# 1. fejezet

# Bevezetés

# 2. fejezet

# Felhasználói dokumentáció

## 2.1 Felhasznált szoftverek

A projekt 2 különböző al-projektből áll:

1. A hoszt operációs rendszeren futó monitorozó applikáció (VSMT\_App)
2. A virtuális gépen futó adatgyűjtő és továbbító kliens (VSMT\_Client)

Mindkét projekt C++ 17 nyelvet és Qt 6.8 keretrendszert használ, viszont a monitorozó applikáció a grafikai felülethez a QtQuick keretrendszert és a QML (Qt Modeling Language) nyelvet is felhasználja.

Az applikációt főleg beágyazott felhasználásra szántam, nagy mennyiségű adatot kell gyorsan kezelnie, mivel akár több virtuális gép teljesítmény metrikáit is soft-real-time vizualizálnia kell, ezért a választásom a C++ nyelvre esett. A nyelv képes magas teljesítményű programok létrehozására, viszont rendelkezik kényelmi funkciókkal (pl. objektum orientált eszközök, template meta-programozás stb..) melyekkel egyszerűsítik a kódírást. Grafikai keretrendszernek a Qt-t választottam, mert megbízható, gyors és grafikai elemek mellett adattárolása, szerializálása és aszinkron kommunikációra is ad lehetőséget. A könyvtár két opciót is nyújt a GUI komponensek implementálására: Qt Widgets és Qt Quick, az előbbi C++ alapú, főleg asztali applikációk létrehozásához használják, a Qt Quick pedig QML alapú mely egy deklaratív nyelv grafikus elemek definiálásához, segítségével dinamikus, platform független GUI-kat valósíthatunk meg. A választásom a Qt Quick-re esett.

## 2.2 A probléma megfogalmazása

Kritikus rendszerekben, ahol kulcsfontosságú a funkcionálisan biztonságos működés, egyre nagyobb számban jelennek virtuális gépek. Ezen gépek használatával magasabb kiber- és funkcionális biztonság szintet lehet elérni, mint azonos funkciót megvalósító külső hardver vagy virtualizálatlan natív szoftver komponensekkel. A virtuális gépek izolációt nyújtanak potenciális hibák esetén, állapotuk gyorsan újratölthető pillanatképek (snapshot-ok) segítségével, valamint hardver függetlenül módon képesek működni.

A virtuális gépek legnagyobb hátránya a natív szoftverekkel szemben a teljesítményben rejlik. Általában elmondható, hogy a kritikus rendszerekben nem csak a biztonság, hanem a gyorsaság is egy fontos tényező. Számos módszer van a virtuális gépek gyorsítására például a hypervisorok vagy a containerek melyek csak egy applikációt és dependenciáit virtualizálják, viszont még nem áll rendelkezésre egy centralizált felület, melynek segítségével több virtuális gépet lehet monitorozni egy központi hoszt számítógépről, szakdolgozatommal erre a problémára keresek megoldást.

## 2.3 Fogalmak és platformok

Az applikáció mind virtuális gépen, mind hoszt operációs rendszeren két platformmal kompatibilis: Yocto Automotive Grade Linux és QNX.

### 2.3.1 Yocto Automotive Grade Linux

Yocto keretrendszerben épített Automotive Grade Linux disztribúció, ezentúl AGL.

#### 2.3.1.1 Yocto

A Yocto Project egy nyílt forráskódú, együttműködés alapú projekt, mely segít a fejlesztőknek testreszabott, hardver architektúra független Linux alapú operációs rendszerek létrehozására. Egy projekt építés végeredménye egy bináris úgy nevezett disztribúció.

A Yocto keretrendszer alapjai a receptek. Egy recept beállítások és feladtatok listája melyek alapján felépíthető és telepíthető egy csomag. Általában egy recept több más receptre hagyatkozik, melyek felépítik és konfigurálják az adott csomag függőségeit. A receptek rétegekbe (layer) vannak rendezve, attól függően, hogy milyen célokat látnak el, például a Qt könyvtár közös verziófüggetlen eszközei és azok függőségei a „meta-qt” rétegben találhatóak, a Qt6.8 verzió ezeken felül nyújtott ezközei pedig a „meta-qt-6.8” -ben.

Egy Yocto projekt (build) receptekből az azokat felölelő rétegekből és konfigurációs fájlokból áll. Egy projektnek tartalmaznia kell: a cél hardver BSP-t (Board Support Package) és a kernel recepteket, valami az azon applikációk és függőségeik receptjét, melyeket használni kívánunk a kész rendszerben.

A virtualizálni kívánt rendszerek építésre nagyon jól használható a Yocto Project, mivel egy adott funkcionalitást megvalósító rendszerhez, lehetséges csak a funkciót megvalósító modulok és dependenciáik beleépítése a rendszerképbe, ezzel egy kisebb, fókuszáltabb rendszert létrehozva.

Felhasznált verzió: kirkstone (4.0.7)

#### 2.3.1.2 Automotive Grade Linux (AGL):

Egy nyílt forráskódú, kollaboráció alapú Linux disztribúció, autóipari felhasználásra tervezve, a Yocto projekt terjedése eredményezte a létrejöttét. Egy monolitikus Linux kernelt tartalmaz, ami azt jelenti, hogy bár logikailag fel vannak osztva az alrendszerei, köztük hozzáférési védelem nincs. Minden modul úgynevezett „kernel space” -ben fut, mely egy speciális területe a rendszer memóriának, ahol a kernel folyamatok futnak, végrehajtási privilégiumokkal.

Az AGL projekt fő fókusza, a jármű-beli „infotainment”, vagyis egy olyan felület, mely működteti a hang- és kép hangrendszereket (pl. rádió, navigáció, zenelejátszó, stb...).

Felhasznált AGL verzió: needlefish

Felhasznált kernel és kernelmodul verzió: 5.10.41-yocto-standard

#### 2.3.1.3 Quick Emulator (QEMU)

Egy nyílt forráskódú, ingyenes emulátor és virtualizátor. Segítségével indíthatunk virtuális gépeket AGL felett. Két módban használható: „user emulation” és „system emulation”, előbbi segítségével futtathatunk olyan Linux programokat, melyek nem a gép CPU architektúrájára lettek fordítva. Rendszer emulációs módban egy teljes virtuális gépet futtathatunk akár más operációs rendszerrel és CPU architektúrával, mint a gépünk, továbbiakban ezt a módot fogom használni.

### 2.3.2 QNX RTOS

A QNX RTOS vagy QNX real-time operating system, egy kereskedelmi felhasználásra tervezett, valós idejű operációs rendszer. Fejlesztését egy kanadai szoftver vállalat a Quantum Software Systems kezdte 1980-ban. Főleg beágyazott biztonság kritikus rendszerekben használják, például autóiparban is mert ASIL D (Automotive Safety Integrity Level) minőségű biztonsági tanúsítvánnyal rendelkezik. Mikrokernel architektúrát használ, amely egy moduláris kernel felépítést eredményez. Maga a kernel, csak a legfontosabb funkcionalitásokat nyújtja, minden más szerepkört folyamatok töltenek be, melyeket a mikrokernel ütemez. Ez a modularitás segít egy operációs rendszer kialakításában, amely flexibilis, átlátható és könnyen tovább fejleszthető. Egy új kernelmodul könnyen tesztelhető, mert egy folyamatként fut a rendszerben, telepítése nem igényli az egész kernel újraépítését. Az esetleges hibák nagyon egyszerűen kezelhetőek, egy modul hibás működése esetén a mikrokernel nem érintett, ezért képes lekezelni a hibát anélkül, hogy újraindulna. Ennek következménye, hogy amíg van egy ép mikrokernel, a hibás rendszerkomponensek gyorsan visszaállíthatóak.

### 2.3.3 Hardver

A megcélzott hardver minden esetben AArch64 (ARM64). A beágyazott rendszerek körében ez a legelterjedtebb processzor család. Ezen processzorok utasításkészlete úgy nevezett RISC (Reduced Instruction Set Computing) tervezési stratégián alapszik. Lényege, hogy több magasan specializált instrukció helyett, kevesebb általános, de jól optimalizált utasítást tartalmaz. Ennek eredménye, hogy egyszerűbb az utasítások dekódolása, több regiszter vagy gyorsítótár telepíthető egy chipre. A szakdolgozatom esetében a felhasznált Renesas R-Car H3 egylapkás rendszer (SOC – System On Chip), is ARM64 processzort alkalmaz (erről a rendszerről a fejlesztői dokumentációban fogok mélyebben írni).

## 2.4 Konfiguráció és használat a virtuális gépen (guest OS)

Bemutatási célokra nem ajánlom a telepítésnek ezt a formáját, sok konfigurációval jár és nagyon sok időt vesz igénybe. Lehetőségünk van az applikációt egy Linux laptopról indítani és előre létrehozott rendszerképeket virtualizálni (vagy virtuális gépek nélkül használni, ha csak a hoszt gép teljesítményére vagyunk kíváncsiak), erről a telepítési módszerről [2.5.3 Ubuntu Linux platform](#_2.5.3_Ubuntu_Linux) szekcióban írok többet.

