# 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. 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:

### 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 project, mely segít a fejlesztőknek testreszabott, hardver architektúra független Linux alapú operációs rendszerek létrehozására. Virtualizálni kívánt rendszerek építésre nagyon jól használható, 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)

Fogalmak:

* Recept (recipe): A Yocto keretrendszer alapja, egy recept beállítások és feladtatok listája melyek alapján felépíthető egy csomag. Egy operációs rendszer csomagokból épül fel, ezen csomagok egyike a kliens alkalmazást tartalmazza.

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

Egy nyílt forráskódú, kollaboráció alapú Linux disztribúció, autóipari felhasználásra tervezve.

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.

### 2.3.2 QNX

### 2.3.3 Hardver

A megcélzott hardver minden esetben AArch64 (ARM64).

#### 2.3.3.1 Texas Instruments TDA4VM

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

Ahhoz, hogy egy virtuális gépet monitorozzunk szükség van a kliens szolgáltatásra, mely összegyűjti az elérhető performansz metrikákat a gépen, majd azokat szerializálva továbbítja az applikáció felé.

### 2.4.1 AGL platform

2.4.1.1 Operációs rendszerkép létrehozása

Előfeltételek:

* Yocto kirkstone környeztet
* Működő Yocto AGL projekt (build)

--DEPENDENCIES--

2.4.1.2 Kernel konfiguráció

A hoszt és virtuális gép kommunikációjához konfigurálnunk kell a kernelt, az alábbi modulokat kell engedélyeznünk:

--DEPENDENCIES--

2.4.1.3 Virtuális gép konfigurálás és indítás

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

--DEPENDENCIES--

A virtuális gépen el kell indítanunk a „VSMT\_Client” alkalmazást. (Érdemes szolgáltatásként indítani.)

### 2.4.2 QNX platform

Előfeltételek:

* QNX SDP 8.0 licensz
* QNX Texas Instruments TDA4VM BSP (board support package)
* A BSP és Qt 6.5 integrálva egy QNX projektbe

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

### 2.5.1 AGL platform

Előfeltételek:

* Qt 6.5 keretrendszer és C++ 17 fordító elérhető a gépen
* CMake verzió minimum 3.16
* A következő Qt modulok telepítve:
  + -DEPENDENCIES-

### 2.5.2 QNX platform

### 2.5.3 Használat --GUI PICS--

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 ki listázza 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

