LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

SKĀRIENJŪTĪGAS SASKARNES IZVEIDE OPERATĪVAI SISTĒMAI DARBINĀMAI UZ VIRTUĀLĀS MAŠĪNAS VIRTUALBOX

MAĢISTRA DARBS

Autors: **Tomass Lācis**

Studenta apliecības Nr.: **tl17007**

Darba vadītājs: **profesors Dr.sc.comp. Guntis Arnicāns**

RĪGA 2019

ANOTĀCIJA

Monitora sienas ir plaši izmantotas, lai attēlotu liela apjomu informāciju un arī pat to manipulēt pārredzamā veidā ar, piemēram, skārienjūtīgie kontroles paneļiem. Tā veidi ir dažādi – no vairāku kopīgu saslēgtu monitoru sienas līdz vairākiem monitoriem, kuri ir izvietoti dažādās telpās, bet pieslēgti pie vienas virtuālās sienas sistēmas.

Viens no tipiskajiem piegājieniem, kā kontrolēt vairākus monitorus sinhronizētā veidā, ir izveidot virtuālo sienu uz virtualizācijas platformām, kurām ir pieejami spēcīgi fiziskie servera resursi. Tālāk vienīgais šķērslis ir izveidot saskarni starp fizisko un virtuālo vidi, kur pareizi parāda virtuālas sienas saturu uz fiziskajiem monitoriem, kā arī komunicē kontroles signālus uz virtuālo vidi.

Šī darba mērķis ir tālāk attīstīt autora bakalaura darba pētījumu par skārienjūtīgām monitora sienām. Šī darba ietvaros tiks aprakstīta arhitektūra un implementācija virtuālas sienas skārienu signālu apstrādei caur VirtualBox virtualizācijas platformu.

Atslēgvārdi: C/C++, skārienjūtīgie monitori, monitoru siena, Infiniviz, Raspberry Pi, VirtualBox

ABSTRACT

TOUCH-SENSITIVE INTERFACE DEVELOPMENT FOR OPERATING SYSTEM WORKING ON VIRTUALBOX

Monitor walls are widely used to display great amount of information and can also be used to manipulate it with, for example, touch-sensitive control panels. Their types are varied: from closely connected monitors in a grid to monitors spread around in different rooms but still connected to a singular virtual monitor wall system.

One typical approach to control many monitors in a synchronized way is to create a virtual wall in virtualization platforms where they have access to powerful physical server resources. After that the only obstacle is to create an interface between the physical and the virtual environment which correctly shows virtual wall's content on physical monitors as well as communicate control signals to virtual environment.

The purpose of this work is to further progress author's bachelor work study about touch-sensitive monitor walls. The scope of this work is to describe architecture and implementation of virtual monitor wall touch signal handling through VirtualBox virtualization platform.

Keywords: C/C++, touchscreen monitors, monitor wall, Infiniviz, Raspberry Pi, VirtualBox

AUTOREFERĀTS

Autora galvenais sasniegums ir praktiska realizācija skārienjūtīgu ekrānu signālu atdarināšana uz VirtualBox uzturētu viesa operētājsistēmu, izmantojot VirtualBox programmatūras izstrādes komplektu. Papildus tam ir veikta dziļa izpēte par šī izstrādes komplekta galvenajām funkcijām un kā integrēt programmatūras ar VirtualBox saskarni, kur var kontrolēt un manipulēt virtuālās mašīnas. Praktiskā risinājuma testēšanai, autors arī ir demonstrējis Raspberry Pi ierīču darbību un to pielietojamību ar papildus moduļiem (LCD skārienjūtīgs ekrāns).

Darba novitātes ziņā ir veikta detalizēta izpēte par praktisku skārienjūtīgo signālu emulāciju ar VirtualBox saskarni. Tas tiek veikts ar domu to izmantot kontroles paneļu izstrādei, kur daži skārienjūtīgi paneļi kontrolē lielas monitoru sienas saturu.

Liela daļa iepriekšējo darbu literatūra nāk no Infiniviz publikācijām, kā arī no autora un citu studentu bakalaura darbiem. Autors ir pārsvarā veicis literatūras izpēti programmatūras izstrādes sfērā, tātad liela daļa avotu ir dažādu bibliotēku, programmatūru vai standartu dokumentācijas. Nozīmīga daļa literatūras avoti arī ietver Linux sistēmu izsaukumu pielietošanu un POSIX standartus.

Darbs sākotnēji vispārīgi izklāsta sava bakalaura darba saturu, citu studentu darbus un Infiniviz publikācijas, lai izveidotu nelielu ievadu par autora monitora sienas sistēmas izstrādes mērķiem. Savukārt, nākošās nodaļas ieiet pamatīgās detaļās par pareizu VirtualBox izstrādes komplekta izmantošanu, programmatūru iestatīšanu, risinājuma implementāciju un testēšanu.

Darba praktiskā daļa ir vidēji aizņēmusi trīs mēnešus, kur sākotnēji autors atdarina Infiniviz risinājumu, lai iegūtu dziļāku izpratni par monitora sienas implementāciju VirtualBox vidē. Pēc tam ir veikti mazi testi un eksperimenti ar virtuālo mašīnu manipulēšanu. Beidzamajos posmos ir izstrādāta klienta-servera programmatūra, kas pārraida un emulē skārienjūtīgos signālus.

Darbs ir pareizi noformēts latviešu valodā pēc vispārīgajiem Latvijas Universitātes un papildus Datorikas Fakultātes norādījumiem. Autors arī ir apskatījis un izpildījis norādījumus kontrolsarakstā no maģistra darba noformēšanas norādījumu 6. pielikumā. Idejas, formulējumi, attēli utt., kuri ir aizgūti no citiem darbi ir attiecīgi atzīmēti ar atsaucēm vai atzīmēti kā teksta aizguvumi.

SATURS

APZĪMĒJUMU SARAKSTS

Apzīmējums	Apzīmējuma apraksts
Infiniviz	Gunta Arnicāna un Rūdolfa Bunduļa projekts, kurā pēta monitora sienas problēmas un piedāvā savus risinājumus [1][2][3][4][5][6]
Raspberry Pi	Ļoti mazs, lēts un enerģiju taupošs dators, kuru izstrādāja mācību nolūkiem, atļaujot cilvēkiem lēti veikt vairākus praktiskus projektus [1]
CMake	C/C++ projektu ģenerēšanas rīks, kas ir spējīgs uzģenerēt dažādu kompilatoru failus, lai kompilētu iekļautos C/C++ failus uz vairākām platformām [1]
Boost	C/C++ bibliotēku kopa, kura sastāv no vairākiem simtiem bibliotēku, kas paplašina programmēšanas valodu ar daudzām papildu palīgkonstrukcijām un veic abstrakciju operētājsistēmas specifiskās konstrukcijās [1]
VirtualBox	Oracle uzņēmuma izstrādāts atvērtā koda rīks, kas atļauj veikt operētājsistēmu virtualizāciju, kas nozīmē simulēt reālu operētājsistēmu, kurā var palaist programmatūras tādā veidā, kādā tās tiktu palaistas uz parastajām operētājsistēmām [1]
SSH	Saīsinājums *Secure Shell*. Drošs datu apmaiņas tīkla protokols, kas atļauj droši pārnest datus (ar kriptogrāfijas palīdzību) caur nedrošu tīklu [1]
SDK	Saīsinājums *Software Development Kit*. Programmatūras izstrādes komplekts. Palīgprogrammu (rutīnu) kopa, kas lietojumprogrammu izstrādātājam atvieglo programmu veidošanu, ievērojot konkrētās to darbības vides īpatnības (piem., grafisko lietotāja saskarni, operētājsistēmu, datu bāzu pārvaldības sistēmu u.c.) [11][1]
POSIX	Saīsinājums *The Portable Operating System Interface*. IEEE Computer Society un The Open Group komitejas definēti standarti, lai nodrošinātu savienojamību starp vairākām dažādām operētājsistēmām, tai skaitā, Unix-veida operētājsistēmām [42][1]
Galvenes fails	C/C++ pirmkodu failu tips (angl. *header file*), kuros parasti ievada funkciju izsaukumu un struktūru deklarācijas. Tie ir bieži izmantoti, lai izveidotu saprotamu programmatūras saskarni un atļauj tā metodes implementēt ar attiecīgajiem .c vai .cpp failiem. Galvenes failu galotnes ir .h (C) vai .hpp (C++)
NVidia	Grafikas kartes ierīču izstrādātājs, kurš nodrošina dažāda veida grafisko operāciju paātrinātājus, kā, piemēram, GTX n-tās paaudzes kartes
AMD	Centrālo procesoru ražotājs, kas konkurē ar Intel dažādos tirgos, kā arī mazliet konkurē ar NVidia ar savām grafikas kartēm
Intel	Viens no lielākajiem centrālo procesoru ražotājiem
COM	Microsoft izstrādāts standarts, kā sazināties starp atsevišķām programmatūrām caur konkrēti definētu objektu modeli [27]

IEVADS

Monitora sienas kā konceptuāla sistēma ir daudzkārt izpētīta un implementēta dažādos veidos. Vispopulārākais veids ir izmantot virtualizācijas platformu kā Hyper-V, VMware vai VirtualBox, kurā strādā virtuālā mašīna ar konkrētu izšķirtspēju, ko var sadalīt vai dublicēt uz vairākiem fiziskiem monitoriem. Diemžēl šis paņēmiens nav viegli uzstādāms, jo nav izstrādāta standartizēta saskarne starp fiziskajiem monitoriem un virtuālo sienu. Saskarnei, kā minimums, ir jānodrošina pareiza virtuāla grafiskā satura izplatīšanu uz pareizajiem monitoriem.

Problēma paliek aizvien sarežģītāka, kad ir nepieciešams kontrolēt virtuālās sienas saturu no ārējiem kontroles paneļiem, kuri varbūt pat nav tuvumā pašam serverim. Ar šādiem paneļiem izveidoto saskarni vajag papildināt ar kontroles signālu apstrādāšanu, kur ir padziļinātāki jāizprot virtualizācijas platformas programmatūras izstrādes komplektu (angl. *SDK – Software Development Kit*). Autors šī darba nolūkos strādās ar VirtualBox vidi un tās piedāvātajiem interfeisiem, kuri apstrādā gan kontroles signālus kā peles spiedienus vai skārienus, kā arī grafisko izvadu no virtuālās grafiskās ierīces.

Liela daļa no šī darba seko no skārienjūtīgo monitora sienas izstrādes pētījuma, kas ir autora bakalaura darbs. Ir svarīgi saprast, ka bakalaura darbā tika izstrādāta un pārbaudīta skārienu apstrāde *Windows* operētājsistēmas ietvaros. Skārienu apstrāde vēl nebija izstrādāta uz pašas VirtualBox platformas, bet bija vispārīgi aprakstīta teorētiskā līmenī. Autora mērķis ar maģistra darbu ir sīkāk izpētīt un implementēt mērogojamu skārienu apstrādi VirtualBox vidē, izmantojot tā programmatūras izstrādes komplektu.

Autors šī darba ietvaros izveido kopsummu savam un citu studentu iepriekšējiem darbiem par monitoru sienas pētījumiem. Pēc tam tiek izklāstītas nozīmīgākākie principi, strādājot ar VirtualBox programmatūras izstrādes komplektu. Tas tiek izpētīts no VirtualBox SDK dokumentācijas un arī praktiski pārbaudīts. Galējā rezultātā autors mēģina izveidot programmatūru, kas iegūst, apstrāde un emulē skārienjūtīgos signāls caur VirtualBox SDK uz kāda no viesa operētājsistēmām. To pēc tam novērtē, apraksta izmantoto aparatūru testēšanai un pašu testēšanas gaitu. Tā kā monitora sienas sistēmas ir diezgan lielas un autors visticamāk veselu komplektu nevarēs izstrādāt šī darba nolūkiem, tad beidzamā pamatnodaļa apraksta nākotnes attīstību, izmantojot autora ieguldīto darbu.

1. SKĀRIENJŪTĪGO MONITORU SIENAS IZSTRĀDES PĒTĪJUMS

Monitora sienas būtība un tās šķēršļi ir vairākkārt izpētīts caur Infiniviz publikācijām [2][3][4][5][6]. Publikācijas apraksta vispārīgo problēmu kopu, kuru mēģina atrisināt monitora sienas, kādi ir ierobežojumi attiecībā pret operētājsistēmām un cik risinājumi ir mērogojami ar fiziskajiem datora resursiem. Arī loģistika monitoru satura izplatīšanai uz sienas paneļiem ir dziļi izpētīta ar VirtualBox virtualizācijas platformas palīdzību. Savukārt, skārienjūtīgo monitoru būtība sienas sistēma ir aizvien maz-izpētīta sfēra.

Skārienjūtīgie monitori piedāvā iespēju izveidot kontroles paneļus monitora sienai, kur ar skārienjūtīgu virsmu var kontrolēt sienas saturu (piemēram, pārbīdīt aplikācijas logu vai veikt iestatījumus). Parasti sienas saturu būtu jākontrolē pašā servera iekārtā, kurā darbojas virtualizācijas platforma un tas arī var kļūt neērti, ja virtuālā siena ir vairākus megapikseļus liela izšķirtspējā. Ar skārienjūtīgu paneli var pārslēgties uz konkrētu kvadrantu virtuālajā sienā un izmainīt tās saturu, pēc tam pārslēgties uz citu kvadrantu un tā tik turpināt.

Autora bakalaura darbs pētīja skārienjūtīgo monitoru sienas izstrādi [1]. Tas, savukārt, turpina no Lauras Ķirsones teorētiskā pētījuma par skāriena signāliem un autores piedāvāto arhitektūru tās izstrādei [7]. Abi darbi arī balstās uz Infiniviz publikācijām, kuru autori ir Rūdolfs Bundulis un prof. Guntis Arnicāns [2][3][4][5][6].

Skārienjūtīgo sienas izstrādi autors sākotnēji pētīja izveidojot vienkāršu programmatūru, kas apstrādā tīros skāriena signālus no Linux operētājsistēmas skāriena ierīces. Tie tiek tālāk pārsūtīti uz servera programmatūru, kas darbojas Windows vidē. Serveris vienkārši atdarina Linux ievāktos skāriena signālus uz Windows operētājsistēmas un līdz ar to virtualizācijas platformas implementācijas nav apskatīta.

Ķirsones darbs līdzīgi apskatās to pašu Linux signāla struktūru un pat bija teorētiski aprakstīta to apstrāde, bet nekas dziļāk par to atdarināšanu vai praktisku implementāciju nebija dokumentēts. Autore lielāku nozīmi savam darba lika uz abstraktu arhitektūru par skārienjūtīgo sienas signālu komunikāciju. Ķirsones arhitektūra bāzējas uz Infiniviz arhitektūru, kurā ir iekļautas *virtuālās mašīnas* no VirtualBox vides.

Infiniviz projekts ir licis vairākus pamatus, no kuriem ir smēlies gan autors, gan Ķirsone, gan arī vairāki citi studenti ar saviem darbiem. Šī projekta publikācijas sevī ietver aprakstus par

monitora sienas niansēm, kā tās ir pareizi jāiestata un kādi ir reālie resursu ierobežojumi. Lielākā problēma monitoru sienās ir to ietverto ierīču daudzums un dažādība. Programmētajā darbs būtu atrast kopīgu saskarni, kā visas šī dažādās ierīces var savienot kopā un izveidot kaut cik saprotamu abstrakciju. Šī abstrakcija ir nepieciešama, lai risinājums būtu kaut cik mērogojams ar dažādiem monitoriem. Jāsaprot arī, ka katrai ierīcei ir savi parametri un monitori nāk dažādos lielumos un izšķirtspējās.

Šajā nodaļā autors apraksta savu bakalaura darba kopsummu un arī to novērtē ar papildus jaunu informāciju, kuru ieguva maģistra darba izstrādes gaitā. Kopsummas saturā arī ir aprakstīts Infiniviz projekts un Ķirsones darbs, kā arī pielikta papildus informācija, kā tie palīdz autora izstrādāt skārienjūtīgo monitoru sienas sistēmu ar VirtualBox.

Sākotnēji būs aprakstīts Infiniviz projekts, lai lasītājiem iedotu nelielu kontekstu par monitoru sienu izpēti Latvijas Universitātes ietvaros. Par projektu arī būs sīki aprakstītas tās implementācijas nianses. Nākošajā apakšnodaļā būs apraksts par Lauras Ķirsones darbu un kā tas palīdz autoram realizēt VirtualBox risinājumu skārienjūtīgajai siena. Beidzamajās apakšnodaļās ir autora bakalaura darba kopsumma, pašnovērtējums un vai sastaptās problēmas ir aktuālas ar virtualizētu risinājumu.

Protams, detalizētāku aprakstu par iepriekšējiem monitora sienas izstrādes rezultātiem var izlasīt autora bakalaura darbā [1] vai arī pašās publikācijās [2][3][4][5][6]. Šajā darbā autors lielu nozīmi veltīs sīkam un detalizētam apraksta, kā izmantot VirtualBox saskarni skārienjūtīgās sienas izstrādei.

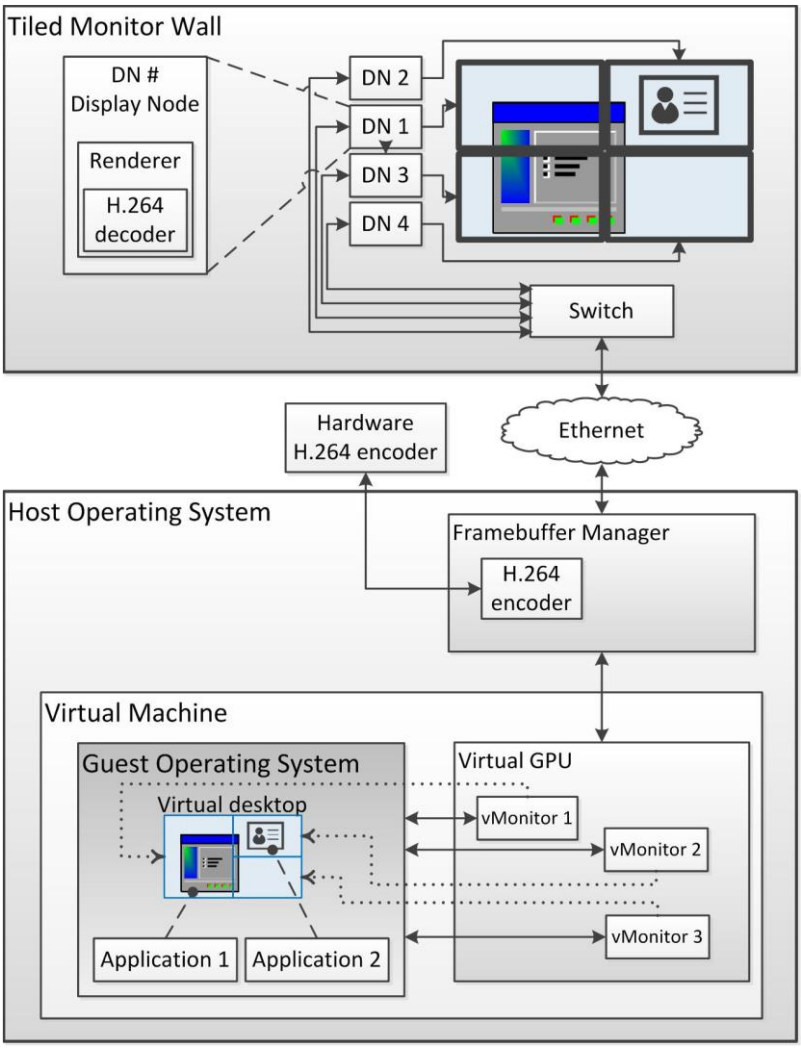
1.1. Infiniviz pētījums un implementācija

Projektu *Infiniviz* sākotnēji veidoja Rūdolfs Bundulis ar prof. Gunti Arnicānu, lai atrastu noderīgu un lētāku monitora sienas risinājumu. Tā sākotnējie darbi apraksta vispārīgo monitora sienas problēmu, kā arī sīki apraksta fiziskos ierobežojumus (piemēram, ir noteikta maksimālā izšķirtspēja, ko Windows operētājsistēma var atbalstīt). Lai atrisinātu dažus no šiem ierobežojumiem, projekta ietvaros izstrādā mērogojamu risinājumu ar VirtualBox, Raspberry Pi ierīcēm un vairākām monitora iekārtām [2][3][4][5][6].

