2.2.1. Emuláció

Az *emuláció* a virtualizációs megoldások eszközkészletének egyik legrégebbi és legtágabb értelmű eleme, többnyire akkor használják, amikor egy teljes architektúra szimulálására van szükség. Használatával lehetőség nyílik a különböző hardver architektúrákat áthidaló virtuális rendszerek építésére. Széleskörű alkalmazhatóságának legnagyobb akadálya, a vele járó szignifikáns teljesítményveszteség. Általánosságban elmondható, hogy az emuláció önmagában való alkalmazása 90%-os adminisztrációs költséggel jár, ezért többnyire kerülendők az ilyen megoldások, azonban alkalmazásának teljes mértékű kiküszöbölése sok esetben nem lehetséges.

2.2.2. Bináris fordítás

Az emuláció egy speciális részhalmazát képezi a *bináris fordítás*, amely azonos architektúrájú gazda és vendég gép esetén alkalmazható. Segítségével a processzor utasítások jelentős része közvetlenül hajtható végre, a problémás utasítások hatását pedig kiváltó kódokkal emulálják. Mindez úgy történik, hogy közben a vendég rendszer az egészből semmit nem vesz észre. Problémás processzor hívásokkal akkor találkozhatunk, amikor olyan hardver architektúrát alkalmazunk, amely a virtualizáció szempontjából nem kívánatos vagy éppen veszélyes utasításokkal és állapotokkal rendelkezik. Ezek az utasítások többnyire a processzor állapotának egészét érintő, módosító utasítások, amelyek egyik virtuális gépből történő meghívásuk esetén valamennyi, az adott processzort használó virtuális gépre kihatással lennének. Például a klasszikus x86 architektúra is ilyen. Mivel ez a megoldás nem igényli a teljes utasításkészlet hatásának kiváltását, ezért az emulációnál sokkal nagyobb hatékonyságot biztosít.

2.2.3. Paravirtualizáció

Azon operációs rendszerek esetében, ahol a forráskód szabadon elérhető és szükség szerint módosítható, a kényes utasítások problémájára nyújtott bináris fordítást alkalmazó megoldásoknál jóval hatékonyabb módszereket is találhatunk, mégpedig a rendszer megfelelő módosításának formájában. A virtualizációnak azon formáját, amikor a vendég operációs rendszeren eszközölnek változtatás a futtathatóság vagy a futási teljesítmény elérése érdekében *paravirtualizációnak* nevezzük. Mivel ez a módszer a bináris fordítást teljes egészében kiküszöböli, az emulációnál jóval nagyobb teljesítmény elérést teszi lehetővé. További előnyei közé tartozik hogy működési elvéből adódóan jóval egyszerűbb felépítésű hypervisort vagy Virtual Machine Monitort igényel. A paravirtualizáció hatékonyságának oka egyben széleskörű használhatóságának egyik akadálya is, hiszen az olyan operációs rendszerek, amelyek nem rendelkeznek nyílt forráskóddal nem futtathatók paravirtualizációt alkalmazó rendszereken.

2.2.4. Hardveresen támogatott virtualizáció

A hatékonyság és a széleskörű alkalmazhatóság céljának szem előtt tartásával született meg a virtualizáció egy régi-új, a korábbiaktól eltérő szemlélettel bíró megoldása, a hardveresen támogatott virtualizáció, melyet teljes virtualizációként is emlegetnek. Régi, mert már a virtualizáció őskorát jelentő 70-es években is alkalmazták az IBM mainframe-jeiben, új mert a jelenkori virtualizációs technikák közül ez jelent meg utoljára a ma leggyakrabban virtualizációra használt x86-os platformon. Mint a neve is sugallja, a a megoldás a szoftveres technikák helyett a hardveres megvalósításra helyezi a hangsúlyt. Ilyen megvalósítás a ma kapható x86 architektúrájú processzorokban található virtuális gépek hatékony futtatását lehetővé tévő bővített utasításkészlet és működési mód. A két domináns processzorgyártó mindegyike rendelkezik ilyen megoldással, amelyek szinte teljes egészében azonos elven működnek. A paravirtualizációval ellentétben a vendég operációs rendszer nem igényel semmiféle módosítást, így zárt kódú vendég operációs rendszerekkel is használható. Bár alkalmazhatóság terén nagyon kecsegtető paraméterekkel rendelkeznek ezen csoportba tartozó megoldások, egy fontos tényt mindenképpen észben kell tartanunk, ez pedig a virtuális gépek közötti és a virtuális gép és gazdagép közötti környezetváltásokkal járó, adminisztrációs költségekből adódó teljesítményveszteség, amely főleg a korábbi architektúrájú processzoroknál okozhat gondot.

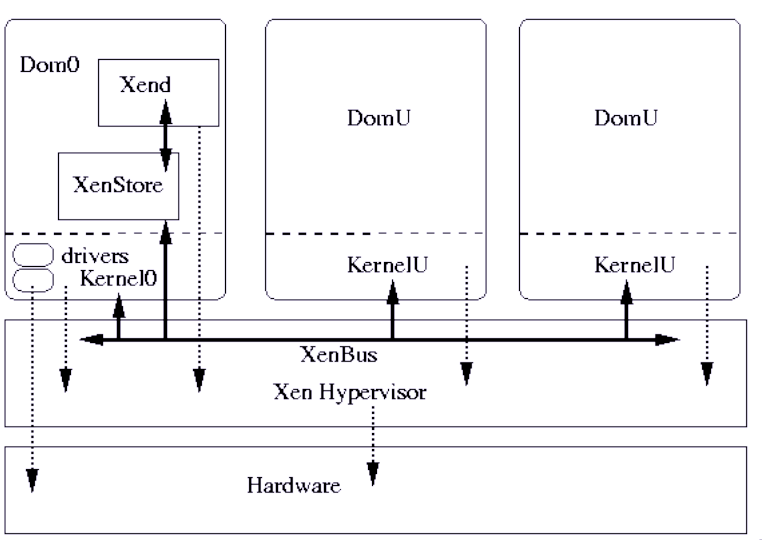
2.3. Memória és I/O virtualizáció

Szűkebb értelemben nem alapfeltétele a hardveresen támogatott virtualizációnak ám megvalósításából és céljából adódóan itt kell megemlíteni a memória elérést és a fizikai eszközök elérését és biztonságossá tételét célzó kiegészítő technológiákat és funkcionális egységeket. Elsőként essen szó a memóriakezelésről. Egy hagyományos, semmiféle hardveres rásegítéssel nem rendelkező rendszer esetén a memória kezelése okozza a legtöbb gondot és adminisztrációs költséget. A probléma egyik legáltalánosabb megoldása az *árnyék laptáblák*

használata. Ilyenkor a hypervisor tart fent miden egyes guest számára egy laptáblát, ezeket nevezzük árnyék tábláknak, a guest nem fér hozzá a valós táblákhoz, és a valós táblák és az árnyék megfelelők között is csak a hypervisor létesít kapcsolatot és végez szinkronizációt, tulajdonképpen a virtuális gép fizikai memória címét felelteti meg a fizikai gép logikai memória címének. Mint látható az amúgy sem egyszerű címfordítási folyamat egy újabb elemmel bővül, amely megkétszerezi a folyamat hosszát. Természetesen ez a probléma hardveres támogatás nélkül is kiküszöbölhető, ám ugyanazokba a problémákba ütközünk, mint a paravirtualizáció esetében, az operációs rendszernek módosíthatónak kell lennie, azt fel kell tudnunk vértezni azzal a képességgel, hogy tudomást vegyen a vele párhuzamosan futó többi rendszerről, azok képesek legyenek kooperatívan működni, a memória erőforrásokat egymás megzavarása vagy veszélyeztetése nélkül használni. Az elmondottakból érezhető, hogy ez egy operációs rendszer radikális módosítását igényelné, hiszen a memória hozzáféréssel szinte minden kapcsolatban áll.

A memóriakezelés gyorsítására a korábbinál egyszerűbb és bizonyos paraméterek és körülmények mellett hatékonyabb megoldást nyújt a hardveresen támogatott memória kezelés, melynek lényege, hogy egy újabb tábla bevezetésével jelentősen csökkenti a címfeloldási folyamat komplexitását, ez a tábla a virtuális gép fizikai memóriacíme és a fizikai gép fizikai címe között létesít megfeleltetést. Az árnyék táblák alkalmazásával ellentétben nem kell minden egyes processzor esetén külön laptáblát fenntartania a hypervisor-nak, ebből adódóan használata kevesebb erőforrást igényel. Alkalmazásának egyetlen negatívuma van, ez pedig a cím fordítás cachelésével kapcsolatos, ha ugyanis a gyorstárban nem található meg a kívánt bejegyzés, annak felkutatása jelentős többletköltséggel jár. Az előbbieknek megfelelően viszonylag nagy méretű memória lapok esetén célszerű alkalmazni, hogy a gyorsítótár szűkös mérete miatt ne kelljen mindig címfeloldást végezni, hiszen ha kis méretű lapokkal dolgozunk, könnyel előfordulhat, hogy egy korábbi példányt kell felülírni a cache táblában, mert már nincs szabad hely.

Részben a memória kezeléshez kapcsolódik részben pedig a kimeneti és bemeneti eszközök kezeléséhez a hardveres támogatás harmadik fontos eszköze, egyértelmű elnevezést nem tudunk rá adni, gyártó függő, hogy ki hogyan nevezi és milyen funkciókat sorol a szolgáltatások közé. Ezen szolgáltatáscsoport célját úgy foglalhatnánk össze, hogy lehetővé teszi a I/O eszközök hatékony elérését, miközben megakadályozza a működés és hozzáférés közben felléphető véletlen és szándékos rosszindulatú hozzáféréseket. Az AMD architektúrát alkalmazó állomásokban IOMMU néven található meg, az Intel pedig a VT-d nevet használja, míg az első generációs megvalósítások - mint például a NetBurst és Core vagy a K8, K10 mikróarchitektúra köré épülő rendszerek - esetén az alaplapi chipset rendelkezhet ezzel a technológiai eszközkészlettel, addig az újabb architektúrájú processzorok esetén már a processzor lapkán kell kutakodnunk.



2.1. ábra. Xen architektúra [19]

2.4. Virtualizációs megoldások

2.4.1. Szerver oldali megoldások

Xen

A Xen az egyik legismertebb és legelterjedtebb nyílt forráskódú virtualizációs megoldás, amely képes a paravirtualizáció és hardveresen támogatott virtualizáció előnyeinek kiaknázására. A Xen tulajdonképpen egy absztrakciós réteg a virtuális gépek és a valós erőforrások között, mely lehetővé teszi azok használatát. Architektúrális szempontból egy erősen célorientált Linux vagy Unix kernelnek tekinthetjük, jelen esetben logikai vagyról kell beszélnünk és nem kizáró, a magyar nyelv kizáró vagyáról. A Xen fejlesztésének és használatának két jelentős, részben együttműködő vonala létezik, az egyik a közösségi a másik az üzleti jellegű. A kettő közötti különbséget főleg az igénybe vehető támogatásban és használható menedzsment eszközökben kell keresni. Az üzleti vonal képviselője a Citrix, Xen Server nevű termékével.

Architektúráját tekintve, ahogy az a 2.1 ábrán is látszik, ez a megvalósítás három viszonylag jól elkülönülő részre tagolódik. Az első és legnagyobb a hpervisor, amely közvetlenül a hardver réteg felett helyezkedik el, elsődleges funkciója, hogy absztrakciós réteget képezzen a virtuális gépek és a fizikai eszközök között. Az absztrakciós réteg jellegből adódóan a hypervisor végzi az erőforrás kezeléssel kapcsolatos műveleteket. Az első domaint, ami a rendszer indítását követően létrejön dom0-néven emlegetik, tulajdonképpen ez a hoszt operációs rendszer. Dom0-ban futó operációs rendszernek bizonyos Linux disztribúciókat, NetBSD vagy OpenSolaris-t választhatunk.

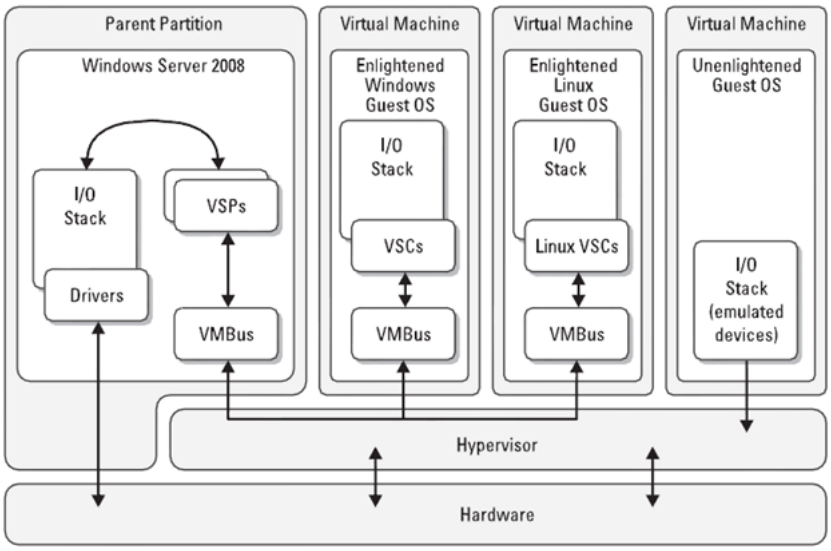
A dom0 speciális jogokkal rendelkezik a virtuális gépek között, hozzáférése van az I/O erőforrásokhoz, és interakciót folytat a többi virtuális géppel. A Xen architektúrájának harmadik összetevője a DomainU vagy röviden domU, ebben a környezetben fut minden kiszolgált vendég operációs rendszer. A domU további két részre osztható az alkalmazott

virtualizációs technika függvényében, így megkülönböztetünk PV Guest domaint és egy HVM Guest domaint. A PV Guest domain tartalmazza azokat a virtuális gépeket, amelyek módosított, paravirtualizációra alkalmas kernellel rendelkező Linux vagy Unix operációs rendszert futtatnak. A virtuális gépek azon csoportjára, amelyek Windows-t, vagy egyéb módosítatlan operációs rendszert futtatnak hardveresen támogatott virtualizációt alkalmazva, HVM Guest domain-ként hivatkozunk.

A virtuális gépek menedzselésében számos démon folyamat vesz részt, ezek a porcesszek a dom0 keretein belül futnak, ilyen a Xend, amely a hypervisor felé közvetíti a hívásokat. A Xend-hez érkező kérések elvárt formátumát az XML-RPC szabvány rögzíti. A PV Guest domain-ben található virtuális gépek számára a blokkos erőforrások elérését paravirtualizált meghajtó programok teszik lehetővé, amely egy a hypervisoron áthaladó az aktuális virtuális gépet és a dom0-át összekötő eseménycsatornán keresztül valósul meg. A módosítatlan vendég operációs rendszerek nem rendelkeznek ilyen eszköz meghajtókkal, számukra a blokkos erőforrások elérését egy, minden virtuális gép számára külön példányban futó Qemu démon teszi lehetővé, amely a dom0-ban található.

Hyper-V

A Microsoft Hyper-V névre hallgató virtualizációs megoldása a Windows Server 2008 platform részét képezi, jelenleg két formában érhető el. A Windows Server 2008 teljes értékű operációs rendszer részeként, amely fizetős termék, illetve Microsoft Hyper-V Server 2008 R2-ként, amely egy szerver alapokra épülő, kizárólagosan virtualizációs feladatokat ellátó, ingyenes szoftver. Működését tekintve a hardveresen támogatott virtualizációt használó termékek közé tartozik, így alkalmazhatóságának szükséges feltétele a processzor virtualizációs támogatásának megléte, ennek hiányában nem használhatjuk. A Hyper-V nem önálló termék, a szerver operációs rendszer egyik funkciója, Microsoft terminológiával élve szerepe. Mivel a kiszolgált operációs rendszerek nem igényelnek módosítást, bármilyen vendég rendszer futtatására képes, de igazából a Windows családra van kihegyezve. Szerencsére a termék térnyerésével párhuzamosan fejlődik a Linux alapú operációs rendszerek esetén nyújtott kiszolgálási teljesítménye is, amely főleg a Microsoft és a disztribúció fejlesztők, különösen a Red Hat hathatós együttműködésének köszönhető. Architektúrális felépítésének, amely 2.2 ábrán látható, alapköve a partíció [3]. Ha a korábban megismert megoldásokkal szeretnénk párhuzamot vonni, és megfeleltetést keresni, akkor azt mondhatjuk, hogy a partíció nagyjából a Xen-es domain-nek felel meg. A partíció tulajdonképpen egy izolációs logikai egységnek tekinthető, amely az egyes operációs rendszerek zavartalan működését teszi lehetővé. Egy partíciónak minden esetben léteznie kell, ezt a kitüntetett szerepű egységet nevezzük szülő partíciónak, a virtuális gépeket magukba foglalókat pedig gyerek partícióknak. A gyerek partíciók nem rendelkeznek közvetlen hardver és megszakítás hozzáféréssel, az általuk látott virtuális erőforrások virtualizációs szolgáltatás kliensek segítségével kommunikálnak a nekik megfelelő szülő oldali szolgáltatókkal, a kommunikációban résztvevő feleket egy dedikált csatorna, a VMBus köti össze. Amennyiben a felhasznált hardver erőforrások rendelkeznek IOMMU

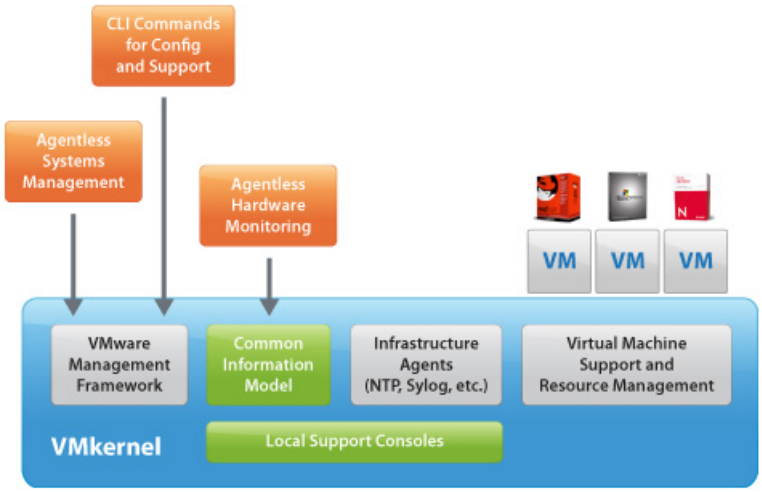


2.2. ábra. Hyper-V architektúra [18]

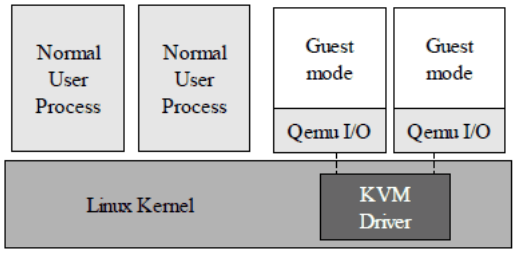
támogatással, az egyes virtuális gépek és a valós gép memóriája közti cím fordítást a Hyper-V a processzor dedikált egységére hárítja, így jelentős többlet teljesítmény javulást érve el. További teljesítmény növekedés érhető el az úgynevezett Enlightened I/O technológia segítségével, amely az erőforrás elérés hatékonyságát hivatott javítani. A megoldás lényege, hogy a magas szintű kommunikációs protokollok implementációját a virtualizáció igényeinek megfelelően átalakították, így lehetővé téve, hogy a kommunikációs folyamat az eszköz emulációs réteg megkerülésével történjen. Ez a funkcionalitás a vendég rendszer megfelelő módosításával, az integrációs csomagok telepítését követően vehető igénybe.

ESX

Az ESX a virtualizációs termékek piacának egyik legismertebb és legnagyobb múlttal rendelkező tagja a VMware üzleti szintű megoldása. Felépítése a jól bevált, módosított Linux-kernel szisztémát követi, a korábban megismert hasonló architektúrájú termékekkel ellentétben ez a kernel nem egy harmadik féltől, valamelyik disztribúció gondozójától származik, mint például a Xen esetén, hanem egy saját fejlesztésű, a saját követelményeknek megfelelően fejlesztett és optimalizált rendszerkomponensről van szó. Az első virtuális gép, amely minden esetben futtatásra kerül a konzol operációs rendszer, amely segítségével menedzselhetjük a rendszerünket. Érdemes megemlíteni, hogy az ESX-nek létezik egy kis és közepes méretű vállalati környezetben használható, ingyenes alternatívája, az ESXi, a két termék közti különbség pusztán annyi, hogy az ESXi nem tartalmazza az imént említett konzol rendszert, így kontrollálásához távoli menedzsment eszközöket kell használni. Piacvezető szerepéhez méltón az egyik legszélesebb és legmodernebb eszköz arzenállal rendelkezik.



2.3. ábra. ESXi architektúra [25]



2.4. ábra. KVM architektúra [20]

Kernel Based Virtual Machine

A Kernel Based Virtual Machine, vagy röviden KVM [6] a hypervisorok egy érdekes csoportját képviseli. Fejlesztésekor a legfontosabb cél az volt, hogy a Linux rendszerek fejlesztése során felhalmozott tapasztalatokat felhasználva olyan terméket készítsenek amely megbízható, gyors és könnyen kezelhető hátteret biztosít egy virtualizált rendszer felépítéséhez. A KVM a Xen-nel ellentétben nem önálló kernel, csak egy a használt processzornak megfelelő, kvm_amd vagy kvm_intel kernel modul, amely nem igényli az egyes disztribúciók gondozóitól, hogy kernel portot készítsenek hozzá, ennek köszönhetően széles körben alkalmazható. A hypervisorok csoportosítását tekintve ez a megoldás a hardveresen támogatott virtualizáció csoportjába tartozik, működésének ez az egyik alapköve, alapállapotában a KVM-et alkalmazó rendszer hardveres processzortámogatás hiányában képtelen virtuális gépeket futtatni. A KVM erősen épít a Linux kernelben megvalósított erőforrás kezelő rutinokra, így nem rendelkezik sem saját ütemezővel, sem önálló memória menedzsmenttel, a virtuális gépek egy egyszerű linux folyamatban (process) futnak, ennek megfelelően a virtuális géphez csatolt memória sem más, mint a folyamat saját memóriaterülete. Mindenképpen említést érdemel egy különleges, de semmiképpen sem egyedi memóriakezelési funkció, melyet Kernel Same-page Merging néven emlegetnek. A megoldás lényegében az egyes virtuális gépek memórialapjait összehasonlítva kiszűri

azokat amelyek azonos tartalommal bírnak és azt egy közös, több virtuális gép által használt memória lapra helyezi el, amennyiben valamelyik gép azt módosítani akarná, akkor saját másolatot kapna, amelyet a továbbiakban már szabadon írhat. Ez főleg olyan statikus memóriatartalmak esetén jelenthet jelentős erőforrás megtakarítást, mint a program könyvtárak vagy a kernel. Hasonló megoldást a korábban tárgyalt ESX rendszerek is alkalmaznak. Az egyes virtuális gépek hagyományos folyamatként történő futtatásának eredménye, hogy azok ütemezését maga a kernel végzi, és így akár konfigurálhatjuk is azt.