Ahhoz, hogy egy virtuális gépet monitorozzunk szükség van a kliens szolgáltatásra, mely a virtuális gépen fut és összegyűjti az elérhető teljesítmény metrikákat a gépen, majd azokat szerializálva továbbítja az applikáció felé. Első sorban szükségünk van egy rendszerképre mely tartalmazza ezt a kliens szolgálatást. 2 operációs rendszerre elérhető a kliens: AGL-re és QNX-re. QNX rendszerképet, csak megfelelő licenccel tudunk létrehozni, a telepítést erre a platformra nem fogom bemutatni.

Elérhetőek előre felépített rendszerképek melyek tartalmazzák a klienseket, ezekről többet [2.5.3 Ubuntu Linux platform](#_2.5.3_Ubuntu_Linux) szekcióban írok. A továbbiakban a rendszerképek felépítéséről lesz szó, ezekhez szükségünk van egy Ubuntu 24.04 LTS operációs rendszert működtető laptopra.

### 2.4.1 AGL platform

#### 2.4.1.1 Operációs rendszerkép létrehozása

Először telepítenünk kell a Yocto környezetet, ehhez le kell töltenünk a következő eszközöket: git, tar, pyhton3.10, curl, tree, gcc, make, chrpath, diffstat, g++, gawk, lz4, python3-distutils, repo.

Töltsük le az AGL gyűjteményt a következő parancsokkal:

|  |
| --- |
| **mkdir** -p AGL**/**needlefish # mappa létrehozása  **cd** AGL**/**needlefish # belépés a mappába  repo init -b needlefish -u https**://**gerrit.automotivelinux.org**/**gerrit**/**AGL**/**AGL-repo  repo sync # letöltjük a gyűjteményt |

Majd készítsük el az AGL projektünket:

|  |
| --- |
| **cd** AGL**/**needlefish  source meta-agl**/**scripts**/**aglsetup.sh -m h3ulcb-kf -b build |

A munkám forráskódja mellett találhatóak konfigurációs fájlok is melyek megkönnyítik a telepítést: az AGL mappán belül a local.conf és a bblayers.conf is felhasználható. Másoljuk be a 2 fájlt az ***AGL/needlefish/build/conf*** mappába. A bblayers.conf definiálja a szükséges rétegeket, a local\_client.conf pedig a recepteket. A szakdolgozat mappájában található a kliens rétege, mely az applikáció forráskódját és a receptjét tartalmazza. A réteg a ***meta-vsmt*** mappában van, melyet másoljuk az ***AGL/needlefish*** könyvtárba. A local\_client.conf fájt nevezzük át client.conf-ra. Lépjünk vissza az ***AGL/build*** mappába és kezdjük meg a rendszerkép építést:

|  |
| --- |
| bitbake agl-image-minimal |

Egy rendszerkép létrehozása több órát is igénybe vehet, valamit az építés hibákba ütközhet, ekkor azt újra kell indítani.

#### 2.4.1.2 Kernel konfiguráció

A hoszt és virtuális gép kommunikációjához konfigurálnunk kell a kernelt, ezt legegyszerűbben a „menuconfig” eszközzel tehetjük meg, egy grafikus interfészen keresztül:

bitbake -c menuconfig virtual/kernel

--menuconfig screenshot goes here--

A fel és le nyilakkal válthatunk a menü pontok között, a bal, jobb nyilakkal pedig az operációt választhatjuk ki (operáció sáv alul), melyet az enterrel hajtatunk végre. A követkető opciókat kell bekapcsolnunk (navigáljuk az opcióhoz majd nyomjuk meg az „**y”** billentyűt):

* Networking support -> Networking options -> Virtual Socket Protocol
* Networking support -> Networking options -> virtio transport for virtual sockets
* Device Drivers -> VHOST drivers -> vhost virtio-vsock driver
* Device Drivers -> Virtio drivers -> PCI driver for virtio devices

A konfiguráció végeztével építsük újra a kernelt:

bitbake -C compile virtual/kernel

Ha kész, másoljuk ki a rendszerképet a jelenlegi könyvtárunkba és tömörítsük ki:

**cp** **\**

tmp**/**deploy**/**images**/**h3ulcb**/**agl-image-minimal-h3ulcb-kf.wic.xz **.**

#### 2.4.1.3 Virtuális gép konfigurálás és indítás

A létrehozott operációs rendszerképet QEMU segítségével virtuális gépként indítjuk a következő konfigurációkkal:

qemu –flags-

Miután betöltöttünk a virtuális gépbe, be kell írnunk a felhasználónevet: ***root***. Majd indítanunk a „VSMT\_Client” alkalmazást. (Érdemes a háttérben indítani.)

**cd** /bin

./VSMT\_Client&

## 2.5 Konfiguráció és használat a hoszt gépen (host OS)

### 2.5.1 AGL platform

#### 2.5.1.1 Hoszt rendszerkép építés

Ha még nem tettük végezzük el a [2.4.1.1 Operációs rendszerkép létrehozása](#_2.4.1.1_Operációs_rendszerkép) és [2.4.1.2 Kernel konfiguráció](#_2.4.1.2_Kernel_konfiguráció) -ban leírt utasításokat. A szakdolgozat AGL könyvtárából másoljuk át a gépünk AGL/build mappájába a local\_host.conf fájlt és nevezzük át local.conf -ra. Majd építsük meg a hoszt rendszerképet.

|  |
| --- |
| bitbake agl-demo-platform  **cp** **\**  tmp**/**deploy**/**images**/**h3ulcb**/**agl-demo-platform-h3ulcb-kf.wic.xz . |
|  |
|  |

#### 2.5.1.1 Hoszt rendszerkép véglegesítése

Ahhoz, hogy a hoszt gépen virtuális gépen indítsunk, először át kell másolnunk rá a rendszerképét.

--mount and copy commands go here--

#### 2.5.1.2 Indítás

Indítsuk el a virtuális gépet és klienst a [2.4.1.3 Virtuális gép konfigurálás és indítás](#_2.4.1.3_Virtuális_gép) szerint.

### 2.5.3 Ubuntu Linux platform (bemutatási célokra)

A telepítés ehhez a formájához csak egy Linux operációs rendszert futtató laptopra lesz szükségünk, én az Ubuntu 24.04 LTS disztribúciót választottam. Kezdésképp telepítenünk kell a Qt6.8 keretrendszert, ehhez létre kell hozni egy Qt fiókot, ha még nincs. Az online telepítő elérhető innen: [Qt-online-installer.](http://qt.io/download-qt-installer-oss)

--screenshot--

Válasszuk ki megfelelő platformot és töltsük le a telepítőt, majd indítás után a „Select Components” ablakban a következőket válasszuk ki és telepítsük őket: --screenshot—

A QtCreator IDE használatával indíthatjuk el az applikációt a forráskódból. Az „Open Project” opcióval válasszuk a CMakeLists.txt file-t, a forráskód mappájából. A következő projekt konfigurációs ablakon válasszuk ki a Qt6.8 keretrendszert, majd fordítsuk és indítsuk az applikációt.

Lehetőségünk van laptopról indítani virtuális gépeket az applikáció funkcióinak teszteléséhez. Ehhez le kell töltenünk az QEMU virtualizátort, a következő parancs lefuttatásával:

sudo apt-get install qemu-system

Navigáljunk az előre létrehozott rendszerképek mappájába és indítsuk el a virtuális gépet:

Fontos kiemelni, hogy a rendszerképek emulált környezetben fognak futni, ha ezt a telepítési módszert választjuk (az emuláció és virtualizáció közti különbségeiről a fejlesztői dokumentációban lesz szó). A gép betöltése után a [2.4.1.3 Virtuális gép konfigurálás és indítás](#_2.4.1.3_Virtuális_gép) szerint tudjuk folyatni az indítást.

### 2.5.3 Használat

#### 2.5.3.1 Főmenü

Indítás után megjelenik a főmenü, melyben a következő funkciók érhetőek el: új kliens konfiguráció hozzáadása, meglévő kliens konfiguráció szerkesztése és kapcsolódás egy klienshez.

Ezek mellet az applikációban mindig látható egy oldalsáv a jobb oldalon, mely kilistázza a kapcsolódott klienseket, a nevükkel, legutolsó mért processzor és memória terhelésükkel százalékos formátumban. Ezek az oldalsáv elemek kattinthatóak, kattintásra megnyílik a kliens összegző oldal. Az első kliens az oldalsávban a speciális hoszt (host) kliens, ez azt a gépet jelenti, ahol futtatjuk az applikációt, a hoszt klienshez mindig csatlakoztatva vagyunk, róla lekapcsolódni nem lehet. Ha valamilyen oknál fogva megszűnik a kapcsolat egy kliensel, az applikáció visszalép a főképernyőre és az adott kliens eltűnik az oldalsávról.

#### 2.5.3.2 Kliens összegző oldal

Az oldal bal szélén általános információkat láthatunk a kliensről (operációs rendszer, CPU architektúra stb..). Alatta az összesített hálózat használatot láthatjuk, letöltésre és feltöltésre bontva, bájtokban. Ez a grafikai elem szintén kattintható és a hálózat használat oldalra visz. Alul a lekapcsolódás gomb található, mely segítségével az applikáció lekapcsolódik a kliensről és visszalép a főmenübe. A hoszt kliensről nem lehet lekapcsolódni.