1.1.1. att. **Infiniviz monitoru sienas prototips [3][6]**

Risinājuma būtība ir uzturēt spēcīgu VirtualBox serveri, kurā var darboties virtuālā mašīna ar jebkuru operētājsistēmu, kurā var piešķir jebkādu izšķirtspēju, ko atbalsta attiecīgā operētājsistēma. Tālāk uzdevums ir izmantot VirtualBox programmatūras izstrādes komplektu, lai reālā laikā straumētu (angl. *real-time streaming*) virtuālās darbvirsmas attēlu uz dažādām monitora ierīcēm caur tīklu. Infiniviz autori šo ir izdarījuši, izmantojot Raspberry Pi ierīces, kuras ir tiešā

veidā saslēgtas ar monitoriem. Tās saņem darbvirsmas attēlu no servera un to attiecīgi attēlo uz monitora iekārtas (skat. 1.1.1. att.).



1.1.2. att. **Infiniviz monitoru sienas prototipa arhitektūra [6]**

Infiniviz sienas arhitektūra (skat. 1.1.2 att.) skaidri parāda datu plūsmas gaitu cauri tīklam, bet arī iekļauj video kodētājus un dekodētājus. Lai reālā laikā straumētu video jeb darbvirsmas attēlu, tad ir jāveic tā izmēra mazināšana. Piemēram, 1920x1080 izšķirtspēja ar 3 krāsas baitiem un 30 kadru sekundē ātrumu sasniedz gandrīz 180 mega**baitus** sekundē (~ 1423 mbit/s). Tādus datu apjomus nevar normālā veidā pārvietot caur tīklu, tik īpaši, ja monitora sienai kopīgā izšķirtspēja būs daudzreiz lielāka.

Izmēra mazināšanai un virtuālās sienas kadru apstrādei pielieto plaši izmantotu H.264 kodēšanas standartu, kas ir izmantots tieši video datu straumes izmēra samazināšanai ar minimālu vai pat nejūtamu kvalitātes zudumu. Tā kā tas ir tikai standarts, pastāv dažādas implementācijas.

Viena no populārākajām programmatūras implementācijām ir *x264*, kuru izveidoja un uztur VideoLAN organizācija (VLC video atspēlētāja programmatūras izstrādāji) [8]. Pastāv arī fiziskās implementācijas, kur, piemēram, izmanto grafisko procesoru kodēšanas paātrināšanai. Infiniviz var izmantot NVENC, kas ir *NVidia* grafikas karšu kodēšanas bibliotēka (NVDEC ir attiecīgi dekodēšanai).

Maģistra darba nolūkiem bija veikts mēģinājums nokompilēt un izveidot Infiniviz programmatūru uz autora laptopa, kurā ir iebūvēta GTX 1060 grafikas apstrādes paātrināšanas ierīce. Diemžēl ātri tika atklāts, ka Infiniviz projekta NVENC versija bija ļoti veca un to vajadzēja atjaunināt. Tas, savukārt, izraisīja operatīvās atmiņas nepareizu izmantošanu, jo Infiniviz pirmkods bija paredzēts vecākai NVENC versijai. Šādas problēmas liecina, ka orģinālais projekts ir slikti novecojis un tā uzstādīšanu uz jaunām iekārtām nav iespējama bez pirmkoda izmaiņām.

Vēl viena nozīmīga problēma ar Infiniviz kadru kodēšanas bibliotēku ir NVidia uzliktie ierobežojumi uz atsevišķiem grafikas kartes modeļiem. Grafikas karte GTX 1060 ir atļauts izveidot tikai divas vienlaicīgas kodēšanas sesijas, kas stipri ierobežo kadru apstrādes ātrdarbību. Plašāk pieejama informācija ir NVidia oficiālajā dokumentācijā, kura ir pieejama tiešsaistē [9]. Lielākām sesiju skaitam ir nepieciešams nopirks citādākas kategorijas NVidia grafikas kartes, kā Quadro P6000, kuras maksā tūkstošos.

1.1.3. att. ***H.264 Pro Recorder* no *Blackmagic Design* izstrādāja [10]**

Citas fiziskās implementācijas ir konkrētas iekārtas izstrādās tīram kadru kodēšanas nolūkiem, kā, piemēram, *Blackmagic Design* kodēšanas ierīces (skat. 1.1.3. att.). Pastāv arī vairāki citi izstrādātāji ar sava veida kodēšanas ierīcēm un tās patiešām ir izmantotas eksistējošajiem monitoru sienas risinājumiem.

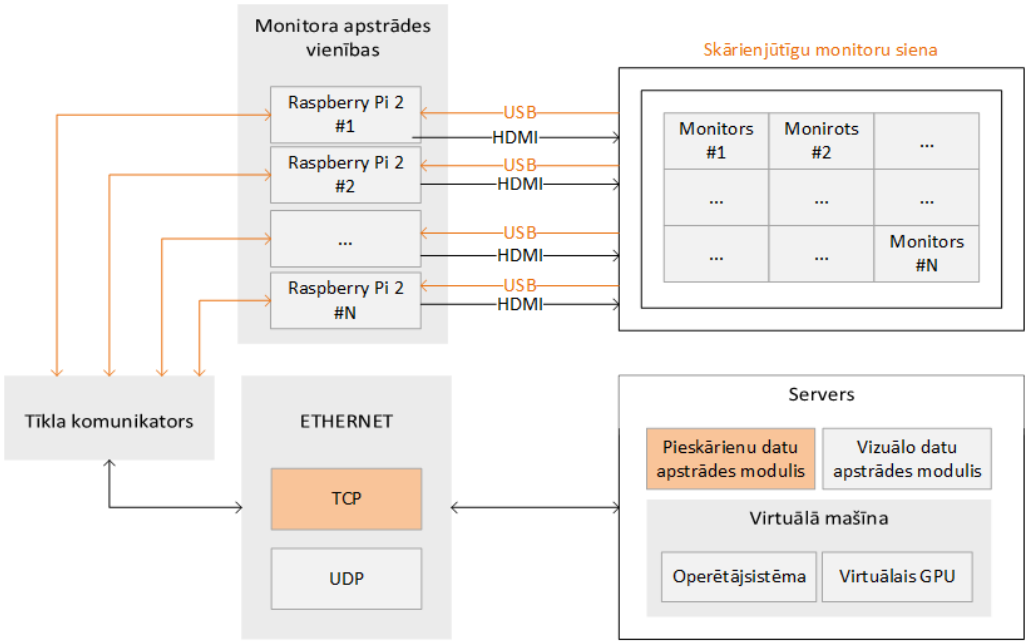
Infiniviz kadru dekodēšanas nolūkiem izmanto *omxplayer* bibliotēkas komponentes un iekodētie kadri tiek saņemti caur RTP protokolu [11][12]. Pašam dekodēšanas procesam nav izmantotas papildus fiziskas iekārtas, kā nu vienīgi parastā centrālā procesoru ierīce. Reāla laika straumēšanas kadru saņēmēju galā ir arī iespējams izveidot operatīvās atmiņas blokus, kuri glabā dažus saņemtos kadrus (angl. *video buffering*), kas parasti palīdz straumēt iepriekš izveidotus video failus. Šajā gadījumā tas nav nepieciešams un lieki tērēs Raspberry Pi operatīvo atmiņu, jo monitora sienas darbvirsmas straumēšana vienmēr piegādās tekošo informāciju (t.i. šī nav eksistējoša video faila straumēšana).

VirtualBox saskarnei ir izmantots tās programmatūras izstrādes komplekts, bet autors, kompilējot Infiniviz programmatūru, sastapās ar līdzīgām kompilēšanas problēmām kādas tika sastaptas ar NVENC kodēšanas bibliotēku. Projekts tika izstrādāts uz vecākām VirtualBox versijām un, kad autors mēģināja to nokompilēt uz VirtualBox 5.2 versiju, atklāja, ka vecais pirmkods apraksta objektus, kuru struktūra ir izmainījusies ar jaunākām versijām. Tas atkal lika autoram dziļi izpētīt orģinālo pirmkodu un atrast kļūdainās vietas, kuras ir jāatjaunina. VirtualBox saskarnes objektus un to atbalstītās darbības ir dziļāk aprakstītas otrajā nodaļā (VIRTUALBOX IMPLEMENTĀCIJA), kur autors izklāstīs papildus detaļas par Infiniviz sastaptajām implementācijas problēmām.

1.2. Ķirsones pētījums

Skārienjūtīgos monitorus un to pielietošanu monitora sienas sistēmām Latvijas Universitātes ietvaros vispirms pētīja Laura Ķirsone [7]. Sākotnēji ir aprakstīti vairāku veidu eksistējošie skārienjūtīgo monitoru sienas risinājumi, kuri iedalās homogēnos (liela izmēra monitori) vai matricu-veidu izkārtojumos. Autores pirmā nodaļā pievieno papildus informāciju par visparīgu ieskatu skārienjūtīgajiem monitoriem un to lomu monitoru sienās, kamēr Infiniviz publikācijas tikai apraksta vizuālā satura apstrādi.

Autora bakalaura darbs visvairāk bija smēlies no Ķirsones skāriena signāla apstrādes nodaļas, kur autore apraksta skārienjūtīga monitora ievaddatu lasīšanu [1]. Ir nozīmīgi saprast, ka gan Ķirsones, gan autora darbi apstrādā signālus no Linux operētājsistēmas. Tas ir pārsvarā darīts, jo ir skaidra un vienkārša dokumentācija Linux ierīču apstrādei, kamēr Windows operētājsistēmā ir vajadzīgs iepazīties ar *draiveru* programmatūras izstrādāšanu un vairākām operētājsistēmas niansēm. Linux gadījumā var tiešā veidā lasīt binārus signāla datus no virtuāla faila ar gatavām operētājsistēmas atbalstītām funkcijām.



1.2.1. att. **Ķirsones Infiniviz arhitektūras pielikums [6]**

Ķirsones arhitektūra (skat. 1.2.1. att.) ir pārsvarā pielikums Infiniviz arhitektūrai, kas ievieš divpusēju signāla apmaiņu ar galveno serveri. Tīrajā Infiniviz arhitektūrā ir paredzēts vienvirziena vizuālo datu padošana no servera uz monitora ierīcēm. Skāriena signāli nāktu no monitora ierīcēm

(Raspberry Pi), kuras tos saņem caur USB vadu no pašām monitora iekārtām. Tie tiek tālāk aizsūtīti uz serveri, kur tos atdarina uz aktīvās virtuālās operētājsistēmas, kuru uztur VirtualBox (vai kāda cita virtualizācijas platforma).

Autores piedāvātā arhitektūra pieņem, ka vizuālā satura monitori un skārien-kontrolējamie monitori ir vienā sienā. Darbs nav tiešā veidā apskatījis variantu, kur galvenā, fiziskā monitora siena satur tikai parastus monitor un ir attālināti kontrolējams ar atsevišķiem skārienjūtīgiem kontroles paneļiem. Šādā gadījumā piedāvātā arhitektūra var aizvien nostrādāt, ja tā attiecas uz kontroles paneļiem un galvenā siena būtu saslēgta tikai ar vienvirziena datu plūsmu.

```
Event: time 1425319271.601632, type 3 (EV_ABS), code 53 (ABS_MT_POSITION_X), value 10306
Event: time 1425319271.601632, type 3 (EV_ABS), code 54 (ABS_MT_POSITION_Y), value 30625
Event: time 1425319271.601632, type 3 (EV_ABS), code 48 (ABS_MT_TOUCH_MAJOR), value 962
Event: time 1425319271.601632, type 3 (EV_ABS), code 49 (ABS_MT_TOUCH_MINOR), value 421
Event: time 1425319271.601632, type 3 (EV_ABS), code 47 (ABS_MT_SLOT), value 1
Event: time 1425319271.601632, type 3 (EV_ABS), code 57 (ABS_MT_TRACKING_ID), value 52
Event: time 1425319271.601632, type 3 (EV_ABS), code 53 (ABS_MT_POSITION_X), value 15416
Event: time 1425319271.601632, type 3 (EV_ABS), code 54 (ABS_MT_POSITION_Y), value 24159
Event: time 1425319271.601632, type 3 (EV_ABS), code 48 (ABS_MT_TOUCH_MAJOR), value 649
Event: time 1425319271.601632, type 3 (EV_ABS), code 49 (ABS_MT_TOUCH_MINOR), value 354
Event: time 1425319271.601632, type 3 (EV_ABS), code 0 (ABS_X), value 10306
Event: time 1425319271.601632, type 3 (EV_ABS), code 1 (ABS_Y), value 30625
Event: time 1425319271.601632, -------------- SYN_REPORT ------------
...
Event: time 1425319271.606626, type 3 (EV_ABS), code 47 (ABS_MT_SLOT), value 0
Event: time 1425319271.606626, type 3 (EV_ABS), code 57 (ABS_MT_TRACKING_ID), value -1
Event: time 1425319271.606626, type 3 (EV_ABS), code 47 (ABS_MT_SLOT), value 1
Event: time 1425319271.606626, type 3 (EV_ABS), code 57 (ABS_MT_TRACKING_ID), value -1
Event: time 1425319271.606626, -------------- SYN_REPORT ------------
```

1.2.2. att. **evtest ierīču ievada notikumu testēšanas rīka izvada piemērs [1]**

Paši skāriena signāli ir izsūtīti no USB uz virtuāla faila (uz Raspberry Pi ierīcēm, faila atrašanās vieta ir parasti ***/dev/input/event0***). Linux operētājsistēma apraksta dažādu ierīču ievaddatus kā binārus notikuma objektus ar tipa kodiem, statusiem un pašām vērtībām [13]. Šie objekti ir notverti un publicēti seriālā veidā un ir viegli testējami ar ***evtest*** rīku [14]. Gan Ķirsone, gan maģistra darba autors izmantoja šo rīku, lai atrastu pieejamos skāriena signāla objektus un aprakstīt to tipus (skat.) [1].

Galvenie nozīmīgie daudz-skāriena signāla objekta (ABS_MT) tipi ir POSITION_X/Y koordinātu pozīciju vērtības, kuras parasti nav X,Y koordinātas pašreizējai izšķirtspējai. Lai iegūtu gala koordinātas izšķirtspējai, tad tās ir jāsadala ar ekrāna ierīces iekšējo (tīro) izšķirtspēju un jāsareizina ar operētājsistēmas izšķirtspēju.

Nākošie svarīgie tipi ir SLOT un TRACKING_ID, kuri apraksta dažādu vienlaicīgu skāriena uztveršanu. Katram vienlaicīgajam skārienam ir savs indekss iekš SLOT masīva un uzģenerēts

TRACKING_ID identifikators. Raspberry Pi operētājsistēma Raspbian atbalsta 10 vienlaicīgus skārienus (ABS_MT_SLOT indekss no 0 līdz 9), ko var pārbaudīt ar ***evtest*** programmu. Windows operētājsistēma parasti šo limitu atstāj fizisko ierīču draiveru ziņā.

Lai saprastu, vai pirksts pirmo reizi pieskarās vai arī atlaižas, tad ir jāskatās uz TRACKING_ID vērtību, kur pirmajam skārienam būs uzģenerēta jauns, unikāls identifikators. Skārienu atlaižot, TRACKING_ID kļūst par -1 pie atlaistā SLOT indeksa.

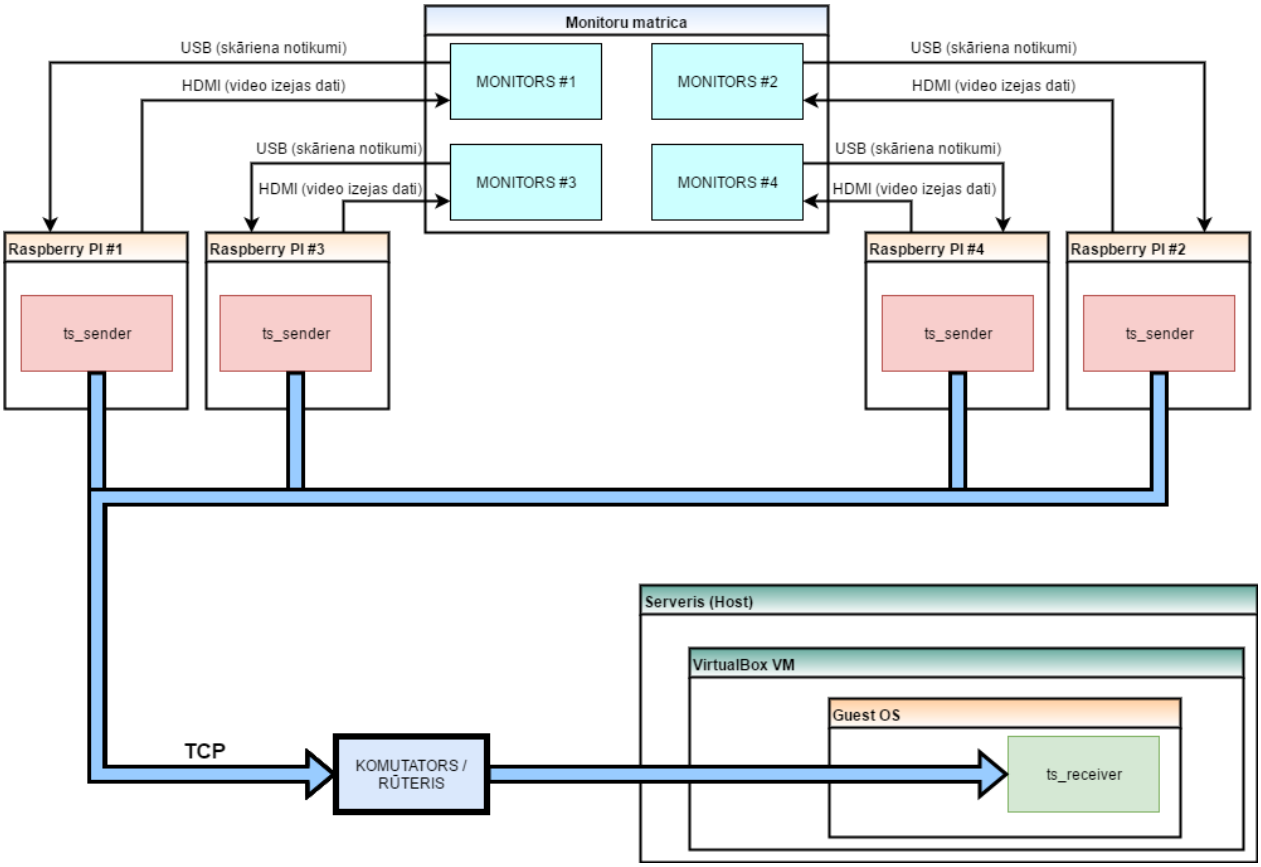
Skārienu signāliem arī ir papildus tipi (piemēram, TOUCH_MINOR/MAJOR), kuri var pateikt vairāk par skāriena īpašībām (piemēram, kāda ir pieskarošā pirksta orientācija). Maģistra darba izstrādes gaitā, autors saprata, ka liela daļa papildus tipa notikuma objekti var netikt implementēti fiziskajā ekrāna ierīcē. Piemēram, Raspberry Pi LCD skārienjūtīgais ekrāns ziņo tikai par minimālo tipu kopu, kamēr bakalaura darba ietvaros izmantotais, lielais un skārienjūtīgais IIyama ProLite T2735MSC-B1 monitors deva arī TOUCH_MINOR/MAJOR objektus.

