Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Balla Dávid, Phan Ngo Anh Tra

OSZTOTT MEMÓRIÁN KERESZTÜL KOMMUNIKÁLÓ VIRTUÁLIS GÉPEK OPENSTACK FELHŐBEN

KONZULENSEK

Dr. Maliosz Markosz
Dr. Simon Csaba
Géhberger Dániel

BUDAPEST, 2016

Tartalomjegyzék

Tartalomjegyzék ... **1**
1 Bevezetés ... **2**
2 Számítási felhő ... **3**
2.1 Szolgáltatás típus szerinti csoportosítás ... 4
2.1.1 Software as a Service ... 4
2.1.2 Platform as a Service ... 4
2.1.3 Infrastructure as a Service ... 4
2.2 Felhasználói kör szerinti csoportosítás ... 4
2.2.1 Nyilvános felhő ... 5
2.2.2 Privát felhő ... 5
2.2.3 Hibrid felhő ... 5
3 OpenStack rendszer áttekintése ... **6**
4 Virtualizáció ... **9**
4.1 Quick Emulator – QEMU ... 9
4.1.1 QEMU Machine Protocol - QMP ... 10
4.2 Kernel-based Virtual Machhine ... 11
4.3 Libvirt ... 12
5 Osztott memória ... **13**
6 Mérőrendszer ... **14**
6.1 OpenStack felépítése a laboratóriumban ... 15
6.2 Osztott memória biztonsági kérdései felhőben ... 16
6.3 QMP engedélyezése ... 16
6.4 Osztott memória használata a virtuális gépekben ... 19
7 Mérések ... **22**
7.1 Mérések specifikálása ... 22
7.2 A mérések implementálása ... 22
7.3 Mérési eredmények ... 24
8. Összefoglalás ... 37
Irodalomjegyzék ... **38**

1 Bevezetés

Napjainkban teret nyert a felhő alapú számítástechnika, mert a szolgáltatások üzemeltetői számára fontos, hogy az üzemeltetés a lehető legköltséghatékonyabb legyen. Ezek a felhők elrejtik a hardvereket, így a szolgáltatások nem dedikált hardvereken futnak, hanem virtualizált formában több eszközön elosztottan. Ezért a felhő szolgáltatásban kulcsfontosságú szerepet játszanak a virtualizációs technológiák, mivel az alkalmazások jellemzően virtuális gépeken futnak. A felhő alapú szolgáltatások nagy előnye, hogy annyi erőforrást használhatunk amekkorára igényt tart az alkalmazás, valamint a felhőben futó alkalmazások számára nagy rendelkezésre állást és könnyebb karbantarthatóságot biztosít.

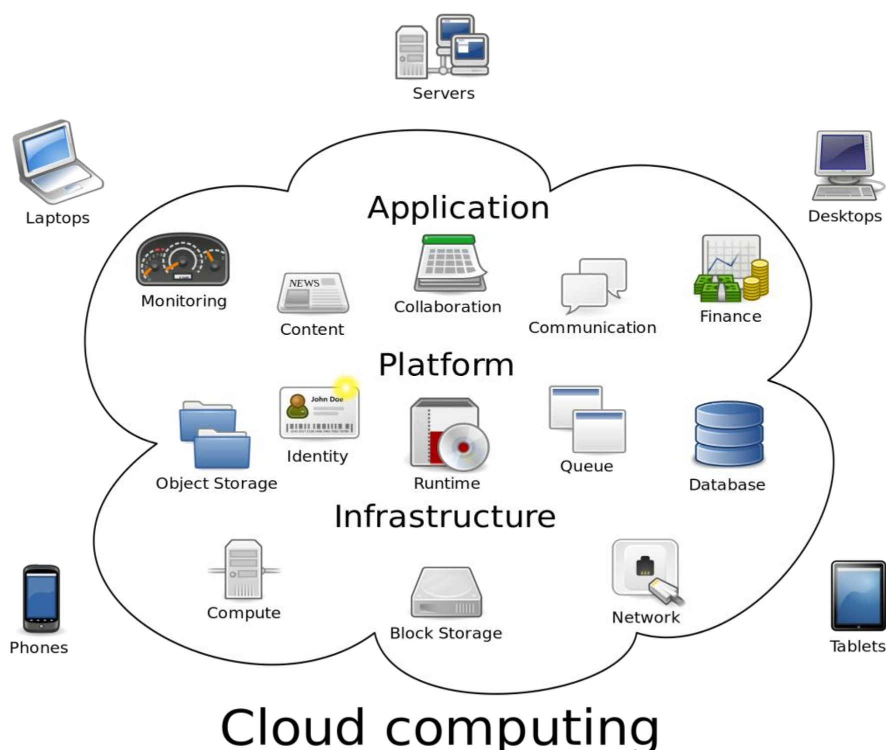
A felhő rendszerekben a virtuális gépek szeparáltan futnak, a közöttük történő kommunikáció pedig a felhő által biztosított virtuális hálózaton keresztül valósul meg. Két virtuális gép kommunikációjára osztott memórián keresztül is lehetőség van, azonban értelemszerűen a nyilvános felhőkben biztonsági okok miatt ez általánosan nem megengedhető opció.

Azonban privát felhő környezetben megengedhetjük az osztott memórián keresztüli kommunikáció megvalósítását. Privát felhők üzemeltetői körében igen népszerű az OpenStack, amely infrastruktúra szolgáltatás kialakítására alkalmas ingyenes elérhető, nyílt forráskódú megoldás. Az OpenStack egy olyan felhő operációs rendszer, amely számítási, tárolási és hálózati erőforrások menedzseléséért felel.

Dolgozatunkban megvizsgáljuk az osztott memórián keresztüli kommunikáció OpenStack felhőbe való integrálását, majd bemutatjuk az architektúrába illeszthetőségét és implementációját, amelyet egy új modul hozzáadásával értünk el. Mérésekkel összehasonlítjuk a hálózaton keresztüli és az osztott memórián keresztüli kommunikáció hatékonyságát, teljesítményét.

2 Számítási felhő

A felhő alapú számítástechnika (cloud computing) napjainkban igen elterjedt ágazat. A szolgáltatások már nem dedikált hardveren, hanem több eszközön elosztottan futnak az üzemeltetési részleteit a felhasználótól elrejtve. Ilyen rendszerre építve könnyebb karbantarthatóság, költségkímélés, könnyű skálázhatóság és nagyobb rendelkezésre állás érhető el, azonban hátránya lehet, hogy így a rendszer komplexitását növeljük. Ezeket a szolgáltatásokat a felhasználók publikus felhő használata esetén interneten keresztül, privát felhő használata esetén helyi hálózaton, VPN-en vagy interneten keresztül érhetik el. A felhőszolgáltatásokat számos szempont alapján csoportosíthatjuk. A két legfontosabb szempont azonban a nyújtott szolgáltatás illetve a felhasználói kör [1]. A nyújtott szolgáltatás szempontjából három nagy csoportot különböztethetünk meg: Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service, amiről áttekintést ad az ***1.* ábra**.



1. ábra - A felhő alapú számítástechnika elemei és kliensei [2]

2.1 Szolgáltatás típus szerinti csoportosítás

2.1.1 Software as a Service

Olyan szoftverszolgáltatási módszer, amely esetén a szoftver egy szolgáltató szerverein egy „felhőben" fut és a hozzá tartozó adatok is a szolgáltató tárhelyén találhatóak. A felhasználók pedig a hálózaton keresztül – leggyakrabban az Interneten keresztül – egy vékonykliens alkalmazás, esetleg böngésző segítségével érik el a szolgáltatást. (pl.: Google Docs, Microsoft Office 360)

2.1.2 Platform as a Service

A PaaS egy olyan cloud szolgáltatás, amely esetén a szolgáltató egy platformot és egy környezetet biztosít az alkalmazásfejlesztők számára. A fejlesztők a szolgáltató eszközeit testre szabhatják, majd ezek segítségével végezhetik a munkájukat. A szolgáltatáshoz a felhasználók az interneten keresztül férnek hozzá, általában webböngésző segítségével. (pl.: Google App Engine, OpenShift)

2.1.3 Infrastructure as a Service

Az IaaS egy olyan cloud szolgáltatás, amely a felhasználók számára hozzáférést biztosít a virtualizált számítási erőforrásokhoz, lényegében virtuális gépek menedzselését adja a felhasználók kezébe. A felhasználók előre telepített virtuális gép sablonok közül választhatnak, majd ezeket saját igényük szerint módosíthatják. A futtatott virtuális gépekhez rendelhetnek erőforrásokat, amelyeket úgynevezett flavor-ök segítségével lehet megadni. A flavor-ben megadhatóak a legfontosabb erőforrások, így a CPU magok száma, a memória és a háttértár méretei. Ezen felül a felhasználónak lehetősége van az elkészült virtuális gépeikről egy később újra betölthető mentést készíteni, ezek az úgynevezett snapshot-ok. (pl.: OpenStack, Amazon Web Services)

Természetesen a való életben számos más szolgáltatással is találkozhatunk, mint például Storage as a Service, Firewall as a Service vagy LoadBalancer as a Service.

2.2 Felhasználói kör szerinti csoportosítás

A felhasználói kör és a hozzáférés szempontjából háromféle megoldást különböztetünk meg: nyilvános, privát és hibrid felhőket.

2.2.1 Nyilvános felhő

Nyilvános felhők esetén az infrastruktúrát a szolgáltató a saját telephelyein üzemelteti. Az infrastruktúrán tetszőleges ügyfelek osztozhatnak. Ebben az esetben a felhasználók adatai ki vannak szolgáltatva a szolgáltatás üzemeltetőjének.

2.2.2 Privát felhő

Privát felhők esetén egy szervezet számára dedikált felhőt alakítanak ki. Ebben az esetben csak az adott szervezet tagjai használhatják a felhőszolgáltatást. Ez a megoldás természetesen biztonságosabb a nyilvános felhőknél, ugyanis az adatok „házon belül” maradnak. Ebben az esetben viszont szembe kell nézni a magasabb költségekkel, ugyanis ezt a szolgáltatást a felhasználó szervezetnek kell kialakítania és üzemeltetnie.

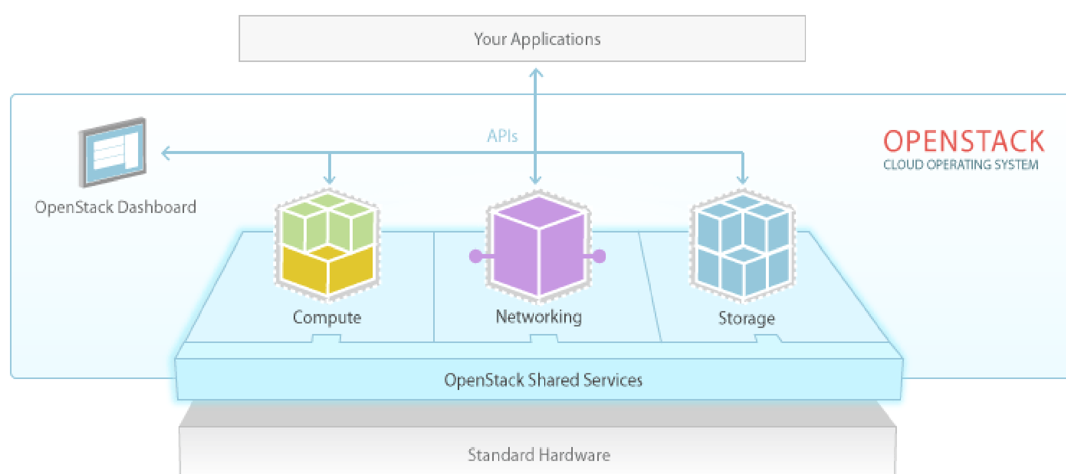
2.2.3 Hibrid felhő

Hibrid felhők esetén a nyilvános és privát felhők kombinációjáról van szó. Ilyen eset lehet, ha a kritikus adatainkat, alkalmazásainkat egy magánfelhőn tároljuk, üzemeltetjük, viszont a kevésbé kritikus adatainkat egy nyilvános felhőn tároljuk. Esetleges erőforrás-kiegészítésként is használhatjuk a hibrid felhőket, abban az esetben, ha kevésnek bizonyulna a magánfelhő számítási kapacitása, akkor nyilvános felhőkkel biztosítható a többletszolgáltatás

3 OpenStack rendszer áttekintése

Az OpenStack egy nyílt forráskódú legnagyobbrészt pythonban megírt alkalmazásrendszer, amellyel infrastruktúra szolgáltatást (IaaS) nyújtó számítási felhőt lehet kialakítani. 2010-ben indult a Rackspace Hosting és a NASA közös projektjeként. Azóta annyira rohamosan fejlődik, hogy számos cég OpenStack alapokra építi a saját privát felhő megoldását, mint például a HP, Ericsson és a Mirantis.