A KVM egyetlen hiányossággal rendelkezik, ha ezt hívhatjuk egyáltalán, hiányosságnak, önállóan nem alkalmas egy virtuális gép felépítésére, nem tudunk vele például olyan virtuális erőforrásokat létrehozni, mint háttértár, vagy hálózati csatoló, erre a célra alkalmazzák a Qemu nevű virtualizációs alkalmazás egy speciális változatát, amely KVM modullal együtt települ.

2.4.2. Kliens oldali megoldások

Az eddig ismertetett termékek kivétel nélkül mind bear metal alapú megoldások voltak Ezeket többnyire kizárólagosan virtualizációs feladatokra használják, többnyire hiszen egy KVM alapú rendszer a virtuális gépek kiszolgálásán kívül bármilyen más feladatot is elláthat, de akár ugyanez elmondható a Hyper-V-ről is, hiszen akár egy webkiszolgáló szerepet is hozzáadhatunk a használt szerver operációs rendszerhez. A mindennapi munka során gyakran felmerülhet az igény más platformok futtatására, vagy csak egyszerűen szükségünk lenne egy másik operációs rendszerre is, amelyen elvégezhetnénk a teendőinket. Ezekben az esetekben többnyire nincs szükségünk az előzőekben megismert megoldások által felkínált lehetőségek széles palettájára, nem akarunk dedikált rendszert telepíteni, konfigurálni még kevésbé. Az ilyen jellegű igények kielégítésének céljából születtek a hosted típusú virtualizációs megoldások, melyek egyszerűen telepíthetőek, mint bármilyen más program, többnyire integrált grafikus kezelőfelületük segítségével könnyen létrehozhatjuk a kívánt erőforrásokat és telepíthetjük a megfelelő operációs rendszert. Az ilyen jellegű alkalmazások közül három nevet kell kiemelnünk, népszerűségüknek és kimagasló teljesítményüknek köszönhetően.

QEMU

A KVM esetén már felmerült a neve, mint a virtuális erőforrások kezeléséért felelős komponens, ennél azonban sokkal többre képes. Egy önálló virtualizációs megoldás, amely tiszta emuláció alkalmazásával az egyes architektúrák közötti átjárást is lehetővé teszi. Működése során bináris fordítást alkalmaz, amennyiben a kiszolgáló és a vendég gép megegyezik. Unix, Linux és Windows operációs rendszerek esetén egyaránt használható ingyenes program, teljesítménybeli képességei emuláció esetén hagynak némi kívánnivalót, amelyet kompenzál az a lehetőség, amelyet a keresztplatformos fejlesztés területén nyit.

VirtualBox

A VirtualBox egy hosszú múltra visszatekintő virtualizációs alkalmazás, mely számos operációs rendszeren elérhető. Dinamikusan fejlődő ingyenes termék, képes kiaknázni a hardverekben rejlő virtualizációs képességeket, mind a VT-x/AMD-V, mind a memória virtualizációs támogatások. Figyelemre méltó képességei közé tartozik az ablakkezelés során 3D kártyára történő hagyatkozása, amely sok esetben érezhetően gyorsabb és felhasználóbarátabb használatot tesz lehetővé. Mindenképpen megemlítendő, hogy bár desktop alkalmazás akár egy kisebb, távolról irányítható virtuális rendszert is kiépíthetünk köré, ugyanis olyan menedzsment felületeket nyújt, amelyet akár távolról is el tudunk érni. Természetesen az olyan komplex funkciókról, mint a migráció le kell mondanunk, ugyanis ilyen szolgáltatásokat nem nyújt.

VMware Workstation

Az alkalmazás alapú virtualizáció egyik piacvezető terméke a VMware Workstation, teljesítményben és funkcionalitásban is felülmúlja társait. A létrehozott virtuális gépek kompatibilisek az ESX által futtatott gépekkel, így akár szükség esetén ott is futtathatjuk őket. A VirualBox-hoz hasonlóan képes kiaknázni a hardveres támogatásokban rejlő lehetőségeket. Az újabb verziók lehetővé teszik az asztal szintű alkalmazás integrációt, amely azt jelenti, hogy a virtuális gépünkön futtatott szoftver a gazda operációs rendszer ablakkezelőjében úgy fog megjelenni és viselkedni, mintha azon a rendszeren futna. Érdemes még említést ejteni egy különleges támogatásról, a Visual Studio illetve Eclipse fejlesztő környezetbe integrálható hibakövető modulról, amely a hagyományos hibakövető módszereknél sokkal részletesebb nyomkövetést és hibadetektálást tesz lehetővé.

3. fejezet

Virtualizált rendszerek távoli menedzsmentje

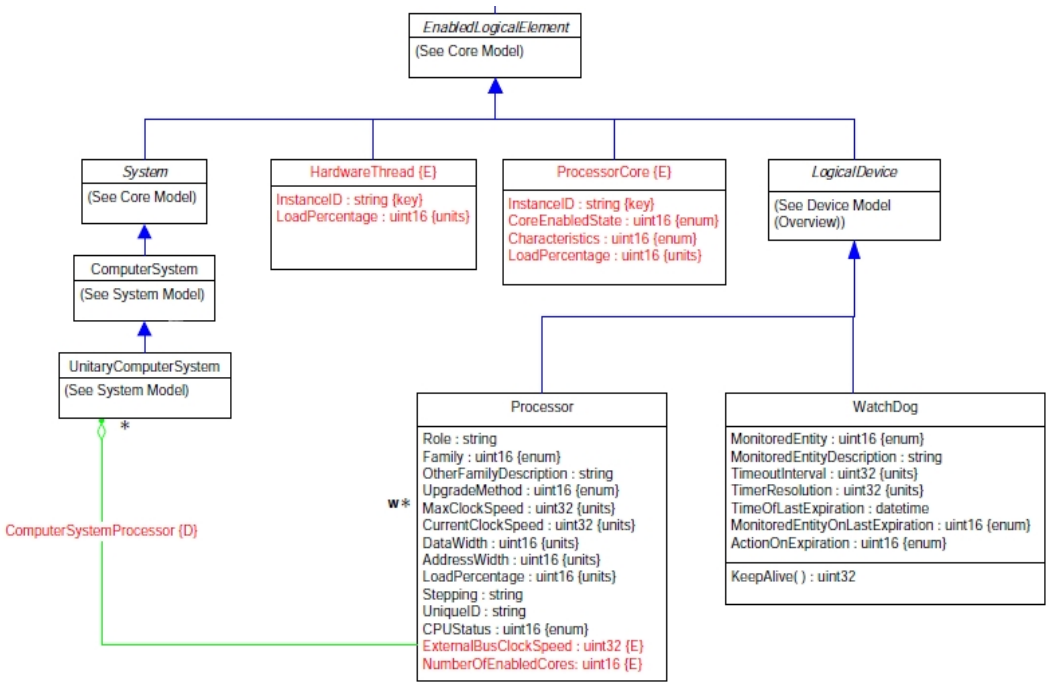
A mai informatikai szolgáltatások, termékek jelentős része valamiféle üzleti rendszer vagy folyamat szerves részét képezi, amelyben a legkisebb leállás vagy hiba is gyakran erős anyagi vonzattal járhat, a kiesések minimalizálása érdekében szükség van a rendszerbe történő mielőbbi beavatkozásra. Mivel a virtualizáció mára az üzleti infrastruktúra egyik meghatározó építő elemévé vált, ezért az ilyen célú megoldásoknál is joggal várhatunk el valamiféle eszközt, amellyel bármikor, bárhonnan elérhetjük rendszerünket és módosításokat végezhetünk benne. Ilyen eszközök a távoli menedzsment interfészek melyekből szinte kivétel nélkül minden virtualizációs megoldás implementál legalább egyet. Ezekről általánosságban elmondhatjuk, hogy gyakran alkalmaznak már meglévő menedzsment szabványokat vagy felületeket a virtualizáció által támasztott követelményeknek megfelelő módosítással.

3.1. Platformfüggetlen menedzsment megoldások

A virtualizációs platformok számának növekedésével arányosan a menedzsment felületek száma is növekedik, ezek a menedzsment felületek sokszor eltérő szabványokon alapulnak és még többször a probléma különböző aspektusait tükrözik. Ahogy növekedik a virtualizációs technikákat alkalmazó rendszerek komplexitása és heterogenitása, úgy csökken a rendszerfelügyeleti teendő kézben tarthatósága. A probléma ellenszere egy egységes menedzsment szabvány lenne, sajnos a jelenlegi formában egyetlen ilyen célú teljes szabvány sem létezik, de léteznek ilyen célú törekvések.

3.1.1. Common Information Modell

A DMTF gondozásában készült és karbantartott szabvány [2] az informatikai szabványok legsikeresebbjei közé tartozik, segítségével lehetőség nyílt a menedzselt erőforrások, szolgáltatások adatainak szabványos leírására, cseréjére, melynek elsődleges színtere a hálózati közeg. A szabvány két részből épül fel: egy meta jellegű szabványból, amely a modellezési nyelvet és ezzel együtt az adattípusokat definiálja, valamint egy séma



3.1. ábra. CIM diagram [16]

gyűjteményből, amely a különböző menedzselni kívánt komponenseket leíró formalizált modelleket tartalmazza. Az objektum orientált szemléletnek megfelelően a modellek alapjait az osztályok képezik, az osztályok és a köztük lévő lehetséges kapcsolatok, mint például származtatás, tartalmazás segítségével szemléletesen és hatékonyan kifejezhetőek a menedzselt komponensek közötti relációk. Az egyes objektumok nem csak tulajdonságokkal rendelkezhetnek, hanem a rajtuk értelmezett műveleteket is definiálhatják. Maga a mag modell számos problémához rendelkezik kész modellel, amennyiben azok nem megfelelők vagy csak részben, tetszés szerint bővíthetjük őket. A CIM nagyszerűsége abban rejlik, hogy használata segítségével egy probléma esetén elegendő egyszer elkészíteni a megoldást, a későbbiekben azt bármikor használhatjuk. Bár önmagában a mag modell nem ad túl sok útmutatást a virtualizált rendszerek menedzsmentjéhez, a benne található fizikai rendszerekre elkészített modellek kiterjesztésével kellően stabil és megalapozott modellt és arra építve jól használható és szabványos menedzsment felületet készíthetünk. Ennek egyik kézzel fogható példája, amely a napi gyakorlatban is segítségünkre lehet, a Microsoft Hyper-V termékéhez készített WMI osztályok, amelyek CIM által definiált modellre épülnek, például egy Msvm_ComputerSystem osztály példány reprezentálhat egy hosztot vagy egy virtuális gépet is, miközben a CIM_ComputerSystem leszármaztatásával keletkezett.

3.1.2. WS-Management

A ma létező menedzsment szabvákony közül talán az egyik legkönnyebben és leghatékonyabban alkalmazható szabvány a WS-Management [12], magyarul webszolgáltatás alapú menedzsmentnek nevezhetnénk. Stabilitásának és hatékonyságának

alapját, olyan kiforrott és mára sokat bizonyított technológiák adják, mint az XML és a rá épülő SOAP és WS protokollok. A szabvány elkészítése egy számos informatikai óriást magában foglaló szakértői csoporthoz fűződik, a szabványosítási folyamatért pedig a DMTF vállalt felelősséget. A WS-Management biztosítja a menedzsment adatok http réteg fölött történő szabványos cseréjét. Nem pusztán get és set műveletek hívására ad lehetőséget, lehetőség van a menedzselt fél által implementált és publikált tetszőleges metódusok használatára. A szabvány önmagában nem hordoz technikai újításokat, meglévő technológiákból és szabványokból építkezve határozza meg az irányítási és információszerzési folyamatok menetét. Stabilitása mellett széleskörű alkalmazhatóságának egyik megalapozója a tűzfal-barát jelleg, amely a felhasznált technológiák egyenes következménye.

WS-Transfer

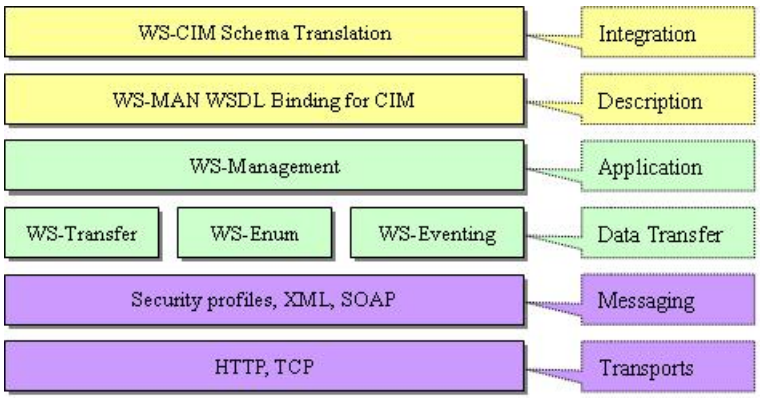
A WS-Transfer [13] szabvány rögzíti, hogy miként lehet webszolgáltatások segítségével egy entitás XML alapú reprezentációját elkérni a kommunikáló féltől. Kétféle entitást kell megkülönböztetnünk, erőforrást, amely egy végpont referenciával adott, illetve gyártókat, amelyek XML alapú reprezentációból képesek erőforrásokat gyártani. Két műveletet használhatunk az adott erőforrás reprezentációjának küldésére és fogadására, get és put, valmint kettőt az erőforrások törlésére és létrehozására, delete és create.

WS-Addressing

A kommunikáció során elérni kívánt webszolgáltatások és a közben használt üzenetek címzésére a WS-Addressing [10] szabványt használják, amely az átvitel módjától független címzést tesz lehetővé. Mindehhez a szabvány két konstrukciót definiál, a végpont referenciát és az üzenet információs fejlécet. A végpont referencia hordozza mindazokat az információkat, amely segítségével az üzenet címzettje egyértelműen azonosítható, míg az üzenet információs fejléc az egy interakcióban résztvevő kommunikációs felek azonosítását teszik lehetővé. Ilyen fél a feladó és a címzett, kétirányú kommunikáció során a válasz üzenet célpontja, illetve a hiba esetén értesítendő fél.

WS-Enumeration

Számos esetben előfordulhat, hogy a kérésünk eredményeképpen kapható entitások, erőforrások számánál nem élhetünk azzal a feltételezéssel, hogy csak egyetlen példány létezhet belőle. Ez a helyzet gyakran tovább bonyolódik, amikor hatékonysági okokból nem elfogadható az eredményhalmaz adatainak egyetlen üzenetben történő továbbítása, ebben az esetben fel kell készülnünk az eredményhalmazok kezelésére. Az ilyen eredmény kollekciók bejárásához nyújt segítséget a WS-Enumeration [11] szabvány. Működését az adatbázisok kurzorához lehet hasonlítani, a kliens egyesével olvassa a felsorolás kontextust mindaddig, amíg az tartalmaz példányokat, így biztosítva a kellően hatékony adatátvitelt.



3.2. ábra. WS-Management és kiegészítő szabványok [24]

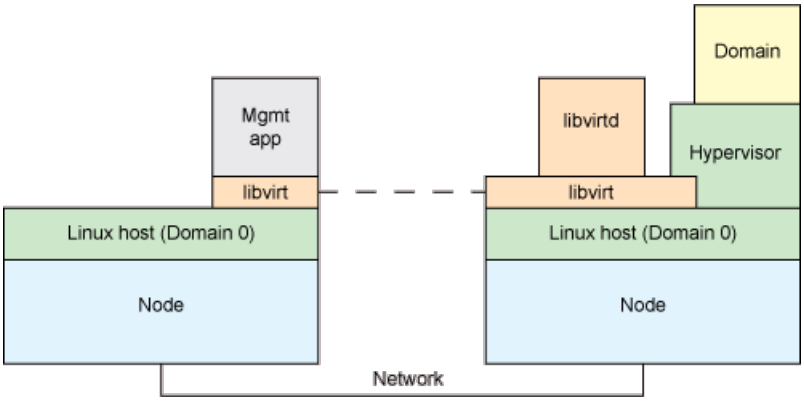
3.1.3. libvirt, mint kvázi szabvány

A libvirt nem más, mint egy eszközkészlet, amely együttműködik a Linux és egyéb operációs rendszerek virtualizációs képességeivel, egy szabadon felhasználható termék, amelynek alapját egy hosszútávra tervezett C API adja. Rendelkezik CIM providerekkel, amelyek a DMTF menedzsment sémájának megfelelő adatokat és funkcionalitásokat szolgáltatja, így biztosítva szabványos felületet a virtuális gépek, hálózatok és tárolók menedzsmentjéhez. A távoli elérés biztonságát TLS protokoll és Kerberos alapú autentikáció biztosítja. A libvirt API szinte minden újabb Linux disztribuciónak a részét képezi, így telepítése nem igényel különösebb előképzettséget. Windows operációs rendszer esetén nem áll rendelkezésünkre a bináris termék, így a fordítást magunknak kell elvégezni, egy MinGW fordító segítségével. Mindenképpen megjegyzendő, hogy Windows esetén csak távoli menedzsment célú kliensként használhatjuk a kapott terméket.

A libvirt terminológia megértéséhez két fogalmat kell tisztáznunk. Az első a node vagy csomópont fogalma, amelyen egy fizikai gépet kell értenünk, a második a domain fogalma, amely lehet egy virtuális gép és lehet a hypervisor vagy libvirt komponenseket futtató operációs rendszer is, ez utóbbit a megkülönböztetés végett domain0 névvel szokták illetni.

Az üzemeltetés biztonságát és nyomon követhetőségét célozza a többszintű naplózási mechanizmus, amely három kulcs pillérre épül, az elsőt az üzenetek képezik, amelyek futási időben generálódnak, tartalmazzák az üzenet prioritási szintjét, keletkezésének időpontját és helyét, kódsor szám pontossággal. A következő építőelemet az üzenet szűrők képezik, amelyek segítségével tetszőleges szabályrendszert alkothatunk az üzenetek kezelésére, a harmadik pillért pedig a log kimenet adja, ahol meghatározhatjuk, hogy a szűrők által átengedett üzenetek közül, melyik hova kerüljön, így például lehetőségünk nyílik olyan konfiguráció beállítására, ahol minden üzenetet elmentünk egy nyomkövetési fájlba, és a rendszer log állományaiba csak a hibáról árulkodó bejegyzéseket engedjük tárolni.

Az API megtervezésekor elsődleges szempont volt egy olyan stabil és hosszú távon alkalmazható illesztő felület megalkotása, amely építőkövéül szolgál más magas szintű menedzsment eszköznek. Ezen cél elérésének egyik legfontosabb feltétele volt egy olyan, kellően általános szerkezet kialakítása, amely nem csak egyetlen virtualizációs környezet



3.3. ábra. libvirt távoli menedzsment [21]

használatát teszi lehetővé. Ennek az erőfeszítésnek azonban ára van, az általánosságra való törekvés közben le kellet mondani számos, specifikus szolgáltatásról, amelyek csak bizonyos termékek esetén lehetnének elérhetők. A felhasználási igények kielégítése érdekében, felületet kell biztosítani az egyes eszközök irányításához és megfigyeléséhez, ilyen felületet tesz lehetővé a libvirt olyan erőforrások esetén, mint a processzor, memória, hálózat vagy a háttértár.

A kommunikációs folyamatok platformfüggetlenségnek biztosítása érdekében a menedzsment folyamatok adatváltásaihoz, számos más termékez hasonlóan önleíró adatstruktúrát használ, amely a libvirt-re épülő alkalmazások fejlesztését is megkönnyíti, hiszen minden valamire való programozási nyelvhez léteznek xml feldolgozást lehetővé tévő könyvtár csomagok.

A libvirt működésében kétféle üzemmódot kell megkülönböztetni. A megkülönböztetés alapjául a menedzser alkalmazás és a menedzselt erőforrás elhelyezkedése szolgál, ezek alapján ezek a komponensek vagy azonos csomóponton helyezkednek el vagy különbözőn. Az első esetben a libvirt közvetlenül éri el az alsóbb rétegeket és az irányítani kívánt rendszer hypervisor-át, míg a második esetben mindez a libvirtd nevű démon segítségével történik.

A libvirt szolgáltatásait használja a virsh nevű konzolos alkalmazás, amely segítségével irányíthatjuk domain-einket, a hívásokat a libvirt továbbítja a megfelelő platform menedzsment felülete felé.

3.2. Menedzsment alkalmazások

A virtualizációs termékek palettájának kiszélesedésével párhuzamosan jelent meg az igény olyan menedzsment eszközök iránt, amelyek segítségével különböző virtualizációs megoldásokra épülő rendszereket lehet egységesen, közös felületen keresztül vezérelni. A következőekben bemutatni kívánt két eszköz ezeknek az elvárásoknak próbál eleget tenni.

3.4. ábra. ConVirt 2.0 [15]

3.2.1. ConVirt

A ConVirt [1] nevű alkalmazás segítségével Xen és KVM alapú rendszerek menedzsmentje válik lehetővé. A termék egy nyílt forráskódú, ingyenesen használható alkalmazás, amelyhez szükség esetén vásárolhatunk különböző szolgáltatásokat magában foglaló támogatást. Három szintű, erőforrás tárház alapú architektúrával rendelkezik, amely lehetővé teszi a stabil működtetést és a könnyű skálázhatóságot. Könnyen telepíthető és konfigurálható, használatának feltétele, hogy az irányítani kívánt gazdagépeken megfelelően telepítve legyen az ssh szolgáltatás. Mind Xen mind KVM alapú menedzselt rendszer esetén elengedhetetlen a megfelelő konfigurációs beállítások megléte. KVM esetén a socat nevű alkalmazás megléte is szükségeltetik, amely segítségével kétirányú bájt folyam alapú átvitel valósítható meg a megadott célpontok között. Az egyes rendszerek uniform menedzsmentjét webes felület teszi egyszerűvé, a nyomon követhetőség érdekében historikus adatok tárolására is lehetőséget biztosít, amely segítségével előrejelzéseket készíthetünk, erőforrás átcsoportosításokat végezhetünk, a szolgáltatások zökkenőmentes voltának fenntartása érdekében. Az adatgyűjtést kiszolgáló monitorozó folyamatok segítségével, olyan erőforrásokról gyűjthetünk adatokat, mint a processzor, memória, háttértár, hálózati erőforrások. Az adatgyűjtés több szinten történik, virtuális gépenként, fizikai hosztonként vagy erőforrás csoportonként, így szolgáltatva minél teljesebb képet a rendszer aktuális állapotáról. A kiépítendő virtuális rendszer homogenitását célozza meg a sablon alapú provisioning. A stabil működés egyik eszközeként szolgál az automatizált migráció, amely nem csak a virtuális gépek valós erőforrások közötti mozgatását könnyíti meg, hanem a környezet paramétereinek figyelembevételével a legideálisabbnak ítélt erőforráshoz rendeli hozzá a virtuális gépet. Ez a funkcionalitás nem csak migrációnál vehető igénybe, hanem új gép létrehozásánál is.