Maģistra darba nolūkiem vairs dziļāk šos signālus aplūkot nav vērts un autors izmantos populāru bibliotēku *tslib*, kas izveidot viegli izmantojamu abstrakciju šo notikumu apstrādei un paralēli neliek raizēties par dažādu ekrānu ierīču atšķirībām [15]. Dziļāk par tās izmantošanu tiks aprakstīts 2. nodaļā (VIRTUALBOX IMPLEMENTĀCIJA).

Ķirsone implementācijā un testēšanas nodaļās pārsvarā testēja USB ekrāna ierīces un to skāriena signālus. Lielāks uzsvars tika likts uz plānu, kā tos atdarināt un iebūvēt Infiniviz projektā. Tā kā Infiniviz projekts balstās uz VirtualBox, tad ir jāizmanto funkcijas no tās saskarnes bibliotēkas, kuras ir pieejamas caur Windows COM vai Linux XPCOM tehnoloģijām. Autora bakalaura darbs bija izvēlējies signālus atdarināt ar tiešajiem operētājsistēmas funkciju izsaukumiem (t.i. *#include <windows.h>* vai *#include <linux/....h>* C kompilatora direktīvas). Maģistra darba ietvaros tiks veikts mēģinājums izmantot VirtualBox *IMouse* funkcijas, kuras atrada un ieteica Ķirsone savā darbā [16].

1.3. Bakalaura darba pētījums

Autora bakalaura darbs pētīja saņemto skārienu atdarināšanu uz citas operētājsistēmas. Ķirsones darbs bija dziļi aprakstījis, kā saņemt skārienus no Linux vides, bet to atdarināšanai tika pieminētas tikai metodes nosaukumi VirtualBox videi. Šīs metodes attiecīgi nebija notestētas autores darbā un visticamāk slēpj papildus nianses, kuras var atklāt tikai to implementācijas izpildes gaitā.



1.3.1. att. **Autora bakalaura darba demonstrētā arhitektūra [1]**

Diemžēl bakalaura darbs arī nebija dziļi aplūkojis VirtualBox implementāciju. Tas vairāk pārbaudīja skārienu atdarināšanu operētājsistēmas līmenī. Autors bija demonstrējis praktiskus testus starp Linux-veida operētājsistēmu kā skāriena avots un Windows operētājsistēmu kā skāriena atdarināšanas vide. Rezultāts ir dokumentēta, implementēta un testēta arhitektūra, kā var uzstādīt attālinātu monitoru skāriena pārsūtītājus un saņēmēja serveri, kas iespējams stāv iekš virtualizētas Windows operētājsistēmas (skat. 1.3.1. att.).

VirtualBox platforma netika pilnīgi ignorēta un autors bija veicis papildus teorētiskos pielikumus par tās izmantošanu. Viena no metodēm, kas tika apskatīta VirtualBox SDK dokumentācijā bija *putEventMultiTouch*, kura ir vēlāk aprakstīta "VIRTUALBOX IMPLEMENTĀCIJA" nodaļā un reāli pielietota. Bakalaura darbā šī metode bija tikai izkonspektēta no oficiālās *IMouse* interfeisa dokumentācijas [16]. Autors arī nebija daudz iedziļinājies par VirtualBox COM/XPCOM saskarnes pielietošanu (t.i. kā var nokompilēt gatavu programmatūru, kura izmanto VirtualBox COM/XPCOM funkciju izsaukumus).

Skārienu signālu ielasīšanai/simulācijai tika izmantots liels IIyama ProLite T2735MSC-B1 skārienjutīgs monitors, kuru aizdeva Latvijas Universitāte pētījuma nolūkiem. Šis monitors pārvada skārienus uz Raspberry Pi 2 B ierīci caur USB savienojumu. Diemžēl monitors izmanto USB 3.0 centrmezglu (angl. *USB 3.0 hub*), kurš nestrādā ar USB 2.0 ierīcēm (t.i. Raspberry Pi 2 B). Tā rezultātā, autoram vajadzēja izveidot speciālu konfigurāciju Raspberry Pi ierīcei, lai savienojums strādātu [20]. Maģistra darba nolūkiem autors izmanto Raspberry Pi 3 ierīci, kurai vairāk nav šādu problēmu.

Implementācija tika veikta izmantojot C++ (C++11 standarts) programmēšanas valodu, kura atļauj izmantot nepieciešamo zema līmeņa piekļuvi operētājsistēmas izsaukumiem (Windows ziņā tie ir SDK izsaukumi no *windows.h* galvenes faila). Izvēlētā programmēšanas valoda arī nodrošina iespēju izveidot operētājsistēmām specifiskus koda apgabalus vienā failā. Tas attiecīgi ļauj nokompilēt vairākus programmatūras variantus dažādām operētājsistēmām, kuri daudz-maz strādā vienādi.

Projekta struktūru un dažādu programmatūru kompilāciju autors pārvaldīja ar plaši izmantotu *CMake* rīku [17]. Šis rīks palīdz izveidot nosacītu loģiku, kā savienot un kompilēt failus, kādas bibliotēkas ir jāpievieno un kādi parametri ir jāizmanto kompilēšanas rīkiem. Autors projektam bija izveidojis vienu koplietotu bibliotēku, kuru izmantoja *ts_sender* skārienu pārsūtītājs un *ts_receiver* skārienu atdarināšanas serveris. Projekts ir pieejams GitHub tīmekļa platformā zem autora konta [19].

Autors arī bija izmantojis populārās *boost* C++ bibliotēkas, kuras iedod papildus palīglīdzekļus, kurus standarta bibliotēkas nepiedāvā. Viena no *boost* apakšbibliotēkām, ko autors izmantoja ir *asio*, kas atļauj izveidot dažādā veida (UDP/TCP) tīkla savienojumus bez operētājsistēmas specifiskiem kodu apgabaliem (t.i. *asio* ir abstrakcijas līmenis virs dažādiem operētājsistēmu izsaukumiem) [18].

Tīkla ziņā *ts_sender* un *ts_receiver* programmas komunicēja ar TCP/IPv4 protokolu ar standarta šablonu, kur sākumā nosūta objektu skaitu un pēc tam nolasa attiecīgos objektus. To visu varēja viegli realizēt ar *boost* apakšbibliotēkas *asio* palīdzību. Raspberry Pi ierīces izmantoja pievienotu USB WiFi adapteri, kuras pievienojas vietējām maršrutētajam un tālāk varēja savienoties ar centrālo serveri, kurā darbojās VirtualBox ar virtualizētu Windows operētājsistēmu. Šajā virtualizētājā operētājsistēmā attiecīgi strādāja *ts_receiver* programmatūra, kura saņem *ts_sender* signālus un tos atdarina ar Windows SDK izsaukumiem.

Viena no būtiskākajām problēmām šajā projektā ir signālu pareiza atdarināšana. Autors nevarēja tiešā veidā pārveidot Linux vidē monitora saņemtos skāriena signālus uz Windows vides. Testēšanā autors bija izmantojis virtualizētu Windows 8.1 operētājsistēmu, kuru programmatūras izstrādes komplekts jeb SDK nebija pilnīgi lietojams skārienu atdarināšanas ziņā. Skāriena emulācijas izsaukumi tika ignorēti un gala rezultātā vajadzēja ienākošos Linux skāriena signālus atdarināt kā peles taustiņu spiešanu. Ļoti iespējams, ka Windows skārienu emulācijas izsaukums netika pareizi lietots, bet diemžēl tā dokumentācija bija diezgan nepilnīga bakalaura darba izstrādes laikā (tagad gan tā ir mazliet uzlabojusies un iespējams strādā korekti uz Windows 10) [21].

Novērtējot bakalaura darbu, var saprast, ka tik vienkārša lieta kā skārienu atdarināšana uz jebkādās operētājsistēmas slēpj vairākus klūpakmeņus, kuri nav aprakstīti dokumentācijās. Autors arī bija veltījis lielu laiku *boost* un *asio* bibliotēku izpētei – to pareizai lietošanai. Tā kā darbs bija pārsvarā ar operētājsistēmas zemajiem izsaukumiem, tad vislabāk būtu bijis palikt pie parastās C valodas un iztikt bez papildus C++ standarta vai *boost* bibliotēkām, kuras stipri pasliktina kompilācijas ātrumu un palielina pirmkoda sarežģītību. Rīks *CMake* tomēr aizvien bija laba izvēle, jo tas atļauj izveidot gudru kompilācijas loģiku un ģenerēt dažādu programmēšanas vides rīku (angl. *IDE*) projekta failus, tātad to autors noteikti tālāk izmantos maģistra darbam.

2. VIRTUALBOX IMPLEMENTĀCIJA

Gan autora, gan Ķirsones darbi līdz šim nebija dziļi detaļās iegājuši skārienjūtīgas sienas risinājuma aprakstam, kas izmanto VirtualBox virtualizācijas platformu. Šis apraksts ir nepieciešams, lai izveidotu galīgo risinājumu, kuru var reāli praksē izmantot. Pirms tam tika apskatītas skārienjūtīgo signālu apstrādes komponentes un kā tās teorētiski var iekļaut sistēmā kā Infiniviz, kamēr praktiskie testi tika veikti ar konkrētas operētājsistēmas izsaukumiem.

Pirms autors iedziļināsies programmatūras izstrādē ar VirtualBox rīkiem, tiks izskaidroti bieži izmantoti jēdzieni par virtualizāciju un kā pareizi uzstādīt datoru, lai uz tā var darboties ar virtualizāciju. Parasti operētājsistēmas virtualizācija nav uzreiz pieejama, jo nepieciešamie iestatījumi ir atslēgti pēc noklusējuma un iespējams lielā uzņēmuma vidē tie var būt aktīvi bloķēti aizsardzības nolūkiem. Viens no šādiem aizsardzības nolūkiem ir novērst ļaunprogrammatūru (angl. *malware*) kā *Blue Pill*, kura ir aprakstīta zinātniskā IEEE konferences publikācijā [22].

Operētājsistēmas virtualizācijas process iekļauj sevī jēdzienus *saimnieku* un *viesi* (angl. *host* un *guest*). Jēdziens *saimnieks* šajā kontekstā nozīmē datoru ar operētājsistēmu kurā strādā VirtualBox programmatūra. Jēdziens *viesis* nozīmē virtualizētu operētājsistēmu, kura strādā VirtualBox programmatūras iekšienē. Abi šie jēdzieni tiek bieži izmantoti, kad runā par virtualizāciju un šajā gadījumā ir nozīmīgi saprast, kurā vidē atrodas dažādas programmatūras.

Vēl viena būtiska lieta, ko virtualizācija atļauj ir ierobežot virtuālo operētājsistēmu redzamos fiziskos resursus un portus. Programmatūra, kā VirtualBox atļauj ierobežot gan operatīvo atmiņu (RAM), gan cietā diskā izmēru, gan video kartes atmiņu (VRAM). Portu ziņā ir iespējams atļaut piekļuvi saimnieka sistēmas pieslēgtajām perifērijām, gan arī pieslēgt virtuālas ierīces, kuras saimnieka vidē emulē to darbību, kamēr viesa sistēma tos uztver kā reālas, fiziskas ierīces. Izteikts piemērs, ko autors arī izmantot savam projektam, ir pateikt virtuālajai sistēmai, ka peles navigācijai ir pieslēgta skārienjūtīga planšete. Saimnieka pusē šī planšete īstenībā ir pieslēgtā USB pele, bet viesa sistēma to vienkārši redz kā reālu un fizisku skārienjūtīgu planšetu (virtualizēta Windows operētājsistēma pat automātiski lejupielādē skārienjūtīgo ierīču draiverus, lai izmantoto šo ierīci).

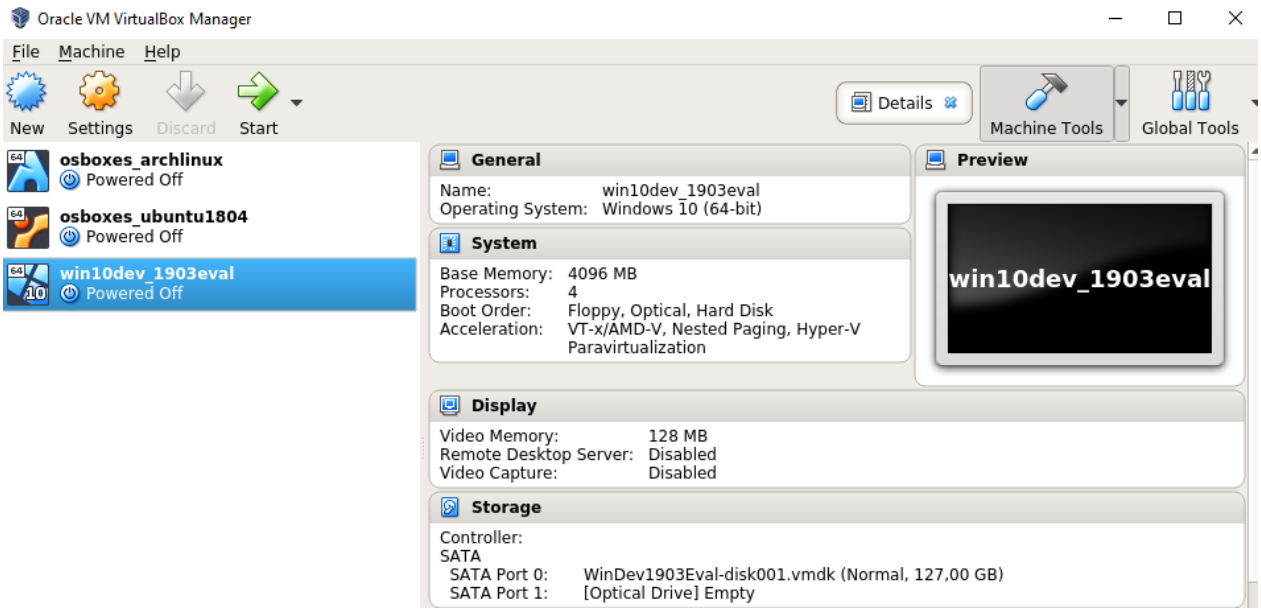
2.1. VirtualBox uzstādīšana Linux vidē

Tā kā operētājsistēmu virtualizācija ir diezgan sarežģīts process, tad bieži vien ir jāieslēdz papildus *saimnieka* datora virtualizācijas režīmi, ja grib to efektīvi pielietot. Gan Intel, gan AMD procesori atbalsta virtualizācijas tehnoloģiju un tos ir speciāli jāieslēdz iekš datora BIOS. Intel iestatījumu parasti nosauc par "Intel Virtual Technology", bet AMD iestatījuma nosaukums var atšķirties (atkarīgs no izvēlētās mātesplates). Vēl viena nianse ir saimnieka operētājsistēmas arhitektūra: nav iespējams virtualizēt 64bit operētājsistēmu uz 32bit saimnieka operētājsistēmas, bet var virtualizēt 32bit operētājsistēmu uz 64bit saimnieka operētājsistēmu.

VirtualBox instalēšana ir diezgan vienkārša Windows vidē, jo tā ir iepakota MSI instalācijas programmatūrā un attiecīgi to var ātri uzstādīt ātri izejot cauri instalācijas dialoga logam ar noklusētājiem iestatījumiem. Autors projekta nolūkiem bija izvēlējies Linux instalāciju, kuru uzstādīt ir mazliet sarežģītāk, ja negrib jaunāko versiju (autors izvēlās VirtualBox 5.2.26 versiju, bet šī darba rakstīšanas brīdi ir jau pieejama 6.0.6 versija).

Linux VirtualBox 5.2.26 instalāciju uzstāda izmantojot pieejamo Linux distribūcijas iepakojuma failu (piemēram, Debian veida distribūcijai ir .deb iepakojuma faili), kuru var palaist ar *apt* rīku uz Debian distribūcijas vai līdzīgiem pakotņu rīkiem no citām distribūcijām [23]. Ja instalācija nav pieejama šādās iepakojuma failos konkrētai Linux distribūcijai, tad vienīgais variants ir lejupielādēt VirtualBox pirmkodu un to nokompilēt – tas attiecīgi aizvilks daudz vairāk laika un būs nepieciešams uzinstalēt papildus kompilēšanas rīkus. Protams, ja izvēlas uzinstalēt jaunāko versiju, tad var vienkārši palaist "*sudo apt-get install virtualbox*" komandu, bet ļoti iespējams būs arī nepieciešams pievienot VirtualBox *apt-get* vietni.

Ar uzinstalētu VirtualBox programmatūru var sākt izveidot virtuālās *mašīnas,* kuras uzturu kādu no virtualizētām operētājsistēmām. Tīrais veids ir iegūt operētājsistēmas .iso failu (t.i. lejupielādēt no Ubuntu vai Windows oficiālās vietnes), kurus parasti ieraksta uz USB diska. Pēc tam šo .iso var izmantot, lai izveidotu virtuālo mašīnu ar norādīto operētājsistēmu ar iestatītiem fiziskajiem resursiem no VirtualBox grafiskās saskarnes. Tiks izveidots virtuāls disks, kur .iso fails tiks pieslēgts kā USB disks pie virtuālās mašīnas, pēc tam operētājsistēmu uzstāda līdzīgā veidā, kā uzstāda tās uz parastajiem datoriem. Beigās operētājsistēma būs uzinstalēt uz šī virtuālā diska un turpmāka virtuālās mašīnās palaišana izmantos uzinstalēto operētājsistēmu no virtuālā diska (t.i. .iso fails vairs nav nepieciešams).



2.1.1. att. **Autora uzinstalētās VirtualBox virtuālās operētājsistēmas**

Tā kā autoram vajadzēja notestēt vairākas operētājsistēmas un lieki nevēlējās tērēt laiku lejupielādēt .iso failus un manuāli uzstādīt operētājsistēmu iestatījumus, tad tika izmantoti gatavi virtuālie cietie diski (.vdi faili). Virtualizētām Linux operētājsistēmām autors bija atradis *osboxes.org* vietni ar gataviem virtuālajiem diskiem, kuros operētājsistēma ir jau nokonfigurēta ar optimāliem iestatījumiem un lietotāju/paroli [24]. Windows gadījumā Microsoft piedāvā izmantot jau sagatavotu VirtualBox eksportētu konfigurāciju ar jaunāko *Windows 10 Dev Evaluation* operētājsistēmu [25].

Uzinstalētās virtuālās operētājsistēmas arī ir spējīgas ciešāk savienoties ar saimnieka operētājsistēmu, ja uz tām ir uzinstalēti VirtualBox *viesu* pielikumi (*VirtualBox Guest additions*). Šie pielikumi atļauj kopēt teksta saturu, pārvietot failus un dinamiski izmainīt virtuālo izšķirtspēju starp saimnieku un viesi [26]. Tipiski VirtualBox šo pielikumu instalācijas failus piedāvā pievienot kā pieliktu virtuālu disku pie viesa sistēmas. Tālāk lietotājs viesa sistēma palaiž instalāciju un uzstāda viesa pielikumus. Sagatavotām viesa sistēmām no *osboxes.org* tie nav automātiski uzinstalēti (t.i. atsevišķi jāinstalē klāt), kamēr gatavā Windows viesa sistēma ar tiem jau nāk klāt uzinstalēta.