Középen további információkat találhatunk a kliensről, 3 teljesítménymérőt melyek százalékos formában vizualizálják az összegzett: CPU, memória és háttértár használatot, ezek az elemek szintén kattinthatóak: a CPU használat, a folyamat és a háttértár használat oldalakra vezetnek. Alattuk egy táblázatot láthatunk, mely megjeleníti a rendszeren futó folyamatokat.

#### 2.5.3.3 Hálózat használat oldal

Az oldalon a hálózati interfészek vannak listázva. Minden interfészhez elérhető a neve, típusa (ethernet, wifi stb..) és két grafikonon az elmúlt 20 mérés alatt kapott (RX) és küldött (TX) bájt, a legutolsó mérés bájtjai láthatóak a grafikon felett is. Amikor egy adott interfész le- vagy fel kapcsolódik a kliensre, a lista frissül.

#### 2.5.3.4 CPU használat oldal

Az oldal tetején egy grafikon látható, melyen vizualizálva van az összesített CPU terhelés. Alatta magokra lebontva látszik a kihasználtság, mindegyik külön grafikonon. A grafikonok dinamikusan frissülnek amikor mérési adat érkezik, az elmúlt 20 mérési eredményt jelenítik meg.

#### 2.5.3.5 Folyamat (process) oldal

Itt a kliensen futó folyamatokról kaphatunk információkat, egy táblázat formájában. A táblázat megegyezik az összegző oldalon találhatóval, mind megjelenített adat, mind nyújtott funkciók tekintetében. Az alábbi információk érhetők el egy folyamatról:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PID | Mag használat | Memória használat | Állapot | Parancs | Argumentumok |
| A folyamat azonosítója. | A mag kihasználtsága mely végrehajtja a folyamatot, százalékos formában (lehetséges 100%-nál magasabb kihasználtság, ekkor több mag hajta végre a folyamatot). | A folyamat memória használata az összes memóriához képest. | A folyamat állapota:   * Futó * Alvó * Várakozik * Zombi * Megállt * Halott * Tétlen | A végrehajtott parancs. | A parancs argumentumai. |

A fejléc akármely elemére kattintva, rendeződik a kiválasztott oszlop szerint a táblázat, először növekvő, ismételt kattintás esetén csökkenő sorrendbe. A táblázat alapértelmezett rendezése: parancs szerint növekvő.

#### 2.5.3.6 Háttértár oldal

Ezen az oldalon egy listában láthatjuk, kliensen található háttértár lemezeket. Minden lemezről elérhető a neve, a használt és összes tárhely megabájt formájában, a fájlrendszer típusa és a felcsatolás (mount) címe.

#### 2.5.3.7 Kliens konfigurálás

A főoldalon található egy lista, melyben az eddig konfigurált kliensek vannak. A lista fejlécében a „+” gombbal vehetünk fel új klienseket, melyek megjelennek a lista végén „New Client” névvel. A listaelemeken 3 művelet hajtható végre: törlés az „X” gomb segítségével, szerkesztés az „Edit” gombbal és csatlakozás a „Connect” gombbal. A törlés gomb hatására, a kliens eltűnik a listából, a csatlakozás gomb pedig csatlakozási kísérletet eredményez, mely, ha sikeres az a kliens meg is jelenik a jobb oldalsávban. A szerkesztés gombra kattintva, egy új ablak nyílik meg, itt adhatjuk meg a kliens konfigurációját: nevét és VSOCK címét: kontextus azonosítóját és port-ját (a VSOCK címzésről a fejlesztői dokumentáció 3.1.3 VSOCK szekciójában írok). Miután beállítottuk a számunkra megfelelő értékeket a mentés (Save) gombbal menthetünk ezzel bezárva az ablakot, ha mégsem szeretnénk menteni a mégse (Cancel) gomb visszaállítja a legutolsó mentett konfigurációt. Új konfiguráció felvételekor és módosítás esetén minden konfiguráció mentésre kerül a háttértárba. Az első kliens a listában speciális hoszt (Host) kliens mely azt a gépet jelöli, ahol az alkalmazás fut, ezt törölni, megváltoztatni nem lehet, mindig csatlakoztatva van.

# 3. fejezet

# Fejlesztői dokumentáció

## 3.1 Fogalmak

### 3.1.1 AArch64 és a virtualizáció

#### 3.1.1.1 A virtualizáció és az emuláció

A virtualizáció és emuláció is hasonló problémára keres megoldást: egy olyan virtuális rendszer futtatását, mely úgy működik mintha közvetlenül a hardveren futna, viszont egy másik gép által létrehozott és felügyelt virtuális környezetben működik. A virtuális gépet, vendég gépnek (guest machine) a hoszt gépet, gazda gépnek (host machine) is szokták nevezni.

Az emuláció lényege, hogy egy fizikai gépen egy másik gép fizikai felépítését szimuláljuk, szoftveres megoldással. Ez nagyon hasznos és kényelmes megoldás, ha egy teljesen más fizikai architektúrájú rendszeren szeretnénk dolgozni, anélkül, hogy az adott hardvert felhasználnánk, például egy Android alkalmazás fejlesztőnek, nem kell magán az okostelefonon tesztelnie az applikációt, ezt a fejlesztői gépén egy emulált Androiddal is meg tudja tenni. Ez a megoldás, magas teljesítmény költséggel jár, így esetemben nem volt alkalmazható.

Maga virtualizáció egy absztrakciós réteg, mely szeparálja a hardvert, az operációs rendszertől. Fő különbség az emulációhoz képest, hogy a hardver architektúra egy virtuális gép esetén ugyanaz, mint a hoszt gép architektúrája. Virtualizáció esetén az erőforrás szeparáció a legfőbb szempont, nem pedig egy idegen architektúra szimulálása. A virtualizáció legfőbb eleme a hypervisor, nélküle a következőkben taglat módszerek nem megvalósíthatóak. Alapvetően több virtualizációs megoldást létezik, én a 2 legelterjedtebbet fogom taglalni: a teljes virtualizáció és a paravirtualizáció.

A teljes virtualizáció egy olyan virtualizálási módszer, mely úgy nevezett bináris fordítást használ (binary translation), ami lehetőséget add arra, hogy kritikus kódrészleteket átfordítson virtuális gép instrukciókészletéből, a hoszt gépébe. Ennek eredményeként bizonyos utasítások majdnem natív gyorsasággal futhatnak le a virtuális gépen. Ez esetben a vendég rendszer azt hiszi, hogy a hardveren fut és közvetlenül a hardverrel kommunikál, nem tud a bináris fordítás működéséről. A virtualizálás ilyen módszere átláthatóbb: nem kell módosítani a vendég gépet a működéshez, viszont hardver támogatást igényel.

Paravirtualizáció esetén a virtuális gép „tudja”, hogy nem közvetlenül a hardveren fut. Így jobb teljesítményt érhetünk a teljes virtualizációval szemben, mert a virtuális gép képes közvetlenül a hardverrel kommunikálni, bináris fordítás nélkül. A teljes virtualizációval szemben előny, hogy nem kell hardver támogatás, viszont a vendég gép kerneljét, fel kell készíteni arra, hogy virtuális fog futni. Ez a megoldás kevesebb absztrakcióval jár, ezért magasabb teljesítményt eredményez, feláldozva az átláthatóságot.

A diagram of a diagram

Description automatically generated with medium confidence

#### 3.1.1.2 ARM magok: A, M és R

Az ARM Cortex processzor családban, több típusú mag található. Különböző mag kategóriák, más-más célokat látnak el. Ahhoz, hogy megértsük, hogyan is történik a virtualizáció processzor szinten, először ezeket a mag típusokat kell megismernünk. Az ARM magok alapvetően 3 családba kategorizálhatóak: Cortex-A, Cortex-M és Cortex-R.

A Cortex-A vagy csak A magok, úgy nevezett „Rich OS” futtatnak, funkciókban gazdag operációs rendszerek, esetükben az erőforrás kezelés és több tevékenység végzése egy időpontban (multitasking), fontosabb mint a valós idejű működés. Ilyen rendszerek az általános felhasználású eszközökön jellemzőek, pl.: asztali gép, infotaintment eszköz stb… Az applikációt futtató AGL is A magon fog futni.