3.2.2. OpenNebula

A virtualizáció a benne rejlő lehetőségeknek köszönhetően egyre szélesebb körben kerül alkalmazásra. Egy fontos, lehetséges alkalmazási terület a felhő alapú számítás rendszerek kiépítése, amit a szakzsargonban csak cloud computing-nak neveznek. A felhő fizikai vagy virtuális gépekből történő kialakítása között annak logikai szintjén nincs lényeges architektúrális különbség, mindkét esetben számítási csomópontok strukturált szerkezetéről beszélhetünk, amelynek célja egy flexibilis, az aktuális igényeknek megfelelő aggregált erőforrás kapacitás létrehozása. A két megvalósítás közti különbségeket a rugalmasságban és az irányítási felületekben kell keresni. Ilyen felületet nyújt az OpenNebula [8] nevű eszközkészlet, amely meglévő termékek, integrációs felületek és technológiák felhasználásával olyan architektúrát definiál és deklarál, amely segítségével hatékonyan hozhatunk létre és felügyelhetünk virtuális gépeken alapuló, felhő szerkezetbe foglalt adatrendszereket. Felépítését tekintve három rétegre tagolódik, a legfelső réteg parancsértelmezőt, ütemezőt és a libvirt felület implementációját foglalja magában. Segítségükkel beavatkozási lehetőséget nyújt a felhasználóknak, valamint felületet biztosít további menedzsment alkalmazások fejlesztéséhez. A középső réteg képezi a megoldás magját, felügyeli és irányítja a hozzá kapcsolt virtuális rendszereket, portolja a megfelelő hívásokat az érintett megvalósítás valós kommunikációs protokolljára és továbbítja azt a megfelelő elérési pont felé. A harmadik réteget az igény szerint használható driverek képezik, melyek segítségével kapcsolódhatunk az egyes hypervisorokhoz vagy felhő szolgáltatásokhoz. A menedzsment adatok perzisztens tárolásához a megvalósítás SQLite3 adatbázist használ, ennek köszönhetően könnyen fejleszthető hozzá különböző szolgáltatásminőséget és egyéb jellemzőket számlázó kliens alkalmazás. Az OpenNebula segítségével Xen, KVM és ESX alapú rendszereket kapcsolhatunk felhőbe. KVM esetében az alkalmazhatóság feltétele, hogy a libvirt alkalmazás csomag is telepítve legyen, ugyanis eléréséhez azt használja. Xen esetében a xend nevű, menedzsment feladatokat ellátó démonra támaszkodik, míg az ESX alapú termékek eléréséhez saját, a VMware Infrastructure API-n alapuló meghajtót használ.

3.3. Implementált platformfüggetlen és platformspecifikus menedzsment felületek

Szinte valamennyi virtualizációs megoldás rendelkezik saját menedzsment ezközzel vagy olyan programozási felülettel, amely segítségével menedzsment célú alkalmazások készíthetőek a kérdéses termékhez. Ezek egy része valamilyen saját szabványra épül, míg mások a meglévő, szélesebb körben elterjedt menedzsment protokollokat részesítik előnyben. A következőkben a vizsgált virtualizációs termékek által elérhetővé tett ilyen célú alkalmazások és interfészek kerülnek áttekintésre.

3.3.1. Xen

xm

Az xm megnevezés a Xen managament user interface alkalmazást rejti maga mögött. Ez a konzolos felület a helyi vezérlést teszi lehetővé. Segítségével létrehozhatunk, törölhetünk és módosíthatunk Xen domaineket. Működése során a kéréseket a xend démon folyamat felé továbbítja, így alkalmazásához mindenképpen szükséges, annak futó példánya. Mivel az xm segítségével fontos rendszerbeállításokon végezhetünk módosítást, csak rendszergazdai jogosultságokkal rendelkező felhasználók használhatják.

xen-tools

A xen-tools egy perl szkript csomag, amely a virtuális gépek, vagy Xen-es terminológiával domainek telepítési és konfigurálási folyamatainak leegyszerűsítését hivatott szolgálni. Használhatóságának a támogatott vendég operációs rendszerek szűk köre szab gátat, csupán egyes Debian és CentOS 4 disztribuciók esetén alkalmazható.

Xen-CIM

A Xen a szabványos menedzsment felületek közül a CIM szabványt implementálja. Az elkészített CIM providerek hozzáférést biztosítanak a gazdagép fizikai erőforrásaihoz és a virtuális gépekhez egyaránt.

Xen Management API

A Xen Management API a Xen saját menedzsment felülete, a korábbiakban említett CIM providerek is részben ezt a felületet használják az általuk elérhetővé tett funkciók megvalósításához. A menedzsment felület távoli elérése HTTP szállítási réteg által továbbított XML-RPC hívások segítségével történhet, amelyek az XML alapú ábrázolásnak köszönhetően platformfüggetlenséget biztosítanak. A felület által elérhetővé tett szolgáltatások és erőforrások osztályokba szervezett hierarchiát alkotnak. Ezeken az osztályokon keresztül hozzáférést nyerhetünk a gazdagép erőrforrásaihoz és a virtuális gépekhez egyaránt.

3.3.2. Hyper-V

Hyper-V Manager

A Microsoft virtualizációs termékének vezérléshez egy egyszerű, de jól használható grafikus felülettel rendelkező eszközt bocsát rendelkezésünkre, mellyel igény szerint testre szabhatjuk virtuális rendszerünket. Segítségével lehetőség nyílik a távoli erőforrások kezelésére is. A Hyper-V szolgáltatás aktiválásával párhuzamosan települ, valamint lehetőség van Windows Vista és Windows Seven operációs rendszerek egyes kiadásaira történő telepítésre is, természetesen ilyenkor csak távoli menedzsmentre van módunk.

Windows Remote Management

A Windows Remote Management rendszer nem más, mint a WS-Managment Microsoft általi implementációja, amely lehetővé teszi hardver és szoftver rendszerkomponensek felügyeletét és így a Hyper-V kezeléséhez is felületet biztosít. Maga a keretrendszer több, önmagában is jól használható komponens együtteséből épül fel. Az architektúra alapkövének a WMI (Windows Management Instrumentation) tekinthető, amely interfészt biztosít az operációs rendszer által nyújtott és felhasznált erőforrásokhoz, segítségével informálódhatunk róluk vagy befolyásolhatjuk működésüket vagy állapotukat. A WMI tulajdonképpen a Common Information Model és a Web Based Enterprise Management szabványok Microsoft általi megvalósítása. Mivel WMI osztályokat több úton is elérhetjük, ezért közvetetten a keretrendszer elérésére is több lehetőség adódik, ezek közül az egyik a WS-Management, de akár használhatjuk a COM vagy DCOM technológiákat is.

Winrm.cmd

Ez a parancssoros eszköz az előzőekben ismertetett menedzsment megoldás részét képezi. Helyi és távoli erőforrásokon egyaránt végezhetünk vele műveleteket. Bár kicsit körülményes módon, de használhatjuk a Hyper-V menedzsmentjére is, ha nem riadunk meg a konfigurációs állományok és tetemes méretű lekérdezések kézzel történő írásától.

3.3.3. ESX

Service Console

Az ESX alapú rendszrek menedzselésének egyik legkézenfekvőbb eszköze a konzol operációs rendszer által nyújtott parancssoros felület, amely teljes felügyeletet biztosít az erőforrások felett. Sajnos ez nem minden változatban érhető el, így teljes biztonsággal nem hagyatkozhatunk rá.

VMware Virtual Infrastructure web felület

A legegyszerűbb és egyben a legszűkösebb funkcionalitást nyújtó menedzsment lehetőség az ESX gazdagépekre alapértelmezetten telepített és engedélyezett webes menedzsment felület használata. A felület által nyújtott szolgáltatások az erőforrás állapotának illetve a virtuális gépek állapotának lekérdezésére, valamint az azokkal végezhető alapfeladatokra - mint például elindítás, leállítás, újraindítás, stb. - korlátozódnak. Ez sajnos szintén hiányzik az ESXi változatokból.

Remote Command Line

Az ESXi változatok esetében a Service Console kikerült a termékből, ennek hiányát Remote Command Line nevű, távoli parancssorral kompenzálhatjuk.

VMware Infrastructure Client

A Hyper-V hez hasonlóan a VMware is rendelkezik a termékeihez készített grafikus menedzsment eszközzel, a VMware Infrastructure Client-tel. Segítségével a virtuális gépek kezelésén kívül olyan magasabb szintű, a rendszer egészére kiterjedő beállításokat eszközölhetünk, mint például az erőforrás poolok létrehozása.

SMASH

Az ESX két viszonylag jól elkülönülő feladatkört ellátó menedzsment interfészt bocsát a felhasználók rendelkezésére. Ezek a felületek nem csak funkcionalitásukban, hanem a felhasznált szabványokban is különböznek. Az első a CIM alapú SMASH (System Management Architecture for Server Hardware), amely a gazdagép erőforráskészletének felügyeletéért felelős. Felépítését tekintve két fő profil csoportra tagolható, az erőforrások széles körét lefedő profilokból álló SMWG (Server Management Working Group) és a kizárólag háttértárakra koncentrálókat tartalmazó SNIA (Strorage Networking Indusrty Association) csoportokra. A felület eléréséhez a WS-Management és a CIM-XML szabvány áll rendelkezésre.

VMware Virtual Infrastructure Management

Az ESX által implementált menedzsment felület második komponense a virtuális erőforrások kezelésére összpontosító VMware Virtual Infrastructure Management interfész. A felület által definiált funkciók webszolgáltatások segítségével érhetőek el. A ESX menedzsment felülete mögött húzódó modell a korábbiaktól eltérő, összetettebb hierarchikus szerkezetet mutat. Az egyes erőforrások szabályos, fa jellegű szerkezetbe tagozódnak, amelyet Inventory-nak hívnak, az egyes erőforrás típusok, mint például a virtuális gépek, vagy maga a gazdagép ennek a fának egy levele, vagy belső csomópontja. Ezen kívül léteznek úgynevezett konténer típusok, amelyek szolgálhatnak az egyes erőforrások csoportosításra vagy éppen lefoglalására, de akár adminisztrációs jellegűek is lehetnek.

3.3.4. KVM

A KVM egy jól alkalmazható virtualizációs megoldás amely teljesítmény és funkcionalitás területén számos pozitívummal bír, sajnálatos módon a menedzselhetőségére ugyanezt nem mondhatjuk el. Önállóan nem rendelkezik távolról elérhető, beavatkozásra és konfigurálási feladatok ellátására alkalmas felülettel, ez a probléma kisebb-nagyobb kerülőutak segítségével leküzdhető. Az egyik ilyen kerülőút, amely szinte minden más terméknél is használható valamilyen formában, a távoli kapcsolat segítségével elért konzolos felület, amellyel szerencsére a KVM is rendelkezik, így egy ssh kapcsolat segítségével könnyedén elérhetjük az irányítani kívánt erőforrást. Bár maga az erőforrás elérés rutin feladatnak tekinthető egy Linux rendszerekben jártas felhasználónak, nem túl hatékony és célravezető, ezért inkább tanácsos a másik menedzsment lehetőséget választani. Ez nem más, mint a

korábban már említett libvirt, amely jelentősen leegyszerűsíti és megkönnyíti a helyzetet. Nem csak egy célirányos, kvázi szabványos felületet használhatunk, hanem elkerülhetjük a konzolos megközelítésből adódó nehézségeket is, hiszen a szoftver csomagnak létezik grafikus menedzsment felülete is, mint például a virt-manager.

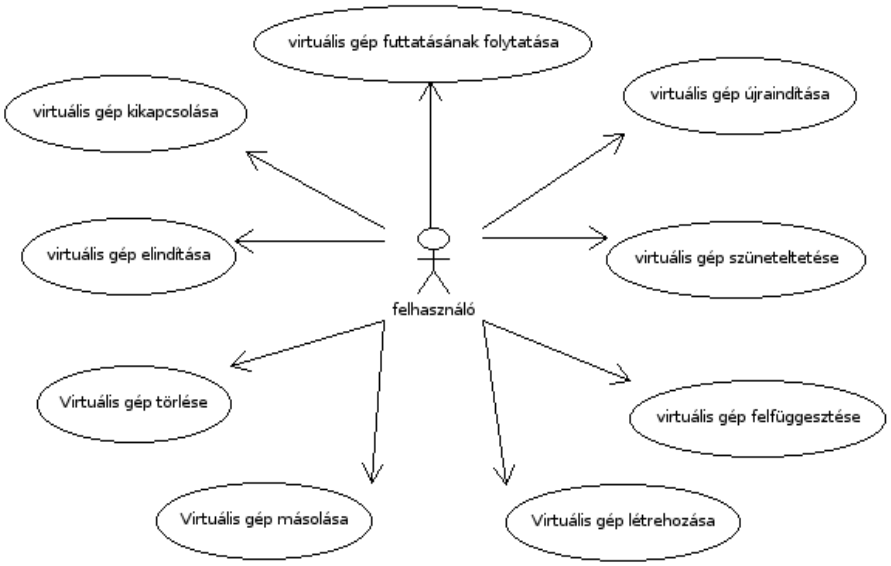
4. fejezet

Multiplatformos menedzsment eszköz tervezési és implementálási folyamata

4.1. Követelmények

A korábban megismert menedzsment felületek az egyes termékek esetén nyújtanak távoli hozzáférést a rendszer felügyeletéhez és működtetéséhez. Minden megoldás rendelkezik speciális, csak rá jellemző szolgáltatással, sokszor az olyan műveletek mögött is egymásnak ellentmondó, szemléletben gyökeresen eltérő megvalósítások rejlenek, amelyek a külső szemlélődő számára teljesen azonosnak tűnhetnek. Egy széles körben alkalmazható eszköz sikeres fejleszthetőségéhez és jól használhatóságához a funkciók olyan szűk halmazát kell megkeresni, amely minden felületen igénybe vehető, amelyek könnyen uniformizálhatók egy megfelelő absztrakciós réteg segítségével, amely teljesen elrejti a valós mögöttes felületeket és megvalósításokat, valamint a folyamatok során felhasznált technológiákat. Ebbe a funkcionalitás halmazba olyan elemek kerülnek, amelyeket minden rendszerrel és virtuális géppel meg kell tudnunk tenni, ezek tételesen a következők.

- **Elindítás:** virtuális gépeket el kell tudnunk indítani.
- **Kikapcsolás:** virtuális gépeket ki kell tudnunk kapcsolni.
- **Újraindítás:** a fizikai gépek reset gombjának hatásával egyenértékű funkció.
- **Felfüggesztés:** számos virtualizációs alkalmazás biztosít lehetőséget az éppen futó vendéggép állapotának háttértárra való mentésére, ilyenkor az állapot leírók stabil tárba való mentését követően. a felhasznált erőforrások felszabadulnak.
- **Szüneteltetés:** egy futó virtuális gép szüneteltetésén azt a szolgáltatást értjük, amikor hosszabb-rövidebb időre megállítjuk azt anélkül, hogy abortálnánk vagy költséges állapotmentéseket végeznénk, ilyenkor csupán a processzor kerül elvételre az aktuális géptől.
- **Folytatás:** a folytatás az előbbi két szolgáltatás inverze, vagyis a háttértárra mentett gép visszaállítása, illetve a megállított gép futtatásának folytatása.

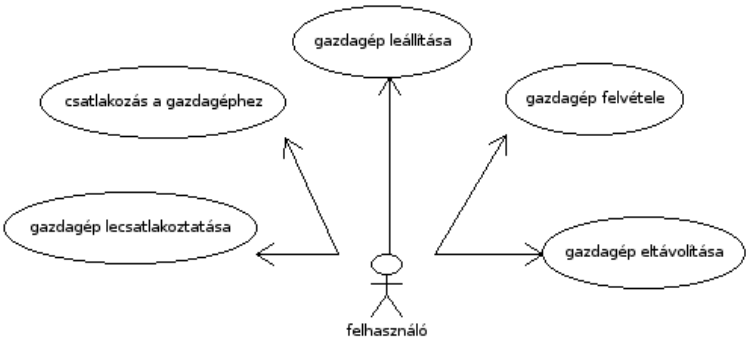


4.1. ábra. Virtuális gép használati esetei

- **Létrehozás:** virtuális gép létrehozása leíró adatstruktúra alapján.
- **Törlés:** virtuális gép fizikai törlése a kiszolgáló gazdagépről.
- **Másolás:** meglévő virtuális gépről való működőképes másolat készítése.

A virtuális gépek életciklusához hozzátartozó folyamatok közül a létrehozás a legkevésbé egységesíthető és legnehezebben megvalósítható funkció, azt is mondhatnánk, hogy nincs két olyan megoldás, amely mögött húzódó koncepció csak részben is hasonlítana. Sok rendszer esetében a létrehozás másik útja a klónozás, néhány virtualizációs termék esetében ez külön beépített funkció, másutt pusztán egy másolással kombinált létrehozási folyamat, amely mögött nincs semmiféle automatizált támogatás. A legtöbb menedzsment felület nem csak a virtuális gépek eléréséhez nyújt eszközt, hanem magát a háttérben szolgáló virtualizációs rendszert is felügyelhetjük, irányíthatjuk, ezeket is jellemezhetjük egyfajta működési ciklussal, azonban itt inkább az indítás és a leállítás folyamata a domináns. Egy fizikai gép távoli bekapcsolása egyszerű, pusztán szoftver erőforrásokat használó megvalósítása gyakran kivitelezhetetlen, bár egy gépet hálózat segítségével fel lehet éleszteni, annak gyakran feltétele, hogy a menedzselni kívánt hoszttal egy hálózati szegmensen helyezkedjünk el. Bár a távolról történő menedzselhetőség egyre fontosabb követelmény egyelőre csak kevés olyan hardver eszköz létezik, amely segítségével helyfüggetlen indítást és hardverbeállítás módosítását végezhetnénk.Ezen okokból fakadóan a leállítás folyamata kerül implementálásra. A gazdagép használati eseteit a 4.2 ábra foglalja össze.

Jelen dolgozat korábbi fejezeteiben négy neves szerver oldali termék jellemzőinek ismertetése történt meg, ezért a fejlesztés során ehhez a négy virtualizációs megoldáshoz

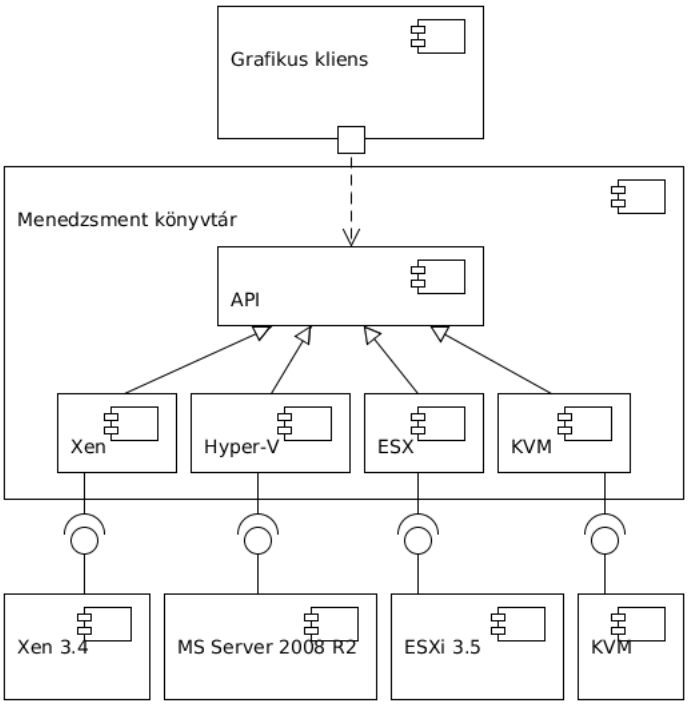


4.2. ábra. Gazdagép használati esetei

történő menedzsment alkalmazás elkészítésére szorítkozunk. Ezek név szerint a következők: Xen, Hyper-V, ESX és KVM.

4.2. Logikai felépítés

A szoftverfejlesztés és ezzel együtt a menedzsment eszközök fejlesztésének egyik fő irányadója és meghatározó eleme az újrahasznosíthatóság és továbbfejleszthetőség szempontja. Ahhoz egy termék hosszú távon sikeres legyen és képes a tovább fejlődésre már a kezdeti lépések során olyan szerkezeti váz megalkotását igényli, amely kellően szoros kapcsolatban álló elemeket tartalmaz ahhoz, hogy a működés minél hatékonyabb legyen és kellően lazán csatoltak, hogy a további fejlesztési folyamatoknak ne szabjon gátat. A vázolt célok és elvárások teljesítéséhez nyújt irányfényt a Model-View-Controller tervezési minta. Jól bevált fejlesztési gyakorlatait és útmutatásait követve az elkészítendő alkalmazás kellő hatékonysággal fog bírni, ahhoz hogy érdemileg is használható legyen mégis jelentős fokú szabadságot biztosít ahhoz, hogy az esetleges komponenseket később tovább bővítsük, vagy lecseréljük, anélkül, hogy az az egész szoftver átdolgozását igényelné. Az MVC-nek megfelelően az implementálni kívánt alkalmazás három fő egységre tagolódik, a modell és kontroller rétegeket magába a menedzser könyvtár tartalmazza míg a nézetet megvalósító grafikus felületet a kliens hordozza. Az alkalmazás fejlesztése Java nyelven történik, amely részben lehetővé teszi az alkalmazás környezet és architektúra függetlenségét, így biztosítva a minél szélesebb körű alkalmazhatóságot. A megvalósítandó eszköz két fő egysége két projektként kerül megvalósításra, mivel a menedzser réteg önmagában nem nyújt közvetlen hozzáférési felületet a felhasználó számára ezért nem is kell belépési ponttal rendelkezni, így keretbe foglalásához elegendő egy osztálykönyvtár projekt. A második komponens teszi elérhetővé a felhasználó számára az osztálykönyvtárban implementált szolgáltatásokat, mindehhez Swing alapú grafikus felületet használva, mely kellően gyors és kis erőforrás igényű asztali alkalmazást eredményez.