Izveidotās virtuālās mašīnās uz Linux vides atrodas lietotājā *'VirtualBox VMs'* mapē jeb */home/user/'VirtualBox VMs'*, kur katra apakšmape attiecas uz vienu no uzstādītajām virtuālām mašīnām. Papildus pie lietotāja izveidotajām virtuālām mašīnām ir VirtualBox konfigurācijas

mape, kura atrodas */home/user/.config/VirtualBox* mapē. Gan virtuālo mašīnu apakšmapēs, gan VirtualBox vispārīgās konfigurācijas mapē arī atrodas notikumu reģistrēšanas faili (angl. *log files*), kuri glabā visus notikumus, kuri attiecas uz to būtiskajiem objektiem.

```
<VirtualBox xmlns="http://www.virtualbox.org/" version="1.12-linux">
  <Global>
    <ExtraData>
      ... VirtualBox grafiskās saskarnes (GUI) iestatījumi ...
    </ExtraData>
    <MachineRegistry>
      <MachineEntry uuid="{c9e77153-32e9-4941-a2fd-fd3c55b6be77}"
       src="/home/azeroc/VirtualBox VMs/osboxes_archlinux/osboxes_archlinux.vbox"/>
      <MachineEntry uuid="{16853400-a977-4441-a85f-b9dc1d1b9be9}"
       src="/home/azeroc/VirtualBox VMs/osboxes_ubuntu1804/osboxes_ubuntu1804.vbox"/>
      <MachineEntry uuid="{04679777-00b7-44e7-97e5-6f13b53b6d6f}"
       src="/home/azeroc/VirtualBox VMs/win10dev_1903eval/win10dev_1903eval.vbox"/>
    </MachineRegistry>
    <NetserviceRegistry>
      <DHCPServers>
        <DHCPServer networkName="HostInterfaceNetworking-vboxnet0"
         IPAddress="192.168.56.100" networkMask="255.255.255.0"
         lowerIP="192.168.56.101" upperIP="192.168.56.254" enabled="1"/>
      </DHCPServers>
    </NetserviceRegistry>
    <SystemProperties defaultMachineFolder="/home/azeroc/VirtualBox VMs"
     defaultHardDiskFormat="VDI" VRDEAuthLibrary="VBoxAuth"
     webServiceAuthLibrary="VBoxAuth" LogHistoryCount="3"
     exclusiveHwVirt="true"/>
    <USBDeviceFilters/>
  </Global>
</VirtualBox>
```

2.1.2. att. ***/home/user/.config/VirtualBox.xml*** **piemērs**

Visi VirtualBox iestatījuma faili ir noformēti pēc XML formāta, kur */home/user/.config/VirtualBox.xml* fails apraksta visas lietotāja grafiskās saskarnes iestatījumus, atrastās virtuālās mašīnas, izmantotās tīmekļa interfeisa ierīces un citus sistēmas iestatījumus (skat. 2.1.2. att.). Var redzēt, ka virtuālās mašīnas ir, galvenokārt, identificētas ar UUID (vispārēji unikāls identifikators), bet savos atsevišķos iestatījuma .xml failos tiem arī pastāv tā lietotāja piešķirtais vārda identifikators kā, piemēram, *osboxes_archlinux* (skat. 2.1.1. att.). Abus no šiem identifikatoriem var izmantot, lai programmatūrā pareizi izvēlētos kādu no virtuālajām mašīnām, bet ieteicams ir izmantot UUID identifikatoru, jo vārda identifikators var mainīties.

Tā kā tīras virtuālās mašīnas neko neredz no ārpuses saimnieka pusē, tad tās arī neredz Interneta savienojumu. Šī iemesla dēļ VirtualBox un citas virtualizācijas platformas nodrošina virtuālu tīmekļa interfeisa adapteri, kas ir izveidots kā tilts starp saimnieka un viesa tīmekli. Pēc noklusējuma VirtualBox interfeiss ir NAT (Network Address Translation) tipa tīmekļa adapteris drošības nolūkiem. Tas principā nozīmē, ka viesis var piekļūt jebkuram interneta resursam, kas ir sasniedzams no saimnieka, bet Internets nevar nekādi pieslēgties jebkādam portam vai adresei viesa iekšienē.

Autora bakalaura darbam šo tīmekļa pieeju vajadzēja nomainīt uz mazliet atvērtāku risinājumu (VirtualBox port-forwarding), lai no lokālā tīkla ierīcēm varētu piekļūt skārienu saņēmēja serverim, kurš atradās viesa iekšienē. Maģistra darba ietvaros neko īpašu nevajadzēs darīt ar tīmekļa pieeju, jo autora risinājums strādās tiešā veidā ar VirtualBox SDK.

VirtualBox pēc noklusējuma arī neļauj strādāt ar *root* lietotāju. Programmatūra paredz, ka ar to strādā ne-*root* lietotāji, kuri ir ***vboxusers*** grupā. Protams, kad VirtualBox ir uzinstalēts, tad liela daļa no programmatūras komponentēm strādā ar *root* tiesībām, bet pats klienta līmenis ir domāts katram normālajam *vboxusers* lietotājam. Katram lietotājam būs savas virtuālās mašīnas un vispārīgais *VirtualBox.xml* iestatījumu fails

```
azeroc@azeroc-omen-laptop:~$ cat /etc/udev/rules.d/60-vboxdrv.rules

KERNEL=="vboxdrv", NAME="vboxdrv", OWNER="root", GROUP="vboxusers", MODE:="0660"
KERNEL=="vboxdrvu", NAME="vboxdrvu", OWNER="root", GROUP="vboxusers", MODE="0666"
KERNEL=="vboxnetctl", NAME="vboxnetctl", OWNER="root", GROUP="vboxusers", MODE="0660"
```

2.1.3. att. lietotāju grupas *vboxusers* VirtualBox ierīču atļauju labojums

Viens no sarežģījumiem, ar kuru autors sastapās un nebija aprakstīts oficiālajā VirtualBox dokumentācijā, ir */dev/vboxdrv* un */dev/vboxdrvu* ierīču tiesības. Uzinstalējot VirtualBox 5.2.26 šo abu ierīču tiesības ir *root:root*, kas neļauj nevienam lietotājam komunicēt ar VirtualBox servisiem, kuri ir nepieciešami virtuālo mašīnu startēšanai un stāvokļa izmaiņai. Pareizās piekļuves tiesības ir *root:vboxusers* (*root* lietotājs un *vboxusers* lietotāju grupa), bet pēc noklusējuma tādas nebija uzstādītas. Linux ierīču tiesības var modificēt caur */etc/udev/rules.d/* mapes iestatījuma failiem, kur autors bija izveidojis iestatījuma failu *60-vboxdrv.rules* VirtualBox ierīcēm (skat. 2.1.3. att.). Pēc jaunu iestatījumu ielikšanas iekš */etc/udev/rules.d/* mapē ir vai nu jārestartē dators, vai jāpalaiž "***udevadm control --reload***" komanda.

2.2. VirtualBox SDK

VirtualBox programmatūras izstrādes komplekts ir balstīts uz COM/XPCOM tehnoloģijas, kas atļauj atsevišķām un atdalītām programmatūrām bināri sazināties viens ar otru (ļoti tipisks piemērs ir veikt citas programmatūras COM publicētus funkciju izsaukumus). COM tehnoloģija ir Microsoft standarta izstrādājums, kur tās implementācija ir atdarināta ar Mozilla XPCOM uz citām platformām, kā, piemēram, Linux Ubuntu [27][28].

Galvenā ideja COM ir ka konkrēti bibliotēkas objekti jeb, VirtualBox gadījumā, funkciju izsaukumi ir pieejami globāli atmiņā un ir pieejami caur globāliem adrešu identifikatoriem. COM ir standarts nevis implementācija un tas nosaka, ka vienīgais veids, kā tikt klāt citas programmatūras izsaukumiem ir caur rādītājiem, tātad atmiņas adreses. Autors projekta izstrādes nolūkiem pārsvarā pielieto XPCOM implementāciju, bet VirtualBox ir abas implementācijas apvienojis zem *COM GLUE* abstrakcijas slāņa.

2.2.1. COM GLUE bibliotēka

Tā kā VirtualBox ir izstrādāts kā platformu neatkarīgs (angl. *platform independent*) risinājums, tad SDK var pielietot dažādos veidos: Java pakotne, COM, XPCOM vai COM GLUE. Autors bija izvēlējies COM GLUE VirtualBox iekšējo risinājuma, ko oficiālā dokumentācija arī iesaka. Visi COM varianti ir piedāvāti kā C/C++ pirmkods (ar galvenes un implementācijas failiem).

COM GLUE tehnoloģija apvieno gan Microsoft COM, gan XPCOM zem viena abstrakcijas slāņa, kur ir pieejamas abstrakti funkciju izsaukumi. Šie funkciju izsaukumi ir nokompilēti caur nosacītiem kompilācijas noteikumiem: ja tiek kompilēts uz Windows (ir pieejams *windows.h* fails), tad tiek izmantota Microsoft COM tehnoloģija, ja nē – izmanto Mozilla XPCOM.

VirtualBox oficiālais programmatūras izstrādes komplekts piedāvā ļoti izpalīdzīgu piemēru, kā tīrajā C programmēšanas valodā pielietot COM GLUE bibliotēku (skat. 1. pielikumu). Piemērs izmantot C GLUE galvenes failu, kurš attiecīgi pievieno pareizās versijas VirtualBox funkciju izsaukumu galvenes failu (autora gadījumā tas ir *VBoxCAPI_v5_2.h* fails). Pēc tam tas pievieno XPCOM vai COM implementācijas failus (t.i. *VirtualBox_i.c* un COM vai XPCOM galvenes failus no to *include* mapēm).

Šo COM/XPCOM apvienoto bibliotēku var izmantot tikai, ja VirtualBox ir jau uzinstalēts un tā */dev/vboxdrv* un */dev/vboxdrvu* ierīču faili ir pieejami (t.i. piekļuves tiesības ir *root*:*vboxusers*). Arī ir ļoti nozīmīgi, ka izmanto pareizo versiju gan uzinstalētajai VirtualBox programmatūrai, gan VirtualBox SDK, jo ir pamanīts, ka COM objektu rādītāju adreses atšķiras starp nozīmīgām versijām (angl. *major versions*). Nepareizas versijas arī visticamāk neiekļaus pareizus funkciju izsaukumu aprakstus (piemēram, vecākā versijā vienai funkcijai ir 3 argumenti, bet jaunākai versijai ir 5).

```
add_executable(MyVirtualBoxApp
    main.c
    ...

    # VBox API
    vbox_api/glue/VBoxCAPIGlue.c
    vbox_api/xpcom/lib/VirtualBox_i.c
)
target_include_directories(MyVirtualBoxApp
    PRIVATE ${CMAKE_SOURCE_DIR}/vbox_api/glue
    PRIVATE ${CMAKE_SOURCE_DIR}/vbox_api/include
    PRIVATE ${CMAKE_SOURCE_DIR}/vbox_api/xpcom/include
)
target_link_libraries(MyVirtualBoxApp m dl ${CMAKE_THREAD_LIBS_INIT})
```

2.2.1.1. att. **CMake kompilācijas iestatījumu fails VirtualBox C COM GLUE bibliotēkas izmantošanai**

Skārienu apstrādei un to atdarināšanai uz virtuālās mašīnās caur VirtualBox C COM GLUE bibliotēku ir nepieciešams pievienot šos galvenes un implementācijas failus pie programmatūras kompilācijas. Tā kā autors izmanto CMake un veidos projektu tikai uz Linux vides, tad ir jāpievieno XPCOM faili (*vbox_api/xpcom*), C GLUE faili (*vbox_api/glue*) un *VBoxCAPI_*.h* faili (*vbox_api/include*). Mape *vbox_api* satur nepieciešamās apakšmapes no VirtualBox SDK *sdk/bindings/c/glue*, *sdk/bindings/c/include* un *sdk/bindings/xpcom* mapēm. Kompilācijai arī nepieciešamas C standarta matemātikas bibliotēka *m*, dinamisko savienojumu bibliotēka *dl* un pavedienu bibliotēka *pthread*. Ar šo minimālo konfigurāciju (skat. 2.2.1.1. att.) ir iespējams izveidot programmatūru, kura var tiešā veidā ietekmēt VirtualBox darbību un kontrolēt pieejamās virtuālās mašīnas.

2.2.2. Nozīmīgās klases, objekti un metodes

Programmatūras izstrādei ar VirtualBox C COM GLUE bibliotēku (vai arī citu izstrādes komplekta veidu) ir jāizmanto tā definētās klases un izveidot šo klašu objektus. VirtualBox oficiālā dokumentācija par tās programmatūras izstrādes komplektu apraksta vismaz simts un vairāk klases, bet kā galvenās klases ir noteiktas *IVirtualBoxClient, IVirtualBox* un *ISession*. No šīm divām klasēm ir iespējams iegūt pārējo klašu objektus un attiecīgi izmantot to metodes [29].

Atmiņas piešķiršana / Objekta izveide

Pirms tiek aprakstīta katras klases būtība un tās asociētās metodes, ir jāsaprot atmiņas piešķiršana šo klašu objektu izveidei. Programmēšanas valoda C vai arī C++ pēc noklusējuma nepārvalda dinamiski izveidotos atmiņas apgabalus. Parasti izmanto metodes *malloc/free* (C/C++) vai *new/delete* (tikai C++) šādam nolūkam, bet VirtualBox ir savas piešķiršanas un atbrīvošanas metodes.

VirtualBox C COM GLUE bibliotēka dinamiskās atmiņas atbrīvošanai piedāvā globāli pieejamus izsaukumus *{Klases/Interfeisa vārds}_Release()*. Piemēram, *ISession* objekta *session_object_ptr* atmiņas atbrīvošanai izmanto *ISession_Release(session_object_ptr)* izsaukumu.

Atmiņas piešķiršanai arī izmanto globālas metodes pēc klašu nosaukuma un ar *_get_* atslēgvārdu pa vidu metodes nosaukumam. Jaunus klašu objektus var pārsvarā iegūt tikai no jau eksistējošiem klašu objektiem, izņēmumi ir klases kā *IVirtualBox* un *ISession*. Piemēram, *IConsole* klases objekta rādītāju (*IConsole* console*) var iegūt no metodes izsaukuma *ISession_get_Console(session, &console)*, kur ir svarīgi saprast, ka otrais arguments ir rādītājs uz rādītāju (metode izveidos atmiņas apgabalu *IConsole* objektam un liks ārpus metodes rādītājam rādīt uz to).

Liela daļa klašu metodes strādā ar rādītājiem un pašas metodes ir deklarēti kā globāli izsaukumi, kur pirmais arguments ir izsaukuma attiecinātās klases objekts. Metodes no kurām ir iespējams iegūt zemāka līmeņa klases objektus ir nepieciešams šo objektu rādītājs, kurš ir padots kā reference (tas nozīmē, ka metode savā iekšienē ir spējīgā ietekmēt, kur šis ārējais rādītājs rāda). Ja rādītājs nerāda uz pareizu adresi vai arī satur speciālo vērtību *NULL*, tad tā izmantošana VirtualBox metodēs sagraus esošo programmatūru.

IVirtualBoxClient

Klase *IVirtualBoxClient* ir palīgrīks, lai iegūtu nepieciešamos *IVirtualBox* un *ISession* objektus. Tās galvenā būtībā ir apskatīt programmatūras darbības kontekstu, kurā lietotāja vidē šī programmatūra tika palaista. Pēc šī konteksta apskatīšanas ir iespējams sazināties ar VirtualBox servisiem caur */dev/vboxdrv* ierīces un saņemt vajadzīgo informāciju par pareizu *IVirtualBox* un *ISession* objektu izveidi.

```
IVirtualBoxClient* vboxclient = NULL;
IVirtualBox*       vbox = NULL;
ISession*          vboxsession = NULL;
g_pVBoxFuncs->pfnClientInitialize(NULL, &vboxclient);
if (!vboxclient)
{
    fprintf(stderr, "FATAL: could not get VirtualBoxClient reference\n");
    return EXIT_FAILURE;
}
rc = IVirtualBoxClient_get_VirtualBox(vboxclient, &vbox);
if (FAILED(rc) || !vbox)
{
    fprintf(stderr, "FATAL: could not get VirtualBox reference\n");
    return EXIT_FAILURE;
}
rc = IVirtualBoxClient_get_Session(vboxclient, &vboxsession);
if (FAILED(rc) || !vboxsession)
{
    fprintf(stderr, "FATAL: could not get Session reference\n");
    return EXIT_FAILURE;
}
IVirtualBoxClient_Release(vboxclient)
```

2.2.2.1. att. **IVirtualBoxClient izmantošanas piemērs**

Šī palīgrīka objektu inicializē ar statiski globālā *g_pVBoxFuncs* objekta metodēm. Pēc tā iniciēšanas var izmantot *IVirtualBoxClient_get* SDK metodes, lai iedalītu atmiņu un sagatavot *IVirtualBox* un *ISession* rādītājus. Darbu beidzot ar *IVirtualBoxClient* objektu, atmiņa ir jāatbrīvo (skat. 2.2.2.1. att.). VirtualBox dokumentācija brīdina, lai veido tikai vienu un vienīgu *IVirtualBoxClient* objekta instanci.

IVirtualBox un IMachine

Klase *IVirtualBox* ir sākuma punkts virtuālo mašīnu pārvaldībai. Šajā kontekstā tas nozīmē atrast un iegūt virtuālās mašīnas jeb *IMachine* objektus pēc to identifikatoriem (UUID vai lietotāja iedotais vārds), izveidot jaunas vai arī pārvaldīt VirtualBox tīmekļa adapterus.

```
IVirtualBox* vbox = ...;
ISession* session = ...;
BSTR id;
BSTR sessionType;
IMachine* machine = NULL;
IProgress* progress = NULL;
g_pVBoxFuncs->pfnUtf8ToUtf16("{6990e729-8762-4993-b19b-78ec0e8c2f57}", &id);
g_pVBoxFuncs->pfnUtf8ToUtf16("gui", &sessionType);
IVirtualBox_FindMachine(vbox, id, &machine);
IMachine_LaunchVMProcess(*machine, session, sessionType, NULL, &progress);
```

2.2.2.2. att. **IMachine atrašana un tās procesa palaišana**

Maģistra darba izstrādes nolūkiem ir izmantotas metodes, lai atrastu iepriekš izveidotās virtuālās mašīnās un caur to objektiem tās palaist (skat. 2.2.2.2. att.). Klases *IMachine* objektam ir metode *LaunchVMProcess*, kas palaiž atsevišķu virtuālās mašīnas procesu. Tas arī automātiski iedod koplietotu slēdzeni (angl. *shared lock*) uz šo procesu, kas atļauj veikt stāvokļa-mainīgas darbības ar *ISession* objektu.

No *IVirtualBox* objekta var arī iegūt meta-informāciju, piemēram, par VirtualBox instalāciju (tās versiju un lietotāja VirtualBox iestatījumu mapes atrašanās vietu). Vēl papildus var iegūt virtuālo mašīnu resursu iestatījumus caur to identifikatoriem (resursi šajā ziņā ir atļautais procesoru kodolu skaits, operatīvā atmiņa, video atmiņa, cietā diska vietas izmērs, u.c.).

Virtuālās mašīnas process, kurš tika palaists ar *LaunchVMProcess* metodi ir pilnīgi atdalīts no esošās programmatūras darbības. Tas nozīmē, ka beidzot programmatūras darbu, virtuālās mašīnas process aizvien strādās un virtualizētā operētājsistēma neko dīvainu nepamanīs. Maģistra darba praktiskais projekts izmanto *IConsole* objektu, lai šo papildus procesu pareizi likvidēt.