Az OpenStack három fő funkcióját mutatja be a **2. ábra**. A Compute a számítási erőforrások kezeléséért, a Networking a komponensek és virtuális gépek hálózatáért, a Storage pedig a virtuális gépek számára szükséges háttértár szolgáltatásért felelős.



2. ábra – OpenStack fő komponensei [3]

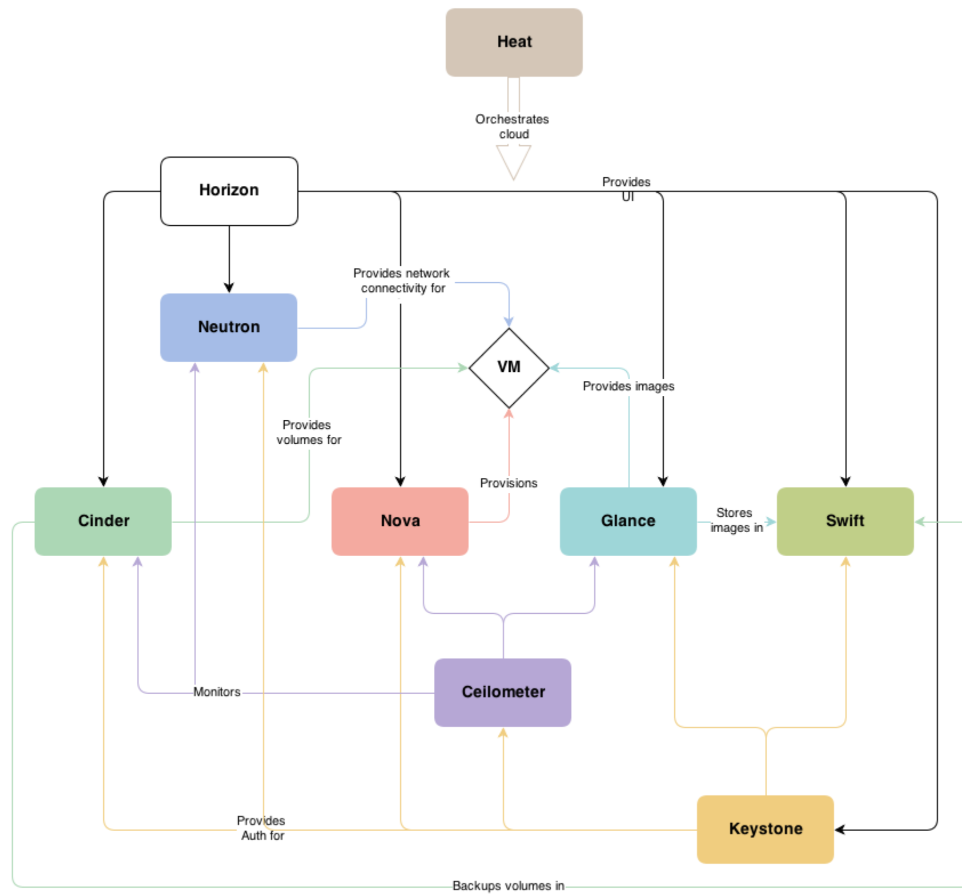
A fenti három fő funkciót, illetve a további funkciókat megvalósító OpenStack komponensek a következőek:

- Nova – Ez a komponens az egész OpenStack „lelke". Ez felelős a virtuális gépek létrehozásáért és futtatásáért, majd leállításáért. Továbbá felelős az OpenStack vezérlő (Controller node) és hálózati (Neutron) csomópontokkal való együttműködésért.
- Keystone – Hitelesítés és hozzáférés engedélyezés szolgáltatást nyújt a felhasználók illetve a további OpenStack belső szolgáltatás számára.
- Neutron – Ez a komponens felelős a virtuális hálózatok menedzsmentjéért. Képes virtuális hálózatot és virtuális útvonalválasztót

(virtual router) létrehozni, továbbá feladata ezt a fizikai hálózatra leképezni.

- Glance – Ez a komponens felelős a képfájlok (image) kezeléséért. A Glance különböző formátumú képfájlokat képes kezelni, mint például a qcow2 (QEMU Copy On Write 2) vagy ISO.
- Horizon – Webes kezelő felületet biztosít a felhasználóknak, ezen keresztül menedzselhető az OpenStack rendszer. A virtuális infrastruktúrát vagy ennek segítségével, vagy parancssorból lehet létrehozni.
- Ceilometer – Ezzel a komponenssel mérhető a virtuális gépek erőforrás használata. Méri többek között a processzor- és memória-használatot és a hálózati forgalmat is.
- Heat – Automatizálást megvalósító (orchestration) modul. Egy sablon (template) alapján képes virtuális gépek és hálózatok létrehozására és összekötésére.
- Swift – Háttértár szolgáltatás, amelyben a feltöltött fájlokat nem direktben érthetjük el a virtuális gépekről, hanem REST API segítségével.
- Cinder – Háttértár szolgáltatás, amellyel direktben érhetjük el a fájlokat, mert a virtuális gépek egy felcsatolt fájlrendszer kötetként (volume) látják a háttértárat.

Ezeknek a komponenseknek a viszonyát és kapcsolódását mutatja be a **3. ábra**.

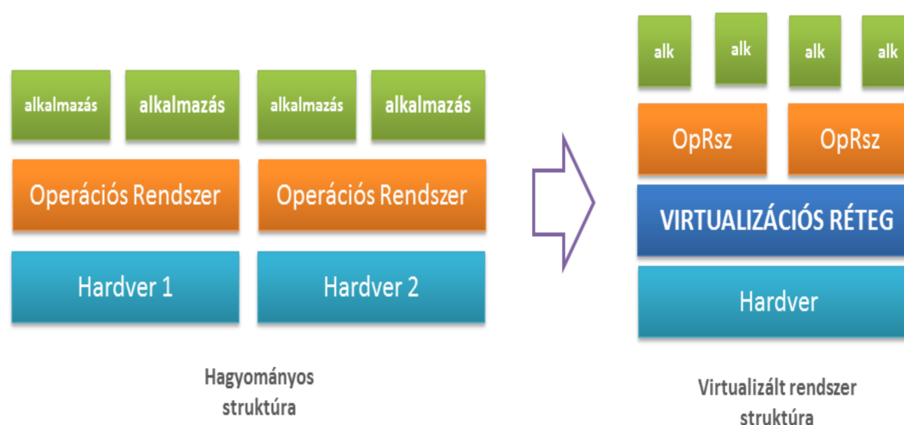


3. ábra- Openstack komponenseinek viszonya [4]

Az OpenStack többféle virtualizációt támogat, mint például a XenServer, a Hyper-V, a VMware vagy mint a QEMU/KVM.

4 Virtualizáció

A virtualizáció egy olyan szoftveres technológia, amely lehetővé teszi, hogy egy időben egyszerre több operációs rendszert futtassunk ugyanazon a számítógépen (**4. ábra**). Így az infrastruktúra egyszerűbbé és hatékonyabbá válik, mivel lehetővé válik, hogy egy fizikai kiszolgáló egyszerre több különálló rendszeren futó logikai szolgáltatást is kezelni tudjon.



4. ábra - Virtualizációs technológia összehasonlítása a hagyományos szerver-koncepcióval [5]

Virtualizáció esetén általában egy fizikai számítógépen futtatott virtuális gépekre gondolunk. Ebben az esetben platform virtualizációról illetve operációs rendszer szintű virtualizációról beszélünk. Platform virtualizáció esetén a virtualizált környezetek egy virtualizációs réteg felett futnak. A virtualizációs réteg gondoskodik a felette futó virtualizált környezetek menedzseléséről.

A virtuális gépek szükség esetén az egyik fizikai kiszolgálóról áthelyezhetők egy másikra. Ezzel nagyban növelni tudjuk szolgáltatásaink rendelkezésre állását, hiszen egy fizikai kiszolgáló meghibásodása esetén a rajta futó kieső szolgáltatásokat egy másik számítógépen újra tudjuk indítani.

4.1 Quick Emulator – QEMU

A QEMU egy nyílt forráskódú processzoremulátor program. A processzor emulálása dinamikus bináris fordítón keresztül történik. A QEMU két fő funkciója

egyrészt teljes architektúra emulálása, ahol a QEMU különböző processzor architektúrákat képes emulálni, mint például: x86, x86_64, PowerPC, Sparc, Coldfire, ARM, ARM-Cortex, másrészt nemcsak processzort emulál, hanem különböző eszközöket is, úgymint floppy lemez, virtuális USB, PS/2 egér és billentyűzet. Az emulátorban különféle operációs rendszerek futhatnak, úgymint például GNU/Linux, BSD, Mac OS X, Windows.

A QEMU alkalmazható kizárólag CPU emulációra is felhasználói szintű folyamatok esetén, ekkor egy adott architektúrára lefordított bináris alkalmazás futtatható egy másik architektúrán, végrehajtva a rendszer hívások fordítását, a POSIX jelzés- és szálkezelést.

A QEMU KVM-et (lásd 4.2 alfejezet) használva képes közel olyan teljesítményre, mintha virtualizáció nélkül használnánk a gépet.

4.1.1 QEMU Machine Protocol - QMP

A QMP egy JSON alapú protokoll, amely lehetőséget ad arra, hogy kommunikáljunk egy futó QEMU példánnyal és azon keresztül menedzseljük a futtatott virtuális gép működését. A QMP-n keresztül információt nyerhetünk egy futtatott példányról, annak egyes eszközeiről, sőt még hozzá is adhatunk eszközöket a futó példányhoz, akár memóriát is. A QMP a QEMU 2.2 verziójától támogatott, a verziószámok növekedésével fokozatosan egyre több funkcióval látták el. A QEMU 2.5-ös verziójában már futás közben akár több memóriát is adhatunk a futó virtuális gépnek.