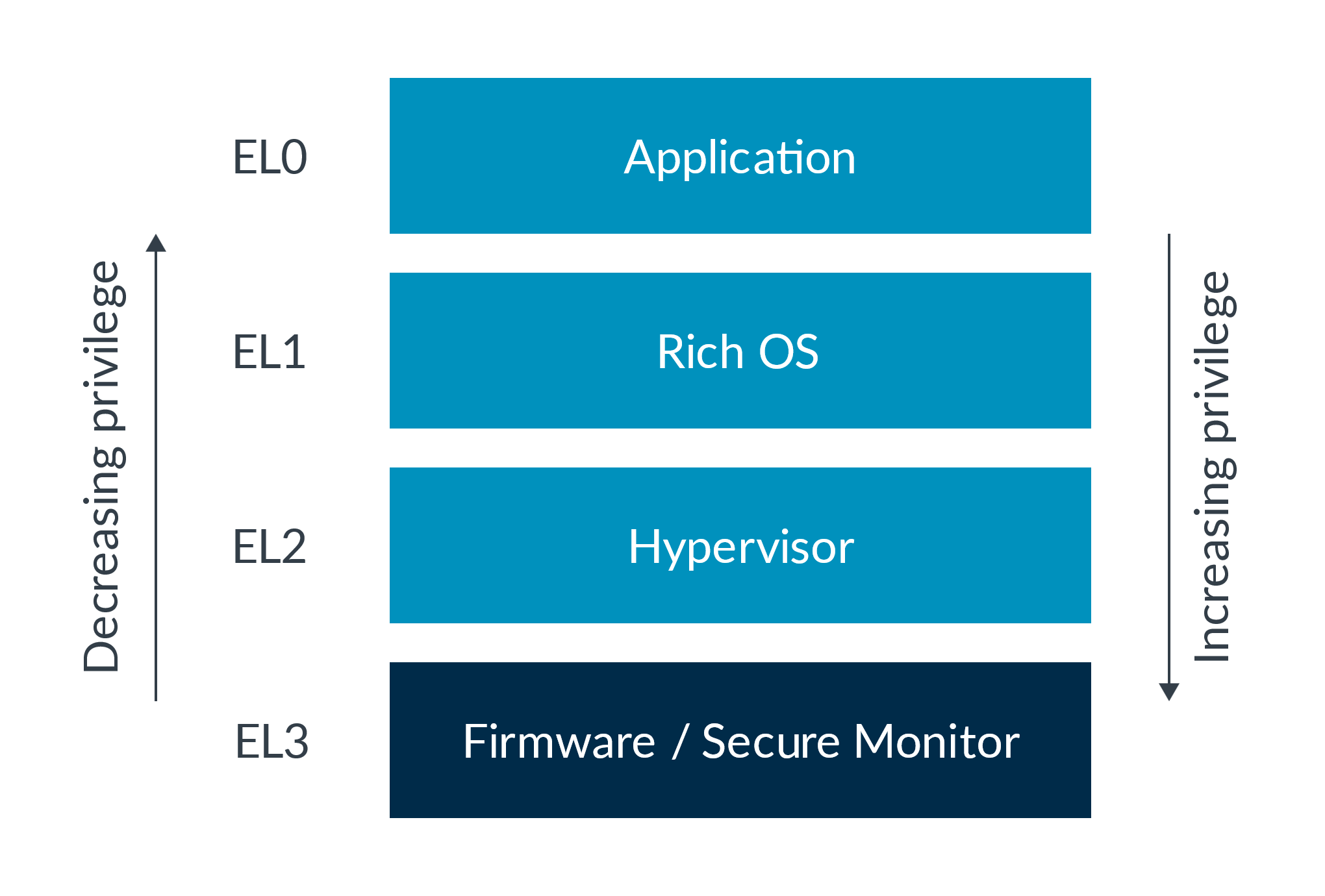
Az M magok a mikrokontroller piacra, lettek fejlesztve. A legenergiahatékonyabb processzor család, általában alacsonyabb teljesítmény igényű funkciókat töltenek be, pl.: szenzorok vagy kamerák vezérlése.

A Cortex-R mag család a magas teljesítményigényű real-time (valós idejű) funkciókat hajtja végre, pl.: fékvezérlők, merevlemez vezérlők. Ezeket a funkciókat valós idejű operációs rendszereken (RTOS) keresztül valósítják meg. A funkcióknak kritikus időbeli követelményeik vannak, ha egy fékvezérlési funkció nem hajtódik végre, az időkeretben fatális következményekkel járhat.

#### 3.1.1.3 Renesas RCar-H3

#### 3.1.1.4 Kivétel szintek (Exception levels)

A modern szoftverek modulárisak, minden modulnak különböző mélységű hozzáférése van a processzor és rendszer erőforrásokhoz, szabályszerű erőforrás hozzáférés biztosításához jött létre AARCH64 architektúrában a kivétel szintek (Exception levels) koncepciója. 4 kivételi szint van meghatározva, EL0-tól EL3-ig, csökkenő hozzáférésekkel.



EL0 szinten fut minden felhasználói applikáció, ezt a szinten „userspace” -nek is szokták nevezni. EL1 szinten úgy nevezett „Rich OS” fut, ez az operációs rendszer (pl. AGL) kernel folyamatait jelenti. EL3 a legmagasabb biztonsági szint, itt a „Trustzone” applikációk pl. firmware-ek működnek. EL2-ben egy speciális program a hypervisor fut.

### 3.1.2 Hypervisor

Hypervisornak nevezünk minden olyan hardver vagy szoftver komponenst, mely virtuális gépek létrehozásával és erőforrásmegosztásával foglalkozik. Egy hypervisor virtualizálja az összes hardver erőforrását a gépnek, ezzel tered adva egy vagy több virtális gép futtatásának. Két típusra oszlanak a hypervisorok: type 1 és type 2.

A type 1 hypervisorok közvetlenül a hardveren futnak és hozzáférnek az erőforrásokhoz (CPU, memória stb…). Nincs szükség hoszt operációs rendszerre a virtuális gépek hardver hozzáférését a hypervisor kezeli, ez nagyon gyors végrehajtást eredményez, viszont szüksége van egy külső rendszerre, mely konfigurálja a virtuális gépeket, melyek nélkül nem futtatható semmilyen applikáció a hardveren. Az autóiparban kifejezetten kutatott terület, mivel több kritikus funkció más-más operációs rendszeren futtatható, anélkül, hogy mindegyik saját ECU-t (Electronic Control Unit) vegyen igénybe.

Ezekkel ellentétben a type 2 hypervisor-ok, avagy „hosted” hypervisorok, egy folyamatként futnak az operációs rendszerben. Szükségük van egy hoszt operációs rendszerre, ez magasabb teljesítmény költséggel jár. Esetükben nincs szükség külső rendszerre, a hoszt OS konfigurálja a virtuális gépeket. A hoszt rendszer képes natívan applikációkat futtatni, ha szükséges. Legfőképp akkor használják, ha van olyan funkció mely natívan fut a hoszt-on, pl.: asztali alkalmazások esetén, mint a VirtualBox vagy VMware.

Hypervisor típusok


Description automatically generated

#### 3.1.2.4 KVM

KVM vagy Kernel Based Virtual machine egy Linux kernel modul, mely segítségével a kernel egy type 1-es hypervisor-ként funkcionál. A KVM egy QEMU-val közösen teremti meg a virtuális gépek futtathatásához használható környezetet, ezért QEMU/KVM -ként is szokás hivatkozni rájuk. A QEMU rendszer emulációs módja, gyakorlatilag egy dinamikus fordító: amikor egy kódrészlethez először érkezik a virtuális gép működtetése közben, átalakítja a gazda gép utasítás készletéhez illővé. Ezt a folyamatot a TCG (Tiny Code Generator) végzi. Először egy belső reprezáncióvá alakul a kód (IR – Internal Representation), melyen optimalizációk történnek. Ekkor a kódrészlet a hoszt hardverén közvetlenül futtatható, a KVM segítségével. AGL felett a QEMU/KVM fogja betölteni a hypervisor szerepét.

A diagram of a server

Description automatically generated

### 3.1.3 VirtIO VSOCK

Amikor egy virtuális gépet konfigurálunk lehetőségünk van eszközöket definiálni, például hálózati interfészeket vagy grafikai vezérlőket, melyeket a gép úgy érzékel mintha fizikailag csatlakoztatva lennének. A legegyszerűbb és legrégebb óta használt módszer az eszköz emuláció: tisztán szoftveresen hozzuk létre az eszközt, hardveres támogatás nélkül. Ez egy kényelmes megoldás, gyakorlatilag akármilyen eszközt definiálhatunk a virtuális gépnek, nem kell egyezzen a hoszt gépen létező eszközzel. Ez a flexibilitás viszont nagyon nagy teljesítmény költséggel jár, beágyazott rendszerekben nem alkalmazható hatékonyan. Erre a problémára jött létre a VirtIO interfész. Ez egy szabványosított interfész, mely segítségével, definiálhatunk olyan eszközöket, amik „tudják”, hogy virtuális gépben futnak és együttműködnek a hypervisor-ral, hogy az eszközműveletek közvetlenül a hoszt hardverén történjenek, ezzel drasztikusan felgyorsítva az operációkat emulált eszközökhöz képest.

A VSOCK vagy VM Socket API egy VirtIO eszköz, mely segítségével, a hagyományos socket interfészen keresztül kommunikálhatnak virtuális gépek és a hoszt gép. A kommunikáció lehet kapcsolat-orientált mint a TCP, vagy kapcsolat nélküli datagram protokoll, mint az UDP. A VSOCK kommunikáció független a virtuális gép hálózati rétegétől, ez lehetőséget nyújt olyan virtuális gépek beállítására melyek, csak a VSOCK kommunikációban vesznek részt.