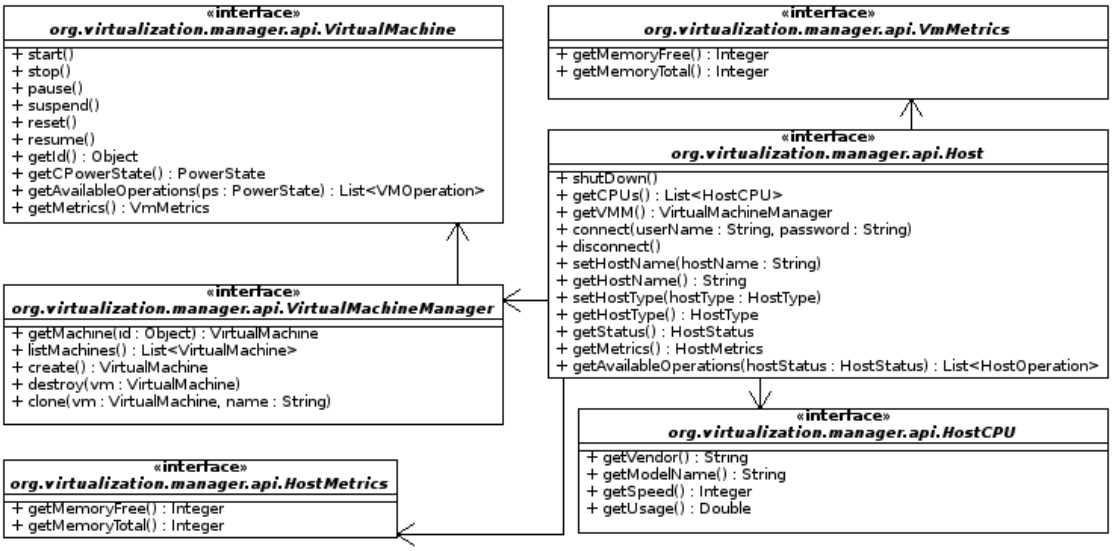
4.3. ábra. Logikai felépítés

4.3. Szerkezeti felépítés

A korábbiaknak megfelelően az alkalmazás két egységből épül fel, az első egység a menedzsment feladatokat ellátó osztálykönyvtár, amely interfészek segítségével definiál egy egységes elérési felületet a későbbiekben megvalósított, konkrét termékek elérését biztosító felületek uniformizált kezeléséhez, a másik egység pedig az ezt felhasználó kliens alkalmazás. A 4.4 ábrán látható osztálydiagram ezt az egyes konkrét menedzsment megvalósítások vázát képező osztályszerkezetet vázolja. A Java nyelv adotságait kihasználva, és a kitűzött feladat karakterisztikáját figyelembe véve, ez a szerkezet pusztán egymással szoros kapcsolatban lévő java interfészek gyűjteménye. Interfészek, mert az egyes konkrét virtualizációs megoldások által tükrözött szemléletmódok és megvalósításbeli különbségek nem teszik lehetővé, hogy olyan funkciókat fogalmazzunk meg és implementáljunk, amelyek minden megvalósítás esetén használhatóak, így a csupán absztrakt metódusok öröklése helyett célszerűbb interfészekkel dolgozni.

4.3.1. Host

Az API legfelső szintjén áll a Host, amely egyetlen, virtuális gépek futtatására alkalmas fizikai gépet reprezentál. Egyértelmű azonosítása IP címmel vagy hoszt névvel lehetséges, mivel valamennyi menedzsment célú felület ilyen jellegű erőforrás azonosítót alkalmaz. Önmagában egy konkrét termékhez implementált Host osztály példányosítása nem jelent semmiféle kapcsolat kiépítést vagy felület elérést, pusztán a későbbiekben elérni kívánt csomópont azonosítására szolgáló leíró információ elraktározását jelenti. Ennek



4.4. ábra. API osztály diagram

köszönhetően elkerülhető az erőforrás pazarlása olyan esetekben, amikor az alkalmazás csak számon akarja tartani az esetlegesen elérendő gépeket, de az adott pillanatban még nem akar hozzájuk csatlakozni. Az irányítani kívánt erőforrással való kapcsolatkiépítés vagy az arra történő referencia szerzést a `connect` metódus végzi el, amely paraméterként az elért csomóponton a felhasználót azonosító név-jelszó adatpárt várja. A csatlakozás folyamatakor lefoglalt adatokat és távoli erőforrásokat a `disconnect` függvény hívásával szabadíthatjuk fel. Mivel az alkalmazás segítségével több virtualizációs megoldáson is végezhetünk beavatkozásokat néha hasznos lehet, ha tudjuk, hogy milyen rendszerrel állunk szemben, ezért a Host példányok célszerűen tudják magukról, hogy milyen típusúak és erről a `getHostType` hívással információt is adnak. A típusán kívül a hosztot reprezentáló osztály példánytól lekérdezhető a hoszt állapota, az erőforrás processzorai valamint az őt jellemző metrikus adatok közül a memóriát leíró teljes és felhasznált mennyiség. Azzal a feltételezéssel élve, hogy egy átlagos, bare metal virtualizációt alkalmazó környezetben egy hoszt csak egyetlen hypervisort futtat, minden Host objektum egyetlen VirtualMachineManager interfészt megvalósító hypervisor példánnyal rendelkezik, amely a `getVMM` metódson keresztül érhető el. A Host felületen keresztül lehetőség nyílik a reprezentált fizikai hoszt leállítására is, ezt a célt szolgálja a `shutDown` hívás. Mivel az egyes megoldások más és más funkciókat tesznek elérhetővé még a fent említett szolgáltatás halmazon is előfordulhat, hogy az egyes rendszerek valamelyiket nem támogatják, ezért szükség van annak ismeretére, hogy a hosztot reprezentáló objektumon milyen funkciókat megvalósító függvényhívásokat hajthatunk végre. Ennek érdekében minden a Host interfészt implementáló osztály példánya információt szolgáltat saját képességeiről, amely a `getAvailableOperations` híváson keresztül érhető el.

Programozási szempontból, mint arról korábban szó volt, a Host típus egy interfész generikus függvényekkel, amely egyrészt a testre szabhatóságot, másrészt az egyes konkrét leszármazottak esetén a fordítási idejű típusbiztonságot szolgálja. A hoszt állapotának,

típusának és engedélyezett műveleteinek konvencionális kezelése érdekében azok külön felsorolás típusban kaptak helyet, így született a HostType, HostStatus és a HostOperation felsorolás típus. Használatuk egyszerűbb, hatékonyabb és biztonságosabb fejlesztést tesz lehetővé, mintha azokat a Host interfész statikus mezőiként definiáltuk volna.

4.3.2. VirtualMachineManager

A VirtialMachineManager interfész egységes felületet biztosít az egyes hypervisorok eléréséhez, a segítségükkel elvégezhető feladatok vezényléséhez. Az interfész öt műveletet definiál, amelyek a virtuális gépek kezelésével, elérésével kapcsolatsak. Ilyen művelet a hypervisor vagy virtual machine manager által kezelt összes virtuális gép lekérdezése, melyhez a `listMachines` függvényt kell hívni, az egyes virtuális gépeket egyenként is lekérdezhetjük a `getMachine` hívás segítségével, amely paraméterként a virtuális gépet egyértelműen azonosító kulcs objektumot várja. Ezen az interfészen keresztül van lehetőség virtuális gépek létrehozására is, bár sok esetben a konkrét virtualizációs megoldások virtuális gépet reprezentáló objektumain értelmezett a törlés és a klónozás jelen esetben, bizonyos architektúrális megfontolások eredményeként ezen a felületen kaptak helyet a virtuális gépet leíró felület helyett. Mivel egy VirtualMachineManager csak egy adott típusú virtuális gép halmaz elemeinek kezelésére alkalmas ezért ez az interfész teljes egészében generikus paraméteres, típusparaméterként az általa kezelt virtuális gépeket leíró, VirtualMachine interfészből származó osztályt várja.

4.3.3. VirtualMachine

Egy VirtualMachine interfészt megvalósító osztály példánya egyetlen virtuális gépet reprezentál, felületet biztosíva a rajta értelmezett műveletek elvégzéséhez, jelen esetben ezek a műveletek a virtuális gép állapotai közötti tranzíciókat hivatottak aktiválni. A virtuális gépet jellemző állapotok a Host esetében alkalmazottakhoz hasonlóan egyetlen közös felsorolás típusba kerültek, amely a PowerState nevet kapta. Mivel az egyes konkrét megvalósítások nem tesznek lehetővé minden ésszerű és elvárható átmenetet, ezért a virtuális gépet reprezentáló osztály információt szolgáltat egy adott állapot függvényében végezhető műveletekről,ezt a célt szolgálja a `getAvailableOperations` függvény, amely egy PowerState példányt vár paraméterként. A virtuális gép állapotát leíró metrikus adatokról a getMetrics híváson keresztül szerezhetünk információt, melynek visszatérési értéke az adott virtualizációs megoldásnak megfelelően implementált, VmMetrics interfészt megvalósító osztály példánya.

4.3.4. VmMetrics

A virtuális gép állapotát leírását segítő szabványos felület, amely jelen esetben a virtuális gépet jellemző teljes memória kapacitás és szabad memória kapacitás értékekre korlátozódik.

4.3.5. HostCPU

Az interfészt megvalósító osztályok példányai egy fizikai processzort reprezentálnak, annak legfontosabb leíró jellemzőivel, ezek az adatok a Host interfészen keresztül érhetőek el. Ez architektúrális szempontból megkérdőjelezhető döntés, hiszen kerülhetett volna a HostMetrics interfészen belülre is az elérést biztosító metódus, azonban sok esetben a processzor nem szolgáltat az adott pillanatbeli állapotról konkrét adatokat, nem tudhatjuk meg az aktuális kihasználtságát vagy dinamikus órajelét, pusztán a hoszt erőforráskészletének statikus elemét képezi. Ez magyarázza hogy miért esett a választás a Host-on keresztül történő elérésre.

4.3.6. Egységes hibakezelés

Mind a kommunikációs során, mind pedig a virtuális rendszerek irányítása során különféle hibák léphetnek fel, amelyek vagy a körülményekből fakadnak vagy a program valamiféle rendellenes működésére figyelmeztetnek. A Java hibakezelésének alapját más programozási nyelvekéhez hasonlóan a kivételek dobása és elkapása képezi. Minden egyes hiba típusnak külön kivétel osztály felel meg, amely az informativitás szempontjából hasznos, azonban már egy kisebb, de többrétű alkalmazás esetén is túl sok hiba lekezelését igényelné, ennek a problémának több lehetséges megoldása van, az egyik legkézenfekvőbb és legcélravezetőbb az, ha saját kivételtípusokkal dolgozunk, amelyek segítségével becsomagoljuk az egyes hibákat, így az elkészített kód olvashatóbb és kisebb lesz, és elkerülhető a hiba által hordozott információtartalom elvesztése is. Jelen esetben is ez a megközelítés került alkalmazásra, ennek megfelelően két kivétel osztály született, az első a virtuális gép hibáinak jelzésére, míg a másik a kommunikáció során fellépő hibák kézben tarthatóságának biztosítására. Bár az osztálykönyvtár esetében mindezek nem jártak túl nagy előnnyel, a hozzá készített kliens alkalmazás esetén jelentős könnyebbséget jelentet a fejlesztés során.

4.3.7. Communicator

Ez az osztály nem implementál semmiféle különleges interfészt, mégis fontos szerepet játszik a program működésében, megvalósítástól függően feladata széles skálán mozoghat. Ez lehet csupán a távoli erőforrással történő kapcsolat felvétel, de lehet maguknak a távoli hívásoknak egyfajta interpretere is, mindezeket a későbbiekben konkrét példákon keresztül is figyelemmel kísérhetjük.

4.4. Hívási szekvenciák

Az előbbiekben definiált programozási felület az egyes függvényhívások segítségével válik érdemileg is hasznossá.A következőkben ezen hívássorozatok bemutatása következik néhány példa segítségével. Mindegyik hívássorozat azzal a feltételezéssel él, hogy a felhasználó elsődlegesen egy Host példányt lát, és azon keresztül kívánja elérni céljait.

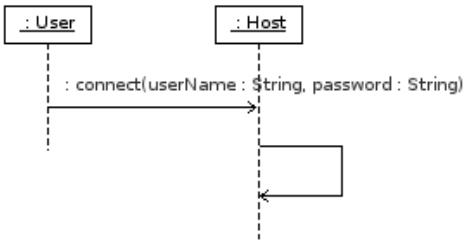
Egy gazdagéphez való kapcsolódás folyamatát írja le a 4.5 ábra. Mint látható egy aszinkron hívásról van szó, a Host osztályon belül hívásokat csak fekete dobozként tudjuk

jellemezni. Ennek fő oka az, hogy az egyes konkrét termékekhez implementált osztályok esetében más és más teendőket kell elvégezni. A kapcsolat bontása az előbbihez hasonlóan történne, értelem szerűen a disconnect hívás hatására.

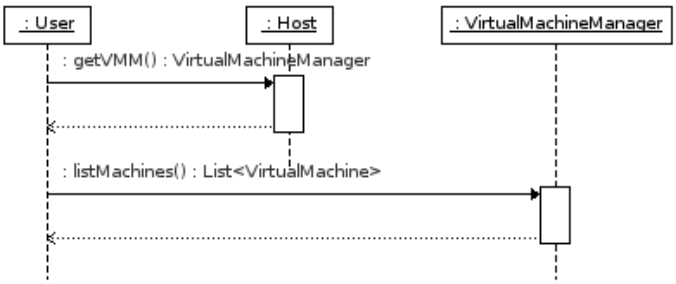
A következő, 4.6 ábrán az egyes csomópontokon található virtuális gépek listázásának menetét láthatjuk. A virtuális gépeket egyesével is elkérhetjük, ehhez ismerni kell a virtuális gép egyedi azonosítóját, ezt ábrázolja a 4.7 ábra.

A virtuális gép állapotát megváltoztató hívások folyamatai azonos analógiára épülnek, eltérés csak az utoló hívásban lehet, amely az aktuálisan elérni kívánt állapothoz vezető átmenetért felelős. A 4.8 ábrán a virtuális gép indításának egy lehetséges szekvenciája található, lehetséges és nem kötelezően végrehajtandó, hiszen a virtuális gépre más úton is szerezhetünk referenciát.

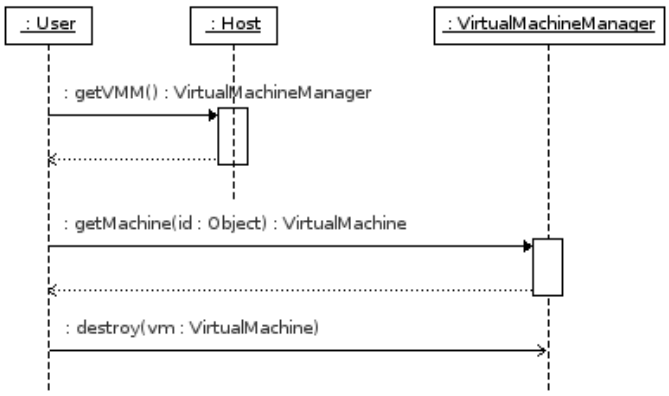
A lehetséges hívási szekvenciák harmadik csoportját a virtuális gép életciklusát érintő funkciók alkotják, ilyen a virtuális gép létrehozása, törlése és másolása. A 4.9 ábra egy virtuális gép törlése során bekövetkező lehetséges hívássorozatot ábrázol.



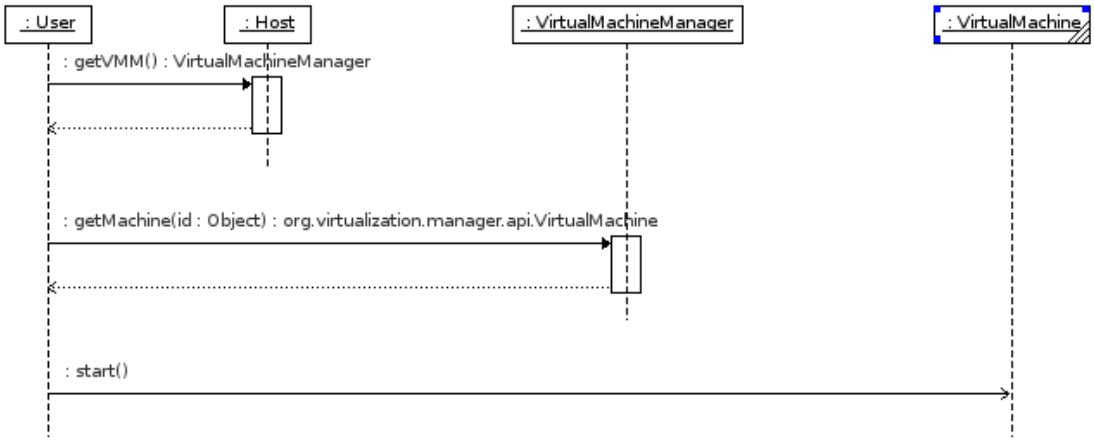
4.5. ábra. Gazdagép csatlakoztatása



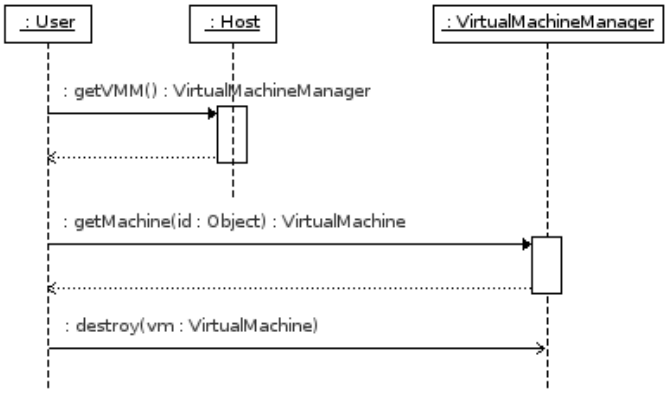
4.6. ábra. Virtuális gépek listázása



4.7. ábra. Virtuális gép lekérdezése



4.8. ábra. Virtuális gép indítása



4.9. ábra. Virtuális gép törlése

4.5. Menedzsment felület implementációk

4.5.1. Xen

A Xen alapú virtuális rendszerek esetében a legjobban használható távoli elérést biztosító protokoll az XML-RPC szabvány, amely segítségével távoli erőforráson kezdeményezhetünk függvényhívásokat xml-ben leírt adatok segítségével. A felület segítségével elérhető osztályok és szolgáltatásokról a Xen Management API [22] ad bővebb felvilágosítást.

Mint az a sikeres és jól használható programozási felületek esetén lenni szokott ez is rendelkezik számos programozási nyelven elkészített könyvtár csomaggal, amely gyors és egyszerű alkalmazásfejlesztést tesz lehetővé. Mivel a nyers XML-RPC hívások sem okoznak jelentős problémát, és viszonylag könnyen használhatóak, ezért a továbbiakban az egyes funkciók megvalósítása közvetlenül azok segítségével történik, kihagyva az egyes könyvtárcsomagok által nyújtotta, előre megvalósított szolgáltatásokat.

Az XML-RPC hívások használatában Java nyelv esetén az Apache XML-RPC nyújt segítséget, az alkalmazás fejlesztése során a kommunikációs nehézségek áthidalásához a könyvtárcsomag 3.13 verziója került felhasználásra. Használata nem csak a közvetlen kommunikációra terjed ki, működése során elfedi a Java típusok használatából adódó konverziós feladatokat illetve az objektumok ki (unmarshall) és becsomagolásából (marshall) adódó többlet munkát.

A Xen API lehetővé teszi az egyes távoli metódusok hívásmódjának megválasztását, így lehetőség van szinkron és aszinkron hívásokra egyaránt. A fejlesztés során több tényező miatt is a szinkron hívások részesültek előnyben, ilyen tényező az, hogy a Java nyelv viszonylag jól használható eszközkészletet bocsájt rendelkezésre a többszálú alkalmazások fejlesztéséhez. Így a tényleges kliens alkalmazás esetén egy hosszú folyamat szálba ágyazásával elkerülhető az egész program lebénulása.

Communicator

A Xen esetében a kommunikáció session alapú, amelyre a bejelentkezés során visszakapot azonosító segítségével hivatkozhatunk. A kilens és a szerver közti kapcsolattartás a XmlRpcClient osztálypéldány feladata, amely egy bővített funkcionalitású http kliens feladatait látja el. Működésének szükséges feltétele a megfelelő konfiguráltság, amely XmlRpcLocalClientConfig osztálypéldány segítségével történik. Megadhatunk olyan beállításokat, mint a Basic Athentication során használt felhasználónév és jelszó páros, a karakterkódolás vagy az üzenetek tömörítésének szükségessége, mindezek mellett a legfontosabb paraméter a címzettet azonosító URL. Jelen esetben ez az egyetlen szükséges paraméter, ugyanis a hitelesítést követően az egyes hívások legitimitásának eldöntése a hívási paraméterben átadott munkamenet azonosítónak a segítségével (session id) történik.

Egy távoli hívás általánosságban három összetevőből áll: a hívott metódus azonosítójából, a munkamenet azonosítóból és a hívás paramétereiből. Ennek az általánosíthatóságnak az eredményeként született meg a Request osztály, amely a függvényazonosító és a paraméterek egységbe fogalását célozza. A hívó osztálynak nincs

```
public Object sendRequest(Request parameter) throws CommunicationException {
    Object[] parameters = new Object[parameter.getData().length + 1];
    parameters[0] = sessionId;
    System.arraycopy(parameter.getData(), 0, parameters, 1, parameter.getData().length);
    try {
        Map result = (Map) client.execute(parameter.getCommand(), parameters);
        if (result.get("Status").equals("Success")) {
            return result.get("Value");
        } else {
            StringBuilder sb = new StringBuilder();
            sb.append(result.get("Status"));
            Object[] error = (Object[]) result.get("ErrorDescription");
            for (Object o : error) {
                sb.append(' ');
                sb.append(o);
            }
            throw new CommunicationException(sb.toString());
        }
    } catch (Exception e) {
        throw new CommunicationException(e);
    }
}
```

4.10. ábra. Communicator sendRequest

tudomása a session id-ról, azzal csak a Communicator példány rendelkezik. Egy hívás kezdeményezése a `sendRequest` metódus meghívásával történik, amely a távoli eljárás paramétereit kiegészíti a munkamenet azonosítójával majd ezt követően az XML-RPC kliens segítségével végrehajtja a hívást. A `sendRequest` függvény forráskódja a 4.10 ábrán látható.

Mivel a távoli fél, jelen esetben a Xen szerver objektumokat tart fent minden munkamenet számára, ezért a munka befejeztével a memória pazarlást a munkamenet szabályos befejezésével kerülhetjük el, ezt a célt szolgálja a `releaseSession` metódus.

XenHost

A Xen API a menedzsment folyamatok megkönnyítése érdekében az összetartozó adatokat és a hozzájuk köthető metódusokat egységbe foglalta, mégpedig az objektum-orientált programozási paradigmát követve menedzsment osztályokat hoztak létre. Ilyen osztály a Host, amely egy konkrét Xen példányt ír le. A XenHost osztály részben ennek a host osztálynak a kliens oldali megfelelője, kibővítve a korábban leírt programozási felületből adódó egyéb funkcionalitásokkal. A Host példányra történő referencia szerzés a host.get_all távoli metódus meghívásával lehetséges, amely eredményül egy tömböt ad vissza, melynek elemei a gazdagép által ismert rendszerek. Többnyire ez a lista egyelemű, ezért jelen esetben is élhetünk azzal az egyszerűsítéssel, hogy a XenHost osztály példányosításához az eredménylista eslő elemét használjuk.