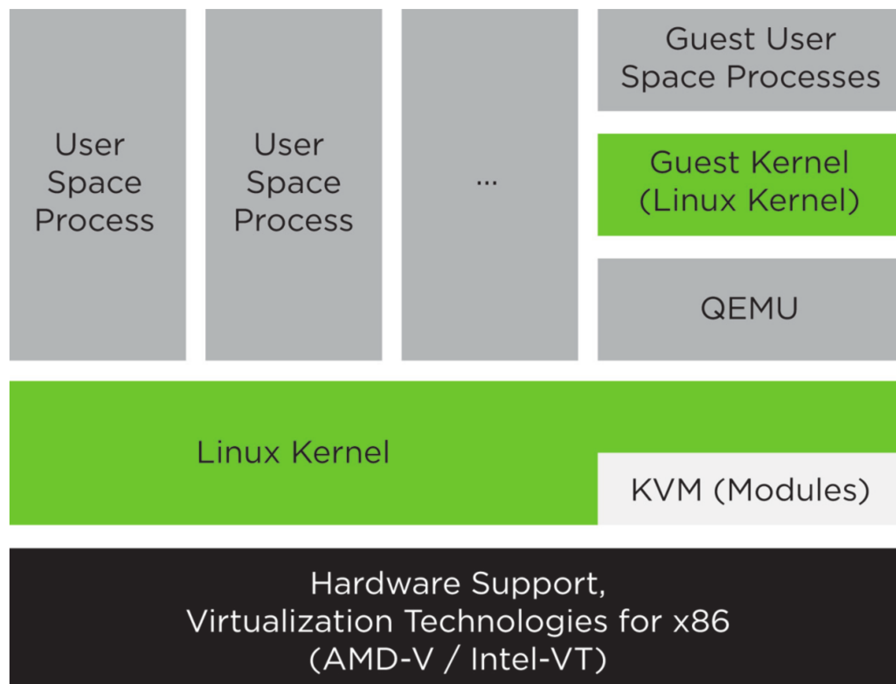
A QMP használatának különböző módjai vannak:

- QEMU-monitor-command használatával
- TCP socketen keresztül
- unix socketen keresztül

Munkánk során a TCP socketen keresztüli kommunikációt alkalmaztuk és integráltuk az OpenStack környezetbe.

4.2 Kernel-based Virtual Machhine

A Kernel-based Virtual Machhine (KVM) technológia egy Linux operációs rendszeren futó (hosted típusú) virtualizáció, amely Linux kernelt virtuális gépeket futtató tulajdonsággal ruházza fel. A KVM csak olyan processzorokon működik, amelyeken hardveresen támogatott a virtualizáció (Intel VT-x vagy AMD-V). Ahogyan az a KVM elnevezésből is látszik, ez a technológia kernel alapú, a működéséhez kernel modulokat kell betölteni. Ezek a modulok a kvm.ko és Intel processzor esetén kvmintel.ko vagy AMD esetén a kvm-amd.ko. A KVM a QEMU nyílt forráskódú processzoremuláló- és dinamikus fordítóprogramot használja, képes a gazda gép operációs rendszerétől vagy architektúrájától függetlenül egy másik operációs rendszert vagy architektúrát emulálni (*5.* **ábra**). Ha a gazda gép nem képes a hardveres virtualizációra vagy a futtatott host operációs rendszer kernelében nincs betöltve a KVM hardveres virtualizációt támogató modul, akkor a QEMU csak emulált módban tud működni. Ez lényegesen lassabb működést eredményez a hardveres virtualizációt támogató KVM megoldáshoz képest.

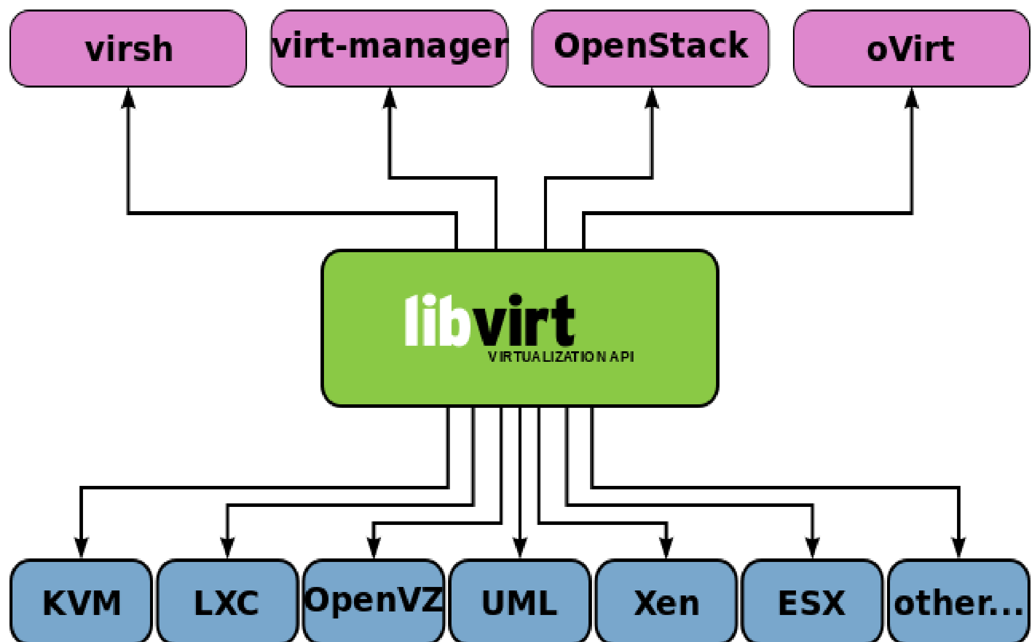


5. ábra - KVM virtualizációs megoldás architektúrája [5]

4.3 Libvirt

A libvirt különböző szoftverek csoportja, mely lehetővé teszi a virtuális gépek könnyű kezelését, továbbá számos egyéb virtuális eszköz menedzselését is lehetővé teszi, így például a háttértár vagy a hálózati interfészek kezelését is.

A fent említett szoftverek többek között egy API könyvtárat, egy démont (Linux szolgáltatást) (libvirtd) és a virtuális gépek kényelmes menedzselésére egy parancssori interfészt, a virsh-t tartalmazzák. A libvirt célja egy egységes, közös interfész létrehozása a különböző virtualizációs technológiákhoz (pl. KVM, Xen, VMWare ESX). A libvirt funkciókat különböző felhasználói interfészeken keresztül érhetjük el, úgymint a virsh, ami parancssori hozzá férést biztosít, vagy mint virt-manager ami grafikus elérést ad a libvirthez. Az Openstack is a libvirt API-t használja, hogy menedzselje a virtuális gépeket. (**6. ábra**)



6. ábra - A libvirt, néhány felhasználói felülete és virtualizációs technológiája [6]

5 Osztott memória

Az osztott memória segítségével egyidejűleg különböző programok érhetik el ugyanazt a memóriaterületet, aminek a segítségével növelhetik az egymás közötti kommunikáció sebességét vagy elkerülhetik az adatok duplikált tárolását.

Aa osztott memória hardveresen és szoftveresen is megvalósítható. Több-processzoros rendszerek esetén a hardveres értelmezése, hogy ugyanazt a memória területet több processzor is használhatja egyszerre. Különböző megoldásai vannak, mint például:

- Uniform Memory Access (UMA) - Minden processzor egységesen fér a fizikai memóriához. A memória elérés idő függvénye független a processzor helyzetétől.
- Non-Uniform Memory Access (NUMA) - Memória elérés idő függvénye függ a processzor helyzetétől. A processzorok a saját memóriájukat gyorsabban elérik, mint a távoli memóriát.
- Cache-Only Memory Architecture (COMA) - NUMA-hoz képest a saját memória úgynevezett cache memória, ami gyorsabb memória elérést ad.

Az osztott memória szoftveres megvalósításban a programok, folyamatok közötti adatközlésre (InterProcess Communication – IPC) alkalmazható. Számos folyamat képes kommunikálni, azaz olvasni a memóriából vagy írni memóriába anélkül, hogy rendszerhívásokat kellene igénybe venni. Természetesen ez módszer szinkronizációs és koordinációs plusz feladatot rejt magában, amennyiben több folyamat írna ugyanarra a területre.

6 Mérőrendszer

A projekt elkészítésekor egy olyan környezet kialakítása volt a cél, amely egy felhő rendszerben fut és támogatja az azonos számítási csomópontokon (compute node) futó virtuális környezetek közötti osztott memóriás kommunikációt.

A fentiek alapján nem meglepő, hogy az osztott memóriának már több olyan alkalmazási területét is javasoltak, amikor számítási feladatokat végző egységek kellett kommunikáljanak egymással. Például a kilencvenes években intenzíven kutatták az osztott memória alkalmazhatóságát Java szálak között [7].

A felhő rendszerek esetében pedig kimutatták a nyilvános felhők limitációit, mivel –ahogy a bevezető fejezetben is említettük- ott nem lehetséges csak a hálózaton keresztüli kommunikáció [8].

A mi munkánk beleillik a napjainkban intenzíven folytatott felhő rendszereken belüli hatékony VM-közti kommunikációra megoldásokat kereső kutatásokba. A kutatások állapotáról egy alapos átfogó képet adnak a [9] szerzői. Ugyanakkor a mi motivációnk a távközlési szektorban alkalmazható VM-közti kommunikáció támogatása. Üzemi körülmények közt a hibák, VM kiesések pótlása során a gyakran a mikroszekundum alatti nagyságrendű válaszidők elérése a cél. Egy ilyen nagyobb léptékű kérdéskör egy konkrét vizsgálandó része az a kérdés, amire mi mérés alapú választ igyekeztünk adni.

A mérőkörnyezetben szerettük volna összehasonlítani a virtuális gépek közötti kommunikáció lehetséges módjait. Így az egyik lehetőség az volt, hogy a virtuális gépek a felhő környezet virtuális hálózatán keresztül érik el egymást, míg a másik lehetőség az osztott memórián keresztüli kommunikáció. A laborkörnyezetben az osztott memórián keresztüli kommunikációt csak azonos számítási csomópontokon tudtuk megvalósítani, ugyanis nem állt rendelkezésünkre RDMA (Remote Direct Memory Access) képes hálózati kártya. Az RDMA támogatással képes egy gép direkt memória elérést biztosítani egy másik gép számára az operációs rendszer támogatása nélkül. Ez a technológia magas áteresztőképességet és alacsony késleltetést igényel a hálózattól. Így az egyik lehetőség az volt, hogy a virtuális gépek a felhő környezet

virtuális hálózatán keresztül érik el egymást, míg a másik lehetőség az osztott memórián keresztüli kommunikáció.

6.1 OpenStack felépítése a laboratóriumban

Az OpenStacket választottuk felhő rendszernek és ennek is a Mitaka verziójára esett a választás, mert az Openstack telepítés során a QEMU 2.5-ös verziója települ integráltan a számítási (compute) csomópontokra. Az OpenStack Mitaka verziónak a minimális konfigurációja egy vezérlő (controller) és egy virtuális gépeket futtató számítási (compute) csomópont (node). Esetünkben több compute géppel rendelkezik a rendszerünk. A controller gép felelős a felhő rendszer menedzseléséért, ezen a gépen futnak a fő komponensek, mint a Nova, a Horizon és a Heat is. A controller gép felelős továbbá a virtuális hálózat megfelelő működéséért, ezen a gépen fut a Neutron szerver. A compute gépek biztosítják a virtualizációt, ezeknek a gépeknek az erőforrásait felhasználva jönnek létre a virtuális gépek. Az OpenStack komponenseket futtató számítógépeknek az egymással való kommunikációhoz szükségük van egy menedzsment hálózatra. A menedzsment hálózat egy privát hálózat, a külvilágból nem elérhető, a 10.1.1.0/24-es alhálózatból áll. Ezen a belső hálózaton lévő számítógépek hálózati címfordítás (Network Address Translation – NAT) segítségével érik el a nyilvános Internetet. A laboratóriumi környezetben ugyanezt a hálózatot használjuk a virtuális gépek közötti hálózati forgalomra is a következőkben leírt módon. A virtuális gépek (VM) közötti hálózati kommunikáció egy fedő hálózat (overlay network) segítségével, Virtual Extensible LAN (VXLAN) technológiával valósul meg. A VXLAN lehetővé teszi, hogy már egy meglévő Layer 3 hálózat fölött alakíthatunk ki virtuális LAN hálózatot. A hálózatok kiépítésekor nem volt lehetőség menedzselhető kapcsolók (switch) használatára, így a Neutron által biztosított VXLAN alagút technika (tunneling) beállításokkal tudnak kommunikálni a virtuális gépek. A VXLAN tunneling LinuxBridge segítségével valósul meg.

