# 1. fejezet

# Bevezetés

A virtualizáció a modern informatika egyik legjobban kutatott területe. A világon számos helyen használnak virtuális gépeket különböző problémák kiküszöbölésére. Napjainkban már nemcsak a szerver-alapú alkalmazások és otthoni végfelhasználók a virtualizáció legfőbb alkalmazói, hanem a beágyazott rendszerek. Ezeken belül az autóiparban is évre-évre nagyobb szerepet kap a virtualizáció.

A mai személy- és tehergépjárművekben igénybe vehető funkciókat elosztott rendszerként működő automotív ECU-kon (Electronic Control Unit), mint speciális beágyazott hardware-en futó valós idejű operációs rendszereken (Real-Time Operating System) futó alkalmazások végrehajtása és együttműködése realizálja. Az ipari trendek a centralizáció és a “Software-Defined Vehicle” (SDV) irányába mozognak: a korábban nagyobb számú, alacsonyabb teljesítményű ECU-k helyett kevesebb és nagyobb teljesítményű mikroszámítógépek kerülnek a gépjárművekbe, melyeken a funkcionális- és kiberbiztonsági követelményeknek eleget téve integrált virtuális gépek látják el a korábbi ECU-k funkcióit.

Egy autóipari rendszerben kulcsfontosságú, hogy egy adott időpontban azonnal információt kapjunk egy virtuális gép állapotáról, gyorsaságáról és a host rendszer kommunikációs csatornáiról. Ennél az architektúránál még nem áll rendelkezésre olyan centralizált felület, melynek segítségével több virtuális gépet lehet monitorozni egy központi host számítógépről. Szakdolgozatom erre a problémára keres megoldást: egy járműfunkció-specifikus, alrendszer specifikus és globális rendszer információs, adatgyűjtő, monitorozó és vizualizáló információs rendszer kifejlesztése, mely az alábbi főbb funkciókat nyújtja:

1. az elérhető járműfunkciók aktuális állapotának grafikus/vizuális megjelenítése,
2. a különböző funkciókat megvalósító alrendszerek konfigurációjának, I/O kommunikációs forgalmának fogadása és megjelenítése,
3. a rendszert alkotó funkciók globális nézete, elérhetőségének és aktuális státuszának, valamint teljesítménymérőszámainak soft-real-time adatgyűjtése és grafikus megjelenítése.

# 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 6.5 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 Qt6.5 verzió ezeken felül nyújtott ezközei pedig a „meta-qt-6.5” -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

### 2.3.3 Hardver

A megcélzott hardver minden esetben AArch64 (ARM64).