A XenHost `connect` metódusa részben eltér a többi megoldásétól, ugyanis ebben az esetben tényleg csatlakozásról van szó, majd ezt követi a Host példányra történő referencia megszerzése. A visszakapott adatok közül egyedül a távoli objektumot azonosító kulcs érték kerül felhasználásra, más a XenHost számára szükséges adatot nem szolgáltat.

A csatalakozással ellentétes disconnect folyamat is eltérő abból a szempontból, hogy

nem pusztán helyi objektumok kitörlését igényli, hanem a távoli objektumok által lefoglalt hely felszabadítását is kezdeményezni kell. Ez a korábban említett `releaseSession` Communicator segítségével történik, így a XenHost objektum delegálja a kérést a Communicator felé. A gazdagép leállítása egy kétlépcsős folyamat, amelyben először letiltjuk az új gépek futtatásának lehetőségét a host.disable hívással, majd a második lépésben kikapcsoljuk a gépet a host.shutdown távoli metódus meghívásával.

XenVMM

Xen esetében nem található olyan osztály, amely a virtuális géptől elkülönül és a megvalósított menedzsment felület VirtualMachineManager osztályának lenne megfeleltethető. A Xen esetében ezek a funkciók a VM osztályhoz vannak rendelve, amely a virtuális gépeket leíró szerver oldali osztály. Tehát ha figyelembe vesszük a szerver oldali megvalósítást, akkor elmondhatjuk, hogy a XenVMM osztály példányok csupán közvetítő szerepet játszanak a kliens oldali és a szerver oldali objektumok között.

A virtuális gépek lekérdezése a VM.get_all hívással történik, eredményeként a gazdagépen található virtuális gépek referenciáinak tömbjével tér vissza. Ezeket a referenciákat felhasználva egyesével tudjuk lekérdezni a virtuális gép példányok adatait, amelyek a virtuális gépet leíró rekordok formájában érkeznek a klienshez. Ez a rekord nem más mint egy kulcs-érték párokat tartalmazó HashMap szerkezet. A virtuális gépek lekérdezése némi figyelmet igényel, ugyanis a Xen az egyes virtuális gép sablonokat (template) is virtuális gépként kezeli, így miután lekérdeztünk egy vendég gépet leíró rekordot a benne lévő adatokból kell eldöntenünk, hogy az sablon vagy nem, erre a célra szolgál az `is_a_template` mező.

A sablonok esetleges felhasználásának egy módjára mutat példát a XenVMM osztály `createByTemplate` metódusa, amely célja egy virtuális gép létrehozása lenne egy sablon segítségével. A sablon másolását követően a hozzá tartozó konfigurációs bejegyzéseket kell módosítani, elsősorban azt kell megadni, hogy hol tároljuk a hozzá tartozó diszk állományokat. A konfigurációs adatok diszkeket leíró adateleme egy xml struktúra, amely hatékony kezeléséhez azt fel kell dolgozni és dokumentum modellt kell belőle készíteni, ezt a célt szolgálja a VMConfig osztály `getDisks` metódusa. A diszk adatok módosítása a `setDisks` hívással valósul meg, amely a paraméterként kapott dokumentum csomópontot visszaalakítja xml adattá.

A megfelelő beállítások elvégeztével a konfigurációs adatokat hozzá kell rendelni a virtuális géphez, ezt a VM.set_other_config távoli hívás segítségével végezhetjük el, ez után már csak egyetlen lépés van hátra, VM.provision, mely hatására a virtuális gép megszületik. A virtuális gép sablon alapú létrehozását megvalósító fő eljárás forráskódja a 4.11 ábrán olvasható.

XenVM

A Xen menedzsment felületének VirtualMachine interfészt implementáló osztályáról, a XenVM-ről már a korábbiakban esett szó a konfigurációs rekord kapcsán. Minden

```
public void createByTemplate(XenVM vm, String name) throws CommunicationException,
DOMException, IOException, ParserConfigurationException, SAXException, IllegalArgumentException,
TransformerException, TransformerFactoryConfigurationError {
      Request templateRequest = new Request("VM.clone", new Object[]{vm.getId(), name});
      XenVM templateVm = new XenVM((String) communicator.sendRequest(templateRequest), communicator);
      VMConfig vmc = templateVm.getConfiguration();
      Object storageId = getDefaultStoragePool();
      Element disks = vmc.getDisks();
      NodeList nl = disks.getChildNodes();
      for (int i = 0; i < nl.getLength(); i++) {
          Node disk = nl.item(i);
          NamedNodeMap nnm = disk.getAttributes();
          for (int j = 0; j < nnm.getLength(); j++) {
              Node attr = nnm.item(j);
              if (attr.getNodeName().equals("sr")) {
                  attr.setTextContent((String) storageId);
              }
          }
      }
      vmc.setDisks(disks);
      templateVm.setConfiguration(vmc);
      Request provisionRequest = new Request("VM.provision", new Object[]{vmc.getVmRefId()});
      communicator.sendRequest(provisionRequest);
}
```

4.11. ábra. XenVMM createByTemplate metódusa

példány rendelkezik a Communicator osztály aktuális példányával, amelyet konstruktor paraméterként kapnak meg. A Communicator ismerete azért szükséges, mert a virtuális gép állapotaival kapcsolatos műveleteket a virtuális gép objektumán történő metódus hívással érhetjük el, valamint a szerver irányú hívásokat az objektum saját maga végzi, anélkül hogy azt valamelyik másik helyi osztályra bízná. A kiszolgáló oldali hívásokat a VM osztályon keresztül érhetjük el, ennek megfelelően a VM.start, VM.resume stb. hívásokat kell beágyazni.

XenVmMetrics

A virtuális gép állapotát jellemző adatokat két elérési pont segítségével szerezhetjük be. A virtuális gép statikus jellemzőit a VM osztályon keresztül, így a maximális memória méret elérése a VM.get_memory_static_max távoli metódus hívásával lehetséges, míg a dinamikus adatokról, mint az aktuálisan felhasznált memória mennyisége, a VM_metrics.get_record hívás által visszaadott struktúrából értesülhetünk.

XenHostMetrics

A gazdagép jellemzőinek elérése az előbbiekben ismertetett virtuális gép adatok elkérésének folyamatához hasonlóan két lépcsőben zajlik, azonban azzal ellentétben nem a dinamikus és statikus adatok eltérő helyen történő tárolása miatt, hanem mert a host osztály csak egy referenciát szolgáltat a saját adatainak eléréséhez. Ezt a referenciát kell használni az általa azonosított távoli objektum adatainak elkéréséhez, amely a host_metrics.get_record hívással tehető meg.

XenHostCPU

A Xen esetében lehetőségünk van a gazdagép processzorainak lekérdezésére. A processzor felsorolás alapegysége nem a fizikai foglalat, hanem a végrehajtó egység. A lekérdezés a host.get_host_CPUs hívással valósítható meg, amely visszatérési értékül a processzor objektumokat azonosító referencia objektumokat adja, a konkrét példányokat jellemző adatokat a host_cpu osztály host_cpu műveletével érhetjük el.

4.5.2. Hyper-V

A Microsoft termékének távoli menedzseléséhez számos elérési pontot biztosít, mindegyikükben közös, hogy működésük alapját a WMI osztályok és szolgáltatások adják. Lehetőségünk van COM, DCOM segítségével kommunikálni a távoli erőforrással, valamint használhatjuk a webszolgáltatás alapú interfészt is. Jelen esetben az utóbbi történt. Sajnálatos mód a választott programozási nyelv miatt ez a termék okozta a legnagyobb fejtörést és igényelte a legtöbb munkát, ennek okát főleg a Microsoft terméktámogatásában és üzleti stratégiájában kell keresni. C, C++ és C# nyelveken beépített utasítások segítéségével tehetjük meg ugyanazokat a hívásokat, amelyekhez java esetén az egyes SOAP üzenetek xml tartalmának aprólékos, lépésről lépésre töténő elemzése és feldolgozása szükséges.

A fejlesztés során a kommunikáció biztosításához szükség volt a WS-Management Java nyelven írt megvalósítására, szerencsére a Wiseman [14] projekt formájában egy nyílt forráskódú, ingyenesen és viszonylag egyszerűen használható könyvtár csomagra talál az ember, amely megoldja a webszolgáltatásokkal történő kommunkáció során felmerülő problémák egy részét. A felmerülő problémák másik részének a kommunikáció során alkalmazott üzenetek összeállítása és értelmezése során jelentkezett, amely főként az xml alapú dokumentumokkal történő operációt jelenti. A dokumentumok szabványos úton történő feldolgozásához a W3C Document Obejct Model-jének implementációja nyújtott jól használható eszközkészletet. A SOAP üzenetek manipulációja pedig a javax.xml.soap API segítségével valósult meg.

Communicator

A Hyper-V esetében a kommunikáció http protokoll segítségével történik, más megoldásokkal ellentétben, amelyek szintén webszolgáltatásokat használnak menedzsment célokra, a kapcsolat felvétel során nem létesül semmiféle hosszútávú munkamenet (session). Minden kérés során újra igazolnunk kell a személyazonosságunkat. Hogy ez a felhasználás szempontjából ne legyen zavaró a Communicator osztály két mezővel rendelkezik, egy PasswordAthenticator-ral és egy HttpClient-tel, amelyek a Communicator konstruktorában példányosodnak, felhasználva ehhez annak paramétereit, a felhasználónevet és a jelszót. A kliens tagváltozóként kapja az autentikátort, amellyel a kommunikáció során igazolhatja magát. A felhasználó azonosítása HTTP Basic Access Authentication segítségével történik, amely hagy némi kívánnivalót a biztonság terén, de a feladat által támasztott követelményeknek megfelel. Az üzenetváltások alapját az Addressing példányok képzeik,

amelyeket a Communicator továbbít a címzett felé a `sendRequest` metódusa segítségével. Minden olyan objektum, amelynek kommunikációra van szüksége a működése során rendelkezik az aktuális kapcsolatot reprezentáló Communicator osztály példánnyal.

HypervHost

A Hyper-V virtualizációs megoldást alkalmazó gazdagépek eléréséhez a Win32_OperatingSystem WMI osztályt kell alkalmazni. Ennek az osztálynak a példányai közelítik meg legjobban a gazdagép fogalmát. A Hyper-V menedzsment kapcsán meg kell említenünk egy fogalmat, amelyet a programozás technika is sokat használ, ez pedig a Singleton. Egy osztályt Singleton-nak nevezünk, ha csak egyetlen példánya létezhet. WS-Managment használata esetén az ilyen egypéldányos osztálytípusokat egyszerű lekérdező (get) üzenetek segítségével érhetjük el. Ez az objektum típus a legritkább, a legtöbb esetben a lekérdezés eredményéül egy felsorolás típust kapunk, amelyen végigiterálva kapjuk meg a valós eredményeket.

A Win32_OperatingSystem osztály sem tartozik a Singleton tulajdonságúak körébe, amely abból adódhat, hogy egy fizikai gépre több operációs rendszert telepíthetünk. Jelen esetben azzal a feltételezéssel élünk, hogy pontosan egy telepített operációs rendszert tartalmaz a gépünk. Ha figyelembe vesszük, hogy a Hyper-V vagy dedikált vagy általános szerver célú operációs rendszerként használatos, ez a megközelítés teljesen helyénvaló.

A fenti egyszerűsítésekből adódóan, a gazdagéphez történő kapcsolódás során visszakapott osztálypéldányok listájából mindig a legelsőt tekintjük a gépen futó gazda operációs rendszernek.

A HypervHost osztály esetében találkozunk először egy osztály metódusának meghívása kapcsán adódó nehézségekkel, mégpedig a `shutDown` metódus implementálásakor. Egy metódus meghívásakor először is azt kell meghatározni, hogy melyik távoli objektumon értelmezett a hívás, majd ezt követően a SOAP üzenet törzsében kell elhelyezni a hívás megfelelően ábrázolt paraméterlistáját. A paraméter átadás általános szabályaként elmondhatjuk, hogy az üzenettest első gyermekeként kell elhelyezni egy osztálynév_INPUT alakú bejegyzést, amely gyermekeiként tartalmazza a paramétereket. Mivel xml dokumentumról van szó, ezért minden egyes elem esetén meg kell adni, hogy melyik sémán értelmezett az adott paraméter.

HypervVMM

A virtuális gépek kezelésével kapcsolatos funkciókat a Msvm_VirtualSystemManagementService WMI osztály példányain keresztül érhetjük el, a Host implementációnál ismertetettekhez hasonlóan ez sem Singleton. Ez azért érdekes, mert egy gazda operációs rendszeren egy időben csak egy hypervisor futhat, és ez az osztály tulajdonképpen azt írja le. Ennek köszönhetően az osztály példányára történő referencia szerzéshez egy felsoroló utasítást kell megfogalmaznunk, majd a visszakapott felsorolás kontextus segítségével lekérdezni azt.

A 4.13 ábrán található kódrészlet szemlélteti ennek a módját. A gazdagéptől kapott

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  xmlns:mdo="http://schemas.wiseman.dev.java.net/metadata/messagetypes"
  xmlns:mex="http://schemas.xmlsoap.org/ws/2004/09/mex"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
  xmlns:wsen="http://schemas.xmlsoap.org/ws/2004/09/enumeration"
  xmlns:wsman="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd"
  xmlns:wsmeta="http://schemas.dmtf.org/wbem/wsman/1/wsman/version1.0.0.a/default-addressing-model.xsd"
  xmlns:wxf="http://schemas.xmlsoap.org/ws/2004/09/transfer" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <env:Header>
    <wsman:ResourceURI xmlns:ns11="http://test.foo" xmlns:ns12="http://examples.hp.com/ws/wsman/user">
      http://schemas.microsoft.com/wbem/wsman/1/wmi/root/virtualization/Msvm_VirtualSystemManagementService
    </wsman:ResourceURI>
    <wsman:OperationTimeout xmlns:ns11="http://test.foo" xmlns:ns12="http://examples.hp.com/ws/wsman/user">
      P0Y0M0DT0H0M30.000S
    </wsman:OperationTimeout>
    <wsa:Action env:mustUnderstand="true" xmlns:ns11="http://test.foo" xmlns:ns12="http://examples.hp.com/ws/wsman/user">
      http://schemas.xmlsoap.org/ws/2004/09/enumeration/Enumerate
    </wsa:Action>
    <wsa:ReplyTo xmlns:ns11="http://test.foo" xmlns:ns12="http://examples.hp.com/ws/wsman/user">
      <wsa:Address env:mustUnderstand="true">http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:MessageID env:mustUnderstand="true" xmlns:ns11="http://test.foo" xmlns:ns12="http://examples.hp.com/ws/wsman/user">
      uuid:baf58b13-2e41-4ebe-abf1-b2d336223ea7
    </wsa:MessageID>
  </env:Header>
  <env:Body>
    <wsen:Enumerate xmlns:ns11="http://test.foo" xmlns:ns12="http://examples.hp.com/ws/wsman/user"/>
  </env:Body>
</env:Envelope>
```

4.12. ábra. WS-Management üzenet

válasz üzenet egy Addressing objektum, amely tulajdonképpen egy Enumeration osztály példányt határoz meg. Ennek segítségével állítjuk össze a lekérdezéshez használt új üzenetet. A kérés `setPull` metódusával adjuk meg a kontextus azonosítóját, a válasz üzenet karaktereinek maximális számát, valamint azt, hogy maximálisan hány elemet szeretnénk egyetlen kérésben megkapni. Természetesen meg kell adni, hogy milyen eseményt szeretnénk kiváltani az üzenettel, ez ebben az esetben egy Pull művelet, amely a lekérdezéssel egyenértékű. Miután megadtuk, hogy milyen erőforráson legyen értelmezett a lekérdezés és elküldtük a kérést a válaszként kapott SOAP üzenet törzsének tartalmán kell végig iterálni, vagy kiválasztani a megfelelő elemet, hogy referenciát szerezzünk a kívánt távoli objektumra. Referenciaként többnyire egy vagy több attribútum által alkotott CIM kulcsot használhatjuk, amely egyértelműen azonosítja a távoli objektumot.

HypervVM

Az osztály egy, a gazda gépen található virtuális gépet reprezentál. A neki megfelelő WMI osztály a CIM_ComputerSystem osztályból származtatott Msvm_ComputerSystem, amely virtuális gépek mellett fizikai gépek reprezentálására is használatos. Amikor a VirtualMachineManager segítségével lekérdezzük a fizikai gépen található vendég gépeket ezt is figyelembe kell venni, ugyanis a lekérdezés visszaadja az aktuális gazdagépet is, így azt ki kell szűrni. A virtuális gépen kezdeményezett állapot változtatásokat a RequestStateChange metódus segítségével hajthatjuk végre, amely paraméterként az új állapotot jelző értéket várja.

Msvm_ConcreteJob

A Hyper-V programozási felületének használata során sok hívás aszinkron módon történik, ami azt jelenti hogy a szerver oldalon egy új munkamenet indul, amely az

```
Enumeration response = new Enumeration(addr);
EnumerationContextType ctx = response.getEnumerateResponse().getEnumerationContext();
String enctx = null;
if (ctx.getContent().size() > 0) {
    enctx = (String) ctx.getContent().get(0);
}
Enumeration pullRequest = EnumerationUtility.buildMessage(null, EnumerationMessageValues.newInstance());
pullRequest.setPull(enctx, -1, 10, null);
pullRequest.setAction(Enumeration.PULL_ACTION_URI);
Management mm = new Management(pullRequest);
mm.setResourceURI(HypervVM.URI);
Addressing machineList = communicator.sendRequest(mm);
SOAPMessage message = machineList.getMessage();
SOAPBody messageBody = message.getSOAPBody();
NodeList nl = messageBody.getFirstChild().getFirstChild().getChildNodes();
int length = nl.getLength();
if (length > 1 || length <= 0) {
    throw new CommunicationException();
} else {
    return new HypervVMM(communicator, nl.item(0));
}
```

4.13. ábra. Host lekédezése WS-Enumeration használatával

elvégzendő feladat mennyiségétől függően hosszabb-rövidebb ideig tart. Emiatt gyakran a metódushívás rögtön visszatér, visszatérési értéke pedig egy munkamenet azonosító lesz, amelyet felhasználva lekérdezhetjük annak állapotát és az azt leíró adatokat. Minden munkamenethez egy Msvm_ConcreteJob WMI osztálypéldány tartozik a szerver oldalon és a Java oldali osztály is ezzel a névvel rendelkezik. Ezeknek az objektumoknak az állapotváltozásai a JobTracker nevű osztály segítségével végezhetjük, amely az azonosító alapján lekérdezi a keresett munkamenetet.

HypervVmMetrics

A virtuális gép aktuális állapotát leíró számszerű adatokat a HypervVmMetrics osztály példányai hordozzák. Ezeknek az adatoknak az elkérése a Msvm_VirtualSystemManagementService WMI osztály GetSummaryInformation metódusán keresztül történik, ezért minden virtuális gép rendelkezik egy referenciával az őt irányító VirtualMachineManager osztályról. Az állapotleíró adatok elkérésekor a virtális gépet reprezentáló osztály delegálja a kérést a manager osztálypéldányhoz így szerezve meg a keresett adatokat. A keresés eredményeként várt adatokat a keresés paramétereként adhatjuk meg, a visszatérési érték egy felsorolás, amely a feltételeknek megfelelő adatokból álló bejegyzéseket tartalmaz minden virtuális gép számára. Az egyes virtuális géphez tartozó sor azonosításához a virtuális gépet egyedileg azonosító Element Name attribútumot kell használni, amely CIM kulcs szerepet tölt be a virtuális gépet leíró WMI osztályban.

HypervHostCPU

A gazdagép fizikai processzorát leíró adatokat a WMI Win32_Processor osztálya segítségével szerezhetjük be. Ez az osztály egyetlen foglalatot benépesítő fizikai eszköz tulajdonságait írja le. Azt hogy a processzoron belül mennyi végrehajtó egység található,

az osztály attribútumai között találhatjuk meg.

HypervHostMetrics

A gazdagépet leíró adatokat hordozó HypervHostMetrics osztálypéldány összeállításához szükséges adatokat a HypervHost adataihoz hasonlóan érhetjük el. Ehhez nem kell mást tennünk, mint beszerezni az aktuális gépet leíró Win32_OperatingSystem WMI osztály példányt, amely teljesítmény specifikus adatokat is tartalmaz, jelen esetben ezek közül a TotalVisibleMemorySize és a FreePhysicalMemory érdekes számunkra, amelyek a rendszer által elérhető és aktuálisan használaton kívüli memória mennyiségéről adnak információt.

4.5.3. ESX

Az ESX alapú rendszerek menedzselhetőségének egyik módja a webszolgáltatás alapú menedzsment. Sajnálatos mód a virtualizációval közvetlenül kapcsolatos erőforrások és folyamatok irányítására biztosított felület saját menedzsment protokollt alkalmaz, bár első hallásra a webszolgáltatás alapúság azt sugallná, hogy ismét WS-Management-tel van dolgunk, az csak a szerver fenntartásával kapcsolatos feladatok esetén alkalmazható.

A menedzsment felület működése is elég különlegesnek mondható, három jól elkülönülő objektumcsoportot definiál. Az első a ServiceInstance, amely minden menedzsment folyamat és adat belépési pontja. A második csoportot a ManagedObject-ek alkotják, amelyek szolgáltatásokat és erőforrásokat reprezentáló objektumok, csak a szerveren léteznek és nem jelennek meg a webszolgáltatás leírójába. A harmadik csoportba a ManagedObject-ekről adatokat szolgáltató DataObject-ek tartoznak.

A felület további érdekességei közé tartozik, hogy részben objektum alapú szemléletet tükröz, azonban az egyes objektumon értelmezett hívásokat mégsem az osztálypéldányokhoz rendeli. Egy ESX menedzsment alkalmazás fejlesztéséhez két irányból kezdhetünk hozzá, vagy mindent magunk csinálunk, és ennek megfelelően a WSDL fájl alapján legeneráljuk a menedzsment felület által definiált DataObject osztályokat vagy olyan könyvtárcsomagot használunk amelyhez ezt már megtették. Jelen esetben a második lehetőségre esett a választás. A VMware VI Java API [9] segítségével viszonylag egyszerűen érhetjük el a menedzselni kívánt ESX erőforrásainkat. Az API alapját a WSDL-ből generált osztályok adják, amelyeket burkoló osztályokkal vettek körül, hogy így szüntessék meg a korábbi objektum-metódus elkülönülésből adódó kellemetlenségeket.

Communicator

A kapcsolat kiépítéséhez egy ServiceInstance példányra kell szert tenni, amely konstruktor paraméterként az erőforrást azonosító URL és a felhasználót azonosító név-jelszó párost illetve az SSL tanúsítvány szükségességét jelző értéket kéri. Mivel az SSL tanúsítványok használata megnehezítette volna a használatot ezért annak szükségességétől eltekintettünk. Miután visszakaptuk a várt objektum példányt a további kommunikáció annak segítségével történik. Egyrészt hordozza a valós kapcsolatot, másrészt minden erőforrás és szolgáltatás

ezen az objektumon keresztül érhető el, miközben a valós kommunikációs folyamatokat elfedi előlünk.

EsxHost

A szerver oldali gazdagép adataira a többi virtualizációs megoldáshoz írt menedzsment osztályokhoz hasonlóan a `connect` metódus meghívásakor szerzünk referenciát. Vagyis jelen esetben nem referenciát, hanem a konkrét jellemző adatokat.

Az egyes erőforrások könyvtár szerkezetekbe szerveződnek, melynek tetején a gyökér könyvtár (RootFolder) áll. Ahhoz, hogy a kívánt erőforrást elérhessük, tudnunk kell, hogy az melyik könyvtárban alkot bejegyzést. Jelen esetben a HostSystem-et keressük, amely közvetlenül a gyökérben található. A lekérdezéshez az InventoryNavigator osztály példányát kell használni, amely konstruktor paraméterként kapja, hogy melyik könyvtárban kell a kereséseket végezni.

A menedzselt elemek lekérdezéséhez a `searchManagedEntities` eljárását kell meghívnunk a navigátor objektumnak, amely visszatérési értékként egy ManagedEntity tömböt ad. Ez várható volt, hiszen léteznek olyan erőforrások, amelyekből nem csak egyetlen példány van a rendszerben. A `searchManagedEntities` metódus paraméterként a keresett erőforrás vagy szolgáltatás nevét várja, amely jelen esetben a korábban említett HostSystem.

EsxVMM

Az EsxVMM osztálynak a XenVMM-hez hasonlóan szintén nincs szerver oldali megfelelője, példányosításához csupán a Communicator aktuális példányára van szükség, melynek `serviceInstance` mezője segítségével végzi a VirtualMachineManager felületen elérhető funkciókat. A gazdagépen található virtuális gépek lekérdezéséhez a Host interfész implementációjánál bemutatott InventoryNavigator osztály példányát kell használnunk, a keresési terület ismét a gyökér könyvtár lesz, most azonban VirtualMachine típusú menedzselt objektumokat keresünk.

Az ESX felület által elérhetővé tett funkciók többsége aszinkron jellegű, a híváskor egy Task osztály példánnyal térnek vissza, amely segítségével lekérdezhetjük a folyamat aktuális állapotát. Az aszinkronitás következtében valamiféle megoldást kell találnunk, hogy a szinkron jellegű hívásokat tartalmazó architektúránk ne váljon heterogénné. Ennek az egyik módja, hogy saját magunk figyeljük a folyamat állapotát és igény szerint várakozunk, ehhez információt a Task osztály `getTaskInfo` metódusával szerezhetünk. A másik megoldást maga a Task osztály szolgáltatja `waitForTask` hívás segítségével. A menedzsment alkalmazás mindkét megoldást használja.

EsxVM

Az Esx virtuális gépeket reprezentáló osztály megvalósítása a többi megvalósításhoz viszonyítva egyszerűbb. Ez főleg a felhasznált könyvtárcsomagnak köszönhető. Az osztály példányosításához szükségünk van a gazdagépet szimbolizáló HostSystem osztálypéldányra