Az OpenStack többféle virtualizációt támogat, csak néhányat kiemelve: QEMU/KVM, LXC, Xen [10]. A dolgozatunkhoz a választásunk a QEMU/KVM virtualizációra esett, mivel szükségünk volt a QMP által biztosított hozzáférésre a virtuális gépekhez, és a KVM segítségével a hardveres virtualizációs támogatottság

miatt a virtuális gépek futása is gyors. Továbbá a QEMU/KVM egy széles körben elterjedt virtualizációs megoldás, aminek telepítése, futtatása nem igényel komolyabb beavatkozást a gazda operációs rendszerbe és az OpenStack teljes támogatást ad a használatára.

6.2 Osztott memória biztonsági kérdései felhőben

Eredetileg az OpenStack nem teszi lehetővé a különböző virtuális gépeken futó alkalmazások kommunikációját osztott memórián keresztül. Ez természetesen érthető is, hiszen az OpenStack egy széles körben alkalmazott felhő megoldás, így publikus felhők esetén is gyakran találkozhatunk vele. Mivel az ilyen típusú felhőkben különböző identitású felhasználók különböző feladatokat futtatnak a virtuális környezeteikben, ezért az osztott memórián keresztüli kommunikáció egy biztonsági rést is jelenthet a rendszerben.

Privát felhők esetén, ahol csak egy zárt csoport használhatja a felhőt és nem tartunk rosszindulatú támadóktól, az osztott memória alkalmazása nem jelent ilyen jellegű problémát.

Olyan virtuális gépeket akartunk futtatni, amelyekhez futás közben tudunk osztott memóriát csatolni. Ezt a funkciót a QEMU lehetővé teszi, ellenben az OpenStack alapértelmezetten nem teszi lehetővé az azonos compute node-okon futtatott virtuális környezetek közötti osztott memóriás kommunikációt, ezért ezt a funkciót ki kellett fejleszteni.

6.3 QMP engedélyezése

Az OpenStack a virtuális gépeket a libvirt segítségével konfigurálja fel. A Libvirt a virtuális gépekhez egy úgynevezett instance xml-t hoz létre, melyben a virtuális gép működését meghatározó paraméterek kapnak helyet. A libvirt által generált xml fájlt adja meg az elindítandó virtuális gép tulajdonságait, emellett tartalmaz KVM/QEMU specifikus elemeket is, mint például a QEMU folyamathoz további egyedi parancssori opciókat adhatunk meg, amivel a QMP elérést tudjuk biztosítani.

A megvalósítás az xml fájl módosításával érhető el: xml instance elemét kiegészítjük egy névtérrel amely lehetővé teszi a QEMU parancssori argumentumok

használatát, továbbá a kívánt parancssori opciókat állítjuk be a instance xml-ben, az erre kijelölt elemeken belül.

Az első ötletünk az volt, hogy egy már futó virtuális gép instance xml-jét manuális szerkesztéssel kiegészítjük a QMP-t engedélyező sorokkal. Sajnos ez a megoldás nem volt azonnali hatású, mivel a módosítások érvényre juttatása érdekében a virtuális gépeket újra kellett indítani, ekkor azonban a leíró fájl is újragenerálódott és elvesztek a hozzáadott módosítások.

Az instance xml fájl már a generálása közbeni módosításához az OpenStack nova komponensében változtatásokat kellett végrehajtani. A virtuális gépeket leíró xml fájlokat a nova csomag virt/libvirt/driver.py python fájljának segítségével állítja elő az OpenStack.

Ebben a fájlban a _get_guest_xml metódus felel az instance xml fájlba írásáért. Mielőtt még megtörténne a fájlba írás, egy xml injection-t alkalmazunk, és a fent ismertetett elemekkel egészítjük ki az xml fájlt.

A következő sorokat vittük be az eredeti xml-be:

```
      <domain type='kvm'
xmlns:QEMU='http://libvirt.org/schemas/domain/QEMU/1.0'>
      <QEMU:commandline>
      <QEMU:arg value='-qmp'/>
      <QEMU:arg value='tcp:<ip_address>:<port>,server'/>
      </QEMU:commandline>
```

A fenti xml részletben látható, hogy a QEMU példányhoz egy IP cím – port párt definiáltunk, annak érdekében, hogy különböző portokat használjanak a különböző virtuális gép példányok. Ennek kezelésére és nyilvántartására egy socket-pool-t hoztunk létre, amely egy külön modulként szerepel a rendszerünkben. A socket-pool 999 portot képes kiajánlani, a használt portokhoz eltárolja a virtuális gép egyedi azonosítóját (UUID), a compute node nevét és IP-címét, amelyen a példány fut. Amennyiben a virtuális gép leáll, a port felszabadul.

A socket-pool a controller node-on fut, mivel a menedzsment szolgáltatások a controller csomóponton futnak. A socket-pool felel azért, hogy a virtuális gépek különböző portszámokat kapjanak. Így tehát a driver.py-ban található _get_guest_xml metódus a socket-pool-hoz fordul egy egyedi port azonosítóért.

A socket-pool-lal való kommunikáció érdekében a driver.py fájtl további két metódussal egészítettük ki. Ezek a _get_qmp_port, illetve a _free_qmp_port metódusok. A _get_qmp_port egy “get” üzenetet küld a socket-pool-nak az indítandó gép azonosítójával, melynek hatására a socket-pool lefoglal egy éppen szabad port-ot és a lefoglalt portszámot küldi el válaszul. A free_qmp_port egy “free” üzenetet küld a socket-pool-nak a leállítandó virtuális gép azonosítójával, melynek hatására a socket-pool felszabadítja a portot, és egy “ACK” üzenetet küld válaszként. A virtuális gépek leállításakor a _destroy metódus meghívódik, ennek megfelelően itt hívjuk meg a _free_qmp_port metódust.

Egy kliens alkalmazást (QMP menedzser) is készítettünk a socket-pool kézi vezérlésére és a benne lévő különböző adatok lekérdezésére. Az alkalmazás segítségével felszabadíthatunk portokat, vagy a portokat foglaló virtuális gépekről kaphatunk információt. Ez a segédprogram nagyban megkönnyítette a fejlesztési fázisban a működés tesztelését és ellenőrzését. Az alkalmazás használatát a mutatja be az *1*. és *2*. táblázat.

Free	
python client.py free u <uuid>	felszabadítja a virtuális gép által használt portot, amit UUID alapján azonosít
python client.py free a	felszabadítja az összes foglalt portot
python client.py free h <hostname>	felszabadítja az adott compute node-on használt portokat

1. táblázat – QMP menedzser port felszabadítás parancsok

Show	
python client.py show	kilistázza a használt portokat a hozzájuk tartozó összes kapcsolódó információval (port-number, uuid, hostname)
python client.py show u <uuid>	kilistázza a megadott uuid által azonosított VM portját
python client.py show p <portnum>	kilistázza a megadott porthoz tartozó információkat
python client.py show h <hostname>	kilistázza a megadott compute node által használt portokat

2. táblázat – QMP menedzser információt lekérdező parancsok

6.4 Osztott memória használata a virtuális gépekben

Virtuális gépek kommunikációja különböző módokon történhet. Az egyik megoldás az osztott memórián keresztüli kommunikáció. Ez azonban csak akkor működik, ha a virtuális gépek azonos gazda fizikai számítógépen futnak, vagy ha a fizikai gépek támogatják a távoli direkt-memóriaelérést, az RDMA-t (Remote Direct Memory Access), azonban a dolgozatunkban ennek megvalósítására nem térünk ki, ugyanis a projekt megvalósítása során nem állt rendelkezésünkre olyan hálózati interfész, amely támogatta volna a távoli memóriaelérést.

A virtuális gépek a host operációs rendszer által nyújtott osztott memória-területet használják, ami a /dev/shm/ könyvtár alatt található. Ez a könyvtár egy virtuális fájlrendszer, amely a gazda gép fizikai memóriájára van leképezve. A virtuális gépekben az osztott memória egy PCI eszközként jelenik meg. Ez a virtuális PCI eszköz három címezhető régióval (Base Address Registers - BAR) rendelkezik: konfigurációs régió, I/O portok és eszköz memória. Az eszköz memória az, amely tulajdonképpen a gazda operációs rendszer /dev/shm/ könyvtárában létrehozott fájlra képződik le. A virtuális gépekben ez az osztott memóriaterület a PCI eszközök leírófájljai között érhető el (/sys/devices/pci0000*).

Az osztott memóriát reprezentáló PCI eszköz kétféleképpen használható a QEMU példányban. Amennyiben a felhasználónak csak osztott memória területre van szüksége, akkor módja van azt szabadon használni, de ennél komplexebb viselkedést is elérhetünk. Lehetőségünk nyílik különböző interruptok küldésére a különböző példányok között, ehhez viszont megfelelő kernel-meghajtók implementálása szükséges.

A virtuális gépek osztott memóriás kommunikációjához az első lépés az, hogy a gépekhez osztott memóriát csatolunk, amit kétféleképpen tehetünk meg. Az egyik, hogy a virtuális gépet már az indításkor úgy paraméterezzük fel, hogy egy osztott memóriát felcsatolunk. A másik megoldás pedig az, hogy a virtuális gép futása közben csatolunk hozzá egy memória-területet a gazda gépről (hot-add). Ehhez az előbbiekben ismertetett QMP protokollt hívjuk segítségül.

A QEMU virtuális gépekkel a QMP-n keresztül lehet kommunikálni. Különböző megoldásokkal lehet hozzá csatlakozni, így telnet-tel netcat-tel, saját fejlesztésű hálózati

socketeket használó programokkal, esetünkben erre egy saját python programot készítettünk. A program csatlakozik a virtuális géphez a QMP protokollon keresztül, majd a kommunikáció JSON üzenetek segítségével történik.

A QMP kapcsolat felépülése után a virtuális gép oldalról egy státuszüzenet érkezik, benne a virtuális gép verziójával, majd "képességek egyeztetése" (capabilities negotiation) üzemmódba lép.

Ebben a módban csak a következő parancsot képes fogadni:

```
{ "execute": "qmp_capabilities" }
```

Így ahhoz, hogy a QEMU virtuális gépnek a QMP-n keresztül további parancsokat küldjünk, először ki kell adjuk a fenti JSON formátumú parancsot, mert ennek hatására lép ki a "képességek egyeztetése" üzemmódból. Ezek után már kiadhatjuk a további utasításokat. Esetünkben ez az osztott memória csatolásáért felelős parancs.

```
{ "execute": "device_add", "arguments": { "driver":"ivshmem" ,
"size": "64M" , "shm": "mem00001" } }
```

A parancs hatására egy a virtuális gép által használt fájl jön létre, amely 64 MByte méretű és egy fizikai memória-területre van leképezve, amely a gazda operációs rendszerben a /dev/shm/mem00001 fájlra képződik le. A parancsból az is kiolvasható, hogy a virtuális gép ezt a fájlt, mint osztott memóriát fogja használni, ez a driver mező ivshmem (intervm shared memory) értékéből derül ki. A többi virtuális gépnek, amelyek ugyanazon a fizikai gazdagépen futnak és szeretnék közösen ezt a memóriaterületet használni, ugyanezzel az azonosítóval (mem00001) kell a PCI eszközt felcsatolniuk.

Az osztott memória sikeres felcsatolásáról a QMP kapcsolaton keresztül is meggyőződhetünk az { "execute": "query-pci"} parancs kiadásával.

A parancs hatására kapott kimenet:

```
{"return": [
      {"bus": 0, "devices": ... [
      {"bus": 0, "qdev_id": "", "irq": 0, "slot": 6, "class_info":
      {"class": 1280, "desc": "RAM controller"},
      "id": {"device": 4368, "vendor": 6900},
      "function": 0,
      "regions": [
            {"prefetch": false,
```

```
                "mem_type_64": false,
                "bar": 0,
                "size": 256,
                "address": 2684354560,
                "type": "memory"},
                {"prefetch": true,
                "mem_type_64": true,
                "bar": 2,
                "size": 536870912,
                "address": 2147483648,
                "type": "memory"}]
                }]
        }]
}
```


Mielőtt azonban az osztott memóriát megpróbáljuk csatolni a virtuális géphez, egy kulcsfontosságú lépést el kell végeznünk. Ennek hiányában a gazda operációs rendszer nem fogja engedélyezni az osztott memória használatát a virtuális gépnek, azaz a QEMU folyamat nem fog tudni hozzáférni a gazdagép osztott memóriájához. Ez a lépés az Apparmor szolgáltatás kiiktatása a rendszerből.