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

2.5.3.7 Kliens konfigurálás

# 3. fejezet

# Fejlesztői dokumentáció

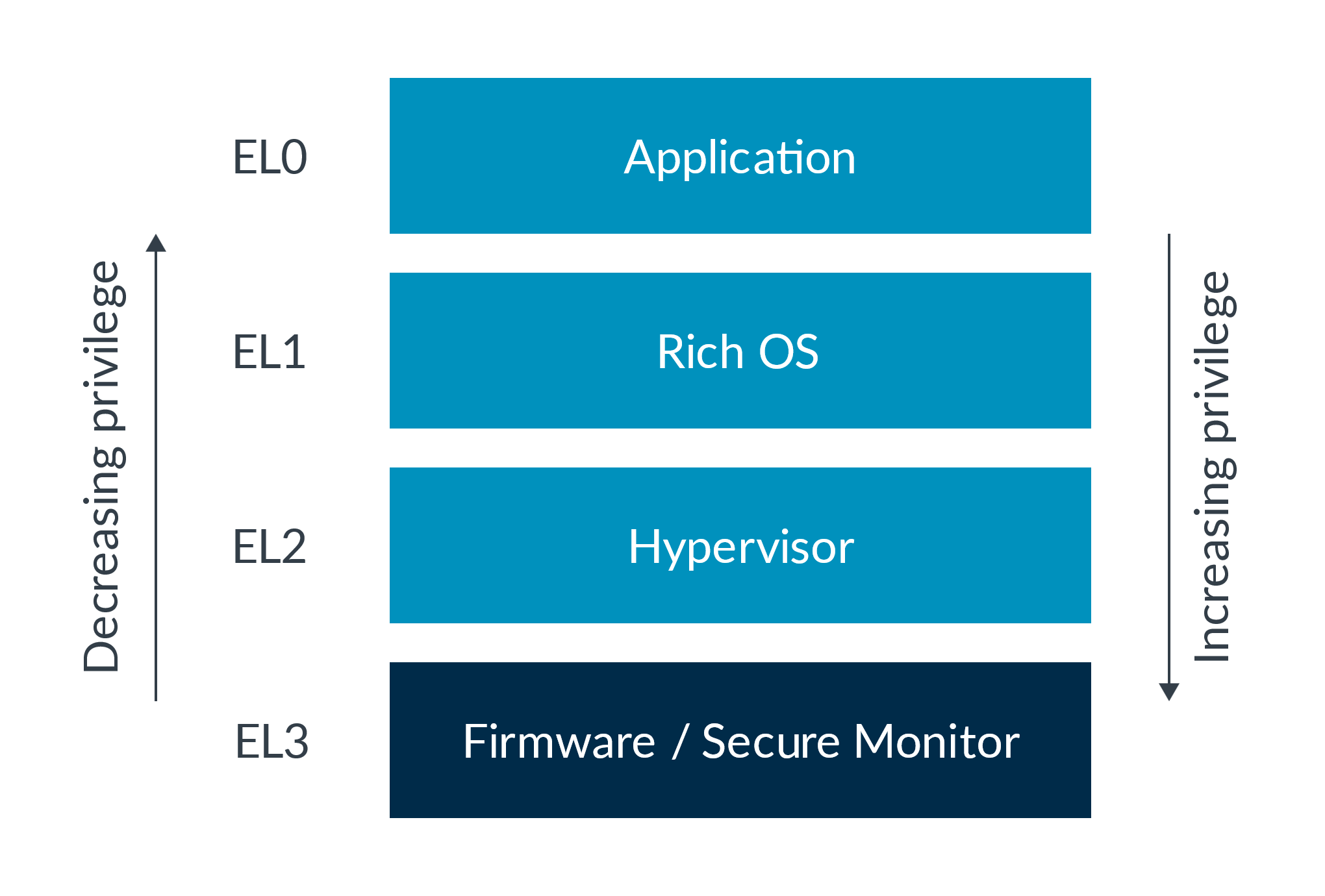
## 3.1 Fogalmak

### 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

Virtio TODO

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 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 AGL

### 3.2.1.1 /proc fájlrendszer

A /proc egy úgy nevezett folyamat információs pszeudó filerendszer, mely 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.

### 3.2.3 QNX

## 3.3 Architektúra

TODO main classes

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.

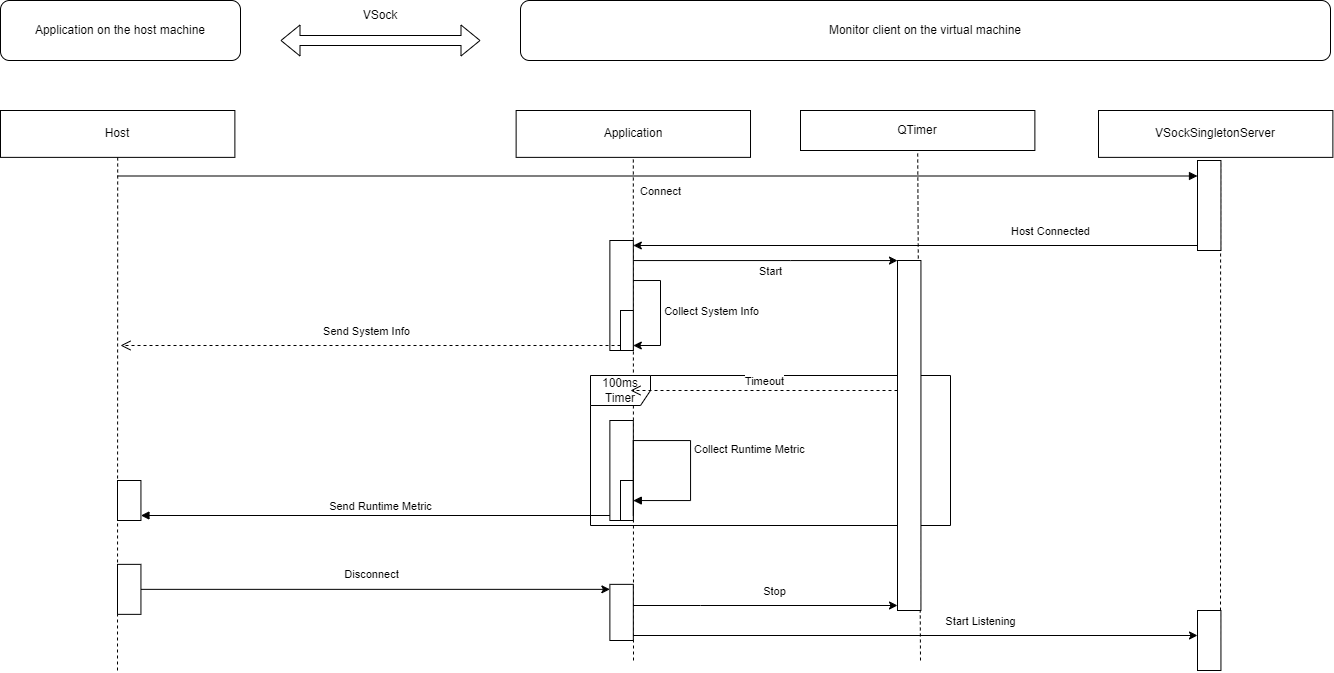
--GUI screenshot with bounding boxes--

### 3.3.4 Kommunikáció és adatfolyam

#### 3.3.4.1 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) servert 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