```
public void start() throws VirtualMachineException {
    try {
        Task task = virtualMachine.powerOnVM_Task(host);
        task.waitForTask();
        TaskInfo taskInfo = task.getTaskInfo();
        if (taskInfo.getState().equals(TaskInfoState.error)) {
            throw new VirtualMachineException(taskInfo.getError().getLocalizedMessage());
        }
        LOGGER.log(Level.INFO, "Virutal machine (" + toString() + ") started");
    } catch (Exception e) {
        LOGGER.log(Level.WARNING, "Virutal machine (" + toString() + ")", e);
        if (e instanceof VirtualMachineException) {
            throw (VirtualMachineException) e;
        } else {
            throw new VirtualMachineException(e);
        }
    }
}
```

4.14. ábra. EsxVM start metódusa

valamint az aktuális virtuális gépet leíró VirtualMachine objektumra. Mindkét osztályt a VMware VI Java API definiálja.

A virtuális gép állapotváltozásait kezdeményező eljárások hasonlóságot mutatnak a többi virtualizációs terméknél tapasztaltakkal, egyetlen lényeges különbség van, az indítás. Míg a többi megoldásnál a virtuális gép és a felhasznált kapcsolat egyértelműen azonosította azt a gazdagépet, amelyiken a virtuális gépet futtatni szándékoztuk, addig az ESX esetében ezt külön paraméterben kell megadni, amely a 4.14 ábrán is jól látható. A többi virtuális gépen értelmezett állapotváltó metódus már csak a virtuális gépre mutató hivatkozást igényéli.

EsxVmMetrics

Sajnálatos mód az ESX által nyújtott távoli menedzsment interfész nem szolgáltat közvetlen adatokat egy virtuális gép aktuális dinamikus jellemzőivel kapcsolatban, csak a statikus adatok direkt lekérdezésére van lehetőség. Ezt a többi megoldáshoz képes könnyen, kommunikáció nélkül megtehetjük, ugyanis a virtuális gépek listázása vagy lekérdezése során visszakapott VirtualMachine típusú VI API-beli osztálypéldányok hordozzák ezeket az információkat.

EsxHostMetrics

A virtuális gépet jellemző EsxVmMetrics osztálynál elmondottak kiterjeszthetőek a vendéggépet jellemző EsxHostMetrics osztályra is. Sajnos ebben az esetben sem szolgáltat a gazdagépet szimbolizáló osztály adatokat a rendszer aktuális állapotáról, amely nagy valószínűsggel annak köszönhető, hogy a gazdagép menedzselésére szolgáló funkciók a SMASH felületben kaptak helyet.

EsxHostCPU

Szerencsére a metrikus adatokkal ellentétben az erőforrás készlet áttekintéséhez jól használható eszközkészletet nyúj a menedzsment felület. A gazdagép fizikai eszközei

közvetlenül lekérdezhetőek a neki megfelelő HostSystem osztálypéldány getHardware metódusa segítségével. A visszakapott HostHardwareInfo típusú objektum tartalmazza minden fontos eszköz adatait, így a processzorét is, amelyekről a `getCpuPkg` függvény segítségével szerezhetünk információt.

4.5.4. KVM

Az egyes termékek által megvalósított és elérhetővé tett menedzsment felületekről szóló részben már szó esett arról, hogy a KVM nem rendelkezik semmiféle önálló távoli menedzsment felülettel. A probléma áthidalásához a libvirt nyújt segítséget. Alkalmazásával lehetőség nyílik a KVM alapú virtualizációs rendszerek távoli menedzsmentjére. A menedzsment alkalmazások fejlesztésének megkönnyítése érdekében a libvirt rendelkezik különböző programnyelvekhez elkészített menedzsment könyvtárakkal, szerencsére ez alól a Java sem kivétel. Sajnálatos mód a helyzet nem annyira egyszerű, mint azt első pillantásra várnánk. Mivel a libvirt működésének feltétele a hozzá kapcsolódó C könyvtár és alkalmazás csomagok megléte, ezért a KVM modul implementálásával elveszítjük a környezetfüggetlen menedzsment alkalmazás fejlesztésének lehetőségét. A Java és C programelemek közötti kapcsolatot a JNA [4] könyvtárcsomag valósítja meg, amely natív osztott könyvtárak elérését teszi lehetővé pusztán Java kód írásával. Ez a csomag a libvirt Java felületének működéséhez elengedhetetlen.

Sajnos a libvirt hiányosságainak köszönhetően néhány funkció nem vagy csak hiányosan került megvalósításra, így a gazdagép processzorainak lekérdezéséről teljes egészében le kellett mondani. Ennek az az oka, hogy a dokumentáció nem tartalmaz a CPU kezelésével kapcsolatos információkat, amely oka lehet a dokumentáltság hiánya, de nagy valószínűséggel az interfész nem támogat ilyen jellegű adatszolgáltatást. Az utóbbit erősíti az a tény, hogy a libvirt-re épülő virt-manager alkalmazás sem mutat részletes információt a processzorokkal kapcsolatban, csupán azok számát közli.

Communicator

Az osztály a korábban ismertetett kommunikációs célú osztályokhoz hasonló feladatot lát el. Mindehhez a Connect osztályú objektumot használja, amelyet a libvirt csomag definiál. A Connect osztálypéldány beállításainak megfelelően a kapcsolat két üzemmódban működhet, csak olvasható és írható-olvasható. A kapcsolat lezárása is ezen az objektumon keresztül történik.

KvmHost

A KVM kiegészítő szolgáltatást nyújtó kernel modul jellegéből adódóan számos alap szolgáltatáshoz nem férhet hozzá, ennek köszönehtően a többi Host implementációhoz képes több implementálatlan funkció maradt az osztályban. Ilyen a gazdagép leállítása funkció, amely eléréséhez nem biztosít felületet a libvirt, és ha biztosítana sem lehetnénk biztosak abban, hogy az lehetséges lenne, melynek oka a Linux jogosultság rendszerében keresendő.

KvmVMM

A VirtualMachineManager implementációja során találkozunk először a fejlesztés során a libvirt domain fogalmával. Mint korábban már említésre került, a libvirt esetében domain néven említjük a gazdagépen futó natív operációs rendszert és a virtuális gépeket is ezzel a névvel illetjük. A virtuális gépek lekérdezéséhez ennek megfelelően a Connect objektum listDomains függvényét kell meghívni, amely a virtuális gépek azonosítóinak listájával tér vissza. Ezek segítségével egyenként kérhetjük el az egyes Domain osztály példányokat, amely szintén a Connect osztály példányán keresztül valósul meg a `domainLookupByID` meghívásával.

Egy domain megsemmisítését az azt reprezentáló objektumon kezdeményezhetjük, a `destroy` függvényhívással.

KvmVM

A libvirt környezetben a virtuális gépek leíró adatainak és referenciájának hordozó osztálya az előző bekezdésben említett Domain osztály. A virtualizációs menedzsment programban ezt a szerepet a KvmVM osztály tölti be. Minden KvmVM objektum hordozza azt a Domain példányt, amelyik által leírt virtuális gépet reprezentálja. A KvmVM osztály tulajdonképpen csomagoló szerepet játszik a Domain osztály körül, minden hívást továbbít feléje.

KvmVmMetrics

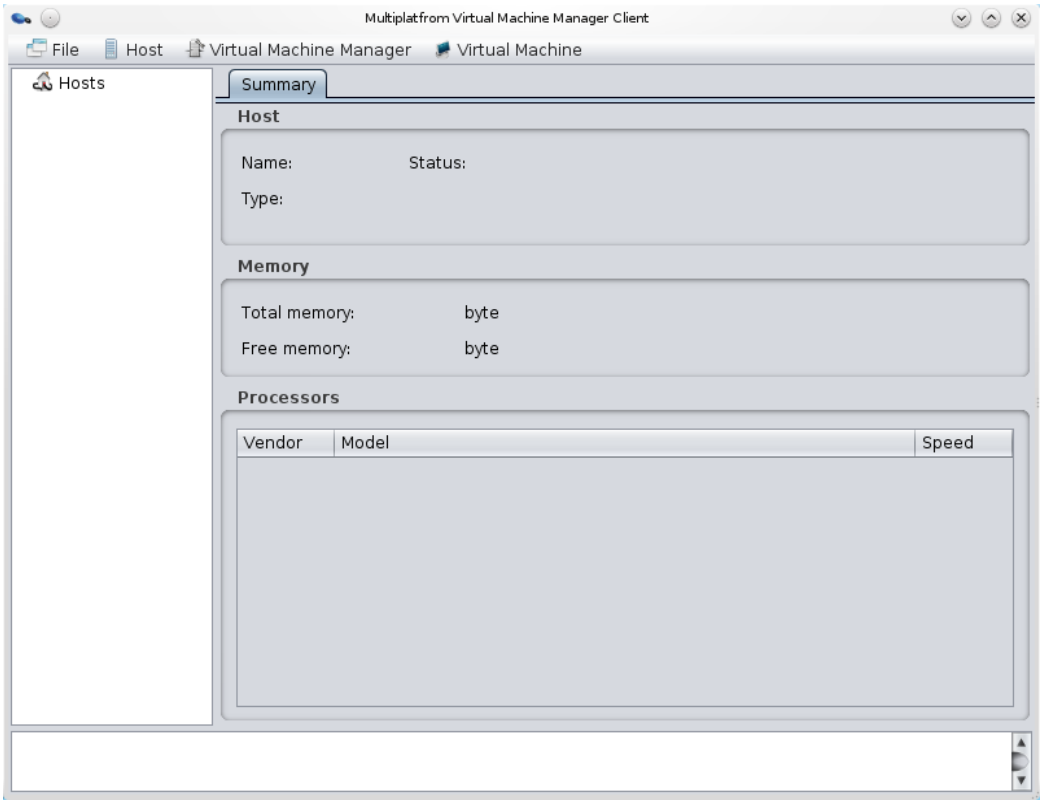
A virtuális gépet jellemző adatokról a Domain osztály példányain keresztül férhetünk hozzá. Így információt szerezhetünk a gép statikus és dinamikus jellemzőiről. Bár a KVM esetében is implementálásra került a VmMetrics interfész, a KvmVmMetrics osztály nem teljes értékű, köszönhetően annak hogy KVM esetében egyelőre bizonyos jellegű adatszolgáltatást a libvirt vagy a libvirt Java csomagjai nem támogatnak, ilyen például a felhasznált vagy a szabad memória nagysága.

KvmHostMetrics

A gazdagépet jellemző KvmHostMetrics osztály esetében is ugyanazt lehet elmondani, mint a virtuális gép esetében, a dinamikus jellemzőkről nem tudunk adatokat beszerezni. Az ilyen jellegű hívások kivételdobást eredményeznek.

4.6. Kliens alkalmazás

Az elkészített menedzsment alkalmazás második egységét a kliens program adja, amely csak a nézeti réteg megvalósítását tartalmazza, az egyes irányítani kívánt erőforrások eléréséhez és vezérléséhez a korábban bemutatott menedzsment osztálykönyvtárat alkalmazza. A kliens kód szinte teljes egésze az általános menedzsment felületre épül, konkrét megvalósítás típusokkal egyedül az egyes virtualizációs megoldást használó gazdagépek felvételekor



4.15. ábra. Kliens alkalmazás

végbemenő konstruktor híváskor találkozhatunk. Ennek köszönhetően a felületet minimális változtatásokkal képessé tehetjük más virtualizációs megoldások kezelésére is.

A kliens program a könnyű kezelhetőség érdekében egy grafikus felülttel rendelkező Swing-es Java asztali alkalmazás formájában látott napvilágot. A felület felépítését tekintve, ahogy a 4.15 ábrán is látható négy részre osztható, a baloldali faszerkezetre, amely a gazdagépek és a rajtuk futó virtuális gépek hierarchikus megjelenítését hivatott ellátni, a terület nagy részét elfoglaló információs panelre, amely a gazdagép vagy a virtuális gép adatairól informál, a felső menü sorra, illetve az alsó nyomkövető területre.

Minden osztályhoz kötött funkció felugró menük segítségével is elérhető, amely az adott operációs rendszeren hagyományosan alkalmazott felugró menü eseményre hozható elő. Ez tipikusan a jobb egér esemény. A kódduplikáció elkerülése és a könnyebb kezelhetőség érdekében a kliens által végrehajtható funkciók az ActionType felsorolás típusba kerültek összefogásra, illetve a megvalósításukra is egyetlen metódusban került sor, amely paraméterében a végrehajtandó akció típusát várja. Ennek köszönhetően könnyen hivatkozhatunk ugyanarra a funkcióra a hagyományos és a felugró menük eseménykezelő osztályaiban.

4.6.1. Listázó

Mivel a gazdagépek és a rajtuk futó virtuális gépek között nincs semmiféle konkrét kötés, ezért a megvalósított menedzsment könyvtár felhasználójának döntésén múlik, hogy azokat hogyan rendeli össze. A kliens alkalmazás esetében ezt a feladatot a listázó és a

mögötte található modell látja el. A Listázó nem más, mint egy JTree osztálypéldány, amely lehetővé teszi az adatok hierarchikus szerkezetbe rendezését. A hierarchia legfelső szintjén a gyökér elem áll, amely egy statikus, Hosts névvel jelölt csomópont. A gyökér gyermekeit a felvett gazdagépek fogják képezni, unokáit pedig a gazdagépekhez csatolt virtuális gépek. Ebből a felépítésből adódóan ha egy virtuális gépen olyan tevékenységet hajtunk végre, amelyhez a hozzá tartozó VirtualMachineManager példányra is szükség van, azt két lépésben beszerezhetjük. Első lépésben megkeressük a kijelölt virtuális csomópont szülőjét, amely nyilván egy Host objektum lesz, majd ettől az objektumtól elkérjük a keresett menedzser példányt. Mindezen folyamatok gyors megvalósításához szükség volt egy saját TreeModel implementációra, amely tárolja a Host és a VirtualMachine osztályok példányait. Míg az első egy egyszerű lista, addig a második egy muti map szerű szerkezet, vagyis olyan HashMap, amelynek kulcsaként Host objektumot használunk, értékként pedig egy VirtualMachine osztályú objektumokból álló listát.

A gazadgépek és a virtuális gépek állapotának jelzése grafikus úton történik, ehhez a használt JTree objektum cella renderelő objektumát kellett kicserélni egy saját, a DefaultTreeCellRenderer osztályból leszármaztatott osztály példányára, amely a az aktuális objektumok tulajdonságai alapján alkotja meg a kirajzolandó komponenst.

4.6.2. Részletező

A faszerkezetben kijelölt csomópont típusának megfelelően a részletező panel az aktuális gazdagép vagy virtuális gép tulajdonságait mutatja. Ehhez egyrészt egy figyelő objektumot kellett hozzáadni a fához, amely a fán kijelölt csomópont változása esetén biztosítja a kívánt adatok megjelenítését, másrészt pedig a csomópont típusának megfelelő oldalt kell mutatni. Mivel az egyes gazdagép típusok nem támogatnak minden műveletet, ezért az adatok lekérdezésénél és megjelenítésénél ez is figyelembe kell venni.

4.6.3. Nyomkövető és menüsor

A nyomkövető panel segítségével kísérhetjük figyelemmel az aktuális történéseket, illetve a historikus adatokat. A működés során mined objektum a fontosabb eseményekről log bejegyzést készít, ezek a bejegyzések kerülnek megjelenítésre a felületen.

A menüsor a korábban említett csoportba zárt funkciókat teszi elérhetővé. Az egyes menüpontok az aktuálisan kijelölt csomópont típusától és állapotától függően kerülnek elérhető vagy tiltott állapotba. Az engedélyezettséget befolyásolja még a csomópont által támogatott akciók halmaza is, az olyan funkciók amelyet az adott Host vagy VirtualMachineManager esetlegesen VirtualMachine példány nem támogat azok mindig tiltott állapotban maradnak.

Mivel távoli erőforrásokon hajtunk végre műveleteket a kliens alkalmazást fel kellett készíteni az ebből fakadó problémák kezelésére. A leggyakrabban jelentkező probléma a kliens alkalmazások egyszálúságából adódó időleges fagyás, amelyet a hosszú kommunikációs folyamatok és a távoli végrehajtás miatt bekövetkező várakozás okoz. Ennek a legjobb ellenszere a többszálúság, mivel a megvalósított menedzsment könyvtár

```
Object node = mp.getSelectedNode();
if (node instanceof Host) {
    final Host host = (Host) node;
    if (host.getStatus().equals(HostStatus.DISCONNECTED)) {
        Authenticator authenticator = new Authenticator(this);
        final String userName = authenticator.getUsername();
        final char[] password = authenticator.getPassword();
        if (userName != null) {
            Thread job = new Thread(new Runnable() {

                public void run() {
                    try {
                        host.connect(userName, new String(password));
                        VirtualMachineManager vmm = host.getVMM();
                        mp.addMachines(host, vmm.listMachines());
                    } catch (CommunicationException ce) {
                        new ErrorMessageDisplay(Main.this, ce);
                    }
                }
            });
            job.start();
        }
    }
}
```

4.16. ábra. Gazdagép csatlakoztatása

tervezésénél és megvalósításánál nem volt szempont a többszálú végrehajthatóság, ezért a kliens elkészítése során kellett azt implementálni. Minden funkció használata során egy új szál keletkezik, amely körbeveszi a meghívott szolgáltatást, így az időigényes folyamatok szeparálódnak a felület futtatásért felelős szál példánytól, erre látható példa a 4.16 ábrán.

A kliens alkalmazás néhány esetben fontos információkat közöl a felhasználóval, vagy éppen igényel tőle. Ezek az üzenetek vagy űrlapok felugró ablakok formájában jelennek meg, amelyek modálisak a szülő ablakra. Ilyen ablak az autentikációt megvalósító felület, illetve a kivétel részletező panel és a gazdagép választó.

Hogy ne kelljen az alkalmazás minden egyes indításakor felvenni a menedzselni kívánt gazdagépeket a kliens kilépés előtt eltárolja a Host objektum példányok legyártásához szükséges adatokat, illetve indításkor beolvassa azokat és segítségükkel példányosítja a megfelelő menedzsment osztályokat. Az adatok a felhasználó home könyvtárában létrehozott .virtclient könyvtárban kerülnek tárolásra. Minden géphez tartozik egy xml állomány, amelynek neve a host nevével egyezik meg és annak típusát és URL-jét tartalmazza. A mentési és beolvasási folyamatokhoz a JAXB API került felhasználásra. Mivel az egyes Host implementációk számos olyan adatot is tartalmaznak, amelyek nem szükségesek magának az osztálynak a példányosításához vagy nem használhatók xml dokumentum létrehozására, ezért született egy XmlHost nevű osztály, amely csak a szükséges adatokat tartalmazza. Az XmlHost példányok adattartalma alapján kerülnek megíhvásra a megfelelő osztályok konstruktorai. Az xml állományokkal végzett feladatokért a HostListHandler osztály statikus `save`, `load` és `removeHost` metódusai felelősek. A 4.17 ábrán a gazdagépeket reprezentáló Host objektumok betöltéséért felelős load metódus látható.