ISession, IConsole, IMouse un IProgress

Klase *ISession* ir sesijas instance, kura ir piesaistīta kaut kādai virtuālai mašīnai jeb *IMachine*. Šis objekts ir nepieciešams, lai manipulētu virtuālās mašīnas darbību (piemēram, simulēt skāriena signālus uz virtualizētās operētājsistēmas).

```
ISession* session = ...;
IConsole* console = NULL;
IMachine* machine = NULL;
ISession_get_Console(session, &console);
ISession_get_Machine(session, &machine);
IMouse* mouse = NULL;
IProgress* progress = NULL;
IConsole_GetMouse(console, &mouse);
IConsole_PowerDown(console, NULL);
IMouse_PutEventMultiTouch(mouse, ...);
```

2.2.2.3. att. **ISession un to apakšklašu objektu daži izmantošanas piemēri**

Visnozīmīgākā apakšklase zem *ISession* ir *IConsole*, kura tālāk iedalās virtuālās mašīnas ierīču interfeisos. Piemēram, ja grib dabūt piekļuvi virtuālās mašīnas pelei un to manipulēt, tad ir vispirms jāiegūst *IConsole* objekts, pēc tam no *IConsole* objekta iegūt *IMouse* objektu.

Klases *IMouse* objekts piedāvā nosūtīt gan peles taustiņu notikumus, gan vairāku pirkstu skārienjūtīgos notikumus. Abas metodes paļaujas, ka programmatūra ievadīs pareizas X, Y koordinātas, kuras atbilst viesa operētājsistēmas izšķirtspējai. Nākošie parametri ir peles taustiņa vai skāriena/pirksta stāvoklis (t.i. labais/kresais/vidējais klikšķis peles darbībām vai arī ir-pieskāries/nav-pieskāries stāvoklis skārienam). Skārieniem arī ir pirkstu identifikators (ja ir piespiesti vairāki pirksti).

Bez *IMouse* objekta, *IConsole* objekts arī piedāvā piekļuvi USB ierīču pārvaldībai, virtuālās mašīnas displejam, virtuālās operētājsistēmas starpliktuvei (angl. *clipboard*), klaviatūrai un citām perifērijām. Klases *IConsole* objektam vēl var mainīt virtuālās mašīnas strādāšanas režīma stāvokli (likt gulēt, likt hibernāciju, uztaisīt pauzi, izslēgt ārā un restartēt).

Klases *IProgress* objekts ir vienkāršs palīgrīks ilgstošām operācijām. To padod ilgai operācijai kā virtuālās mašīnas startēšanu. Pēc tam to var izmantot, lai gaidītu (bloķēt programmatūras izpildi uz tekošā pavediena) uz attiecīgo ilgstošo operāciju.

IFramebuffer

Klase *IFrameBuffer* ir objekts, kas apraksta virtuālās mašīnas viesa operētājsistēmas darbvirsmas grafisko saturu. Šis ir zema līmeņa interfeiss, kurš apraksta atmiņas apgabalus, kuros glabā pikseļu datus, kuri fiziskajām mašīnām būtu kadru apgabali monitoriem. Tam ir metodes lai iegūtu grafisko bitmapu saturu un kontrolētu virtuālo izšķirtspēju, kā arī pieprasīt papildus meta informāciju par video kadru apmaiņu.

Šīs klases objektu tiešā veidā nevar iegūt no standarta klases objektiem kā *ISession*, *IMachine* vai *IConsole*. Tas ir jāimplementē ar savu *IFramebuffer* klasi, kura atbalsta visas vajadzīgās interfeisa funkcijas, pēc tam no tā ir jāizveido objekts. Objektu padod tālāk *IDisplay* metodei *AttachFramebuffer*. Klases *IDisplay* objektu iegūst no *IConsole* objekta un *AttachFramebuffer* metode, saņemot *IFramebuffer* objektu, to tālāk izmanto periodisku kadru uzņemšanas notikumu ziņošanai. Virtuālā mašīnā savā iekšienē uzņems viesa darbvirsmas kadrus un paziņos par tās būtību visiem pievienotajiem *IFramebuffer* objektiem

Iegūtos kadrus, par kuriem noziņo *IFramebuffer* objekta *notifyUpdate* metodē, nāk ar noteiktu izšķirtspēju, kuru var kontrolēt caur tām pašām *IFramebuffer* metodēm. Tas uzreiz var nestrādāt un virtuālā mašīna iespējams neatbildēs uz tādu izsaukumu, ja nav uzinstalēti *VirtualBox Guest additions* uz attiecīgās viesa operētājsistēmas (skat. 2.1. nodaļu).

Autors šo interfeisu nav implementējis projekta nolūkiem, bet tas ir aizvien nozīmīgs, ja vēlas izveidot video signālu pārraidi. Infiniviz projekta risinājums ir patiešām izmantojis šo klasi, lai iegūtu viesa operētājsistēmas darbvirsmas kadrus un tos pārraidītu uz atsevišķām monitora ierīcēm. Diemžēl šis apraksts ir ņemts no savāktās informācijas VirtualBox SDK dokumentācijā un, no autora pieredzes, tās pareiza izmantošana noteikti slēpj vairākas sīkas nianses.

2.3. Infiniviz VirtualBox implementācija

Infiniviz publikācijas un pētījumi tika veikti caur praktisku darbu ar programmatūru, kura bija izstrādāta ar VirtualBox SDK. Risinājuma galvenais mērķis bija izveidot mērogojamu monitora sienas sistēmu, kura ir bāzēta uz VirtualBox platformas. Ar mērogojamību ir saprotams tas, ka var pieslēgt patvaļīgu skaitu monitoru pie virtuālās viesa operētājsistēmas.

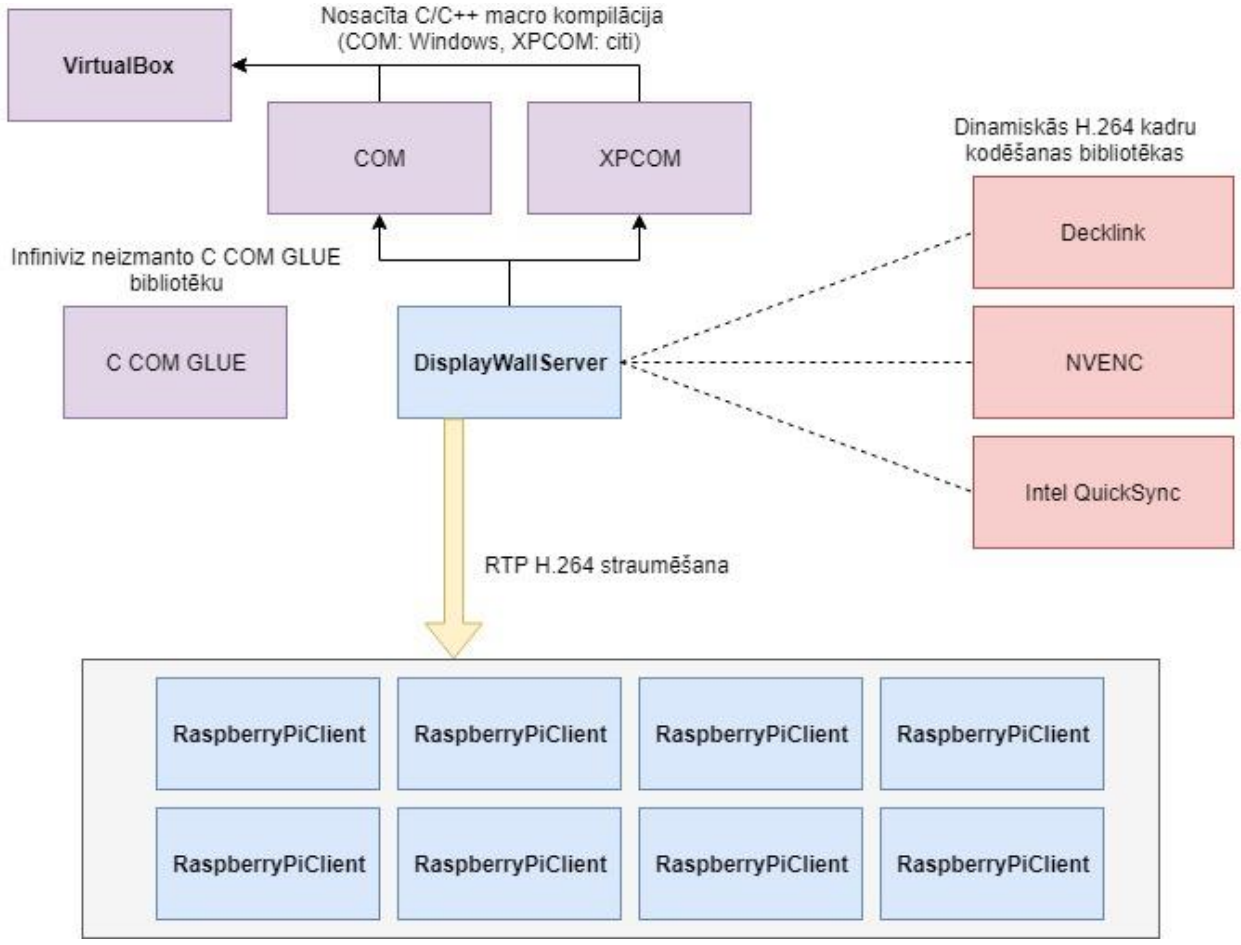
Risinājums arī paredz, ka serveris ir pietiekami spēcīgs un tam ir pieejama NVidia grafikas karte. Tas ir nepieciešams, lai varētu nokompilēt H.264 kodējuma bibliotēku ar CUDA kompilātoru *nvcc* [32]. CUDA ir milzīga NVidia grafikas karšu programmatūras izstrādes komplekts, ar kuru palīdzību ir iespējams nokompilēt asinhronu jeb paralēlu pirmkodu, kas ātri darbojās uz simtiem līdz tūkstošiem CUDA procesoriem [33].

NVidia izstrādes komplekta CUDA kompilators *nvcc* arī tiek izmantots, lai nokompilēt pirmkodu, kas izmanto NVENC programmatūras izstrādes komplektu [34]. NVENC (un NVDEC) ir izmantots, lai iekodētu un dekodētu H.264 kadrus no VirtualBox viesa darbvirsmas grafiskā satura. Straumēšanas iespējas caur H.264 kodēšanu ir atkarīgas no H.264 izvēlētā profila un līmeņa, kas beigās ietekmē minimālo tīmekļa pārraides ātrumu, kas ir nepieciešams, lai straumēšana būtu veiksmīga. Piemēram, 1280x720 izšķirtspējas kadram ar ātrumu 30 kadri sekundē ir vajadzīgs 14,000 kbit/s ātrums [35].

Diemžēl Infiniviz risinājuma projekts izmantoja VisualGDB Visual Studio paplašinājumu [36], lai veiktu attālinātu programmēšanu gan ar Raspberry Pi ierīcēm, gan ar serveri, kas uztur VirtualBox. Šis paplašinājums ir maksas produkts un piedāvā tikai 30 dienu izmēģinājumu un nelies laiks arī ir jāvelta, lai izlabotu attālināto ierīču adrešu iestatījumus. Autors šo problēmu atrisināju pārveidojot projektu uz CMake kompilēšanas sistēmu, kur Visual Studio 2019 versija atļauj strādāt ar attālinātām ierīcēm caur CMake projektiem.

Tā kā gala rezultātā Infiniviz pirmkoda projekts tika pārveidots uz CMake tipa projektu, tad nākošais solis ir izveidot pareizu kompilēšanas iestatījumus (skat. 2. pielikumu). Pirmkoda failu pielikšana ir diezgan vienkāršs process, bet svarīgāk ir saprast, kādas bibliotēkas ir jāsavieno ar gala nokompilēto programmatūru. Bieži vien programmatūrām arī ir vajadzīga pavedienu bibliotēka kā *pthread*, kuru var iegūt caur *FIND_PACKAGE(Threads REQUIRED)* izsaukumu.

Autors šī darba ietvaros nav publicējis galējo *Infiniviz* pirmkoda glabātuvi. Tā tika iegūta no BitBucket Git glabātuves, kur tiesības tās piekļūšanai iedeva Rūdolfs Bundulis. Pirmkoda detalizētai izpētei būtu jākontaktējas ar R. Bunduli, lai iegūtu līdzīgu pieeju. Autors vienkārši apraksta kā Infiniviz pirmkoda failus var savienot kopā ar CMake projekta risinājumu.



2.3.1. att. **Infiniviz risinājuma komponenšu mijiedarbība**

Sākotnēji Infiniviz pirmkods bija nokompilēts uz 4. VirtualBox versijas un vēlāk uz 5. versijas. Autors vēlējās to nokompilēt ar, precīzi sakot, 5.2.26. versiju. Viena no būtiskām lietām, ko autors bija pamanījis ir C COM GLUE bibliotēkas neizmantošana. Infiniviz risinājums ir izveidojis savus C/C++ macro kompilatora direktīvas, lai pareizi nokompilēto tiešos COM/XPCOM izsaukumus. Tas iespējams iedeva lielāku kontroli pār izstrādes komplekta pielietošanu, bet uzreiz tika pamanītas problēmas, kad risinājums bija jāatjaunina uz augstāku VirtualBox versiju. Autoram vajadzēja pielabot vairākas VirtualBox izsaukumu funkciju struktūras, lai tas nokompilētos ar jaunāko 5. versiju (5.2.26).

Vēl viena interesanta nianse ir H.264 kadru kodēšanas rīki – tie nav statiski pievienoti programmatūrai *DisplayWallServer* (monitora sienas serveris). Tie ir nokompilēti kā dinamiskās bibliotēkas, kuras *DisplayWallServer* programmatūras darbības laikā pievieno pēc vajadzības. Risinājums pārsvarā izmanto NVENC kodēšanas bibliotēku, bet autors bija Infiniviz mapē atradis gatavus apakšprojektus ar Decklink [37] un Intel QuickSync [38] implementācijām. Tās attiecīgi balstās uz citiem fiziskām ierīcēm (attiecīgi Decklink kodētājs/dekoderis un Intel procesors).

Pēc visām savienotājām bibliotēkām un COM/XPCOM pielietošanu, servera programmatūra palaiž virtuālo mašīnu *headless* režīmā, kas nozīmē, ka virtuālā mašīna ir startēta bez grafiskā interfeisa procesa. Pat ja šāds process nav palaists, VirtualBox aizvien var ģenerēt darbvirsmas kadrus, ja tam ir pievienots kaut viens *IFramebuffer* objekts.

Pēc virtuālās mašīnas palaišanas, serveris skatās savos iestatījumos, kur var atrast monitora ierīces (IPv4 adreses, kādu izšķirtspēju katra ierīce grib, kādu kodētāju pielietot). Ja kadru saņēmēja programmatūra *RaspberryPiClient* ir palaista uz šīm attālinātājām ierīcēm, tad serveris veic tīmekļa savienoju un straumē H.264 iekodētus kadrus caur RTP protokolu [12].

Attālināto ierīču saņēmēju programmatūra *RaspberryPiClient* izmanto OMX Player bibliotēku [11], lai dekodētu un parādītu H.264 iekodēto attēlo. Šis attēls automātiski aizņem visu ierīces pieslēgto ekrānu, jo zemā līmenī OMX izmanto Linux kadru bufera ierīci [39]. Jebkas, kas tiek ierakstīts šajā buferī ir parādīts uz pieslēgtā ekrāna un tā nepareiza lietošana var nobloķēt grafisko saturu, kamēr ierīce nav restartēta.

Novērtējot Infiniviz implementāciju, autors secināja, ka risinājums ir mazliet jāpārveido, lai izmanto oficiāli ieteikto C COM GLUE bibliotēku, kura ir izveidota tieši vienkāršības iemesla dēļ. Papildus problēmas, ar ko autors sastapās, ir NVENC kodēšanas bibliotēkas nokompilēšana – tā kā Infiniviz bija lietojis vecas grafikas kartes (GTX 6. paaudze), tad programmatūra kļuva nestabila ar jaunākām paaudzēm (autors izmantoja GTX 1060, kas ir jaunāka par 4 paaudzēm). Nestabilitāti izsekoja līdz NVENC bibliotēkas izsaukumiem, kur atmiņas apgabals netika pareizi lietots un tas sagrāva servera programmatūras darbību ar SIGSEGV signālu (Linux signāls, kas parasti liecina par neatļautu atmiņas apgabala izmantošanu). Pēc autora viedokļa, vislabāk būtu aizstāt NVENC ar x264 bibliotēku, kas izmanto parasto procesoru, lai kodētu kadrus [8].

3. SKĀRIENJŪTĪGO MONITORU SIENAS IMPLEMENTĀCIJA

Autors praktiskajā projektā mēģina pielietot VirtualBox programmatūras izstrādes komplektu, lai izstrādātu sistēmu, kas pārraida skāriena signālus no skārienjūtīgiem ekrāniem uz serveri, kas tos atdarina uz viesa operētājsistēmas. Risinājums paredzēs kontrolēt patvaļīgus virtuālās sienas apgabalus ar skāriena signāliem, kur tie var vienlaicīgi nākt no vairākām ekrāna ierīcēm. Šī ir klasiskā servera-klienta problēma, bet ar papildus VirtualBox saskarnes ierobežojumiem.

Sākotnējā apakšnodaļa aprakstīts izstrādātās arhitektūras vispārīgo bildi, kurā arī būs iekļautas plānotas funkcionalitātes, kuras netiks praktiski implementētas šajā darbā. Autors salīdzinās savu arhitektūras plānu ar citām arhitektūrām no autora bakalaura darba un Ķirsones darba. Salīdzinājumos būs apskatīts, kas tiks paņemts no citām arhitektūrām un kas ir jāuzlabo, lai rastu risinājumu monitora sienas sistēmai ar VirtualBox saskarni.

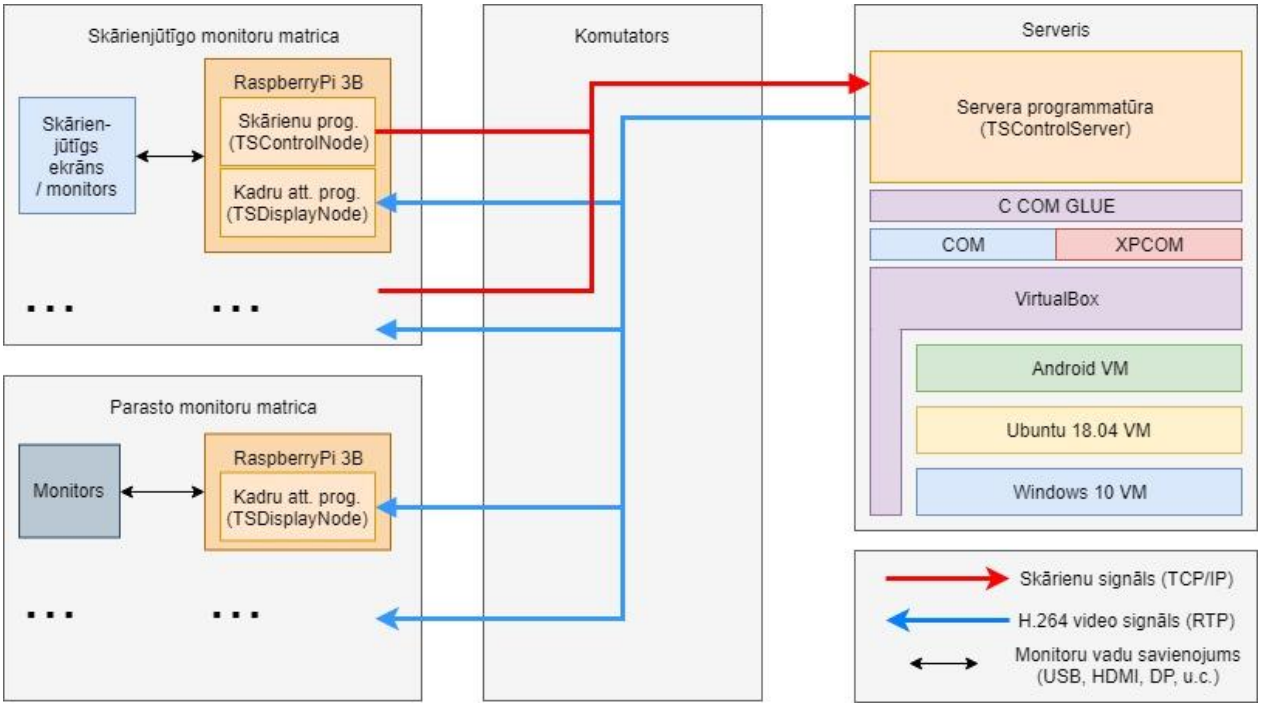
Otrajā apakšnodaļā ir detalizēts apraksts par skārienu saņemšanu no ekrāna ierīcēm (t.i. kā tās ir saņemtas ar ierīcēm, kuras ir pieslēgtas pie ekrāniem/monitoriem), kā arī to apstrādi. Ar skārienu signālu apstrādi ir domāts pielabot koordinātu vērtības un pārkodēt skāriena signāla datus dažādos formātos. Līdzīgi pirmajai apakšnodaļai, skārienu signālu apstrāde arī būs salīdzināta ar apstrādes teoriju no autora bakalaura un Ķirsones darbiem.