#### 2.3.3.1 Texas Instruments TDA4VM

#### 2.3.3.2 Renesas R-Car

## 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. 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&

### 2.4.2 QNX platform

## 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.

### 2.5.2 QNX platform

### 2.5.3 Ubuntu Linux platform (bemutatási célokra)

### 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

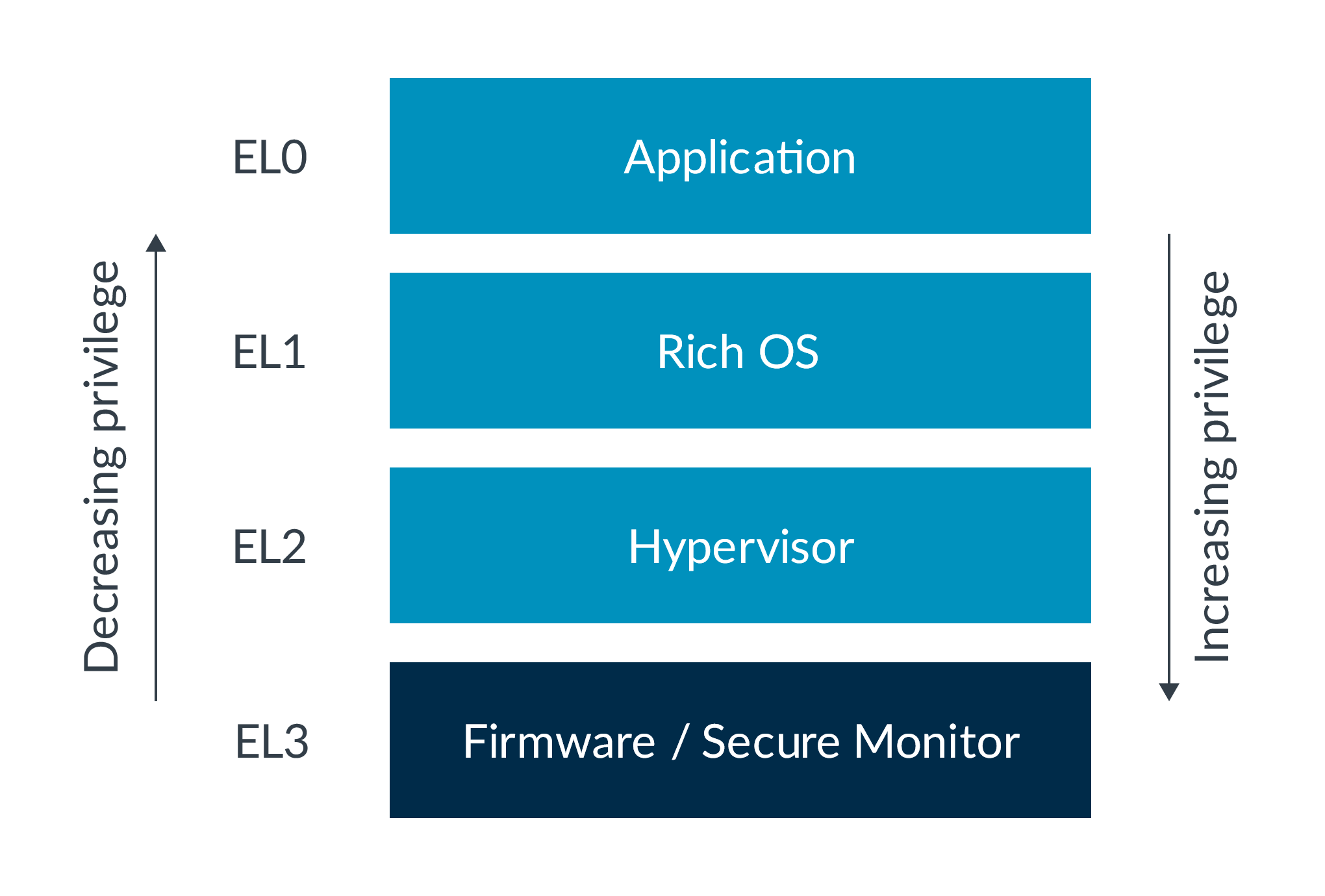
## 3.1 Fogalmak

### 3.1.1 AArch64 és a virtualizáció

#### 3.1.1.1 ARM magok: A, R és M

#### 3.1.1.2 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. Két típusra oszlanak a hypervisorok: type 1 és type 2.

A type 1 hypervisorok közvetlenül a hardveren futnak és hozzáférnek az erőforrásokhoz (CPU, memória). 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. TODO

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.

#### 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.

#### 3.1.2.5 QNX Hypervisor

### 3.1.3 VSOCK

--TODO Virtio--

A VSOCK vagy VM Socket API egy speciális socket család, 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 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 6.5 könyvtár által nyújtott lehetőségeket.

### 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.

### 3.3.2 Controller

### 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.

--GUI screenshot with bounding boxes—

### 3.3.4 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.

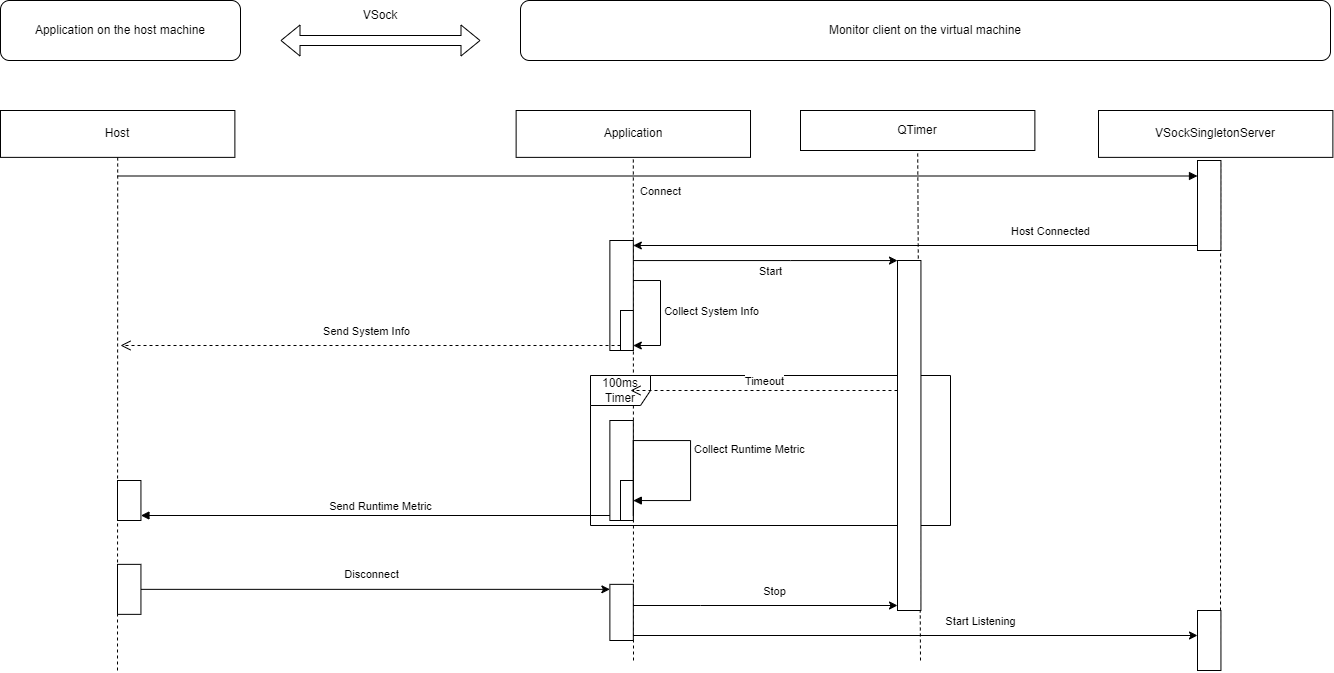
### 3.3.6 COMM névtér

### 3.3.4 Kommunikáció és adatfolyam

#### 3.3.4.3 Hoszt és kliens kommunikáció

--VSOCK DIAGRAMM--

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