Egy gépet egy előjel nélküli 32 bites egész szám (uint32) azonosít: az úgy nevezett „context identifier” vagy CID, minden virtuális gép egyedi CID-el kell, hogy rendelkezzen melyet a */dev/vsock* fájlból olvashatunk ki és a virtuális gép indításakor expliciten adhatunk meg. A hypervisor CID értéke mindig 0, a hoszt gépé 2, az 1 pedig egy fenntartott érték, ezeket virtuális gépnek nem oszthatjuk ki. A VSOCK kommunikációhoz szükségünk van a kontextus azonosítón felül, egy port számra is, ez a két érték határoz meg egy VSOCK címet, segítségükkel használhatjuk a C standard könyvtár socket interfészét az üzenet küldéséhez, fogadáshoz.

A screenshot of a computer

Description automatically generated

A hoszt gépen futó monitorozó applikáció és a virtuális gépen működő kliens VSOCK-on keresztül kommunikálnak egymással.

## 3.2 Teljesítmény metrika és rendszer információ gyűjtés

### 3.2.1 Teljesítmény metrikák és rendszer információk

Minden kliensről elérhető információ két csoportra van osztva: teljesítmény metrika és rendszer információ, a két csoportra általánosan metrikaként fogok hivatkozni. Közöttük a legfőbb különbség, hogy míg a rendszer információk futás közben nem változhatnak, a teljesítmény metrikák igen, ezt azt eredményezi, hogy azokat folyamatosan frissíteni kell. A rendszer információk a SystemInfo struktúrában, a teljesítmény metrikák pedig a RuntimeMetric struktúrában találhatóak, a kettő metrika általánosított változata a Metric osztály, melyben egy típusbiztos unióként (std::variant) van eltárolva a struktúrák valamenyike.

#### 3.2.1.1 Rendszer információk (SystemInfo)

Ebben a struktúrában találhatóak azok az információk melyekről, tudjuk, hogy nem tudnak megváltozni futás közben, ezért csak egy alkalommal kell lekérdezni őket: a kliens inicializációjánál. A struktúrában string-ként van eltárolva az operációs rendszer platform (pl.: Linux) és a konkrét disztribúció (pl.: AGL). Ezeken felül található tartalmaz egy CpuInfo struktúrát, mely a CPU modelljét, sebességét (GHz-ben) és a magjai számát foglalja magába.

#### 3.2.1.2 Teljesítmény metrikák (RuntimeMetric)

Itt a futási teljesítményről találhatunk információkat. A RuntimeMetric osztály több másik struktúra kompozíciója, információt tárol: memória információt (MemoryInfo), és az összes futó folyamat (ProcessInfo), csatlakoztatott háttértár (StorageInfo) és elérhető hálózati interfész információját (NetworkInterfaceInfo) egy-egy C++ vektorban. Valamint még egy vektort mely a CPU magok kihasználtságát tárolja, százalékban, lebegőpontos formátumban.

##### 3.2.1.2.1 Memória információ (MemoryInfo)

Ebben a struktúrában két 32-bites előjel nélküli egész található: az összes felhasználható memória a gépen és a jelenleg használatban lévő. Mindkettő kilobájtos nagyságrendben értelmezendő.

##### 3.2.1.2.1 Folyamat információ (ProcessInfo)

A ProcessInfo struktúra egy adott folyamatról tárol információkat. Egy folyamat minden lehetséges állapota a Status enumerációban van számontartva. A lehetséges állapotok:

* Fut (Running)
* Alszik (Sleeping)
* Várakozik (Waiting)
* Zombi (Zombie)
* Megállt (Stopped)
* Halott (Dead)
* Tétlen (Idle)
* Ismeretlen (Unknown)

A struktúrában el van tárolva a folyamat azonosító (pid), az állapot, a folyamat által használt memória és CPU mag használat százalékosan, a végrehajtott parancs és annak argumentumai string formájában.

##### 3.2.1.2.1 Háttértár információ (StorageInfo)

Egy csatlakoztatott háttértárat ír le a StorageInfo struktúra. Tartalmazza a nevét, a felcsatolási pontot és a fájlrendszer típusát string formában, az összes elérhető és felhasznált tárhelyet megabájt nagyságrendben 64bites előjel nélküli egészekben tárolva.

##### 3.2.1.2.1 Hálózati interfész információ (NetworkInterfaceInfo)

Ebben a struktúrában egy elérhető hálózati információja található. Az interfész típusa mely QNetworkInterface::InterfaceType enumeráció, fontosabb elemei: Loopback, Ethernet, Wifit, CanBus, Virtual. A NetworkInterfaceInfo struktúrában rögzítve van az interfész neve és a legutolsó mérés óta fogadott és továbbított adatmennyiség bájtokban (rxBytes és txByes).

### 3.2.3 /proc fájlrendszer

A /proc egy úgy nevezett folyamat információs pszeudó filerendszer, mely nem tartalmaz „valós” fájlokat, csak információt a rendszeren futó folyamatokról és magáról rendszerről (pl. memória, háttértár, hardver konfigurációk). A fájlrendszer tekinthető egy interfésznek is, mely segítségével hozzáférünk a kernel belső adatstruktúráihoz. A rendszer eszközök jelentős része az itt található fájlokból nyeri az információkat. A fáljrendszer elérhető AGL és QNX alatt is.

### 3.2.4 Információk begyűjtése

A következő fejezetekben a rendszer információk és teljesítmény mérőszámok gyűjtésének menetéről lesz szó. Az adatokat a ResourceMonitor osztály gyűjti, két alosztállyal rendelkezik: CpuMonitor, mely a CPU információiért és kihasználtsági adataiért felel és NetworkMonitor, mely a hálózati interfészekről nyújt információt.

#### 3.2.4.1 SystemInfo

A platform és disztribúció elérhető a QSystemInfo osztályból, melyet a Qt keretrendszer nyújt. A CpuInfo adattagjait a /proc/cpuinfo pszeudófájlból nyerem ki. A fájl a CPU magok adatait tartalmazza soronként a következő szerkezetben: <kulcs> : <érték>. A „model name” kulcs alatt található a CPU mag model neve és sebessége „@” -al elválasztva. A „processor” kulcs alatt található az adott mag száma 0-től indexelve. A magok számát az utolsó bejegyzés „processor” értéke + 1 adja. A fájl értelmezését a SystemMonitor::CpuMontior osztály gatherProcessorInfo metódusa végzi, mely egy SystemInfo objektummal tér vissza.

#### 3.2.4.2 MemoryInfo

Az adattagokat a /proc/meminfo feldolgozásával kapom meg. A fájl szerkezet itt hasonló a /proc/cpuinfo-hoz: soronként kulcs-érték párok. Az összes memóriát a „MemTotal” kulcs adja. A felhasználható memória a „MemAvailable” kulcs alatt található, ez alatt azt a memóriát értjük, mely lefoglalható akármely folyamat által, de nem feltétlen szabad, gyorsítótárak és pufferek által használt memória is ide tartozik. A használt memória az összes, mínusz felhasználható memóriából adódik. A fájl elemzése a SystemMonitor osztály gatherMemoryInfo metódusában történik.

#### 3.2.4.4 Mag kihasználtság

Ehhez az adatokat a /proc/stat fájlból nyerem ki. Soronként tartalmaz információkat a magokról. Számomra a mag név utáni 4. érték a tétlenül töltött idő (idle), az 5. érték IO várakozással töltött idő (iowait) és az összegzett mag idő (az értékek összege: totalTime) volt releváns. A mag IO várakozásnál is tétlen, ezért a tétlen időhöz adva kapjuk meg a teljes tétlen időt:

idleTime = idle + iowait

Az előző mérés értékei legyenek p\_idleTime és p\_totalTime (előző mérés hiányában 0), ekkor a két mérés közti tétlen idő delta\_idleTime, az összes idő pedig delta\_totalTime:

delta\_idleTime = p\_idleTime - idleTime

delta\_totalTime = p\_ totalTime – totalTime

A két mérés közti átlagos mag kihasználtság pedig (százalékban):

(delta\_idleTime / delta\_totalTime) \*100

A SystemMonitor::CpuMonitor osztálya tárolja a legutolsó mérések eredményeit a previousIdleTimes és previousTotalTimes vektorokban, melyek a konstruktorban vannak inicializálva, CPU magszámnyi 0 értékkel. A magok kihasználtság számítását a gatherCoreLoads metódus végzi a fentebb leírt képlet szerint.

#### 3.2.4.5 NetworkInterfaceInfo

Egy hálózati interfész általános információit a QNetworkInterface osztályon keresztül kérdezem le, az összes csatlakoztatott interfészt a QNetworkInterface::allInterfaces() statikus metódus adja vissza QList formájában. Ez az osztály viszont nem tartalmaz az interfész kihasználtságáról információt, ezért a fogadott és küldött bájtok mennyégét máshonnan kell meghatározni, erre a célra a ***/proc/net/dev*** pszeudó fájt használom. A fájl első sora egy fejléc, mely megadja az oszlopok értékeit, ezt a beolvasás során átugrom, a második sortól kezdve soronként ír le egy interfészt kezdve a nevével majd a „:” karakter után az oszlopok értékeivel szóközzel elválasztva. Az első oszlop az interfész összes fogadott bájtjainak, a kilencedik pedig a küldött bájtjainak a számát mutatja.