Az Apparmor az Ubuntu operációs rendszer hozzáférés-vezérlő rendszere, amely tulajdonképpen a kernel egy olyan kiegészítése, ami képes a programok erőforrásokhoz való hozzáférését korlátozni.

Esetünkben az Apparmor teljes kiiktatása hozta meg a sikert. Ezt úgy oldottuk meg, hogy az update-rc.d segítségével letiltottuk az Apparmor rendszerindításkor történő elindulását. Ezt a következő két parancs kiadásával érhetjük el.

```
sudo systemctl stop apparmor.service
sudo update-rc.d -f apparmor remove
```

A virutális gépekkel való socket kommunikáció és az elosztott memória virtuális gépek közötti használatának megteremtésével a mérések elvégzéséhez szükséges környezet elkészült.

7 Mérések

7.1 Mérések specifikálása

A mérőkörnyezetben futtatott mérésekben megvizsgáljuk a virtuális gépek közötti kommunikációt mind a felhő környezet által alapértelmezetten nyújtott virtuális hálózaton, mind osztott memórián keresztül.

A mérések során mind az osztott memóriás, mind a virtuális hálózaton keresztüli kommunikációt csak azonos számítási node-on vizsgáltuk meg. Különböző számítási node-ok közötti méréseket nem végeztünk, ugyanis távoli osztott memóriás elérésre a laborkörnyezetben nem volt lehetőség, így nem tudtuk volna összehasonlítani a két fizikai gép közötti mérések eredményeit.

A mérések szempontjából fontosnak tartottuk, hogy valós feladatok nagyszámú elvégzését hajtsuk végre. Mivel a műveletek pontos időtartamára is kíváncsiak voltunk, ezért ehhez nagypontosságú időmérés volt szükséges.

7.2 A mérések implementálása

A mérések megvalósításakor a specifikációban megfogalmazott követelményeket implementáltuk. A méréseket egy programmodulon belül valósítottuk meg. A modult C++ nyelven írtuk, ennek a mérések időtartamának vizsgálatakor lesz jelentősége, ugyanis az adott műveletek időtartamát egy pontos óra szerint szerettük volna lemérni. Ehhez a processzor timestamp counter-ét (TSC) használtuk. Mivel a TSC értékének változásai a virtuális gépek esetén nem konzisztens a műveletek valódi időtartamával, ezért a Qemu virtuális gépeket olyan módban indítottuk, amely lehetővé tette, a gazda számítógép CPU-jának használatát. Ehhez az OpenStack Nova moduljának a /etc/nova/nova-compute.conf fájljának a szerkesztésére volt szükség. A fájlt a cpu_mode=host-passthrough bejegyzéssel kell kibővíteni. A TSC kiolvasását assembly nyelven tudjuk a legkönnyebben megvalósítani. A C++ nyelv lehetőséget teremt arra, hogy inline assembly hívásokat vigyünk a kódunkba absztrakciós rétegek igénybe vétele nélkül.

A mérőprogram képes megvalósítani a virtuális gépek közötti kommunikáció általunk vizsgált mindkét típusát. Amennyiben a programot a virtuális hálózaton keresztüli kommunikáció késleltetésének mérésére szeretnénk használni, akkor a kommunikáló virtuális gépeken egy-egy példányt kell belőle indítani. A programpéldányok ilyenkor kliens-szerver módban futnak. Amennyiben az osztott memória használatának késleltetéséről szeretnék képet kapni, akkor elég csak az egyik virtuális gépen futtatni a programot, ugyanis ebben az esetben nincs szükség egy fogadó fél jelenlétére, mivel az osztott memória közvetlenül elérhető.

A mérések esetén külön kezeltük az adatok írását és olvasását. Mivel valós felhasználást akartunk szimulálni, ezért különböző mérési mintákat hoztunk létre. A mintákban kombináltuk az adatokhoz való hozzáférés sorrendjét, és a hozzáférések között eltelt időt. Az adatok hozzáférése között eltelt időt aktív és passzív várakozással is szimuláltuk. Aktív várakozás esetén a folyamat két adathozzáférés között egy processzorigényes műveletet végez. Ez a művelet az első N prímszám meghatározása, majd a meghatározott prím négyzetgyökének meghatározása. Ebben az esetben arra számítottunk, hogy a virtuális gép ütemezője nem veszi el a futás jogát a mérőprogramtól, így a kontextusa is megmarad, így az osztott memóriába történő írás sebessége sem fog csökkenni nagy valószínűséggel. Passzív várakozás esetén a mérőprogam egy sleep hívást hajt végre, melynek hatására a virtuális gép ütemezője nagy valószínűséggel elveszi tőle a futás jogát. Ebben az esetben azt szerettük volna megfigyelni, hogy a folyamat elveszti a kontextusát a processzorban és így az osztott memóriába történő írások és olvasások gyorsasága is csökkenni fog az újra betöltéssel eltöltött plusz idő miatt. A jelenség, amire számítunk az úgynevezett cache hiba. Ha egy folyamat elveszíti a futás jogát, a CPU-ban más folyamat kerül a helyére. Arra számítunk, hogy a folyamat elveszíti a CPU gyorsítótárába betöltött adatait. Amikor újra visszakapja a CPU-t, akkor az általa hivatkozott memóriaterületek már nem lesznek a CPU gyorsítótárában, így a lassabb elérésű fizikai memóriához kell forduljon a szükséges adatokért. Ellenben ha a folyamat aktívan várakozik, akkor kisebb valószínűséggel veszíti el a processzort, így kevesebb cache hibába is ütközik.

Mérési minták:

- Soros olvasás / írás
- Soros olvasás / írás aktív várakozás

- Soros olvasás / írás passzív várakozás
- Véletlen olvasás / írás
- Véletlen olvasás / írás aktív várakozás
- Véletlen olvasás / írás passzív várakozás

A kommunikáció késleltetésének vizsgálatára 32 bites egészértékű (uint32_t) típusú változókkal végeztük a méréseket, továbbá a virtuális hálózati kommunikációt TCP portok segítségével valósítjuk meg.

7.3 Mérési eredmények

A mérőkörnyezetben végzett mérések során az osztott memórián keresztüli és a virtuális gépek közötti virtuális hálózaton keresztüli kommunikáció késleltetését vizsgáltam meg.

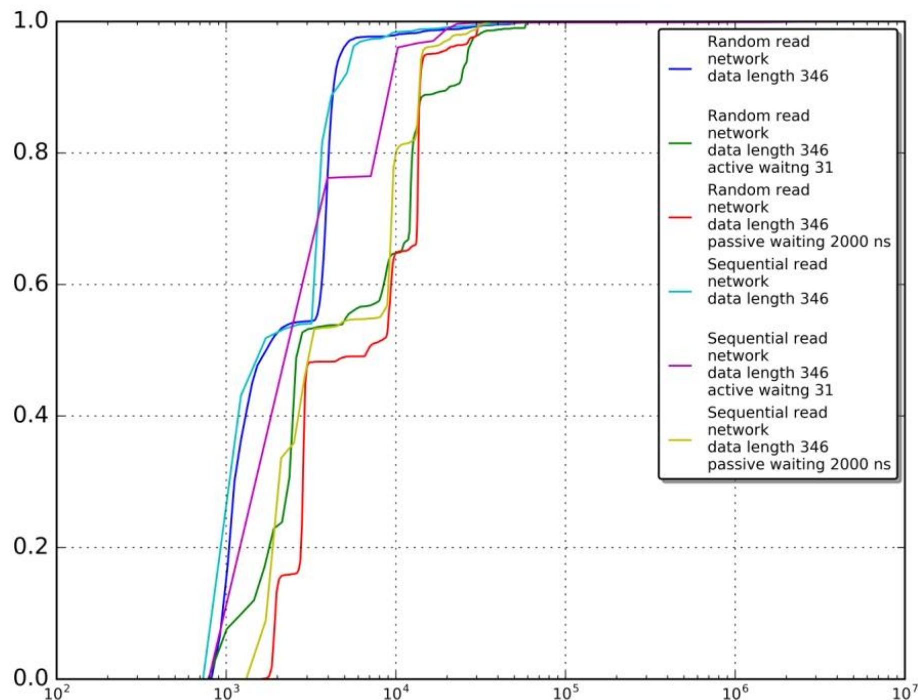
A mérőkörnyezetben végzett méréseket különböző adatméretekkel hajtottuk végre. Azonban az összehasonlítás végett a virtuális hálózat és az osztott memória használat késleltetésének vizsgálatát azonos adathosszakkal végeztük.

Mivel pazarlónak tartottuk, hogy egy hálózati csomagban csak egy rövid adatot vigyünk át, ezért kiszámoltuk, hogy hány 32 bites integer típusú változó fér bele egy hálózati csomagba. Az OpenStack virtuális hálózatában a maximális átviteli mennyiség (Maximum Transfer Unit - MTU) 1450 byte. Ez azért tér el az általánosan megszokott 1500 byte méretű MTU-tól, mivel az OpenStackben a VXLAN tunnelezési protokollt alkalmaztuk a virtuális hálózatok kialakítására, amelynek a többletköltsége 50 byte. Így az 1450 byte méretű csomagba 362 darab 32 bites integer fér bele. A TCP csomagok alap méretét figyelembe véve további 64 byte-ot veszítünk az átvihető adatok mennyiségéből, így összesen 346 darab 32 bites egészértékű változó fér el egy csomagban. Ezt szem előtt tartva az első mérésünket 346 darab integer átvitelével végeztük el. Az egyes mérések során 10000 műveletet végeztünk. Az aktív és passzív várakozások idejét megközelítőleg azonosra állítottuk. Ebben az esetben 2000 ns volt a passzív várakozás ideje, amely megfelel az első 31 prímszám meghatározásának és annak négyzetgyökeinek kiszámolásának időtartamának az általunk használt CPU-n.

A következő grafikonok az egyes műveletek késleltetését mutatják be. A grafikonok vízszintes tengelye a műveletek késleltetését ábrázolja nanosecundumokban,

míg a függőleges tengely pedig annak a valószínűségét, hogy a művelet adott időn belül befejeződött.

A 7. ábrán látható, hogy a hálózati olvasás esetén az adatok átviteléhez szükséges idő mediánja körülbelül 1100 ns a várakozás nélküli esetekben, az aktív és passzív várakozások esetén pedig az 1500 ns-os értéket közelíti meg. Az is látható, hogy egy adat átvitele megközelítőleg 1 valószínűséggel végbemegy 15000 ns alatt.