Pēdējā apakšnodaļa dokumentē, kā servera programmatūra saskarās ar VirtualBox programmatūras izstrādes komplektu. Serveris veido saskarni ar VirtualBox, lai kontrolētu un palaistu virtuālās mašīnas, uz kurām griežas viesa operētājsistēmas. Ar šo saskarni arī ir realizēta skārienu signālu atdarināšana/emulācijas uz viesa operētājsistēmas caur VirtualBox C COM GLUE bibliotēkas funkciju izsaukumiem.

Tā kā autors neveiks implementāciju pilnai monitora sienas sistēmai, bet būs vismaz aprakstījis tās pilno arhitektūru pirmajā apakšnodaļā, tad tās nākotnes implementācija ir aprakstīta pēdējā lielajā pamatnodaļā *5. ATTĪSTĪBA NĀKOTNĒ*.

3.1. Monitoru sienas arhitektūra

Ar parasto monitoru sienas arhitektūru kā Infiniviz ir saprasts, ka pastāv viena vienvirziena datu plūsma, kur serveris no virtuālās sienas pārraida grafiskos datus uz fiziskajiem monitoriem caur tīmekli un monitoru pieslēgtajām ierīcēm. Šādu šablonu ir vienkārši realizēt un tas arī vienkāršo iespējamo monitoru konfigurāciju (ir vairāki monitoru saslēgti matricās vai arī vairāki monitoru izvietoti dažādās istabās, bet pieslēgti pie viena servera).



3.1.1. att. **Autora ierosinātā arhitektūra**

Ar skārienjūtīgo monitora sienas arhitektūru, risinājums paliek daudz sarežģītāks. Vairs nav runa par vienu vienīgu vienvirzienu datu plūsmu, bet jauktu datu plūsmu. Jauktajā datu plūsmā ir iesaistīti grafiskie kadri, skāriena signāli un potenciāli speciāli kontroles signāli sarežģītākām sienas konfigurācijām.

Autors ar savu sienas arhitektūru apvieno Ķirsones arhitektūru ar sava bakalaura darba skārienu apstrādes un emulācijas metodiku. Pie apvienotās versijas ir paredzēta arī parasto monitoru iekļaušana, kur skārienjūtīgie monitori var būt kaut kāda veida kontroles paneļi. Tas atspoguļojās trīs dažādās programmatūrās – servera programmatūra *TSControlServer*, skārienu pārraides programmatūra *TSControlNode* un video atspēlēšanas programmatūra *TSDisplayNode* (skat. 3.1.1. att.).

Servera programmatūrai ir cieša saskarne ar VirtualBox COM/XPCOM ietvaru. Infiniviz projekts bija vairākkārt izlaidis C COM GLUE izmantošanu, kur tiešā veidā sazinājās ar COM vai XPCOM izsaukumiem. Autora programmatūra šo abstrakcijas līmeni neizlaiž, lai nākotnē atvieglotu atjaunināt programmatūru ar jaunākām VirtualBox versijām. Tas arī samazina programmatūra sarežģītību un nav daudz jādomā par nosacītu loģiku, kā pārslēgties starp dažādu operētājsistēmu loģiku (Microsoft COM vai Linux Mozilla XPCOM).

Lai iegūtu lielāku kontroli pār virtuālajām mašīnām, tad serveris arī atbild par virtuālo mašīnu palaišanu un attiecīgu procesu pārvaldību. Šī papildus kontrole atļauj pilnībā rezervēt viesa operētājsistēmu servera programmatūrai, kas iedod iespēju veikt jebkādā veida ārējo manipulāciju (t.i. simulēt perifēriju darbības). Arhitektūras piemērā ir parādīts, ka serveris var izvēlēties palaist kādu no Android, Ubuntu vai Windows virtuālajām mašīnām, vai arī uzstādīt un palaist vēl citas viesa operētājsistēmas.

Autors ir izvēlējies, ka ārējās programmatūras (t.i. programmatūra, kura strādā uz Raspberry Pi ierīcēm) atbild tikai par vienu uzdevumu. Šajā gadījumā ir divi dažādi uzdevumi, kurus apstrāda ar divām dažādām programmatūrām:

1. TSControlNode – skārienu pārraide no ekrānu ierīcēm, kur skāriens ir uztverts kā reģistrēts notikums no monitora/ekrāna ierīcēm un tālāk aizsūtīts caur TCP/IP protokolu uz serveri,
2. TSDisplayNode – video signālu apstrāde, kur serveris nepārtraukti nosūta virtuālās sienas grafiskā satura kadrus uz monitora/ekrāna Raspberry Pi ierīcēm, kurus pēc tam rāda uz visa ekrāna.

Kā redzams “Skārienjūtīgo monitoru matrica” blokā, Raspberry Pi ierīce var darbināt abas šīs programmatūras vienlaicīgi. Tas nozīmē, ka tā vienlaicīgi saņem video kadru signālu no servera, kas ir jāatspēlē uz monitora/ekrāna ierīces, gan arī vienlaicīgi saņem skāriena signālus no monitora/ekrāna ierīces, kuri ir jāapstrādā un jāpārraida uz serveri.

Blokā “Parasto monitoru matricā” ir saslēgti monitori ar Raspberry Pi ierīcēm, kuras tikai atspēlē saņemto video signālu no servera. Tā kā skārienjūtīgie monitori ir daudz dārgāki par parastajiem monitoriem, tad šis variants var atļaut uzstādīt skārienjūtīgos monitorus kā kontroles paneļus parastajiem monitoriem. Tas nozīmētu, ka skārienjūtīgie monitori saņemtu to pašu video signālu, ko saņem parastie monitori. Iespējams varbūt var tikt izstrādā papildus programmatūra,

kur skārienjūtīgie monitori var brīvi pārslēgt dažādus video straumēšanas signālus, kuri ir domāti dažādiem parastajiem monitoriem.

Komutators šajā arhitektūrā ir atstāta kā neformāls apgabals. Tas var kļūt par ļoti sarežģītu tīmekli, ja, piemēram, vienā sistēma ir pieslēgti vairāki monitori, kuri ir ļoti tālu viens no otra (piemēram, dažādās ēkās ar dažādiem tīmekļiem). No otras puses komutatora apgabals var būt vienkāršs lokālais tīkls, kur pat kabeļi nav jāsaslēdz kopā (var izmantot lokālo WiFi tīmekli). Autors šajā gadījumā neparedz nepieciešamību pēc speciālas fiziskās tīmekļa konfigurācijas. Vienīgais, kas ir paredzēts, ir tīmekļa fiziski atbalstītais ātrums, jo tas ierobežo, cik lielas izšķirtspējas kadrus var sūtīt caur tīklu.

Loģiskajā tīmekļa konfigurācijā ir paredzēts atļaut caurlaidi vismaz diviem portiem (skārienu un video signāli). Visas attiecīgās programmatūras atļauj izvēlēties patvaļīgu portu, ar kuru strādāt. Arī ir paredzēts, ka nebūs problēmas izmantot gan TCP, gan UDP protokolu, kur jebkurš no diviem var tikt izmantots RTP protokola implementācijai [12].

Autors arī nepievērš īpašu uzmanību pārraidīto datu drošībai un paredz, ka tīkls pats par sevi būs pietiekami norobežots, lai nepiederīgās programmatūras nesūtītu vai nenolasītu risinājuma pārsūtītos datus. Ja ir nepieciešama papildus datu šifrēšana, tad tas pats par sevi noteikti būs cita darba pētījums un praktiska implementācija.

Tā kā programmatūra cieši saskarās ar fizisko līmeni, tad šo programmatūru implementāciju autors veiks ar C programmēšanas valodu un CMake projektu kompilācijas rīku. Tā kā VirtualBox C COM GLUE bibliotēka ir implementēta ar tādu pašu programmēšanas valodu, tad nav jātaisa papildus saskarnes slānis starp dažādām programmēšanas valodām. Ar CMake autors paredz ātru un vienkāršu veidu, kā automātiski uzģenerēt kompilācijas loģiku, lai pareizi “sasietu” visus pirmkoda failus kopā un izveidot attiecīgās programmatūras. Realitātē autora parādītā arhitektūra atļauj izmantot arī vairākas citas programmēšanas valodas, lai izveidotu risinājumu, bet tām ir jābūt spējīgām sazināties ar VirtualBox programmatūras izstrādes komplekta saskarni.

3.2. Skārienu saņemšana un apstrāde

Ar skārienu saņemšanu ir domāts gan saņemt skārienu no monitora ierīces pašas, gan arī saņemt skāriena signālu servera pusē. Jebkurā gadījumā tam arī ir jābūt pareizi noformētām apstrādes procesā. Tas nozīmē to iegūs dažādos datu formātos un viena skāriena signālu vajadzēs pārveidot vai paplašināt ar papildus informāciju. Ir arī paredzēts, ka skāriena signāli var nākt no dažādiem pirkstiem, kuri ir vienlaicīgi pieskarušies pie skārienjūtīgā ekrāna. Protams, tādi gadījumi parasti ir tikai divi vai trīs pirksti, kuri mēģina izmantot speciālos žestus (piemēram, pietuvināt ekrāna saturu).

Skārienu pārraidi var teorētiski sūtīt gan caur TCP, gan UDP protokolu. Ar UDP protokolu būtu vieglāk iekodēt un pārsūtīt viena signāla vienību datus, jo UDP pēc būtības pārraida datu paketes. Autors gan bija izvēlējies TCP variantu, kur dati ir pārraidīti nepārtrauktā datu straumē, kur ir papildus jāiekodē sākuma un beigu simboli, lai atšķirtu dažādu signālu vienības. Tas tika izlemts, lai pasargātos no UDP pakešu zušanas, jo UDP protokolam nav iebūvēts kontroles signāli, lai pārbaudītu veiksmīgu datu nosūtīšanu, kamēr TCP ir. Veiksmīga nosūtīšana ir svarīga, jo skāriena signāliem ir dažādi pirksta nospiešanas stāvokļi un papildus pārbaudes tik ļoti neietekmē ātrdarbību, ja nu vienīgi pašam tīklam ir kaut kādas problēmas.

```
// Common touch signal struct
#define TOUCH_SIG_STRFMT "%u %u %hhu %d %hhd\n"
struct touch_sig {
       uint32_t x;
       uint32_t y;
       uint8_t slot;
       int32_t tracking_id;
       int8_t touch_state; // -1: doesnt change state / still touching,
                           //  0: touch release, 1: initial touch
};
```

3.2.1. att. **Skāriena signāla loģiskā struktūra**

Skārienu signālus saņem no monitora jeb ekrāna ierīcēm, bet šie signāli var tikt reģistrēti dažādos veidos, atkarībā no operētājsistēmas. Tā kā ir skārienu pirmajai saņemšanai ir izmantotas lētās Raspberry Pi ierīces, tad šī operētājsistēma pārsvarā būs Raspbian [40]. Tā ir viena no Linux distribūcijām, tātad skāriena signāli tiek aprakstīti kā perifērijas ierīču ievada notikumi (*/dev/input/eventX* ierīces faili) [13][14]. Ar programmatūru *TSControlNode* šo signālu struktūra tālāk ir pārveidota uz loģisko struktūru, kuru tālāk izmanto apstrādei un tīmekļa pārraidīšanai (skat. 3.2.1. att.).

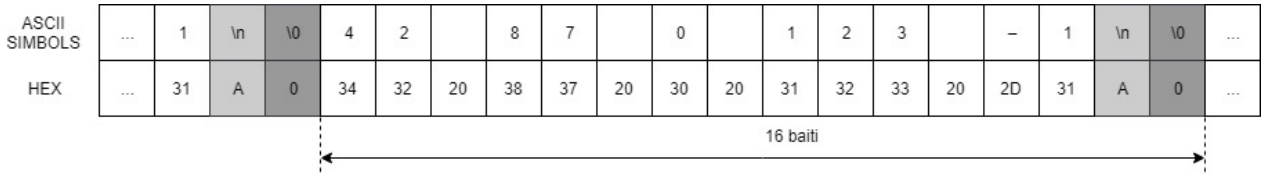
Autora bakalaura darbā skārienus patiešām ielasīja tiešā veidā, bināri nolasot straumējošos datus no */dev/input/eventX* speciālajiem failiem. Šajā projektā autors bija izvēlējies izmantot gatavu bibliotēku *tslib*, kuru plaši iesaka skārienu signālu ielasīšanai un apstrādei [15]. Tā automātiski spēj nolasīt skāriena notikuma bināros datus C struktūru objektā, kuru autors var tālāk izmantot, lai piekļūtu skāriena signāla vienības dažādiem vērtību laukiem. Bez šādas bibliotēkas autoram būtu pašam jāizveido sava struktūra un ielasīšanas algoritms, kas to struktūru aizpilda. Šajā situācijā autors vienkārši pārveido *tslib* struktūru uz savu loģisko struktūru, ar kuru tālāk veic papildus apstrādi.

```
// Calculate VM coord from given coord, its max val and VM boundaries
uint32_t calc_vmcoord(uint32_t coord, uint32_t coord_max, uint32_t vmc1, uint32_t vmc2)
{
	// Force coord to be within coord_max bounds
	if (coord > coord_max) { coord = coord_max; }

	// Scale coord from one coord system to another
	uint32_t vm_coord_diff = vmc2 - vmc1;
	float mod = 1.0 * vm_coord_diff / coord_max;
	float res_scaled = coord * mod;
	uint32_t res = vmc1 + res_scaled; // vm-coord-1 offset + scaled result coord
	return res;
}

// Convert touch coords to chosen rect VM coords
tsig.x = ts_x; // tslib extracted X
tsig.y = ts_y; // tslib extracted Y
tsig.x = calc_vmcoord(tsig.x, scrn_max_x, vm_x1, vm_x2);
tsig.y = calc_vmcoord(tsig.y, scrn_max_y, vm_y1, vm_y2);
```

3.2.2. att. **Skāriena signālu koordināšu apstrāde**

Galvenā apstrāde tīrajam skāriena signālam no *tslib* struktūras ir to koordinātu pārveidošana. Raspberry Pi ierīces skārienjūtīgā ekrāna koordināšu maksimumi (*coord_max*) ir pavisam citādāki nekā VirtualBox virtuālās operētājsistēmas izšķirtspēja (*vmc1* un *vmc2*). Autors arī ir paredzējis, ka Raspberry Pi ierīce var patvaļīgi kontrolēt konkrētu gabalu no virtuālās sienas, kas nozīmē, ka lokālās koordinātas no fiziskā ekrāna ir jāpārveido virtuālo mašīnu koordinātās, kā arī jāpieskaita virtuālo koordinātu minimumi (*vmc1*) un maksimumi (*vmc2*). Šim nolūkam autors ir izveidojis vienkāršu atsevišķu koordināšu pārveides funkciju. Šī funkcija saņem tīro *tslib* koordinātes vērtību *coord*, šīs koordinātes maksimāli iespējamo vērtību *coord_max*, virtuālās mašīnas koordināšu sistēmas sākumu *vmc1* un beigas *vmc2*. Tā pēc tam atgriež pielāgoto koordinātes vērtības, kas darbosies uz virtuālās mašīnas padotās koordināšu plaknes sistēmas (skat. 3.2.2. att.). Līdzīga koordināšu apstrāde arī ir aprakstīta autora bakalaura un Ķirsones darbos [1][7].

Pēc signāla apstrādes Raspberry Pi pusē, tas tiek serializēts un nosūtīts pa tīklu uz galveno serveri. Serializācija ir ļoti vienkāršota un reālajā produkcijas vidē to noteikti būtu jāoptimizē. Lai bināri iekodētu vienu signāla vienības *touch_sig* struktūru un to nosūtītu caur TCP datu straumi, tās vērtības ir pārveidotas uz ASCII formātu un iekodēta vienā simbolu virknē, kura beidzas ar '\n' baitu simbolu (ASCII jaunas līnijas simbols). Ja ir iekodētas vairākas struktūras vienā virknes sūtījumā, tad katra struktūra ir atdalīta ar '\n' simbolu. Jebkurā gadījumā simbolu virknes arī beidzas ar '\0' jeb nulles terminējošo baita simbolu (angl. *null-terminator*). Visas vērtības ir ciparu vērtības un ASCII simbolu virknē tās ir atdalītas ar tukšuma atstarpi (skat. iekodēšanas formātu 3.2.1. att. otrajā līnijā TOUCH_SIG_STRFMT).