A ResourceMonitor::NetworkMonitor osztályban tárolom el a jelenlegi és az azt megelőző kihasználtsági adatokat egy-egy vektorban (currentLoads és previousLoads). Az updateCurrentLoads metódus végzi ezen értékek frissítését, melyet gatherInfo metódus hív, majd ezután végig iterálva a QNetworkInterface -től kapott listán, párosítja az interfészeket a kihasználtsági adataikkal.

#### 3.2.4.5 ProcessInfo

A folyamat információk eléréséhez a ps parancsot veszem igénybe, mely a használt kapcsolóktól függően különböző információkat listáz ki a folyamatokról, a standard kimenetre. A használt kapcsolók és magyarázataik:

|  |  |  |  |
| --- | --- | --- | --- |
| --no-headings | -w | -e | -o pid,state,pcpu,pmem,comm,args |
| Fejléc nélkül jeleníti meg a kimenetet. | Széles oszlopokban jeleníti meg a táblázatot (olvashatóbb kimenet). | Minden folyamat listázva legyen. | A kilistázott információk minden folyamatról:   * pid: folyamat azonosító * state: állapot * pcpu: mag használat százalékban * pmem: memória használat százalékban * comm: futtatott parancs * args: parancs argumentumok |

A parancs kimenete a táblázat utolsó oszlopában taglalt információk, szóközzel elválasztva, soronként egy-egy folyamathoz, ezeket értelmezve kapjuk a ProcessInfo vektort.

#### 3.2.4.5 StorageInfo

A Qt keretrendszer által nyújtott QStorageInfo osztály használom fel. QStorageInfo::mountedVolumes() statikus metódus, visszaadja a csatlakoztatott háttértár eszközöket egy QList formájában, melyek elemei QStorageInfo objektumok. Ezekből kiválogatom azokat, melyek helyesen vannak felcsatolva (isValid metódus), használhatóak (pl.: ha nincs behelyezve CD a beolvasóba, az nem használható, ezt az isReady metódus jelzi) és írhatóak (az isReadOnly metódus akkor igaz, ha csak olvasható az eszköz). A háttértárak neve, felcsatolási pontja és fájlrendszer típusa elérhető a QStorageInfo metódusain keresztül, rendre: displayName, device és fileSystemType. Az összes és szabad terület is lekérdezhető az osztály bytesTotal és bytesAvailable metódusain keresztül, bár ezek, mint a nevük is sugallja bájt nagyságrendben adják vissza az értékeket, ezért ezeket át kell váltani őket. Erre a util névtér byteToMb függvényét használtam. Az összes háttértár információját a ResourceMonitor gatherStorageInfo, metódusa adja vissza, a leírt műveletek szerint.

## 3.3 Architektúra

--SWA diagramm goes here--

Az applikáció Model-View-Controller (MVC) architektúrát használ. Minden model, illetve controller osztály, objektum orientált C++ 17-ben lett implementálva, felhasználva a Qt 6.8 könyvtár által nyújtott lehetőségeket.

Az Application osztály tartalmaz, minden más model és controller objektumot, valamit a grafikus felületet megjelenítő motort. Egyke osztály, az objektum elérhető a statikus Instance metódussal.

### 3.3.1 Model

A model osztályok felelősek az adat perzisztens tárolásáért és a megjelenítés értesítésért, ha azok megváltoznak. A főbb model osztályok egyke tervezési mintát használnak, ezzel le egyszerűsítve a view-val történő kommunikációt. Az osztályok általában valamelyik QAbstractItem model alosztályból származnak (pl.: QAbstractListModel vagy QAbstractTableModel). Ezek a Qt keretrendszer által nyújtott osztályok előre implementálnak bizonyos funkciókat, amik segítik a hatékony adatelérést. Használatukhoz implementálni kell többek között a rowCount metódust, mely a sorok számát adja vissza és a data metódust, amely a kapott index-nél található adattal tér vissza.

A legtöbb model osztály tárolja, a jelenleg kiválasztott kliens azonosítóját valamilyen formában, amikor a View felől adat lekérdezés történik a tárolt kliens adatait adja vissza. Például, ha a hoszt kliens van kiválasztva és a CPU használat oldal lekérdezi a jelenlegi kihasználtságát, a CpuLoadDataModel a hoszt CPU kihasználtságát adja vissza. Ezzel a módszerrel tárolhatjuk egy objektumban a hasonló metrikákat, nem kell minden kliensre más-más model objektumot példányosítani. A következőkben a főbb model osztályok működését fogom bemutatni.

#### 3.3.1.1 Kliens konfiguráció tárolás

A kliens konfigurációk tárolásáért a ClientConfigListModel osztály felelős. Új konfiguráció hozzáadása, meglévő módosítása vagy törlése esetén az összes konfiguráció mentésre kerül egy „configs.xml” nevű fájlba, az applikáció indításakor a model megpróbálja betölteni a konfigurációkat ebből a fájlból, ha a fájl nem létezik (pl.: első indításkor) automatikusan létre hozza azt. A fájlban XML formátuman egymást követik a kliensek információi egy gyökér tag után, minden adat saját tag-ben: azonosító, név, kontextus azonosító, port.

#### 3.3.1.2 Kliens státus tárolás

Az oldalsávban megjelenő kliens státusz információkat a ClientStatusModel tárolja. Ez az osztály nem tartja számot a kiválasztott klienst, a többi model osztállyal ellentétben. Nincs szükség rá, mert az összes kliens státuszát frissíteni kell egy adott pillanatban. Amikor a grafikus felületen a felhasználó egy kliensre kattint az oldalsávban, az osztály itemTriggered szignálja váltódik ki, a kiválasztott kliens indexével. Ezt a Controller osztály kezeli le.

#### 3.3.1.3 CPU és mag használat

A CPU-ról és annak magjainak kihasználtságáról a CpuLoadDataModel tárol információkat. A CPU oldalon lévő diagram megjelenítéséhez szükségen van az előző 20 kihasználtsági adat tárolására. Ezt a 20 adatot úgy kell eltárolnunk, hogy a diagramm lekérdező metódusa iterálni tudjon rajta, vagyis valahogy sorfolytonossá kellett tennem az elmúlt 20 mérés adatait. Ehhez egy template FlowBuffer osztály hoztam létre, mely előre meghatározott számú adatot tárol egy körkörös pufferben. Az osztály tárol eltolódás értéket, mely megadja mennyire van elcsúszva az adattárolás a sorfolytonos tároló többhöz képest. Új elem beillesztésekor a legrégebb óta tárolt elemet írjuk felül és frissítjük az eltolás értéket. Egy adott indexen lévő adat helye a tömbben kiszámolható, az indexből és az eltolásból.

--SRC CODE--

A model osztály számára úgy tűnik, mintha egy tömbben lennének tárolva a mérések értékei. Új CPU használat adat érkezésekor, minden korábbi mérés eggyel hátrébb kerül a tömbben, felülírva a legrégebbi mérést, az újonnan érkezett mérés a sor végére kerül. Egy naiv tömbös megoldásban minden elemet át kellene helyezni, FlowBuffer használatánál egy helyre kerül új érték. Ezzel az optimalizációval sokat gyorsul az applikáció működése, mert másodpercenkét több kliens CPU és mag kihasználtági adatait is frissíteni kell.

--FIX flowbuffer push parameter double -> T

#### 3.3.1.4 Folyamat táblázat

A klienseken futó folyamatokat a ProcessTableModel tárolja, ami a QAbstractTableModel-ből származik és egy táblázatot valósít meg, mely soronként tartalmazza egy-egy folyamat információit. A folyamatok helyes sorrendben történő megjelenítéséhez egy segédosztályt a ProcessIndexer-t használja. Ez az osztály 4 adattaggal rendelkezik maguk a folyamatok egy vektorban (entries), a folyamatok indexei a rendezésben (indicies), a mező mely szerint rendezve vannak (sortedBy) és az indexelő függvény (indexer). Amikor egy mező szerint rendezi a folyamatokat, csak az index vektor változik, tartalma a folyamatok helye a kiválasztott mező növekvő rendezése szerint. Az indexelő függvény két értéket vehet fel: növekvő és csökkenő, az előbbi az indicies vektor elejétől kezdi az indexelést, az utóbbi a hátuljától. Ha megváltozik a rendezései módja (pl.:növekőről csökkenőre), csak az indexelő függvény cserélődik le, ezzel spórolva egy rendezést.

### 3.3.2 Controller

A Controller osztály felelős az applikáció vezérlésért. A kliensek menedzselését is ez az osztály végzi. A felhasználó a grafikai interfészen kapcsolódni kíván egy virtuális klienshez, a Controller felelős a kliens konfigurációjának lekérdezésért a kapcsolat felállításáért és a model értesítésért amikor üzenet érkezik a klienstől. Amikor egy kapcsolat létrejött egy ClientViewController objektum veszi át az adott kliens menedzselését, minden metrika érkezésekor frissíti azon model objektumokat, melyek érdekeltek a kapott metrikában (pl.: a magok használatát továbbítja a CpuLoadDataModel-nek, a memória használatot a MemoryLoadDataModelnek stb…).