```
public static List<Host> load() {
    List<Host> hosts = new ArrayList<Host>();
    try {
        String home = System.getProperty("user.home");
        String storage = home + File.separator + ".virtclient";
        File workDir = new File(storage);
        workDir.mkdir();
        JAXBContext context = JAXBContext.newInstance(XmlHost.class);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        File[] files = workDir.listFiles();
        for (File file : files) {
            if (file.getName().endsWith(".xml")) {
                XmlHost xh = (XmlHost) unmarshaller.unmarshal(file);
                if (xh != null) {
                    hosts.add(xh.getHost());
                }
            }
        }
    } catch (JAXBException jaxbe) {
        jaxbe.printStackTrace();
    }
    return hosts;
}
```

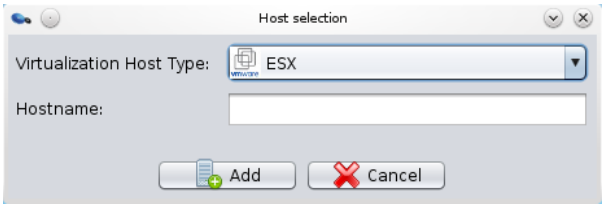
4.17. ábra. Host példányok beolvasása

4.6.4. Futtatási előkészületek

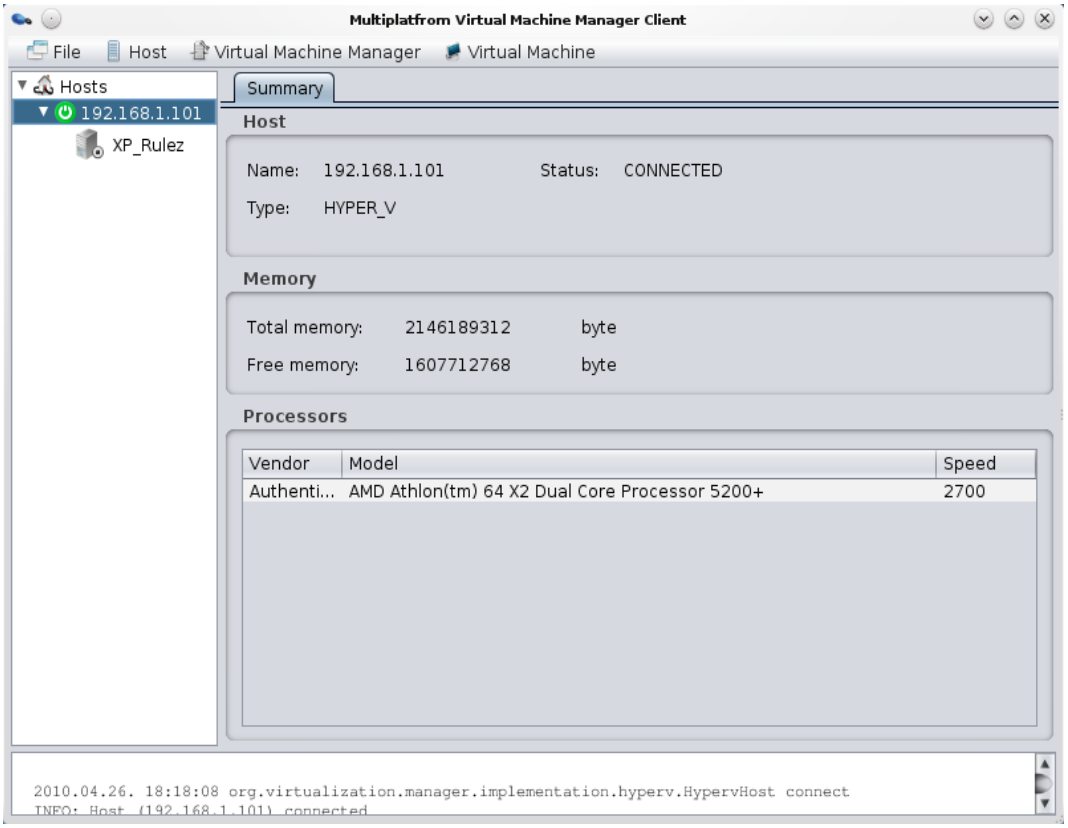
Mivel az elkészített program nyelve a Java, szükségünk lesz valamilyen, annak futtatására alkalmas Java Virtual Machine implementációra. A fejlesztés során Sun JRE 1.6.18 biztosította a futtatást, így ennek használata ajánlott. Továbbiakban megemlítendő, hogy a munka Linux platformon történt, bár a Java platformfüggetlen programozási nyelv szép számmal előfordulnak ennek ellentmondó jelenségek, így érdemes a futtatáshoz is azt használni. Az elkészített menedzsment alkalmazás futtatása előtt bizonyos beállításokat kell eszközölnünk az irányítani kívánt erőforrásokon és esetlegesen az alkalmazás futtatására használt gépen. Elsősorban biztosítanunk kell, hogy a felhasznált könyvtárcsomagok elérhetőek legyenek, Java terminológiával élve azt mondhatjuk, hogy rajta legyenek a class path-on. Ezt a NetBeans a végtermék fordítása közben biztosítja a megfelelő könyvtár struktúrával és manifest állomány Class-Path bejegyzésével. Másodsorban pedig a Hyper-V-t alkalmazó gazdagépeken kell a Windows Remote Management Framework-öt megfelelően beállítani. Mivel a megvalósított menedzsment komponens Basic Authentication-t alkalmaz, így azt engedélyezni kell, illetve a működés biztosítása érdekében a többi autentikációs módot le kell tiltani. Ez indokolatlannak tűnhet, azonban az elkészített menedzsment alkalmazás hiába közli a szerver féllel, hogy Basic Authentication segítségével akarja magát azonosítani, és azon hiába van az engedélyezve, ha más mód is aktív, akkor a szerver fél azokat fogja kérni, így az autentikáció mindig meghiúsul. WinRM 2.0 esetében a menedzsment felület csatolási pontja http esetében a 5985 port, míg https esetében 5986 port, mivel az implementált alkalmazás a WinRM 1.0-nak megfelelően a 80-as http portot használja a kommunikációhoz így be kell állítani, hogy a menedzsment keretrendszer ott is fogadjon kéréseket. A fejlesztés és tesztelés során használt WinRM konfigurációt a 4.18 ábra szemlélteti. Ahhoz, hogy KVM alapú erőforrásokat is elérhessünk rendelkeznünk kell a libvirt telepített példányával.