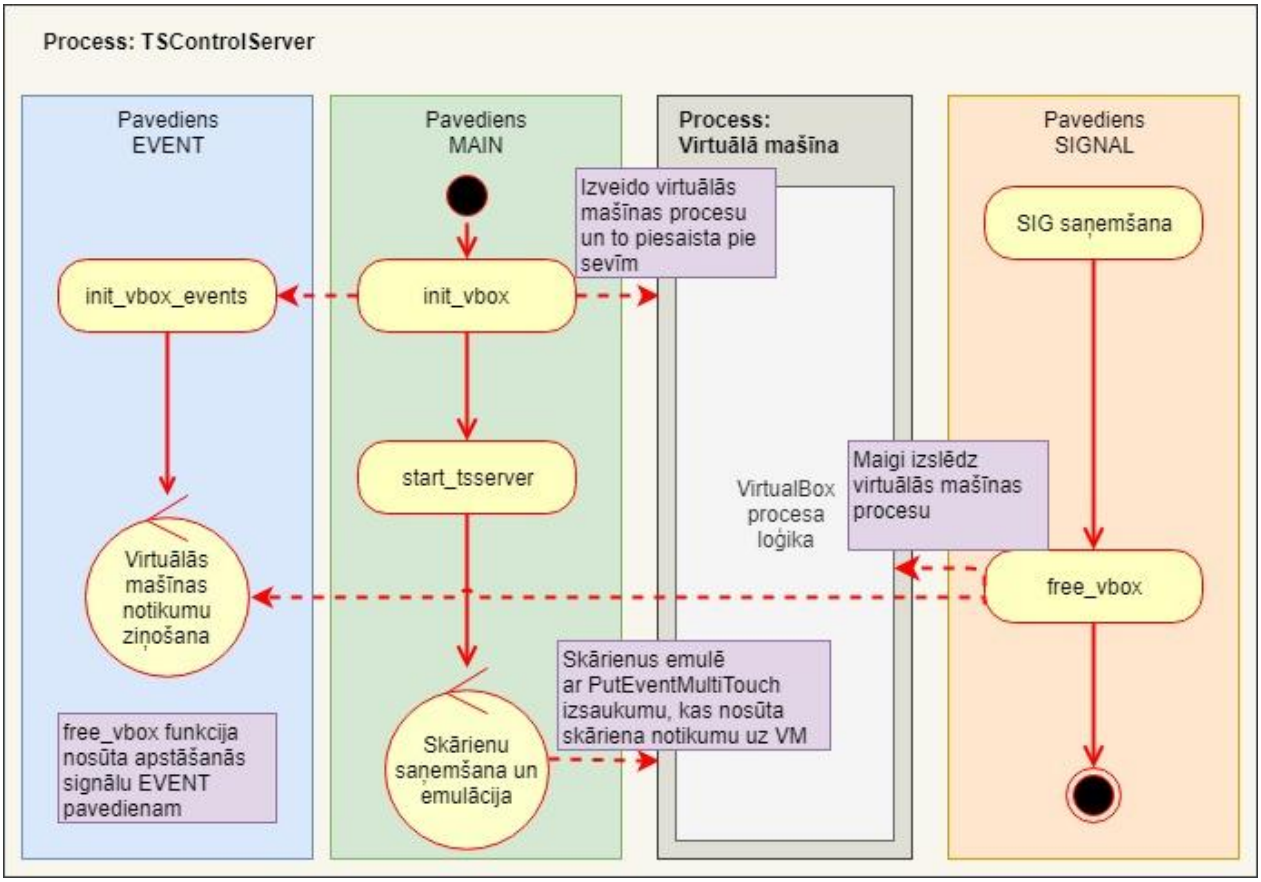
3.2.3. att. **Viena skāriena loģiskās struktūras objekta attēlošana tīmekļa TCP binārajā straumē**

Signāla nolasīšana servera pusē ir veikta, skatoties uz ierakstu atdalīšanas simbolu '\n' un kamēr nav sasniegts nulles terminējošais simbols. Pārsvarā šīs būs vienas struktūras simbolu virkne, kā, piemēram, "`42 87 0 123 -1\n\0`" virkne (skat. 3.2.3. att.). Autors šādu struktūru bija izvēlējies, lai viegli atkļūdotu sliktus signālus, jo tie ir pārsūtīti ASCII formātā un pat var tiešā veidā izvadīt uz ekrāna kā simbolu virknes (ar '\n' simbolu katrs signāla struktūras objekta saturs būs izvadīts uz jaunas līnijas). Produkcijas veida risinājumu būtu labi izvēlēties Ķirsones piedāvāto iekodēšanas struktūru [7], kur ciparu vērtībās ir iekodētas bināri tīmekļa baitu secībā (angl. *network byte order*) [41] un atdalītas ar speciāliem atdalītājsimboliem. Struktūras *touch_sig* pilno kodēšanas pirmkodu var apskatīt 3. pielikumā.

Servera pusē, pēc signāla nolasīšanas, tas ir pārveidots no *touch_sig* struktūras trešā struktūrā, kas ir 64-bitu vesels skaitļa atmiņas apgabals, ko izmanto *PutEventMultiTouch* VirtualBox SDK izsaukumam [16]. Tajā pirmie 32 biti ir atvēlētie X, Y vērtību iekodēšanai, pēc tam 8 biti iekodē skāriena identifikatoru un pēdējie divi biti iekodē skāriena stāvokli. Pārējie 22 biti netiek izmantoti un katram jābūt nulles vērtībai. Detalizētāk par izmantotajiem VirtualBox izsaukumiem ir aprakstīts nākošajā apakšnodaļā.

3.3. VirtualBox saskarnes izmantošana

Programmatūra *TSControlServer* izmanto VirtualBox programmatūras izstrādes komplektu jeb SDK, lai tiešā veidā kontrolētu virtuālo mašīnu dzīvesciklu un veikt manipulācijas ar perifērijas ievadu. Pārējām programmatūrām *TSControlNode* un neizveidotajai *TSDisplayNode* SDK netiek izmantots. Serveris tiešā veidā izmanto C COM GLUE bibliotēku, lai inicializētu VirtualBox objektus un ar tiem palaistu virtuālās mašīnas procesus. Tas arī pielieto vairākus pavedienus un ir nepieciešams veikt dažādas asinhronas darbības, kur dažas no tām pat aiziet līdz atsevišķām virtuālās mašīnas procesam.



3.3.1. att. **Procesa TSControlServer darbības vispārinājums**

Servera dzīvescikls pavisam izveido divus pavedienus un papildus procesu, kā arī var tikt nobeigts (servera process), saņemot dažus no Linux POSIX definētajiem signāliem. Tas uzsāk darbību ar VirtualBox objektu inicializāciju funkcijā *init_vbox*, kur vispirms mēģina izveidot *IVirtualBoxClient* objektu. Pēc šī objekta izveides, tas tālāk tiek izmantots, lai iegūtu nepieciešamos *IVirtualBox* un *ISession* objektus. Ar abiem šiem objektiem ir izvadīts vispārīgs

teksts par izmantoto VirtualBox versiju un lietotāja iestatījumiem. Ar *IVirtualBox* un *ISession* objektiem, funkcija *init_vbox* ieiet dziļāk palīgfunkcijās, kur tiek izvadīta informācija par pieejamām virtuālajām mašīnām, kuras serveris var palaist. Programmatūra šajā brīdī apstājās un gaida uz lietotāja ievadu, kas pasaka, kuru virtuālo mašīnu palaist un lietot. Ar virtuālās mašīnas palaišanu ir iegūtas unikālas tiesības serverim manipulēt šo virtuālo mašīnu ar koplietotu slēdzeni (angl. *shared lock*).

Tālāk serveris inicializē VirtualBox virtuālo mašīnu notikumu reģistrēšanu (izsaukums *init_vbox_events*). Tas tiek inicializēts kā bezgalīgi ejošs, atsevišķs pavediens *EVENT*. Pavedienu izveidei izmanto POSIX [42] *pthread* bibliotēku (izsaukums *pthread_create*) [43], kuru var pievienot ar kompilatora "*-pthread*" argumentu. Galvenais programmas pavediens un šo notikumu pavediens sinhroni koplieto skaitļa mainīgo, kas vienkārši pasaka, kad programmatūrai ir jābeidzas. Piemēram, saņemot SIGINT signālu, kas piespiež beigt programmas darbību, tiek nomainītā šī mainīgā vērtībā. Tas attiecīgi bezgalīgajā notikumu reģistrēšanas pavediena ciklā liks tam apstāties un "maigi" iznīcināt šo *pthread* pavediena resursus, kas arī iekļauj VirtualBox objektu atbrīvošanu. Pirmkods notikumu reģistrēšanai pārsvarā dublējas ar VirtualBox C COM GLUE piemēra programmatūru, kuras pirmkods ir pieejams SDK mapē.

Pašu notikumu reģistrēšanas tiek izmantota, lai regulāri izvadītu uz ekrāna informāciju par virtuālās mašīnas darbību. Notikumu tipi ir daudz un dažādi, kurus var izvēlēties klausīties no virtuālās mašīnas [44]. Autors stingri neiesaka iestatīt, ka jāklausās pilnīgi visi notikumi, jo tad radīsies lielas grūtības izsekot, kas notiek uz servera ekrāna. Papildus tīrai informācijai, viens no notikumiem, kas liecina par virtuālās mašīnas apstāšanos, automātiski nosūta POSIX SIG signālu pats uz sevīm, lai apstādinātu visu programmatūru. Nav vērts turpināt servera darbību, ja virtuālās mašīnas process tika apstādināts no ārējas ietekmes (piemēram, lietotājs manuāli izslēdza palaisto virtuālās mašīnas procesu).

Pēc notikumu reģistrēšanas pavediena inicializācijas, galvenais pavediens inicializē pēdējo VirtualBox komponenti – *IMouse* objektu. Šis objekts nāk no *IConsole* objektu, kas attiecīgi nāk no *ISession*, kuru ieguva *init_vbox* funkcijas sākumā. Objekts no klases *IMouse* tiek izmantots, lai piekļūt galējai *PutEventMultiTouch* skārienu emulācijas metodei. Tā ir attiecīgi pielietota skārienu signālu emulācijai uz esošās virtuālās mašīnas operētājsistēmas. Ar šo pašu objektu arī ir iespējams izsaukt metodes *AbsoluteSupported*, *RelativeSupported* un *MultiTouchSupported*. Šīs papildus metodes pasaka, vai viesa operētājsistēma atbalsta peles absolūto vai relatīvu koordinātu klikšķus

un vai atbalsta vairāku (vai pat jebkādu) skārienu emulāciju. Šis atbalsts var izmainīties dažādos virtuālās mašīnas vai pat tās operētājsistēmas stāvokļos. Piemēram, tikko uzsākot virtuālās mašīnas darbību, visas šīs metodes atgriezīs negatīvu rezultātu (t.i. neatbalsta nevienu peles/skāriena emulācijas metodi). Šī atbalsta izmaiņas arī var noklausīties ar VirtualBox *IEvent* notikumiem, kas arī tiek darīts programmatūrā.

Autors bija pamanījis, ka dažas IMouse metodes kā, piemēram, *PutEventMultiTouchString* nav vispār implementētas un nestrādā. Metode *PutEventMultiTouchString* būtu mazliet atvieglojusi skārienu datu pārkodēšanu, kur šajā variantā to var iekodēt kā simbolu virkni ar atdalītājsimboliem, kamēr parastā *PutEventMultiTouch* metode liek uzmanīgi iekodēt datus 64-bitu skaitļa mainīgajā.

Ar *IMouse* objektu inicializāciju, *init_vbox* funkcija ir pabeigta un pēc tam servera programmatūra aiziet uz *start_tssserver* izsaukumu, kur startē bezgalīgo skārienu datu nolasīšanas un emulācijas ciklu. Katrā cikla iterācijā tiek akceptēti jauni tīmekļa TCP savienojumi vai arī nolasīti dati no esošajiem savienojumiem. Dati ir attiecīgi pārveidoti uz *touch_sig* loģisko struktūru un padoti tālāk skāriena emulācijas metodei, kas *touch_sig* struktūru tālāk iekodē vajadzīgajā 64-bitu skaitļa mainīgajā. Kā aprakstīts iepriekšējā apakšnodaļā, šis mainīgais tālāk ir padots *PutEventMultiTouch* metodei, kas veic COM/XPCOM komunikāciju un liek atdarināt skārienu uz esošā virtuālās mašīnas procesa viesa operētājsistēmas.

Jebkurā brīdi programmatūra var saņemt POSIX SIG signālu (vai nu no procesa iekšienes vai ārienes), kas automātiski visu sāk apstādināt. Sākotnēji ir apstādināts atsevišķais pavediens, kas klausās notikumus. Pēc tam mēģina apstādināt un iznīcināt virtuālās mašīnas procesu. Beigās slēdzenes uz šo konkrēto virtuālo mašīnu ir atbrīvotas un visi VirtualBox atmiņā turētie objekti ir iznīcināti.

Programmatūrai *TSDisplayNode* būtu jāizveido atsevišķs tīmekļa pārraides pavediens RTP H.264 kadru straumēšanai, kā arī būtu jāizveido sava *IFramebuffer* klases implementācija. VirtualBox šajā gadījumā nepiedāvā gatavus *IFramebuffer* objektus. Virtuālā mašīna paredz saņemt programmatūras izstrādātāja definētu *IFramebuffer* klases objektu ar savu pašu speciālo loģiku, kura izmanto tos pašus metožu nosaukumus, kuri tad tiek periodiski izsaukti no virtuālās mašīnas. Šie ārējie izsaukumi principā pasaka, kad kadrs ir gatavs konkrētā atmiņas apgabalā. Autors šo implementāciju atstāj nākotnes attīstībai.

4. PROGRAMMATŪRAS TESTĒŠANA

Praktiskā projekta testēšanai ir izmantotas dažādas fiziskās ierīces, lai simulētu reālu monitora sienu. Ir jābūt datoram ar pietiekamiem resursiem, lai atbalstītu VirtualBox un to viesu operētājsistēmām, kā arī ir jābūt lētām un mazām ierīcēm, kuras var pieslēgt pie monitoriem un pārraidīt datus uz serveri. Pa vidu visām ierīcēm arī jābūt komunikācijas veidam, kā sazināties viens ar otru, kas ir abstrakti aprakstīts kā komutators.

Šajā nodaļā autors apraksta testēšanu monitora sienas skārienjūtīgās daļas programmatūrai. Visas monitora sienas komponentes, kā, piemēram, *TSDisplayNode* video signālu atspēlētāja programmatūra, nav implementētas. Pārsvarā tiks aprakstīta esoši izstrādātās programmatūras testēšana. Nodaļas pamatsastāvs ir iedalīts divās apakšnodaļās – izmantotā aparatūra un tās izstrādes komunikācija/atkļūdošana.

Pirmā apakšnodaļa par izmantoto aparatūru apraksta fizisko ierīču sastāvu un to konfigurāciju. Ar konfigurāciju ir saprasts, kā tās ir pieslēgtas tīklā un kā tās ietilpst vispārīgajā arhitektūras "bildē". Ierīču sastāvs arī apraksta ierīču fizisko resursu daudzumu un papildus iebūvētās ierīces ierīcē.

Otrā apakšnodaļa detalizēti izklāsta, kā tika komunicēts ar ierīcēm programmatūras izstrādes gaitā, kā tās tika atkļūdotas. Tas nozīmē, kā tika nokompilēta programmatūra un kā tā tika testēta uz dažādām ierīcēm attālināti. Šī pamatnodaļa arī var noderēt citu projektu testēšanai (t.i. kā pareizi uzstādīt attālinātu testēšanas sistēmu).

Abās nodaļās arī ir aprakstīti sarežģījumi, ar kuriem autors bijis sastapies. Šie sarežģījums pārsvarā iekļauj sīkas, bet nozīmīgas nianses, kuras citi var palaist garām. Pie reizes arī ir neliels atspoguļojums uz bakalaura darba sarežģījumiem un vai līdzīgas situācijas nav sastaptas maģistra darba izstrādē.

4.1. Izmantotā aparatūra

Kā galveno aparatūru autors izmanto Raspberry Pi ierīces un spēcīgus datorus (arī laptopa formas datori), kuri ir spējīgi pavilkt VirtualBox slodzi un ir spējīgās veikt smagas kompilācijas dažādiem projektiem. Monitora iekārtas īpaši netiks pieminēts, izņemot Raspberry Pi skārienjūtīgo LCD ekrānu, jo parastie monitori (gan arī normālie skārienjūtīgie monitori) tika apskatīti autora bakalaura darbā. Servera aparatūra izmanto spēcīgo aparatūru, kurai ir pietiekami daudz operatīvās atmiņas un jaudīgs centrālais procesors.

4.1.1. att. **Raspberry Pi 3 B ierīce saslēgta ar LCD skārienjūtīgu ekrānu un tā korpusu (barības bloka vads un MicroSD karte nav iekļauta)**

Autors Raspberry Pi 3 B ierīces un tā piederumus ir iegādājies no vietējā Baltijas elektronisko preču piegādes centra *LEMONA*. Tā piedāvā pilnu Raspberry Pi ierīču komplektu, kā arī citu līdzīgu elektroniku [45]. Pilnais produktu komplekts, ko autors iegādājās saistībā ar Raspberry Pi ierīci, kura ir saslēgta ar skārienjūtīgu LCD ekrānu (skat. 4.1.1. att.), ir:

- Minidators RASPBERRY PI 3 B+ (preces kods: RASPBERRY-PI-3-B+, cena: 42 EUR)
- Raspberry Pi barības bloka vads, Micro USB (preces kods: T5875DV, cena: 13 EUR)
- MicroSD karte, 16GB, satur Raspberry Pi NOOBs programmatūru (preces kods: TSRASPI10-16G, cena: 19 EUR)
- Septiņu collu LCD skārienjūtīgs ekrāns Raspberry Pi 3 B ierīcei (preces kods: RASPBERRYPI-DISPLAY, cena: 94 EUR)
- Korpuss septiņu collu LCD skārienjūtīgajām ekrānam un Raspberry Pi 3 B ierīcei (preces kods: MMP-0162, cena: 19 EUR)

Kopējas izmaksas šādi nokomplektētai ierīcei ir, apmēram, 187 EUR. Cenas ir noapaļotas un ņemtas no 2019. gada maija mēneša preču vērtības. Ir svarīgi saprast, ka cenas var tāpat mazliet mainīties. Nopietnām monitoru sienām arī ir ļoti iespējams nosist cenu nost, ja pērk lielos daudzumus un maksā juridiska persona (t.i. produktus var norakstīt, kā biznesa izdevumus).

Autors arī bija nopircis papildus Raspberry Pi ierīci ar dzidru plastmasas korpusu, kuru pieslēdz pie parastā monitora. Sākotnējā ideja bija izveidot arī *TSDisplayNode* programmatūru, kura vienkārši atspēlē video signālu no virtuālās sienas, bet diemžēl autoram nepietika laika. Plastmasas korpuss no *LEMONA* maksā, apmēram, 10 EUR (preces kods: RB-CLEAR). Saliekot Raspberry Pi 3B+, MicroSD karti, barības bloka vadu un plastmasas korpusu, cena kopumā kļūst 84 EUR (apmēram, simts EUR lētāk nekā LCD skārienjūtīgā ierīce).

Raspberry Pi 3 B+ ierīcei ir 64-bitu četru-kodolu 1.4 GHz Broadcom procesors, Broadcom Videocore-IV grafikas procesors, 1GB LPDDR2 SDRAM operatīvā atmiņa, 2.4 GHz un 5 GHz iebūvētā 802.11b/g/n/ac Wi-Fi ierīce. Autora bakalaura darbā tika izmantota Raspberry Pi 2 B ierīce, kurai nebija iebūvētā bezvada tīmekļa ierīce, kur vajadzēs speciāli pirkt, piemēram, TP-LINK Wi-Fi adapteri. To arī var savienot ar vienu parasto tīmekļa vadu (Ethernet), četriem USB 2.0 vadiem, HDMI, *Camera Serial Interface* (CSI) un *Display Serial Interface* (DSI) [46].

Raspberry Pi ierīču un to piederumu komplektā nav iekļautas perifērijas. Tās, protams, nenāk līdzi ar Raspberry Pi iegādi un jāpērk atsevišķi. Kā minimums ir ieteicams iegādāties papildus USB

2.0 savienojuma klaviatūru un peli, jo citādāk nebūs iespējams veikt pirmos uzstādījumus uz ierīces. Autors šim nolūkam bija iegādājis mazu Logitech USB 2.0 klaviatūru un mazu bezvada Logitech peli. Pēc sākotnējās konfigurācijas, autors pieslēdza Raspberry Pi ierīci pie iekšējā, lokālā Wi-Fi bezvada tīkla un iegaumēja ierīces lokālo IPv4 adresi. Tālāk tai varēja pieslēgties caur SSH no galvenā izstrādes datora.

Galvenais izstrādes dators satur Intel i7-8700K sešu-kodolu procesoru ar 5 GHz frekvenci (overlock), 32GB DDR4 operatīvo atmiņu un GTX 1080 Ti grafisko karti. Servera dators, savukārt, ir laptops arī ar i7-8700K procesoru, bet tikai 16GB DDR4 atmiņu un GTX 1060 grafisko karti. Autors bija izvēlējies servera programmatūru uzstādīt uz laptopa-veida datoru, lai to viegli varētu pārvietot un veikt klātienes demonstrāciju.

Viena no sarežģītībām, ko autors pamanīja ir LCD ekrāna ierīces pārvietošanas grūtības. Raspberry Pi ierīce pēc noklusējuma saņem strāvu no sava barības bloka vada, kas ir kaut kur tuvumā iesprausts pie kontakta. Lai tā taptu par ērti lietojamu, mazu kontroles paneli monitora sienai, tad tai būtu jāpieslēdz baterija, kas atļautu to viegli pārvietot pa istabu vai pa ēku, kurā atrodas monitoru siena.