### 3.3.3. View

A grafikus elemek QML nyelvben lettek implementálva. Minden QML fájl egy grafikai komponenst realizál. Az komponensek két csoportra vannak osztva: oldalak és általános grafikai komponensek. A grafikus interfész felépítése a csatolt képen látható. Az AppWindow definiálja magát az ablakot, ezt egy PageRoot komponens tölti ki, itt azok a grafikus elemek vannak, melyek mindig láthatóak az applikáció működésekor: jobb oldalt a kliens oldalsáv, alul az asztal. Az ablak fentmaradó részét egy loader elem foglalja el, mely képes dinamikusan QML komponenseket betölteni, így az oldalak váltását el tudja végre tudja hajtani.

Az általános komponensek az oldalok található grafikus objektumok pl.: folyamat táblázat, CPU mérőóra vagy hálózat használat grafikon. Bizonyos komponensek a model osztályok adatait vizualizálják (pl: a mag használat grafikon vagy folyamat táblázat), ezen komponensek rendelkeznek egy model adattaggal, melyhez egy C++ QAbstractItemModel objektumot lehet hozzárendelni, melyből az adott grafikai komponens kinyeri a vizualizálandó értékeket. Ahhoz, hogy egy ilyen model objektumot QML-ből elérjük, az osztályát egykének kell definiálnunk, ez azt jelenti, hogy protected konstruktorral egy statikus create metódussal rendelkezik, mely visszaadja az egyke példányt. Az egyszerűség érdekében egy QmlSingleton osztályt hoztam létre, mely teljesíti ezeket a feltételeket azokra az osztályokra, melyek leszármaznak belőle.

### 3.3.5 Serialization névtér

A szerializáció egy olyan folyamat, melyben egy strukturált adatot, olyan módon alakítunk át, hogy azt platformfüggetlenül eltárolni vagy továbbítani lehessen, úgy, hogy utána pontosan rekonstruálható lehet az eredeti adat. A kliens és az applikáció közötti adatcseréhez elengedhetetlen az adatok szerializációja. Nagy adatmennyiséget kell gyorsan továbbítani, ezért a VSOCK-csatornán keresztül bináris formátumban vannak továbbítva az adatok. Szerencsére a Qt keretrendszer nyújt lehetőséget, bájtsorrend és CPU független szerializációra és deszerializációra, a QDataStream osztályon keresztül. Erre az osztályra többek között a folyamba író (<<) és a folyamból olvasó (>>) operátorok vannak definiálva. Az operátorok túl vannak terhelve minden C++ primitív és a legtöbb Qt konténer típusra. Egy adott típus folyamba írásánál, az egy előre definiált platformfüggetlen bináris reprezentációban került eltárolásra, ezt azt eredményezi, hogy kiolvasásnál minden esetben visszaállítható az eredeti objektum. A metricserializer.hpp és cpp fájlokban definiáltam, a két operátor túlterhelését az összes metrika osztályra és a standard C++ konténerekre, ezzel szerializálhatóvá téve őket. Az implementáció során fontos volt, az adattagok ugyanabban a sorrendben legyenek a folyamba írva, mint onnan kiolvasva, ha ez nem történik meg a szerializáció inkonzisztens lesz.

### 3.3.5 VSOCK névtér

A vsock névtérben lévő osztályok felelősek a VSOCK kommunikáció megvalósításáért. A névtér 4 fő osztályból és azok alosztályaiból áll: Message, VSocket, VSockSingletonServer és VSockClient.

#### 3.3.5.1 Message

A Message osztály egy vsock üzenetet és rajta értelmezett műveleteket definiálja. Az üzenetet bináris formában tárolja egy QBuffer -en keresztül, ami lekérdezhető a getDataStream metódussal, mely QDataStream csomagoló (wrapper) osztályként adja vissza, ez használható a szerializációhoz. A send metódus elküldi az üzenetet a paraméterként megkapott socket -nek, először az üzenet hosszát 32-bites nagy endián formátumban, majd magát az üzenetet binárisan. Az isDisconnect metódus hívásával, dönthető el, hogy az üzenet a speciális lecsatlakozó üzenet-e. A lecsatlakozó üzenet, gyakorlatilag 4 bájtnyi 0 egymás után, mivel egy normál üzenet legelső 4 bájtja az üzenet hossza, más értelmes üzenet nem kezdődhet 4 bájt 0-val.

A statikus receive metódus, a paraméterként megadott socket-en várakozik, ha üzenet érkezik egy shared\_ptr<Message> objektummal tér vissza, ami tárolja az üzenetet. Statikus adattagként elérhető a fent említett lecsatlakozó üzenet disconnectMessage néven.

#### 3.3.5.2 VSocket

A VSocket osztály célja, hogy a C szabványú socket -eket és a rajtuk végezhető funkciókat enkapszulálja. Ahhoz, hogy egy vsock szerverhez csatlakozzunk, tudnunk kell a címét. A címet az Address osztály határozza meg: egy context azonosító és port érték kompozíciója. Ebben az osztályban segéd enumerációkat is találhatunk CID és Port néven, melyek speciális kontextus azonosító és port értékeket tartalmaznak. A VSocket osztály 3 fő műveletet tartalmaz: csatlakozás, üzenet küldés és lecsatlakozás. A connect metódus egy Address-t vár, melyre megkísérli a kapcsolódást, ha ez sikertelen ConnectionError kivételt dob, a hibaüzenettel. Az üzenet küldést a sendMessage metódus végzi. A disconnect metódus megállítja a hallgató szálat és elküldi a speciális lecsatlakozó üzenetet.

Az osztály tartalmaz egy VSockListener objektumot, mely egy külön hallgató szálat valósít meg. Ez az osztály a QThread-ből származik, ennek következménye, hogy a run metódus felül bírálásával (override) külön szálon futhat, hallgatás funkció. Miután megadtuk a socket-et a VSockListener objektumnak, a start metódussal indíthatjuk a szálat, mely végtelen ciklusban olvassa a socket-et és ha üzenet jön a messageReceived szignállal értesíti a VSocket osztályt, ha a kapott üzenet a lecsatlakozó üzenet a socketDisconnected szignál váltódik ki. A VSocket osztály a messageReceived és disconnected szignálok kiváltásával kezeli le a hallgató szál értesítéseit. Az osztály egy statikus metódussal rendelkezik, ami már nyitott C socket-et csomagol be egy unique\_ptr<VSocket> objektumba, a visszaadott objektum már csatlakoztatva van.

#### 3.3.5.3 VSockSingletonServer és VSockClient

A VSockSingletonServer egy VSOCK szerver, mely csak egy kapcsolódott kliens, fogadására képes. A virtuális gépen a kliens alkalmazásban fut, az elnevezés megtévesztő lehet, a VSOCK kommunikációban a szerver szerepét tölti be ezért csak VSOCK szerverként fogok hivatkozni rá. Adattagjai között található egy ServerWorker objektum, mely külön szálon futtatható és a megadott socket-en keresztül VSOCK kliensek csatlakozására várakozik, amit ez megtörténik a clientConnected szignál váltódik ki. A szignál hatására, a VSOCK szerver leállítja a hallgató szálat és egy VSocket objektumban tárolja el a csatlakozott kliens címét. A szerveren keresztül lehetőség van metrikák küldésére a kliensnek. Amikor a kliens elküldi a lecsatlakozó üzenetet, a szerver bezárja a kapcsolatot fenttartó VSocket-et és újraindítja a hallgató szálat.

A VSockClient osztály a grafikus applikáció csomagjában van, egy adott virtuális gépre csatlakozó klienst valósít meg. Konstruktorában egy VSOCK címet vár, melyre csatlakozni kísérel meg (a VSocket osztály kivételeit nem kezeli le, ez a hierarchiában felette álló osztály feladata). Amit VSOCK üzenet jön az adott virtuális gépről, az onMessageReceived metódus deszerializálja azt és ellenőrzi helyességét, majd metric-ként tovább küldi a metricReceived szignálon keresztül.

### 3.3.6 COMM névtér

A COMM névtér a hoszt-on futó applikáció csomagjában van. Ezek az osztályok felelnek a kliensekkel történő kommunikációért. Három osztály felelőssége a kapcsolattartás: ClientDispatcher, LocalClientDispatcher és RemoteClientDispatcher. Egy osztály a kapcsolat kialakításért felel: ClientConnector.

#### 3.3.6.3 Dispatcher osztályok

A ClientDispatcher absztrakt osztály definiál egy kapcsolat interfészét egy élő kliensel, legyen az a Host kliens vagy egy virtuális gép kliense. Három szignált definiál: kapcsolat létesítés sikeres (clientConnected), teljesítmény metrika érkezett (runtimeMetricReceived) és a kliens nem válaszol (clientTimedOut), melyeket a leszármazott osztályok objektumai válthatnak ki.