```
Config
    MaxEnvelopeSizekb = 150
    MaxTimeoutms = 60000
    MaxBatchItems = 20
    MaxProviderRequests = 25
    Client
        NetworkDelayms = 5000
        URLPrefix = wsman
        AllowUnencrypted = true
        Auth
            Basic = true
            Digest = false
            Kerberos = false
            Negotiate = false
            Certificate = false
        DefaultPorts
            HTTP = 80
            HTTPS = 443
        TrustedHosts = localhost
    Service
        RootSDDL = O:NSG:BAD:P(A;;GA;;;BA)(A;;GR;;;ER)S:P(AU;FA;GA;;;WD)(AU;SA;G
WGX;;;WD)
        MaxConcurrentOperations = 100
        EnumerationTimeoutms = 60000
        MaxConnections = 5
        AllowUnencrypted = true
        Auth
            Basic = true
            Kerberos = false
            Negotiate = false
            Certificate = false
        DefaultPorts
            HTTP = 80
            HTTPS = 443
        IPv4Filter = *
        IPv6Filter = *
    Winrs
        AllowRemoteShellAccess = true
        IdleTimeout = 900000
        MaxConcurrentUsers = 5
        MaxShellRunTime = 2147483647
        MaxProcessesPerShell = 5
        MaxMemoryPerShellMB = 80
        MaxShellsPerUser = 2
```

4.18. ábra. WinRM konfiguráció



4.19. ábra. Gazdagép felvétele



4.20. ábra. Csatlakoztatott gazdagép

4.6.5. Kliens használata

A kliens alkalmazás első indítását követően a 4.15 ábrán látható felület fogad bennünket. Új gazdagép felvételéhez a Host menü Add menüpontját kell kiválasztani, amelynek hatására a gazdagép adatait rögzítő felugró ablak jelenik meg, mint az a 4.19 ábrán is látható. Az elérni kívánt erőforrás nevének vagy ip címének megadását követően a gazdagép felvehető a menedzselni kívánt gépek listájába, amely rögtön meg is jelenik a listázóban.

Az imént hozzáadott erőforrás kijelölését követően a részletező felületen megjelennek az azt leíró adatok, melyek csatlakozás nélkül elérhetők, illetve engedélyezetté válnak a Host menü Remove és Connect pontjai. A Connect menüpont használatát követően ismét egy felugró ablakkal találkozunk, amely a felhasználót azonosító adatok felvételére szolgál. A csatlakozást követően az erőforrás csomópont ismételt kiválasztásával további mezők kerülnek kitöltésre a felületen, ezt mutatja a 4.20 ábra.

Ezt követően, ha kibontjuk a gazdagépet jelképező csomópontot megjelennek a rajta

található virtuális gépek. Az egyes virtuális gépeket kijelölve a jobboldali részletező panelon megtekinthetjük a róluk kapott információkat, illetve a menüpontok vagy a jobb egér menü segítségével az aktuális virtuális gép által támogatott és az állapota által lehetővé tett funkciók végrehajtását kezdeményezhetjük.

5. fejezet

Tesztelés

Az elkészített alkalmazás helyességének ellenőrzéséhez tesztekre van szükségünk, a tesztelés történhet programozott úton és manuálisan is. Mivel az alkalmazás munkájának jelentős részében a távoli erőforrásokkal kommunikál vagy az általuk adott válaszokat dolgozza fel, ezért önmagában nem tudunk teljes körű tesztelést végezni, így mindenképpen be kell vonni azokat a tesztbe. A tesztelés másik hátráltató tényezője, hogy a hibák előfordulása a legtöbb esetben az aktuális válaszok és állapotok következménye, a távoli erőforrásokon végzett feladatok munkatárgyai az általuk korábban adott válaszokba burkolt erőforrások és szolgáltatások. A megvalósított tesztesetek az egyes komponensek közötti helyes együttműködésre valamit a menedzselt erőforrással végzett információ cserére koncentrálnak. Az alkalmazott struktúrák és adatmodell természetéből adódóan az ennél finomabb felbontású tesztesetek kis mértékben járulnának hozzá a program helyes működésének elbírálásához.

A grafikus kliens vizsgálata teljes egészében manuális teszteken alapul. Bár léteznek grafikus kliensek automatizált tesztelésére szolgáló eszközök, amelyek között találunk ingyenesen alkalmazhatót is, jelen esetben azok bonyolultsága miatt eltekintünk a használatuktól és a könnyebben kiértékelhető és megvalósítható kézi tesztelésre hagyatkozunk. A programozott tesztek végrehajtásában a JUnit [5] nevű teszt keretrendszer nyújt segítséget. Mint ahogy a neve is sugallja unit tesztek létrehozását támogatja. Használata nem igényel különösebb konfigurációs lépéseket, egyszerűen, annotációk segítségével definiálhatunk teszteseteket és azokat megelőző inicializáló vagy követő rendrakó metódusokat.

5.1. Automatizált tesztelés

5.1.1. Xen

CommunicatorTest

A Communicator osztály tesztelésének feladata, hogy megállapítsa a kommunikációs folyamat kezdésének, folytatásának és befejezésének helyességét.

- **testNewInstanceNotExist (X1):** a teszteset feladata, hogy megvizsgálja

milyen viselkedést mutat a Communicator osztály példányosítása közben, ha nemlétező erőforrás azonosítóval paraméterezzük fel. Az elvárt viselkedés egy CommunicationException típusú kivétel dobása lenne.

- **testNewInstanceExists (X2):** ez a teszteset az előző kiegészítő esete, amely a helyes adatok esetén mutatott viselkedést és visszaadott eredmény vizsgálja, a helyesség feltétele, hogy a Communicator osztály példánya egy érvényes objektum referencia legyen, ne pedig null pointer.

- **testSendRequest (X3):** a harmadik teszteset célja, hogy megvizsgáljuk a `sendRequest` metódus helyességét. A küldendő üzenet tetszőleges lehet, akár érvénytelen is, ugyanis mint korábban említésre került a tesztesetek többségében nem tartalmi helyességet teszteljük, hanem a kommunikációs folyamatok helyességét. Ebben az esetben pedig pontosan erre van szükségünk, ugyanis a működés helyességének bizonyítéka a null-tól különböző visszatérési érték.

- **testReleaseSession (X4):** a következő teszteset a fenntartott munkamenet elengedését teszteli, sajnos ebben az esetben nem hagyatkozhatunk semmiféle összehasonlítási értékre, mivel azok a mezők, amelyeken ezt végezhetnénk a Communicator osztály privát mezői. Így helyesnek tekintjük a futást, ha az kivétel kiváltása nélkül történik.

XenHostTest

A XenHost osztály metódusainak tesztelése azon az előzetes feltevésen alapul, hogy minden olyan metódus, amely egy másik, a menedzsment alkalmazás által definiált osztály példányával tér vissza, hiba esetén vagy kivételt dob vagy null értékkel tér vissza.

- **testConnect (X5):** a XenHost `connect` hívása sikeres, ha nem lép fel kivétel, és az objektum `getStatus` metódusa `HostStatus.CONNECTED` értékkel tér vissza.

- **testDisconnect (X6):** az előzővel hasonló teszteset, csak ennek futása akkor tekinthető sikeresnek, ha a `disconnect` hívás visszatérési értéke `HostStatus.DISCONNECTED`.

- **testGetVMM (X7), testGetCPUs (X8), testGetMetrics (X9):** ezek a tesztesetek a XenHostTest-nél általánosságban elmondott helyes futás feltételén alapulnak, vagyis visszatérési értékük nem lehet null referencia és nem lehet üres gyűjtemény sem. Rendre a VirtualMachineManager példány helyes megkonstruálását, a processzor adatok lekérdezését és a Host példány által reprezentált fizikai gép tulajdonságait lekérdező eljárások helyességét tesztelik.

XenVMMTest

A XenVMM példányosításának tesztjét már az előző unit teszt keretében, a testGetVMM eset keretében megvalósítottuk ezért attól eltekintünk.

- **testListMachines (X10):** ezzel a tesztesettel tulajdonképpen két metódust helyes működését ellenőrizhetjük, elsőként és direkt módon a `listMachines` metódust, amely első lépésként lekérdezi a virtuális gépek felsorolását reprezentáló referencia tömb példányt. Az egyes virtuális gép példányok lekérdezéséért a `getMachine` metódus felelős, így a teszteset sikeres volta esetén ennek helyessége is bizonyított. Az utóbbi metódus önálló tesztelése azért került mellőzésre, mert egyetlen virtuális gép azonosító példány nem tudunk lekérdezni, csak az összeset, így értelmét veszti ez a teszteset.

- **testGetDefaultStoragePool (X11):** a teszt az alapértelmezett tároló erőforrás tömböt lekérdező eljárást teszteli, sikerességének feltétele a függvény null-tól különböző visszatérési értéke.

5.1.2. Hyper-V

CommunicatorTest

- **testSendRequest (H1):** mivel a Hyper-V esetében nem történik valós kapcsolat kiépítés, és minden kérés során azonosítania kell magát a kommunikációt kezdeményező félnek, ezért elég csak az üzenet küldés helyességét megvizsgálni. A teszteléshez tetszőleges, szintaktikailag érvényes üzenetet használhatunk, a tesztet sikeresnek tekintjük, ha a válasz üzenet null-tól különböző objektum, annak tartalma a teszt kimenetele szempontjából közömbös.

HypervHostTest

- **testConnect (H2):** a Hyper-V esetében a `connect` metódus tesztje nem csak a Host implementáció vizsgálatára korlátozódik, ugyanis a VirtualMachineManager interfészt implementáló HypervVMM osztály is ilyenkor példányosodik, tehát annak tesztjét is jelenti egyben.

- **testDisconnect (H3):** bár a teszteset belekerül az osztályba igazi jelentőséggel nem bír, ugyanis a WS-Management működéséből adódóan nincs munkamenet alapú kapcsolat, így nincs is honnan lecsatlakozni, bár a kliensen tárolt Host példány állapotát változtatni kell és a megfelelő példány mezőket felszabadítani.

- **testGetVMM (H4), testGetCPUs (H5), testGetMetrics (H6):** a Xen esetén elmondottakkal megegyező tesztesetek.

HypervVMMTest

- **testGetInstance (H7):** bár a VirtualMachineManager példányosításának tesztelését a HypervHostTest is tartalmazza indirekt, ez a teszteset direkt módon próbálja annak sikerességét megállapítani. Ennek feltétele, hogy a példányosító metódus null-tól különböző eredménnyel térjen vissza.

- **testListMachines (H8):** szintén a Xen esetében bemutatott testListMachines tesztesettel analóg.

5.1.3. ESX

CommunicatorTest

- **connectToNonExist (E1):** a teszt a nem létező erőforrás esetét vizsgálja, a korábbiaknak megfelelően itt is CommunicationException kivételt várunk.
- **testGetServiceInstance (E2):** a csatlakozás sikeres lefutását kétféle módon tesztelhetjük le, vagy készítünk egy tesztesetet, amely csak a Communicator konstruktorának lefutását tartalmazza, vagy azt egy inicializáló eljárásba tesszük és a tesztesetben pedig a ServiceInstance osztálypéldány értékét vizsgáljuk, amely csak sikeres konstruktor lefutás esetén példányosodhat, ebben az esetben az utóbbi került implementálásra.
- **testLogout (E3):** a kijelentkezés tesztesete akkor tekinthető sikeresnek, ha nem lép fel kivétel, és az előbbiekben említett ServiceInstance példány egy null referencia.

EsxHostTest

- **testConnect (E4):** a Host-hoz történő csatlakozás tesztelése akkor tekinthető sikeresnek, ha futása közben nem lép fel kivétel és a futás eredményeként a Host `HostStatus.CONNECTED` státuszú lesz.
- **testDisconnect (E5):** a kapcsolat bontásának tesztelése a Xen-nél látottakkal analóg módon történik. A teszt sikeres, ha a Host állapota a lefutást követően `HostStatus.DISCONNECTED`.
- **testGetVMM (E6), testGetCPUs (E7), testGetMetrics (E8):** teljes egészében azonos a Xen-nél és Hyper-V-nél látottakkal.

EsxVMMTest

- **testListMachines (E9):** az EsxVMMTest unit teszt egyetlen tesztesetet definiál, ez a testListMachines, amely a virtuális gépek lekérdezésének helyességét vizsgálja, a Xen-es párjával ellentétben ez a teszteset csak a csoportos lekérdezéshez szolgáltat ellenőrzési lehetőséget, mivel a felhasznált metódusok az összes virtuális gép példánnyal térnek vissza, és nem kell őket egyenként lekérdezni.

5.1.4. KVM

Az elkészített alkalmazás KVM-hez implementált osztályaihoz nem születtek tesztek, ennek főképp az az oka, hogy az megvalósított szolgáltatások a hiányos Communicator osztályból adódó olvasásra korlátozódó hozzáférés miatt nem futtathatóak. Azok a szolgáltatások,

amelyek ezek mellet a feltételek mellet is igénybe vehetők a manuális tesztek keretében lettek validálva.

5.2. Manuális tesztelés

A fejezet korábbi bekezdéseiben részletezett okok miatt az elkészített alkalmazás egyes részeinek ellenőrzéséhez, mint például a grafikus felület és az összetettebb erőforrás kezelési folyamatok, nem készültek automatizálható tesztek. Annak érdekében, hogy ezek hibamentességéről is meg tudjunk győződni kézzel végrehajtható tesztesetek születtek, amelyek megpróbálják lefedni a korábban nem vizsgált területeket. Minden teszteset középpontjában vagy egy Host vagy egy VirtualMachine példány áll, melyet felhasználva vizsgáljuk meg, hogy az elvárásnak megfelelő történések mennek végbe a menedzselt távoli erőforrásokon és a grafikus felületen egyaránt.

5.2.1. Gazdagép kezelése

Az első tesztcsoport eseteiben az gazdagépekkel kapcsolatos grafikus és egyéb háttér komponensek működése került megvizsgálásra.

Host felvétel (M1)

A teszt célja, hogy megállapítsuk a gazdagép felvételének hibamentességét. Az elvárt viselkedés szerint a Host példány felvételét követően a baloldali erőforrás fában megjelenik egy új csomópont, amelyet kiválasztva a részletező panelban megjelenik annak neve, típusa és állapota, amelynek DISCONNECTED értékűnek kell lennie. A teszteset második lépéseként a felvett csomópontok maradandóságát kell elvégezni, ehhez az alkalmazás bezárása és újraindítása szükséges, amennyiben a két futtatás során az erőforrásfában szereplő gazdagépek száma megegyezik a teszt ezen részét sikeresnek tekinthetjük. Ezt a tesztet minden egyes menedzselt erőforrás típusra (Xen, Hyper-V, ESX, KVM) el kell végezni.

Host eltávolítása (M2)

Az előző tesztesettel ellentétes irányú folyamat a gazdagép eltávolításának tesztelése, ehhez az eltávolítás után meg kell vizsgálni, hogy az erőforrás fából valóban eltűnt-e a kívánt bejegyzés valamint, hogy a felhasználó home könyvtárából is eltávolításra került-e a megfelelő xml állomány, amely a gazdagép adatait hivatott tárolni.

Csatlakozás a Host-hoz (M3)

A teszt futtatásának előfeltétele, hogy rendelkezzünk már felvett gazdagéppel, amely elérhető állapotban van, illetve hogy rendelkezzünk hozzáférési jogokkal a távoli erőforráshoz. A teszt futásának feladata, hogy a szolgáltatott eredmények alapján megállapítsuk, hogy a menedzselt erőforráson lévő virtuális gépek megfelelő mennyiségben, formában és állapotban jelennek meg a felületen. Az ellenőrzés alapjául szolgáló referencia

adatok beszerzéséhez szükség van minden egyes virtualizációs megoldáshoz valamiféle menedzsment eszközre, amellyel lekérdezhetjük a megfelelő adatokat.

Kapcsolat bontása (M4)

A teszt feladata, hogy megállapítsa a kapcsolat bontása során szükséges módosítások, mint az erőforrásfában a gazdagépnek megfelelő csomópont gyermekeinek eltávolítás és a csomópont állapotának megváltozása, végbe mennek-e.

Host-hoz kapcsolódó funkciók elérhetőségének tesztelése (M5)

A gazdagépet reprezentáló Host példány állapotának megfelelően az egyes funkciók elérhetősége változik, más tehetünk egy lecsatlakoztatott és mást egy kapcsolódott gazdagéppel. A teszteset feladata annak megállapítása, hogy a funkciók mindig csak a megengedett állapotban vehetők igénybe, illetve, hogy akkor biztosan használhatóak-e.

5.2.2. Virtuális gép kezelése

A JUnit tesztek esetében nem nyílt lehetőség a virtuális gépek kezeléséhez implementált funkciók helyességének megállapítására. Ezt a hiányosságot próbálják pótolni az alábbi tesztesetek.

A virtuális gép állapotát megváltoztató funkciók tesztelése (M6)

A teszt célja annak megállapítása, hogy a virtuális gépen végrehajtott funkciók valóban eredményesek. Ennek eldöntéséhez szükségesek a korábbi tesztekben validációs célra már használt menedzsment eszközök. Ezek segítségével megbizonyosodhatunk, hogy a kiválasztott virtuális gép a használt funkció hatására a menedzselt rendszeren a megfelelő állapotba került-e.

A virtuális géphez kapcsolódó funkciók elérhetőségének tesztelése (M7)

A gazdagép esetéhez hasonlóan a virtuális gépeknél is meg kell vizsgálni, hogy a virtuális gép típusának és állapotának megfelelő funkciók vehetőek-e igénybe. Mint látszik ehhez, olyan csatlakoztatott gazdagépekre van szükség, amelyek rendelkeznek virtuális gépekkel is.

Virtuális gép törlése (M8)

A KVM kivételével valamennyi termékekhez implementált menedzsment felület esetén lehetséges a virtuális gép törlése, a teszt célja megvizsgálni, hogy a kijelölt erőforrás valóban megsemmisítésre kerül. A tesztet valamennyi VirtualMachine típusra el kell végezni. Kimenetele sikeresnek tekinthető, ha minden virtualizációs megoldás esetében a neki megfelelő virtuális gép állapota megfelel az ott definiált törölt állapotnak.

Virtuális gép adatainak helyes megjelenítése (M9)

A teszt célja, hogy megvizsgálja miden virtuális gép típus esetében, hogy a részletező panelen megjelennek-e az osztálya által engedélyezett és biztosított leíró adatok. A tesztet sikeresnek tekintjük, ha minden esetben megjelennek az elvárt adatok.

5.3. Tesztkörnyezet

A tesztek végrehajtása során minden esetben Sun JRE 1.6.18-as Java futtató környezet került felhasználásra. A grafikus kliens tesztelése során Kubuntu 9.10-es verziójú operációs rendszeren futott, valamint Windows 7 Professional-on is kipróbálásra került. A Hyper-V és a Xen tesztek esetében a menedzselt virtualizációs rendszer és a menedzselő kliens alkalmazás külön-külön fizikai erőforrásokon futott. Az ESXi esetében hardver követelmények okozta kényszerek miatt a virtualizációs erőforrás szerepét VMware Workstation keretei között üzemeltetett virtuális gép töltötte be, amely a klienssel azonos fizikai erőforráson helyezkedett el.

- Hyper-V: a Microsoft termék teszteléséhez Windows Server 2008 R1 került felhasználásra, melyen a korábban már részletezett beállítások voltak érvényben.
- Xen: Xen platform teszteseteinek futtatásához Xen Cloud Platform 0.1.1-el ellátott rendszer került felhasználásra, amely 3.4.2 verzió számú hypervisort tartalmaz.
- ESX: a tesztesetek futtatása során a távoli, menedzselni kívánt erőforrásként ESXi Server 3.5 szolgált.
- KVM: a KVM teszteléséhez annak 1.8-as verziója futott Kubuntu 9.10-es Linux operációs rendszeren.

5.4. Teszteredmények

Teszt azonosító	Eredmény	Teszt azonosító	Eredmény
H1	siker	H2	siker
H3	siker	H4	siker
H5	siker	H6	siker
H7	siker	H8	siker
X1	siker	X2	siker
X3	siker	X4	siker
X5	siker	X6	siker
X7	siker	X8	siker
X9	siker	X10	siker
X11	siker	E1	siker
E2	siker	E3	siker
E4	siker	E5	siker
E6	siker	E7	siker
E8	siker	E9	siker
M1	siker	M2	siker
M3	siker	M4	siker
M5	hiba	M6	siker
M7	hiba	M8	siker
M9	hiba		

Mint látható a unit tesztek futásuk során az elvárt eredményt produkálták, azonban a kézi tesztesetek végrehajtása során fény derült néhány hibára és érdekességre. Elsőként az ESX-hez tartozó virtuális gépek esetében nem jelentek meg a részletező felületen a lekérdezett adatok, ez a hiba a felfedezést követően javításra került. A másik két rendellenesség nem tekinthető teljes egészében a megvalósított program hibájának, sem a Host sem pedig a VirtualMachine példányokhoz kapcsolt funkciókat tartalmazó felugró menü nem jelenik meg a jobb egér esemény hatására Windows operációs rendsteren. Ez utóbbi valószínűsíthetően a Windows-os JRE hiányosságából fakad, amelynek javítása a MouseEvent osztály isPopupTrigger metódusának helyes implementálását igényelné.

6. fejezet

Értékelés és összefoglalás

6.1. Értékelés

A tervezés során egyik fő célkitűzés volt, olyan menedzsment alkalmazás elkészítése, amely az újrahasznosítható komponensekre helyezi a hangsúlyt. Ez részben megvalósult, azonban ezzel nem lehetünk teljes egészében elégedettek, ugyanis ez főleg a tervezési munkának tudható be és nem a felhasznált menedzsment felületeknek. Így az elérni kívánt cél mindössze a munka során megalkotott saját absztrakt felületre és az erre épülő kliens alkalmazásra korlátozódik. Az egyes virtualizációs termékekhez nyújtott menedzsment felületek olyan eltérő technológiai és strukturális háttérrel rendelkeznek, amelyek nem tették lehetővé az alacsonyabb szinteken elhelyezkedő újrahasznosítható komponensek írását, így minden termékhez azokat külön külön implementálni kellett. Az egyes termékek esetében ez a munka eltérő mennyiségű befektetett energiát igényelt, amely a dokumentáltság és a terméktámogatás mértékével fordítottan arányos. A következőkben egyéb tényezők mellett ezek lesznek a fő értékelési szempontok, amelyek alapján osztályozhatjuk az egyes virtualizációs megoldásokat és az általuk biztosított távoli interfészeket. Az egyes értékelésekben közrejátszhat a Java nyelvű implementáció során szerzett szubjektív tapasztalat is, amely nem feltétlenül esik egybe a tárgyilagos megítéléssel vagy más programozási nyelvet használó fejlesztő véleményével.

6.1.1. Xen

A Xen esetében igénybe vehető Xen API használata a megismert felületek közül talán a legegyszerűbb. Jól strukturált osztályhierarchiát definiál, amely segítségével célirányos lekérdezéseket és beavatkozásokat tudunk eszközölni. A kommunikációhoz alkalmazott XML-RPC is nagyságrendekkel egyszerűbben használható, mint a többi esetben használt protokollok. Egyetlen negatívumként talán azt tudnánk felhozni, hogy egy kicsit koros és túlhaladott szabványról van szó, amely önmagában nem alkalmas önleíró menedzsment struktúrák továbbítására, az adat jelentését mindig csak a felhasználás helyén nyeri el. Dokumentáltságát tekintve a Xen messze felülmúlja társait, nem csak egy jól összeállított referencia kézikönyv segíti a fejlesztés, hanem számos példaalkalmazást is találhatunk a világhálón. A mintapéldákhoz használt programozási nyelvek széles skálán mozognak,

amelyben szinte mindenki megtalálja a neki megfelelőt. Talán mindezek is közrejátszanak abban, hogy a ma fellelhető, több virtualizációs terméket támogató menedzsment eszközök szinte mindegyike rendelkezik Xen menedzsment képességekkel is.

6.1.2. Hyper-V

A Microsoft termékéhez használható menedzsment felületen érezhető, hogy maximálisan törekedtek az új igényeket kielégítő szolgáltatások meglévő szabványos felületbe történő integrálására . Ennek köszönhetően a négy vizsgált megoldás közül ez az egyetlen, amelyről elmondható, hogy valóban szabványos úton teszi lehetővé a gazdagép nem csak fizikai, hanem a virtualizációval kapcsolatos erőforrásainak távolról történő irányítását. Ez a szabványos út pedig nem más, mint a Windows Remote Management Framework és annak alapját képező CIM szabványon alapuló WMI osztályok. Ha az elkészített menedzsment alkalmazás Hyper-V-hez kapcsolódó funkcióinak megvalósítása során az ős CIM osztályok kerültek volna felhasználásra a származtatott WMI-beliekkel szemben egy általánosabb körben használható alkalmazást kaptunk volna. Azonban számos lényeges dolgot csak a leszármaztatott osztályok tartalmaznak, így a szabványosságból származó előny pillanatok alatt szertefoszlónak tűnhet. A viszonylagos előnyei és pozitívumai ellenére a Hyper-V felülete rendelkezik jó néhány negatívummal, amelynek okai részben a Microsoft üzletpolitikájában gyökereznek. Ha az ember nem C, C++ vagy C# nyelveken akar menedzsment alkalmazást készíteni felkészülhet, hogy mindent az alapokról kell felépíteni, teljesen egyedül. Az említett nyelvek esetében előre elkészített könyvtárcsomagok teszik egyszerűvé és gyorssá a fejlesztés folyamatát, más nyelvek esetében azonban marad a kísérletezés és a tapasztalatszerzés. Ezzel igazából egy platformfüggetlen alkalmazás megvalósítása válik nehézzé. A Hyper-V számára létrehozott osztályokon még némileg érződik a kiforratlanság, jó néhány metódus van, amely az osztályleíróban már régóta szerepel, egyelőre azonban implementálatlan. Ha a dokumentáltság és a használhatóság mellet a teljesítményt is értékelési szempontnak használjuk, akkor elmondható hogy a Hyper-V WinRM segítségével történő távoli menedzsmentje során tapasztalhatóak az elkészített menedzsment alkalmazásnál a legnagyobb reakcióidők.

6.1.3. ESX

Az implementációs folyamat során mélyebben megismert alkalmazások és menedzsment felületeik közül az ESX esetén volt talán a legellentmondásosabb az összkép. A termék egészét nézve egyszerre vannak jelen a széles körben használt, standardnak tekinthető menedzsment szabványok és az cég által készített saját protokollok. Sajnálatos mód a szabványosított rész a termék azon részére koncentrálódik, amely a jelen dolgozat témakörén kívül esik, így az itt megfogalmazott gondolatok és benyomások a felhasznált házi szabványra korlátozódnak.

Elsőként meg kell említeni az ESX szerver által nyújtott felületet és a hozzá adott natív nyelvi és dokumentációs támogatást. Összességében nézve számos programozási nyelvhez rendelkezik előre elkészített bináris csomagokkal, melyeknek egy részét mi magunk

is létrehozhatjuk, illetve Java és C# nyelvhez rendelkezik egy részletesebb fejlesztői dokumentációval, amely elég széleskörű ismeretanyagot tartalmaz, azonban pont a lényeg az ami felett elsiklik, hogyan tudunk az adott nyelv segítségével szolgáltatásokat igénybe venni. A legnagyobb problémát a többi menedzsment felülettől gyökeresen eltérő adat ábrázolás mód okozta, amely a menedzselt objektumok és a róluk információt hordozó adat objektumok közötti, a szokásosnál lazább csatolás eredménye. Ez eredményezte azt, hogy az ESX menedzselésére alkalmas kódrészletek nem közvetlenül a natív Java könyvtárat használják, amely a VI API webszolgáltatás leírójából generálható, hanem az átszabott és felhasználóbarátabbá tett VMware VI Java API csomagot. Már az a tény is jelzés értékű, hogy a gyári felületnek létezik módosított, konkurens változata. Az mindenesetre egyértelmű, hogy a korábbi termékekkel való kompatibilitás miatt az API nem változtatható meg teljes egészében, de a VMware VI Java API által használt csomagolásos megoldásokkal sokkal egyszerűbb és használhatóbb menedzsment felületet lehetne elérhetővé tenni, miközben a visszafelé kompatibilitás is megmaradna.

6.1.4. KVM és libvirt

Mint a korábbiakban említésre került, a KVM nem rendelkezik önálló távoli menedzsment interfésszel, sajnos ez jelentős mértékben megnehezíti az irányítására használható menedzsment eszközök készítését. Bár a libvirt használatával részben áthidalható ez a probléma, mégsem lehetünk teljesen elégedettek ezzel a megoldással sem. Ennek főképp az az oka, hogy a libvirt-et egy több virtualizációs megoldáshoz használható menedzsment felületnek és eszközkészletnek szánták. Annak érdekében, hogy minden igényt kielégítsen egy nagy komplexitású felületet definiál. Az egyes megoldások ennek a felületnek csak egy-egy részét támogatják és arról semmi nem szolgál információval, hogy ez a részhalmaz pontosan milyen elemekből áll, így csak a tesztelés közben derült ki, hogy egy adott funkció nem életképes az implementált saját alkalmazásban. A másik sajnálatos nehezítő tényező a libvirt kettős üzemmódjából adódik, annak is inkább a helyi üzemmódhoz kötött felhasználó azonosításához, amely nem teszi lehetővé, hogy a menedzsment klienst futtató felhasználó jogosultságaival érjük el a libvirt szolgáltatásokat. Mindezeknek megfelelően a jelen sorokban megfogalmazott értékelés a libvirt minősítése is egyben. A fent említett hibái mellett pozitívumként elmondható, hogy a libvirt számos programozási nyelvhez rendelkezik támogatással és használható dokumentációval, amely jelentős segítség, bár az kétségkívül érezhető, hogy elsődleges célterületei a Linux/Unix rendszerek alapjának tekinthető C és C++ illetve Python nyelvek.

6.1.5. Elkészített alkalmazás

Az elkészített alkalmazásról összességében elmondható, hogy megfelel az előzetes elvárásoknak, négyből három virtualizációs megoldás esetében annak adottságait és képességeit figyelembe véve minden előzetesen megjelölt funkció elérhető, az egyetlen kivétel a KVM. Az utóbbi esetében sajnos csak szemlélőként vehetünk részt a rendszer működésében.

Sikerült egy olyan felület megalkotása, amely lehetővé teszi szélesebb körű virtualizációs megoldások egységesített menedzselését, az implementáltak körének egyszerű bővítését. Az elkészített grafikus felület egyszerű kezelhetősége mellett nagyfokú rugalmasságot biztosít a továbbfejlesztéshez. Az alkalmazás egyetlen fájó pontja, hogy nem lehet vele a korábban már részletezett okok miatt virtuális gépeket saját kezűleg létrehozni, ezt megjelölhetjük, mint egy fontos továbbfejlesztési területet, amely segítségével az alkalmazás átlépheti a demonstrációs jelleget és teljes értékű termékké válhat.

Egy másik lehetséges fejlesztési irány lehetne a jelenlegi viszonylag szoros API-implementáció kódszintű kapcsolat lazítása. Az ezt követő megvalósításban az API-t alkotó interfészek és közös osztálytípusok külön könyvtárban kapnának helyet. Az egyes virtualizációs termékekhez megvalósított osztályok plugin jellegű egységeket alkotnának, amelyek igény szerint tölthetőek be. Bár a grafikus felület csak minimálisan érintkezik az egyes konkrét implementációkkal, mert az interfészeket használja, ezt a minimális függést is eliminálni lehetne. Ennek eszközéül szolgálhatna egy konfigurációs állomány, amely tartalmazza a használható hypervisor típusok nevét, és az azok esetén használandó konstruktor osztály minősített nevét, amelyet annak példányosításához felhasználhatnánk.

6.2. Összefoglalás

Az elvégzett munka során első lépésben körüljártam a számítógép virtualizáció jelentését, érintőlegesen megismerkedtem ezen gyűjtő fogalom mögé sorolható technikákkal, mint a platform, az operációs rendszer és az alkalmazás virtualizáció. Ezt követően behatóbban tanulmányoztam a platform virtualizáció fogalomkörét. A forráselemzés közben szerzett ismeretek rávilágítottak arra, hogy egy számítógép architektúrának milyen feltételeknek kell megfelelnie ahhoz, hogy virtualizálható legyen. Ezek alapján érzékelhetővé váltak azok a problémák, amelyek a ma legelterjedtebb architektúra, az x86 esetében jelentkezhetnek.

Ezt követően áttekintettem az erre a problémára megoldást nyújtó technológiákat és eszközöket, mint az emuláció, a bináris fordítás, paravirtualizáció és a hardveresen támogatott virtualizáció. Kisebb kitekintés keretében tanulmányoztam mik azok a tényezők, amelyek a virtualizáció használata során fellépő teljesítményveszteséget okozzák, és erre milyen megoldások léteznek, így jutottam el a memória és az I/O virtualizáció területére.

A munka következő állomásaként számba vettem a mai széles körben elterjedt virtualizációs alkalmazásokat. Megvizsgáltam melyek azok az általános vagy éppen különleges jellemzők, amelyekkel az egyes termékek rendelkeznek. A dolgozat lényegi részének az elkészített menedzsment alkalmazás tervezési és implementálási folyamata tekinthető, ezért a munka következő részében áttekintettem, melyek azok az elterjedt szabványok, amelyek a későbbiekben a megvalósítás során segítségemre lehetnek. Ennek folyományaként jutottam el a WS-Management és az azt kiegészítő WS szabványokhoz és a Common Information Model-hez, amelyek széles körben alkalmazott menedzsment eljárásokat rögzítenek, illetve a kimondottan virtualizációs termékek irányítására használt libvirt csomaghoz. Következő lépésként megvizsgáltam, hogy milyen multiplatformos

menedzsment eszközök léteznek és azok milyen jellemzőkkel bírnak, ennek eredménye a ConVirt és az OpenNebula című fejezet. A munka következő lépésében a figyelem középpontjába az Xen, Hyper-V, ESX és KVM alapú rendszerekhez használható menedzsment eszközök, és ezen rendszerek esetében használható menedzsment API-k kerültek, amely a tervezési és implementálási folyamat megalapozását szolgálta.

Ezt követően az eddig megszerzett ismeretanyag segítségével megterveztem egy absztrakt menedzsment felületet, amely segítségével lehetőség nyílik a különböző virtualizációs alkalmazások, rendszerek egységes irányítására. Ez az API a dolgozatra rendelkezésre álló erőforrás és idő kereteknek megfelelő méretű funkcionalitás halmazt fogalmaz meg, amely segítségével alapvető műveleteket tudunk elvégezni távoli erőforrásokkal és azokon található virtuális gépekkel. A programozási felület megalkotását követte az implementáció, amely során a korábban megismert menedzsment protokollok felhasználásával megvalósításra kerültek az előzetesen megfogalmazott funkciók. A menedzsment funkciókat biztosító könyvtár csomag elkészítésével párhuzamosan született meg az azokat elérhetővé tévő grafikus alkalmazás, amely egyben a fejlesztési folyamatban is fontos szerepet játszott az éppen megvalósított funkció helyességének ellenőrizhetővé tételével. Az implementálást követően az elkészített alkalmazás tesztelése következett, amely részben automatizáltan részben kézi úton történt. A felfedezett és egyértelműen az elkészített alkalmazáshoz kapcsolható hibák javításra kerültek.

A teljes munkafolyamat termékeként született alkalmazás összességében megfelel a fejlesztési folyamat kezdetén megfogalmazott elvárásoknak. Jól szemlélteti a virtuális rendszerek egységes menedzsmentjének nehézségeit és az abból adódó problémákat, egyben egy kezdetleges megoldást is nyújt azokra. Jelenlegi állapotában demonstrációs célok betöltésére alkalmas, a korábbiakban megnevezett hiányosságait - mint például a virtuális gépek konfiguráció alapú létrehozása - pótolva azonban eredményes munkavégzésre alkalmas szoftverré válhat.

A munka során egyértelművé vált, hogy a bár jelen pillanatban a virtualizációs eszközök alacsonyabb számának köszönhetően az egységesített menedzsment eszközök fejlesztése kézben tartható és beiktatott absztrakciós szintekkel megoldható feladat, a termékcsaládok és verziók szaporodásával ez az út egyre nehezebben lesz járható. Elfogadható és hosszú életű megoldásként csak egy egységes szabvány jöhet szóba, amelyre már vannak kezdeményezések, de sajnos kis támogatottságuk miatt a jelentőségük elenyészik a probléma fontossága mellett. Egy lehetséges és jól járható út lenne a CIM szabvány még szélesebb körű alkalmazása, amely egyrészt kellő háttérrel rendelkezik a menedzsment területen, hogy ne az alapoktól kelljen kezdeni az építkezést, másrészt megoldaná az általános-termékspecifikus felület megalkotásának dilemmáját is. Ha olyan felületet akarunk létrehozni amely sok terméket támogat akkor le kell mondani a specifikus dolgokról, ha a termék specifikus részeket is belevesszük a menedzsment felületbe, akkor gyakran a széleskörű felhasználhatóságot kell feláldozni. A CIM esetében az osztályok közötti leszármaztatási kapcsolatokkal mindez könnyedén áthidalható lenne, hiszen az általános, minden termékre jellemző részek egy ős osztályba kerülhetnek a specifikus részek pedig ezek leszármazottjaiba.

Bár a jelenlegi szabványosítási folyamatok által mutatott lehetséges utak nem teljesen egyértelműek, de jó eséllyel mondhatjuk, hogy a CIM és a rá épülő termékek a virtualizáció területén is meghatározó menedzsment szabvánnyá válhatnak.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani családomnak, hogy áldozatos munkájukkal lehetővé tették tanulmányaimat és az egyetemi évek során mindvégig mellettem álltak. Köszönettel tartozom Micskei Zoltánnak a dolgozat megírása közben adott hasznos tanácsaiért és egész féléves munkájáért.

Irodalomjegyzék

[1] *Convirture.* http://www.convirture.com/.

[2] *DMTF Common Information Model.* http://www.dmtf.org/standards/cim/.

[3] *Hyper-V Architecture.* http://msdn.microsoft.com/en-us/library/cc768520%28BTS.10%29.aspx.

[4] *Java Native Access.* https://jna.dev.java.net/.

[5] *JUnit.* http://www.junit.org/.

[6] *KVM - Kernel Based Virtual Machine.* http://www.redhat.com/f/pdf/rhev/DOC-KVM.pdf.

[7] *Open Virtualization Format.* http://www.dmtf.org/initiatives/vman_initiative/.

[8] *OpenNebula.* http://www.opennebula.org/.

[9] *VMware VI Java API.* http://vijava.sourceforge.net/.

[10] *Web Services Addressing (WS-Addressing).* http://www.w3.org/Submission/ws-addressing/.

[11] *Web Services Enumeration (WS-Enumeration).* http://www.w3.org/Submission/2006/SUBM-WS-Enumeration-20060315/.

[12] *Web Services for Management.* http://www.dmtf.org/standards/wsman.

[13] *Web Services Transfer (WS-Transfer).* http://www.w3.org/Submission/WS-Transfer/.

[14] *Wiseman Project.* https://wiseman.dev.java.net/.

[15] convirture. *Overview.* http://www.convirture.com/images/screenshot_1c.png.

[16] Distributed Management Task Force, Inc. *Common Information Model (CIM) Standards.* http://www.dmtf.org/standards/cim/.

[17] Gerald J.Popek and Robert P.Goldberg. Formal requirements for virtualizable third generation architectures. 1974.

[18] Kurt Roggen. *Windows Server 2008 (R2) blog by Kurt Roggen [BE].* http://trycatch.be/blogs/roggenk/archive/2008/10/20/hyper-v-parent-partition-vsp-vsc-and-vmbus.aspx.

[19] libvirt. *Libvirt Xen support.* http://libvirt.org/architecture.html.

[20] LinuxInsight. *Finally user-friendly virtualization for Linux.* http://www.linuxinsight.com/finally-user-friendly-virtualization-for-linux.html.

[21] M. Tim Jones. *Anatomy of the libvirt virtualization library.* http://www.ibm.com/developerworks/linux/library/l-libvirt/.

[22] Ewan Mellor, Richard Sharp, and David Scott. *Xen Management API*, api revision 1.06 edition, 2007.

[23] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Computer*, 38:32–38, 2005.

[24] Sun Hao. *Enabling Web Service with Common Information Model.* http://www.ibm.com/developerworks/xml/library/ws-CIM/cimandwsman.JPG.

[25] Tom Howarth. *The End of ESX is Near – Is ESXi Ready for the Enterprise?* http://www.virtualizationpractice.com/blog/?p=3837.