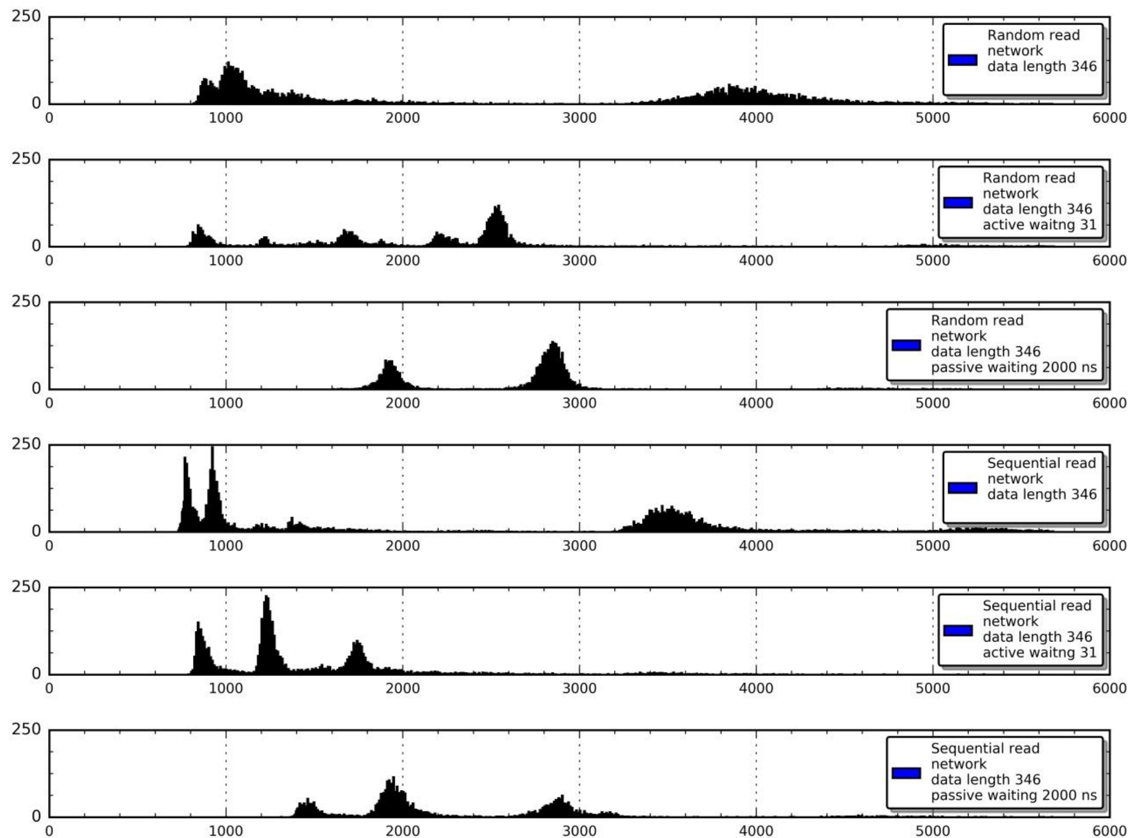
7. ábra – Hálózati olvasást megvalósító mérések 346 darab integer átvitelével

Az eredmények könnyebb értelmezésének érdekében a CDF diagramokon kívül hisztogramokat és percentiliseket megjelenítő diagramokat is készítettünk. A 8. ábra megmutatja, hogy az egyes késleltetésekhez hány művelet tartozik, míg a 9. ábra bemutatja, hogy az adatok hány százaléka alacsonyabb egy bizonyos késleltetésnél.

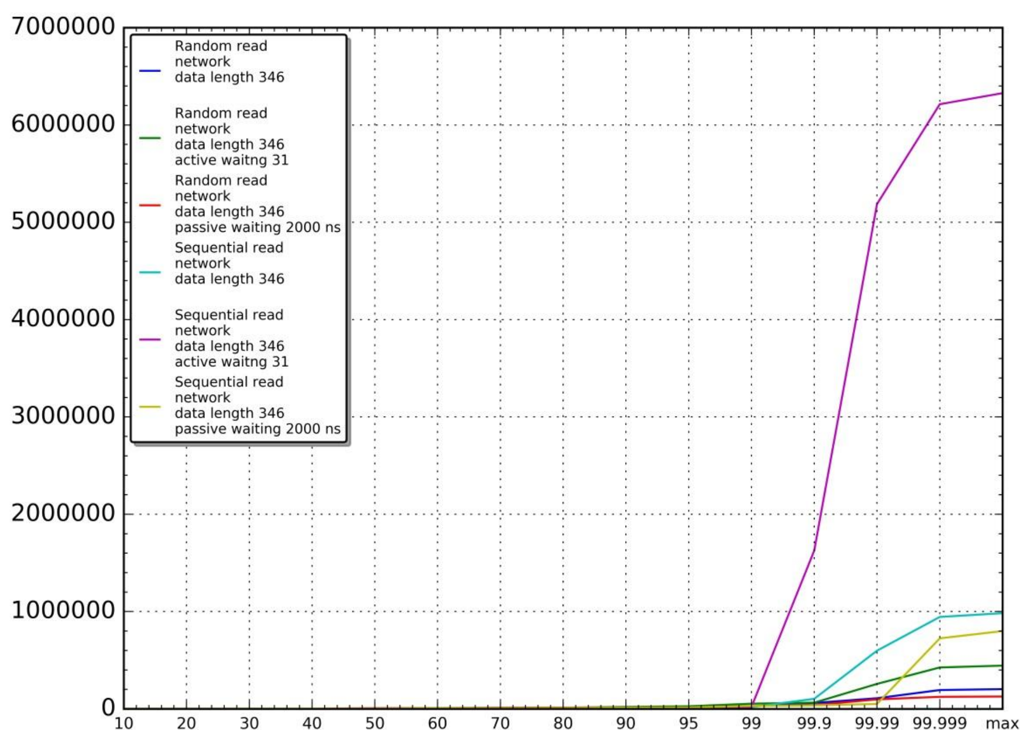
A hisztogramokon a függőleges tengely reprezentálja, hogy az adott késleltetésekhez hány darab művelet tartozik, míg a vízszintes tengelyen a késleltetések értékei olvashatóak.

A percentiliseket ábrázoló diagram azt mutatja be, hogy hány százalékos biztonsággal tudjuk megállapitani, hogy a késleltetések egy bizonyos érték alatt vannak.

A 8. ábrán látható, az adott mérési mintákhoz tartozó késleltetések eloszlása, a mérési mintáktól függően 0,5 – 1 mikroszekundumos jellemző eltolódással, míg a 9. ábráról leolvashatjuk, hogy a szekvenciális írás esetén 1%-a, míg a többi mérési minta esetén mért késleltetéseknek csak 0.1%-a tér el nagyban az átlagos értékektől.

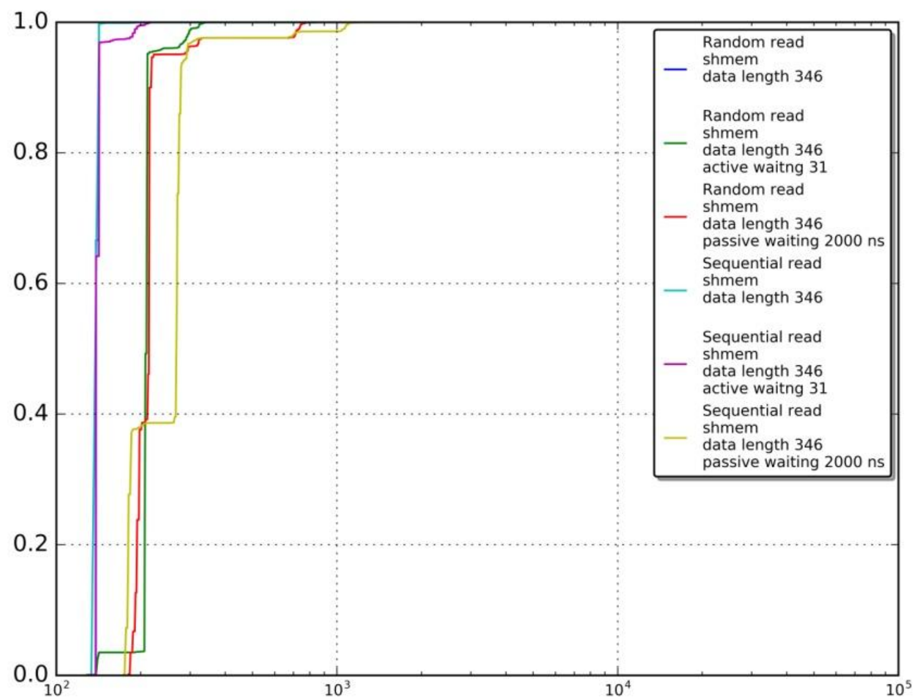


8. ábra – Hálózati olvasást megvalósító mérések 364 darab integer átvitelével – hisztogram



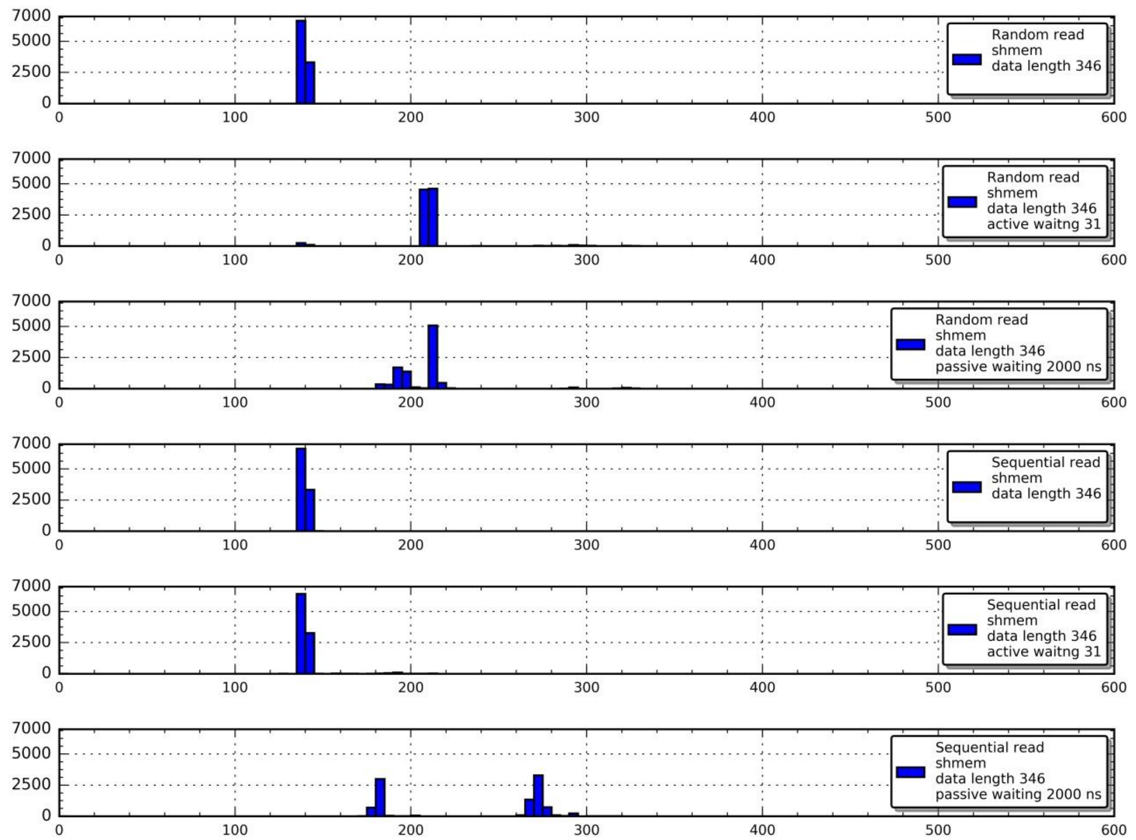
9. ábra - Hálózati olvasást megvalósító mérések 364 darab integer átvitelével – percentilisek

Az osztott memóriás olvasás esetét a 10. ábra mutatja be. Látható hogy a várakozás mentes műveletek késleltetésének mediánja megközelítőleg 140-150 ns között található. Továbbá ebbe a tartományba esik az aktív várakozást megvalósító soros olvasás mediánja is. A többi eset mediánja körülbelül a 200 és 280 ns közé eső tartományba esik.