Egy virtuális géppel fenttartott kapcsolatot a RemoteClientDispatcher valósítja meg. A kapcsolat aktívitásáról egy 5 másodperces időzítővel nyerünk információt, mely lejáratakor a kapcsolat inaktívnak minősül és a clientTimedOut szignál váltódik ki. Amit a kapcsolódás megtörtént egy VSOCK kliensen keresztül, az időzítő elindul. Amikor a VSOCK kapcsolaton keresztül üzenet érkezik, az időzítő újraindul.

LocalClientDispatcher gyakorlatilag egy virtuális klienst emulál, célja, hogy a hoszt gép metrika gyűjtését is úgy lehessen kezelni, mintha egy virtuális kliens lenne. Valójában az osztályban található egy ResourceMonitor objektum, mely segítségivel tud metrikát gyűjteni. A metrika küldés helyes ütemezéséről egy 1 másodperces időzítő gondoskodik.

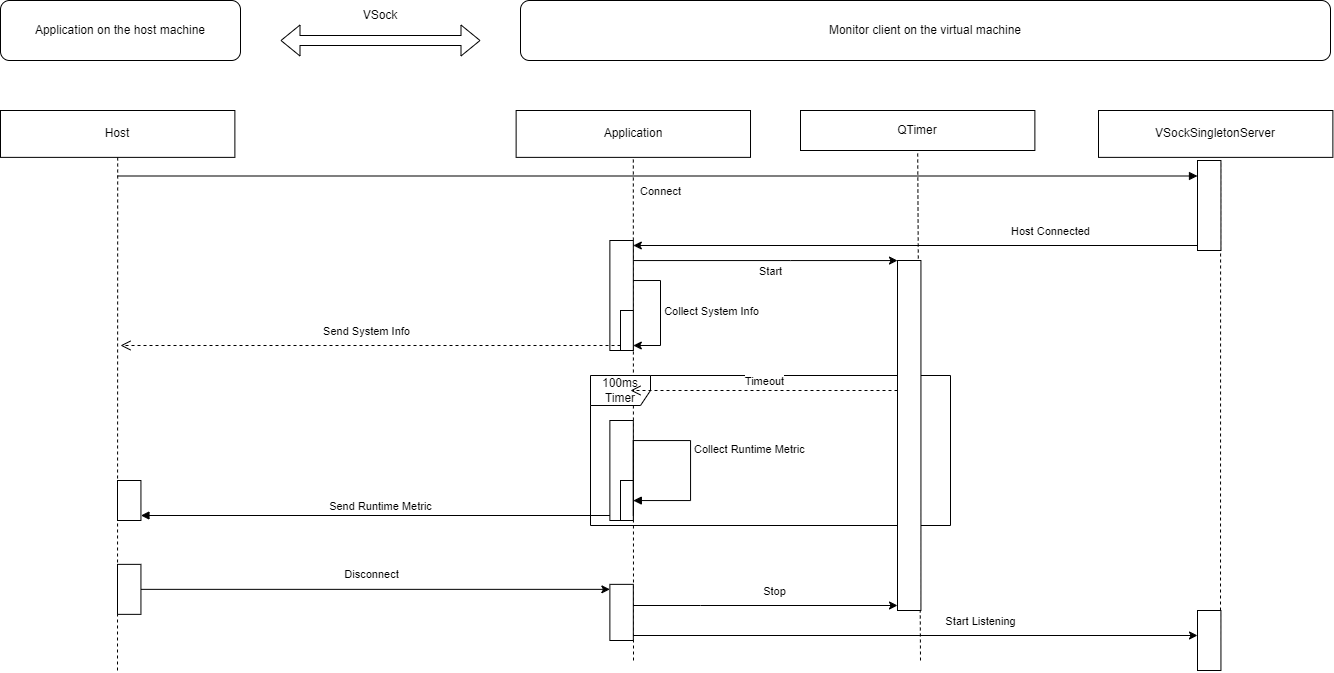
#### 3.3.6.2 ClientConnector

A ClientConnector osztály vezérli a csatlakozást különböző kliensekhez. Az attemptRemoteConnection metódus csatlakozást kísérel megy virtuális gépen futó klienshez, az attemptLocalConnection pedig a hoszt klienshez. Mindkét metódus egy ClientConfiguration struktúrát vár. Ebben a struktúrában a kliens neve, azonosítója és VSOCK címe van tárolva (a hoszt kliensnek az azonosítója 0 és nincs VSOCK címe). Egy csatlakozási kísérletnek két végkimenete lehet: a kísérlet sikeres, ezt a clientConnected szignál jelzi és a kliens nem válaszolt időben, ekkor a clientTimedOut szignál váltódik ki. Amikor egy csatlakozási kísérlet elkezdődik, egy virtuális gépen futó kliensre, egy RemoteClientDispatcher objektum kerül eltárolása és egy 500 milliszekundumos időzítő indul el. Ha az időzítő lejártakor még tárolva van a RemoteClientDispatcher azt jelenti, hogy a virtuális gép nincs rendesen konfigurálva, a kapcsolódási kísértet sikertelen volt. Ha a kliens időben csatlakozik, a diszpécser objektum jelzi a ClientConnector-nak, mely kiváltja a clientConnected szignált és leállítja az időzítőt.

### 3.3.4 Kommunikáció és adatfolyam

#### 3.3.4.3 Hoszt és kliens kommunikáció

Az alábbi szekvencia diagram mutatja be a hoszt applikáció és a kliens monitorozó applikáció interakcióját.



A két komponens között 3 interakció van definiálva: kapcsolódás, teljesítményszám küldés és lekapcsolódás.

Amint a felhasználó a grafikus felületen kapcsolódást kezdeményez az egyik kliensel, az applikáció VSOCK-on kapcsolódási próbálkozásba kezd. A virtuális gépen a monitorozó kliens egy VSOCK (VSockSingletonServer) szervert futtat, mely jelzi az Application vezérlő osztálynak, hogy a hoszt kapcsolódni kíván. Az Application ezután leállítja a szervert és VSOCK üzenetben szerializálva továbbítja, a rendszer információt (SystemInfo struktúra) a hoszt felé, majd indít egy 100ms-el konfigurált időzítőt.

Az időzítő 100ms-enként jelez az Application-nek, mely összegyűjti a teljesítmény metrikákat (RuntimeMetric struktúra) és ezeket továbbítja a hoszt felé.

Ha a felhasználó le kíván kapcsolódni egy kliensről, az applikáció egy speciális lecsatlakozó üzenetet küld a kliensnek. A kliensen működő Application ezt fogadja, majd leállítja az időzítőt és elindítja a VSOCK szervert.

# 4. fejezet

# Tesztelés

Az alkalmazás teszteléséhez az alábbi rendszert használtam: egy hoszt gép, mely egy AGL operációs rendszert futtat. A hoszt AGL kernel modulba integrált KVM hypervisor, mely kezeli a hardver erőforrásokat. Hardvernek a Renesas RCar H3e egylapkás rendszert választottam, egy Renesas RCar-KF kingfisher bővítő táblára integrálva. A hoszton három vendég rendszert virtualizálunk QEMU/KVM -en keresztül, a táblázat szerint:

|  |  |  |  |
| --- | --- | --- | --- |
| Név | Guest-AGL-1 | Guest-AGL2 | Guest-QNX |
| Operációs rendszer | AGL | AGL | QNX |
| Magok | 0,1,2,3 | 4,5,6,7 | 0,1 |
| Memória | 2 GB | 4GB | 4GB |
| Kontextus azonosító | 4 | 5 | 6 |

A hardver 8 maggal rendelkezik (0-7 indexelve), az első 4 mag Cortex-A57, a második 4 pedig Cortex-A54 (a magok közti különbség fontosságáról a fejlesztői dokumentáció tartalmaz több információt). KVM hypervisorral csak homogén magokat használó virtuális gépet lehet indítani, ezért a Guest-AGL-1 az összes A57-es magot, a Guest-AGL-2 pedig A54 magot használja. A QNX vendég gép az AGL-1 vendégel osztozik az első két magon.

## Összegzés és továbbfejlesztési lehetőségek

Diplomamunkám írása során bepillantást nyertem a beágyazott virtualizáció világába, mely az egyik legfrissebb és legjobban kutatott terület az autóiparon belül. Sok lehetőség van virtuális rendszerek teljesítményének optimalizálására és ehhez ezt a teljesítményt valahogyan monitorozni kell. A munkámat egy technológia kutatási projekt keretein belül végeztem, ezért bár lezártnak tekinthető, késznek nem. Számtalan fejlesztési lehetőség van mind új platformok integrálása, mind új funkciók terén. Az applikációt integrálni lehetne valós idejű operációs rendszerekre, melyeknél a legkritikusabbak az idő követelmények. Új funckiókat is lehetne implementálni a programba, például a hoszt gépről az applikáción keresztül, közvetlenül konfigurálhatnánk és indíthatnák a virtuális gépeket, vagy akár a csatlakoztatott gépeken parancsokat is végrehajthatnánk, ezek az új funkciók mind előnyösek lehetnek, egy olyan projektben, mely egy vagy több virtuális gépet alkalmaz.