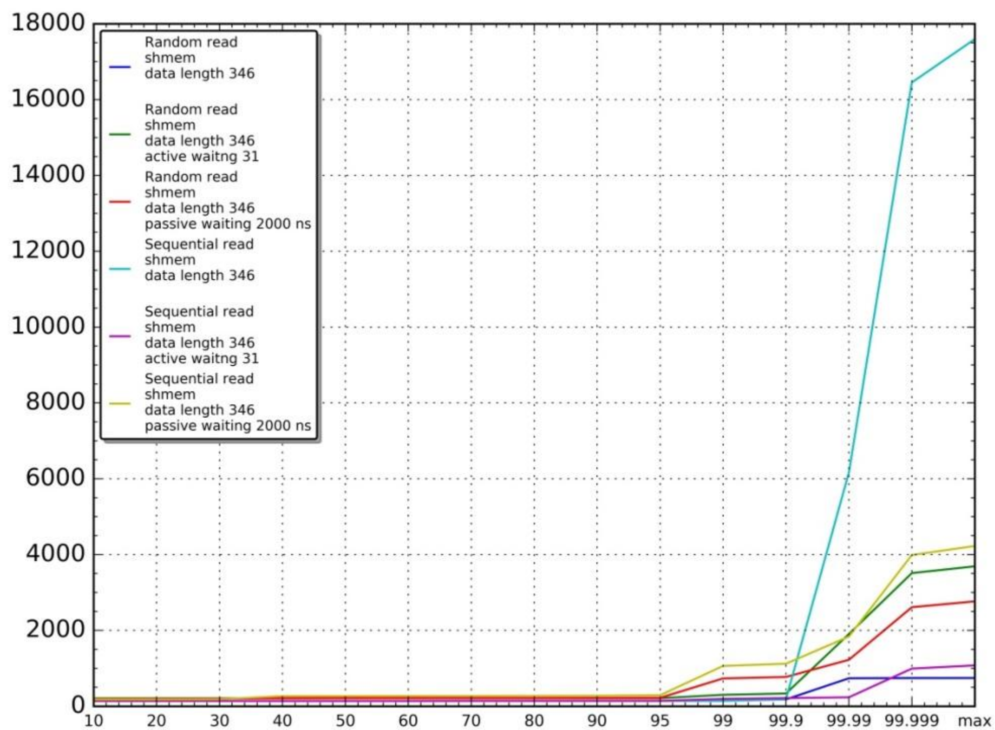


10. ábra – Osztott memórián keresztüli olvasást megvalósító mérések 346 darab integer kiolvasásával

A 11. ábrán láthatjuk az osztott memóriás olvasás késleltetésének eloszlását, látható, hogy a mért késleltetések valóban 140-150 ns közöttiek. A 12. ábra pedig bemutatja, hogy a mért adatok körülbelül 5%-a mutat nagyobb kiugrást az átlagos értékekhez képest, azonban ezek a kiugrások lényegesen kisebbek, mint a virtuális hálózaton keresztüli kommunikáció mérése esetén.



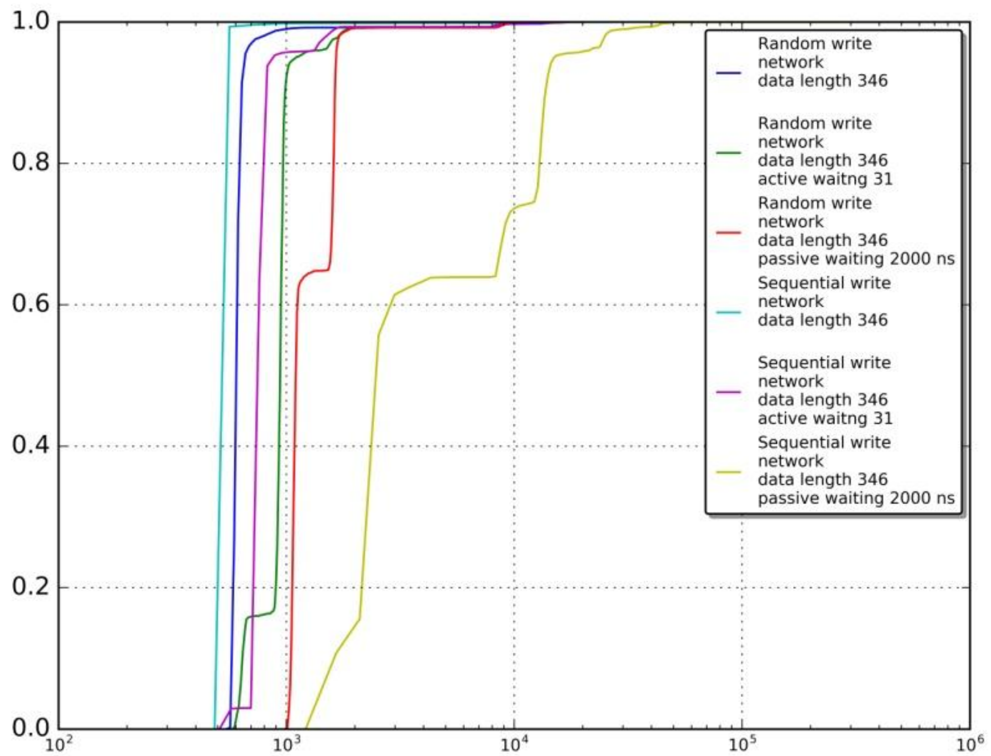
11. ábra **– Osztott memórián keresztüli olvasást megvalósító mérések 346 darab integer kiolvasásával – hisztogram**



12. ábra – Osztott memórián keresztüli olvasást megvalósító mérések 346 darab integer kiolvasásával - percentilisek

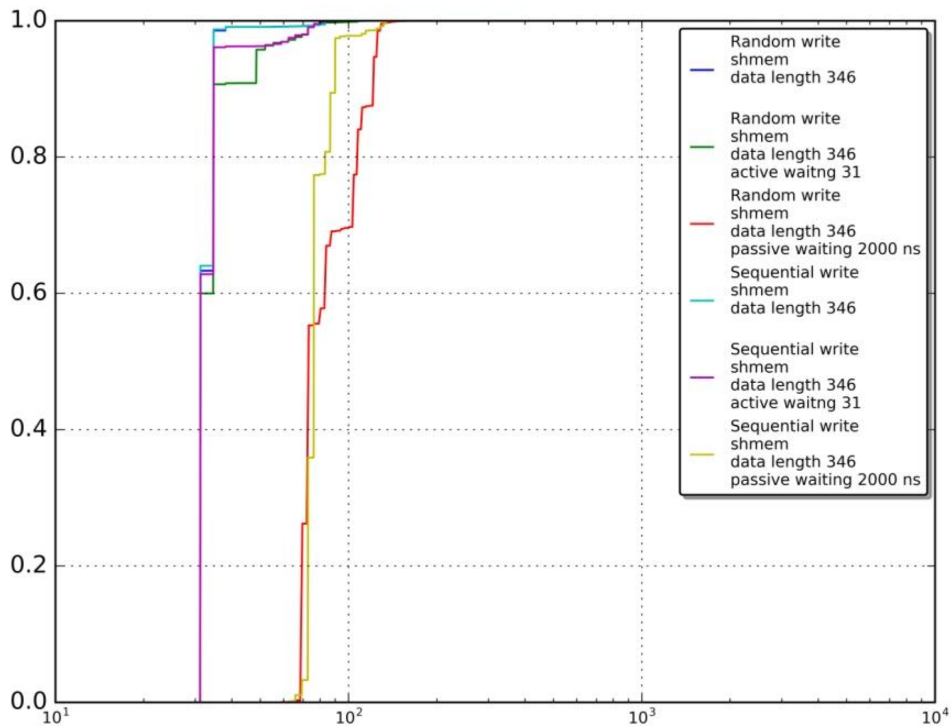
A további eredmények bemutatását csak a CDF diagramok segítségével mutatjuk be.

A hálózaton keresztüli adatok írását a 13. ábra szemlélteti. Szembetűnő, hogy ebben az esetben a legkisebb költségű műveletek is megközelítik az 500 ns-ot. A különböző mérési esetekben pedig a műveletek késleltetésének mediánjainak átlaga pedig körülbelül 1000 ns környékén lehet.



13. ábra – Virtuális hálózaton keresztüli írás 346 darab integer átvitelével

Az osztott memórián keresztüli írások eredményeit a 14. ábra szemlélteti. Ebben az esetben a várakozásaink beigazolódtak a mérési mintákat illetően. Látható, hogy a várakozás mentes és az aktív várakozást megvalósító mérések körülbelül ugyanolyan teljesítményt mutatnak, míg a két passzív várakozást megvalósító mérés eredményei ettől elmaradnak.

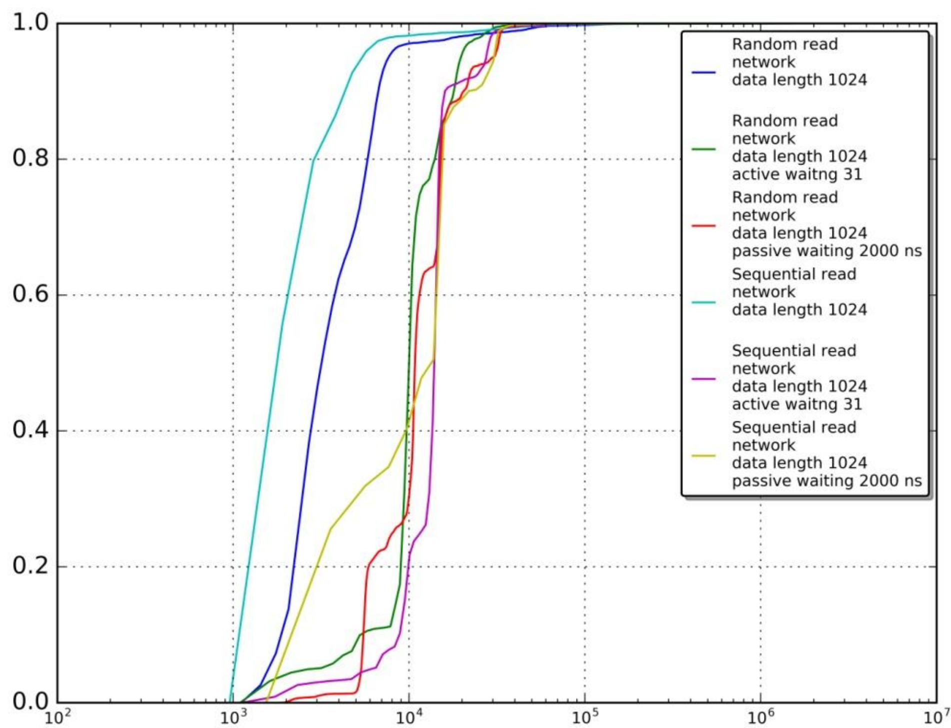


14. ábra – Osztott memórián keresztüli írás 346 darab integer beírásával

A mérések során természetesen mind a gazda, mind pedig a virtualizált operációs rendszer ütemezője kis mértékben megváltoztathatja a várt eredményeket.

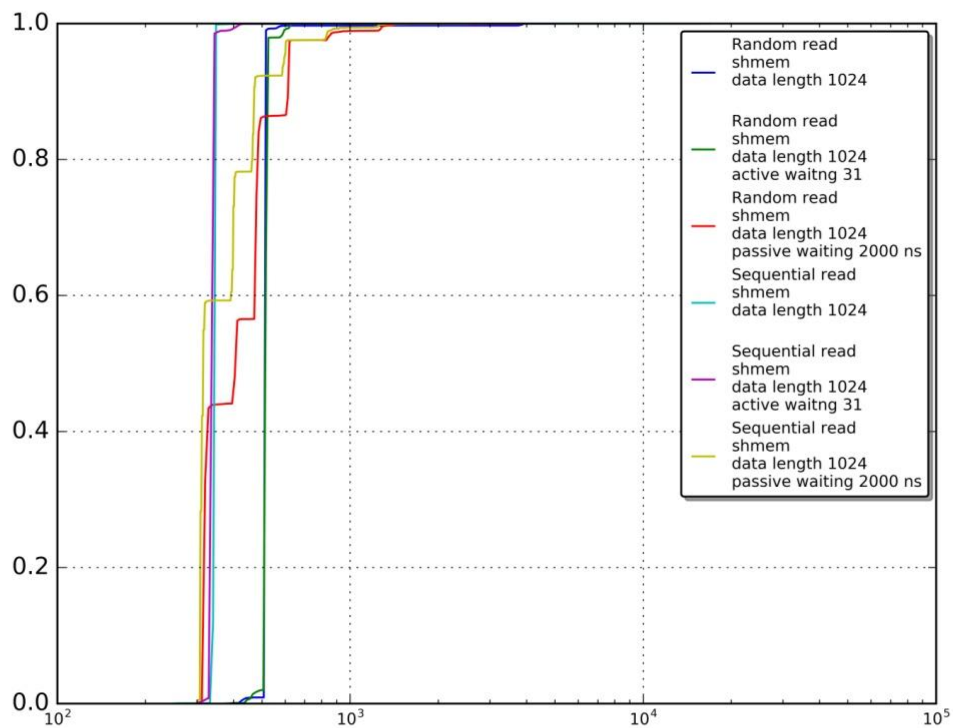
Mivel kevésnek találtuk az egy hálózati csomag méretének megfelelő adatmennyiséget, ezért nagyobb mennyiségű adattal is megvizsgáltuk a rendszerben fellelhető késleltetéseket. Ebben az esetben 4 kilobyte-nak megfelelő adatmennyiséggel végeztük a méréseket. Ez 1024 darab 32 bites integer írását vagy olvasását jelenti. Ebben az esetben is 10000-szer ismételtük meg a műveleteket.

A 15. ábrán látható, hogy a várakozás mentes soros és véletlenszerű olvasás mediánja körülbelül 1500 ns körül található, míg a többi esetben ez 10 és 11ezer ns között található. Az is leolvasható az ábráról, hogy az egyes műveletek körülbelül 100ezer ns alatt biztosan végrehajtódnak.



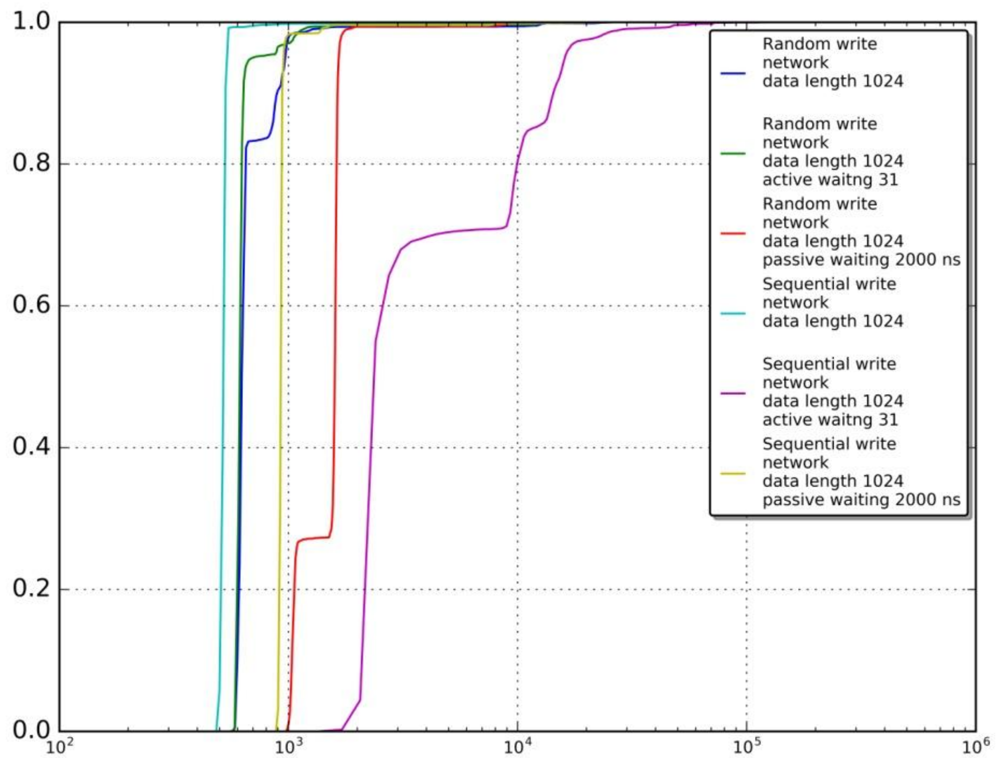
15. ábra – 4 kilobyte méretű adatmennyiség olvasása virtuális hálózaton keresztül

A 16. ábrán 4 kilobyte mennyiségű adat osztott memórián keresztüli kiolvasása látható. Ebben az esetben látszik, hogy az adatok kiolvasása nagy valószínűséggel 4000 ns alatt megtörténik. Viszont a mérési eredmények nagy része 1000 ns alatt marad.



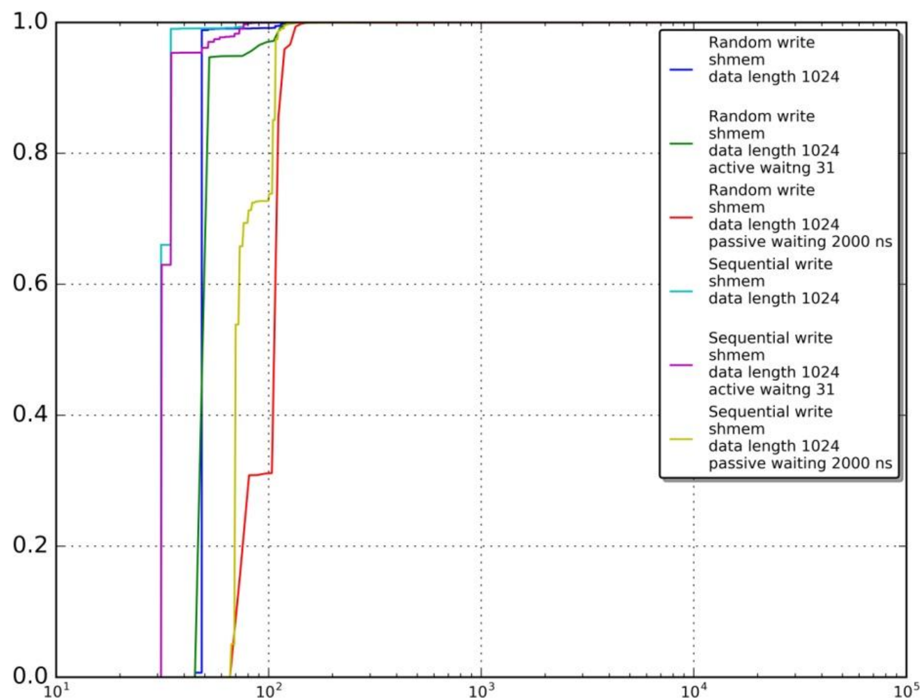
16. ábra – 4 kilobyte méretű adatmennyiség olvasása osztott memórián keresztül

A 17. ábrán 4 kilobyte mennyiségű adat hálózaton keresztüli írásának késleltetéseit láthatjuk. Ebben az esetben a műveletek késleltetésének mediánjainak átlaga körülbelül 1000 ns-nál találhatóak. Az is látható, hogy az összes művelet megközelítőleg 70ezer ns-nál nem tartott tovább.



17. ábra – 4 kilobyte méretű adatmennyiség írása a virtuális hálózaton keresztül

A 18. ábrán a 4 kilobyte méretű adategységek osztott memóriába történő írásának késleltetései láthatóak. Látható, hogy a műveletek késleltetésének nagy többsége nem lépi át a 200 ns-ot.



18. ábra – 4 kilobyte méretű adatmennyiség írása az osztott memóriába.

7.4 A mérési eredmények kiértékelése

A mérési eredményekből kiderül, hogy a felhő rendszer által nyújtott virtuális hálózaton történő kommunikáció késleltetés szempontjából sokkal gyengébb eredményt ad az osztott memórián keresztüli adatátvitelnél. Azt is megfigyelhettük, hogy az adatokhoz történő hozzáférés nem függ a hozzáférések sorrendjétől, viszont az adatok mérete megváltoztathatja az eredményeket. Azt is megfigyelhetjük, hogy az operációs rendszer ütemezője is nagyban befolyásolja az eredmények kialakulását. Azonban a legfontosabb észrevétel, hogy az osztott memóriás kommunikáció az elvárásainknak megfelelően jobban teljesített a virtuális hálózaton keresztüli társánál. A méréseket megvizsgálva az osztott memórián keresztüli kommunikáció körülbelül 10-szeres sebességjavulást mutatott a virtuális hálózaton történő kommunikációhoz képest.

8. Összefoglalás

A felhő rendszerekben a virtuális gépek szeparáltan futnak, a közöttük történő kommunikáció pedig a felhő által biztosított virtuális hálózaton keresztül valósul meg. Két virtuális gép kommunikációjára osztott memórián keresztül is lehetőség van.

Dolgozatunkban megvizsgáltuk az osztott memórián keresztüli kommunikáció OpenStack felhőbe való integrálását, bemutattuk az architektúrába való illesztését és implementációját, amelyet egy új modul hozzáadásával értünk el. Mérésekkel összehasonlítottuk a hálózaton keresztüli és az osztott memórián keresztüli kommunikáció hatékonyságát és teljesítményét.

A mérések elvégzéséhez egy C++-ban implementált mérőprogramot készítettünk, amely segítségével az osztott memóriás és hálózati kommunikáció késleltetését egyaránt meg tudtuk vizsgálni. Az eredmények kiértékelésére egy python programot készítettünk, amely a numpy matematikai és a matplotlib grafikonkészítő könyvtár segítségével a mért adatokból CDF, hisztogram és percentiliseket ábrázoló diagramokat készít.

A mérési eredmények kiértékelésével bebizonyítottuk, hogy az osztott memóriás kommunikáció körülbelül 10-szeres teljesítménybeli növekedést eredményezett a virtuális hálózaton történő kommunikációhoz képest. A projektet azonban még nem zártuk le. A jövőben szeretnénk a méréseket megismételni különböző számítási komponensek között is. Erre sajnos a dolgozat írásakor nem volt lehetőségünk, ugyanis nem álltak rendelkezésünkre távoli memóriaelérést megvalósító hálózati interfészek.

Irodalomjegyzék

[1] Peter Mell, Tim Grance, "The NIST Definition of Cloud Computing (version 15)", National Institute of Standards and Technology (NIST), Information Technology Laboratory (www.csrc.nist.gov), October 7, 2009.

[2] https://en.wikipedia.org/wiki/Cloud_computing - utoljára megtekintve 2016.10.27.

[3] https://www.openstack.org/software/ - utoljára megtekintve 2016.10.27.

[4] http://docs.openstack.org/juno/install-guide/install/apt/content/ch_overview.html - utoljára megtekintve 2016.10.27.

[5] Balla Dávid - BME VIK Virtualizált környezetben futtatott alkalmazások mérés alapú vizsgálata - szakdolgozat 2015. december

[6] https://en.wikipedia.org/wiki/Libvirt - utoljára megtekintve 2016.10.27.

[7] T. Suezawa, „Persistent execution state of a java virtual machine", In Proceedings of the ACM 2000 Conference on Java Grande, 2000.

[8] JACKSON, Keith R., et al. Performance analysis of high performance computing applications on the amazon web services cloud. In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010. p. 159-168.

[9] Ren, Y., Liu, L., Zhang, Q., Wu, Q., Guan, J., Kong, J., ... & Shao, L. (2016). „Shared Memory Optimizations for Inter-Virtual-Machine Communication", *ACM Computing Surveys (CSUR)*, *48*(4), 49.

[10] https://wiki.openstack.org/wiki/HypervisorSupportMatrix - utoljára megtekintve 2016.10.27.