Vēl viena sarežģītība var notikt, ja monitoram nav HDMI porta (ļoti rets gadījums). Raspberry Pi ierīcēm parasti ir tikai viens HDMI ports un cita veida porti (piemēram, DisplayPort) nav. Risinājums būtu iegādāties pāreju uz HDMI no monitora iespējamiem portiem.

Tā kā autors skāriena signālus testēja ar Raspberry Pi skārienjūtīgu LCD ekrānu, tad nav vēl zināms, vai jaunāki Raspberry Pi modeļi ir salabojuši problēmu ar USB 3.0 kopnes centrmezgla komunikāciju [20]. Autora bakalaura darbs bija ar šo problēmu sastapies ar Raspberry Pi 2 B modeļiem, kuriem bija nepieciešams piespiest USB 1.1 darbības režīmu, lai komunicētu ar skārienjūtīgu monitoru, kurš komunicēja caur savējo USB 3.0 centrmezglu. Šī darbības režīma piespiešana attiecīgi neļāva darboties ar citām USB 2.0 perifērijām. Šo testu vajadzētu atkārtot uz LU DF skārienjūtīgajiem monitoriem, lai pārliecinātos, ka šādas problēmas ir salabotas ar jaunākiem Raspberry Pi modeļiem.

4.2. Aparatūras izstrādes komunikācijas un atkļūdošana

Tā kā monitora siena sevī ietver vairākas un dažādas ierīces, uz kurām ir palaista dažāda veida programmatūra, tad to testēšana izvēršas ļoti sarežģīta. Viens no tipiskajiem veidiem būtu nokompilēt programmatūru no viena centrāla izstrādes datora, kur pēc tam to var nokopēt uz monitora sienas ierīcēm. Tā kā visas ierīces atbalsta SSH piekļuvi, tad var vienkārši attālināti pieslēgties pie ierīces un manuāli palaist nokopēto programmatūru. Diemžēl šāds veids aizvien iekļauj sevī daudzus manuālus un laikietilpīgus soļus. Centralizēta izstrādes datora pieeja ir laba, bet daudz labāk būtu, ka nokompilētā programmatūra ir attālināti uzstādīta automātiski bez lietotāja iejaukšanās.

Autors bija eksperimentējis ar Visual Studio 2019 *Preview* veida programmatūru [47], kurā piedāvā atvērt jebkādus CMake projekta mapes un ar tiem strādāt. Šī veida Visual Studio projekts arī neģenerētu savus tipiskos XML veida projekta iestatījumu failus. Tas tikai uzģenerē failu *CMakeSettings.json*, kurā var iestatīt CMake kompilācijas mapes atrašanās vietu (t.i. kurā mapē kompilēt kodu un izveidot programmatūru). Tas arī atļāva iestatīt attālināto ierīču piekļuvi caur SSH protokolu, kur var automātiski pārkopēt un izvietot projekta failus un attālināti nokompilēt risinājumu. Ne tikai tas attālināti izvieto projektu failus, bet arī nokopē attālinātās sistēmas operētājsistēmas C galvenes failus (skat. 4. pielikumu).

Šīs attālināto ierīču funkcijas atļauj veikt pilnu programmatūras izstrāde uz sava vietējā izstrādes datora un pēc tam notestēt programmatūru uz attālinātās ierīces. Tas strādā pat ja izstrādes datorā iet Windows operētājsistēma, kamēr attālinātā ierīce ir Raspbian operētājsistēma. Protams, šīs funkcionalitātes attiecas tikai uz C/C++ programmēšanas valodu. CMake iekšēji ir spējīgs uzģenerēt Visual Studio *Solution* tipa projektus, gan arī UNIX *Makefile* kompilācijas nosacījumu failus.

Ierīču komunikācija autoram tika veikta caur savu komutatoru, kas atbalstīja Wi-Fi piekļuvi. Katrai ierīcei bija kaut kāda IPv4 adrese no 192.168.1.2 līdz 192.168.1.254 (pirmo aizņēma komutators un 255 ir *broadcast* adrese). Savienojuma ātrumam jābūt pietiekami labam, lai ātri pārraidītu vairākus simts vai pat tūkstošus failu, jo Visual Studio 2019 CMake projekti mēģina sinhronizēt attālināto ierīču galvenes failus. Šie faili ir iekopēti lokālajā kešatmiņas mapē, parasti lietotāja mapē. Ar šo informāciju Visual Studio var izmantot pilnu IntelliSense komplektu, kas

automātiski iedod autoram iespēju apskatīt attālinātās operētājsistēmas funkciju izsaukumus ar dokumentāciju un koda ieteikumiem.

Diemžēl lielā Visual Studio programmatūra strādā tikai uz Windows operētājsistēmas. Lai strādātu līdzīgā veidā uz Linux vai citām operētājsistēmām, autors iesaka izmantot Visual Studio Code programmatūru [48], kura ir izstrādāta vairākām platformām (Windows, Linux, macOS). Ar Visual Studio Code ir iespējams iegūt paplašinājumus, kuri nodrošina failu sinhronizāciju starp attālinātām ierīcēm. Tas nebūs tik ideāli, kā pilns IntelliSense komplekts no parastā Visual Studio, bet vismaz var nodrošināt programmatūras kopēšanas automatizāciju.

Projekta ietvaros ir izveidotas divas programmatūras – TSControlNode un TSControlServer. Abas ir nokompilētas ar CMake nosacīto kompilāciju caur CMakeLists.txt failiem (skat. 5. pielikumu). Programmatūru TSControlNode ir paredzēts likt uz Raspberry Pi ierīcēm, kurām ir piekļuva skārienjūtīgam ekrānam un var to signālus ielasīt no */dev/input/eventX* failiem. Programmatūru TSControlServer, savukārt, ir jāliek uz pietiekami spēcīga Linux servera, kurā ir uzinstalēta un pareizi nokonfigurēta VirtualBox 5.2.26 versija.

Autors arī bija pārsteigts, ka Visual Studio CMake projektos var attālināti atkļūdot. Kad CMake projekta kompilācija ir pabeigta, ir iespējams ieslēgt attālinātu GDB atkļūdošanas serveri. Visual Studio pie šī servera pieslēdzās un veic atkļūdošanu programmatūrai, kur GDB soļi un ekrāna izvads ir pārraidīts atpakaļ uz izstrādes datoru.

Viens no sarežģījumiem radās, kad vajadzēja nokompilēt tikai vienu no šiem projektiem un katram bija atsevišķā ierīce. Pēc noklusējuma CMake projekti uz Visual Studio mēģina nokompilēt visus atrastos apakšprojektus (tātad visas programmatūras), tāpēc autoram vajadzēja izveidot divas dažādus CMake konfigurācijas režīmus *RPi-Debug-TSControlNode* un *RPi-Debug-TSControlServer*. Pārslēdzoties no šiem režīmiem, bija iespējams izmainīt ierīču IPv4 adresi un attiecīgo vietu uz kurienes pareizā programmatūra ir nokompilēta.

5. ATTĪSTĪBA NĀKOTNĒ

Darba izstrādes gaitā autoram radās vairākas idejas, kuras uzlabotu esošo programmatūru. Tā kā praktiskā risinājuma mērķis ir tikai parādīt, kā implementēt skārienjūtīgo monitoru signālu apstrādi monitoru sienas sistēmā, autors citām funkcionalitātēm sākotnēji nebija veltījis daudz laika. Papildus funkcijas, kā, piemēram, video signālu apstrādi, padarītu risinājumu daudz pilnīgāku gatavai monitora sistēmai. Protams, pilna sastāva monitoru sistēmu nav reāli izveidot vienam cilvēkam dažu mēnešu laikā, tāpēc autors ir atveltījis veselu nodaļu ar potenciāliem nākotnes attīstības soļiem.

Pats maģistra darbs arī var tik izmantots kā piemērs, kā izveidot programmatūru ar VirtualBox saskarni. Autors ir detalizēti aprakstījis VirtualBox programmatūras izstrādes komplektu un tā saistītos objektus vai funkciju izsaukumus. No šīs informācijas ir iespējams veikt nelielus paplašinājumus monitora sienas sistēmām, kuras ir bāzētas uz VirtualBox. Viens no paplašinājuma piemēriem būtu attālināti pieslēgt klaviatūru, kur tās taustiņu būtu emulēti uz visa operētājsistēmas. Vēl viens piemērs būtu izveidot tīmekļa vietnes lapu, kurā var kontrolēt virtuālo mašīnu stāvokļus, iegūt diagnostikas informāciju un statusu par viesa operētājsistēmām. Visiem šādiem paplašinājumiem ir vajadzīgs izmantot galvenos SDK objektus (*IVirtualBox* un *ISession*).

Šajā nodaļā autors apraksta dažus būtiskus nākotnes attīstības soļus savai izstrādātajai programmatūrai. Sākotnēji ir sakopoti apraksti par dažādiem maziem uzlabojumiem jau esošajai programmatūrai, kura ir izstrādāta. Nākošā apakšnodaļa apraksta programmatūras TSDisplayNode būtību un kā tā būtu jāizveido, lai iegūtu pareizu video signālu apstrādi. Beigās ir neliels ieskaits Infiniviz atjaunināšanai.

Daži no nākotnes attīstības soļiem var kalpot kā ievads citu studentu darbiem, kuri mēģina kaut ko līdzīgu izstrādāt. Autors cer, ka viņa ieguldījums par pareizu skārienjūtīgo apstrādi ir tālāk nākotnē izmantots citiem projektiem vai pētījumiem.

5.1. Esošā projekta uzlabojumi

Programmatūrām TSControlNode un TSControlServer var veikt vairāku veidu uzlabojumus. Vienkāršāko veidu uzlabojumi būtu optimizācijas soļi vietās, kur autors ir izvēlējies izstrādāt vienkāršotus algoritmus vai metodes, lai tās ir vieglāk atkļūdot. Vēl viena veida uzlabojumi ir paplašināt skārienu apstrādi un pielikt papildus grafisko funkcionalitāti TSControlNode vai TSControlServer programmatūrai.

Optimizāciju abām programmatūrām var veikt ar skāriena loģiskās struktūras iekodēšanu. Trešajā nodaļā par monitora sienas sistēmas implementāciju ir apakšnodaļa par skārienu signālu apstrādi, kur autors bija izvēlējies signālus iekodēt ar ASCII formatējumu. Tā kā dati ir sūtīti cauri TCP datu straumēšanas protokolam, tad ir arī jāievieš speciāli kontroles simboli, lai atšķirtu dažādas struktūru objektu vienības straumē. Autors to bija izdarījis ar papildus ASCII speciālajiem simboliem. Neliela optimizācija ir pārveidot datu kodēšanu ar Ķirsone piedāvāto risinājumu. Ķirsones risinājums ir vienkārši iekodēt struktūras binārā veidā ar konkrētiem baitu simboliem, kas atdala dažādus ierakstus. Šī optimizācija mazliet ietaupītu patērēto datu apjomu un padarītu dekodēšanu mazliet ātrāku.

Vēl viena optimizācija ir nomainīt programmatūras *TSControlServer* TCP servera *select* veida algoritmu pret jaunāku *epoll* izsaukumu, kas atļautu efektīvāk apstrādāt vairāk tīmekļa savienojumus. Galvenās atšķirības starp abiem izsaukumiem ir kā tiek pārbaudīts, vai savienojumam ir pieejami dati. Izsaukums *select* to dara lēnāk ar lielākiem savienojumu skaitiem, jo tas iet katram apkārt un manuāli tos pārbauda. Savukārt, *epoll* izveido savu speciālo *pastkasti,* kur katrs savienojums vienkārši informē programmatūru caur *epoll* pastkasti (t.i. *epoll* nav manuāli jāstaigā cauri N savienojumiem) [49].

Programmatūru *TSControlNode* noteikti būtu jāpaplašina ar vismaz minimālu grafisko saskarni, kas aizstāj Raspberry Pi darbvirsmas attēlu. Pašlaik autors ir vienkārši padarījis ekrānu melnu, lai ekrāna skārieni neaiztiktu nekādu darbvirsmas elementus no Raspberry Pi ierīces. Reālam produkcijas risinājumam vajadzētu izveidot pāris grafiskās saskarnes pogas, ar kurām var beigt programmatūras darbību vai pārslēgties uz citu monitoru.

Programmatūru *TSControlServer* var papildus uzlabot ar skārienu pārveidošanu uz peles taustiņu signāliem. Tas būtu ļoti noderīgi viesa operētājsistēmām, kuras noklusēti neatbalsta vairāku skārienu funkcionalitāti. Daudz no operētājsistēmas vispār neatbalsta nekāda veida

skāriena signālus, kuras attiecīgi nevar izmantot projekta demonstrācijas mērķiem. Viegls apkārtceļš šādai sarežģītībai būtu sūtīt peles taustiņu signālus uz viesa operētājsistēmas, kuri nāk no pārveidotiem skārienu signāliem.

Paši skārieni arī var labi nedarboties uz viesa operētājsistēmas, kura uztur lielu virtuālo sienu. Piemēram, var saņemt desmit vai pat vairākus pirkstu skārienus vienā reizē. Tas tīrajā veidā var izpausties ar kursora lēkāšanu pa ekrānu un iespējams neļaus bīdīt divus logus vienlaicīgi. Šādas problēmas var vienīgi risināt ar papildus informāciju par viesa operētājsistēmu. Autora bakalaura darbs bija apskatījies skārienu apstrādi tiešā veidā uz Windows 8.1 operētājsistēmas, tātad ir iespējams iegūt vismaz kaut kādu pamata bāzi, uz kā strādāt. Visticamāk šāda veida skārienu papildus apstrāde nāks kā viesa operētājsistēmas draiveris, kas sinhronizēs vienlaicīgu skārienu piekļuvi, ja nu vienīgi operētājsistēmas to neatbalsta (Windows 10 to var atbalstīt, bet pārējām operētājsistēmām nav liels atbalsts).

Tā kā VirtualBox C COM GLUE bibliotēka apvieno gan Microsoft COM, gan Mozilla XPCOM, tad nevajadzētu būt grūtībām izveidot servera programmatūru Windows Server 20XX operētājsistēmām. Lielākais darbs šajā gadījumā būtu patērēts paplašināt TCP servera komponenti, lai strādātu uz vairākām platformām. Pašlaik programmatūra *TSControlServer* izmanto Linux tīmekļa programmatūras izstrādes galvenes failus, kuri attiecīgi stipri atšķiras no *windows.h* piedāvātājām struktūrām. Attīstot šo uzlabojumu būtu iespējams palaist servera programmatūru gan uz Linux serveriem, gan Windows serveriem.

Būtisks fizisks uzlabojums Raspberry Pi ierīcēm ar mazu pieslēgtu skārienjūtīgu LCD ekrānu būtu baterijas pieslēgšana, lai to varētu pārvietot bez barības bloka vada. Tas atļautu izveidot speciāla veida kontroles paneļus, kur lietotājs varētu patvaļīgi pieslēgties pie vietējā parastā monitora caur monitora sienas serveri un izmainīt tā saturu. Protams, šāda doma arī ietver pietiekami labus grafiskās saskarnes uzlabojumus, lai pārslēgtos uz dažādiem monitoriem.

5.2. Programmatūras TSDisplayNode implementācija

Programmatūra *TSDisplayNode* bija ieplānota, bet beigās netika izstrādāta maģistra darba ietvaros. Autoram nebija pieticis laika, lai iekļautu šo komponenti izstrādes posmā, kaut arī tā bija ļoti nozīmīga. Komponentes galvenā būtība ir video signālu apstrāde, kas atdarina Infiniviz Raspberry Pi klienta programmatūras funkcionalitāti.

Ar video apstrādi šajā kontekstā ir saprasts, ka ienāk H.264 iekodēta video kadru straume, kuru dekodē un parāda uz visa ekrāna. Šāda video signālu atspēlēšana būtu nozīmīga lietotāja prasība, ja vēlētos izmantot kaut kāda veida kontroles paneļus, kur var kontrolēt citu monitoru saturu. Tas arī būtu nepieciešams parastajām monitoru matricām, lai VirtualBox serveris no virtuālās sienas pareizi pārraida grafisko saturu uz attiecīgajiem monitora paneļiem (nav svarīgi, vai monitora panelis ir skārienjūtīgs, vai nē).

Klienta pusē jeb *TSDisplayNode* programmatūrai būtu jāizmanto gatavas H.264 dekodēšanas un attēlošanas bibliotēkas. Atvērtā pirmkoda bibliotēka *omxplayer* ne tikai atļauj dekodēt kadrus, bet arī tos saņemt no RTP protokola tīmekļa datu straumes [11][12]. OMX Player bibliotēka iekšienē izmanto Linux operētājsistēmas kadru bufera failu [39], kas atļauj rakstīt pikseļus uz visa pieslēgtā ekrāna. Tiklīdz ir saņemti pirmie dekodētie kadriem, bufera fails ir aizpildīts ar kadru saturu un viss ierīces ekrāns attēlo to saturu. Šajā gadījumā saturs būtu iestatīta daļa no virtuālās sienas. Lielu daļu piemēra pirmkodu šādai klienta programmatūrai var ņemt no Infiniviz praktiskā projekta.

Servera programmatūrā ir vissarežģītākā daļa – grafisko kadru iegūšana no VirtualBox virtuālās mašīnas. Pašlaik vienīgais atbalstītais veids tikt klāt grafiskā satura atmiņa apgabaliem ir caur *IDisplay* un *IFramebuffer* VirtualBox SDK objektiem. Klases *IDisplay* objektu ir iespējams iegūt no *IConsole* (kas attiecīgi nāk no *ISession*). Gatavs *IFramebuffer* objekts neeksistē un programmatūras izstrādātājam ir tas manuāli jāizveido ar tām pašām metodēm, kuras apraksta *IFramebuffer* klase. Šo pašizveidoto objektu var tālāk padod *IDisplay* objektam, kas periodiski sagatavos grafiskos kadrus un izsauks *IFramebuffer* objekta metodes, lai liecinātu par kadru gatavību. Kadru izmērus un sienas izmantotos apgabalus var aprakstīt izveidotajā objektā, kurus attiecīgi interpretēs *IDisplay* objekts.

Infiniviz projekts arī ir izmantojis *IFramebuffer* pieeju, bet ir izlaidis C COM GLUE bibliotēku, lai iegūtu ciešāku saskarni ar XPCOM vai COM izsaukumiem. Projekts arī ir izveidojis savu *IFramebuffer* klases implementāciju, kura ir izmantota kadru ielasīšanai no *IDisplay* objekta.

Iegūtie kadri ir pēc tam jāpārkodē H.264 formātā, lai tos pēc tam var straumēt ar RTP prokolu uz klienta programmatūru. Kadru kodēšanai uz H.264 var izmantot vairākas gatavas bibliotēkas, bet ir nozīmīgi saprast, vai bibliotēkām nav nepieciešama papildus speciālās ierīces. Piemēram, NVENC/NVDEC bibliotēkai ir nepieciešama strādājoša NVidia grafikas karte [9][34]. Tāpat arī pastāv fiziskās ierīces, kuras ir domātas tikai kadru kodēšanai, kā, piemēram, Blackmagic Design H.264 Pro Recorder [10]. Infiniviz risinājums kadru kodēšanas veidus bija implementējis kā atsevišķas dinamiskās bibliotēkas, kuras ir piesaistītas pēc servera programmatūras vajadzībām.

Tā kā *IDisplay* izsauc izveidotā *IFramebuffer* objekta metodes asinhroni un RTP straumēšanai būs vajadzīgs atsevišķs TCP vai UDP servera cikls, tad tas izveidos divus papildus pavedienus servera programmatūrai. Klases objektam *IDisplay* var padot arī vairākus *IFramebuffer* objektus, kas nozīmētu, ka pavedienu skaits var pieaugt augstāk par diviem. Visi šie pavedieni programmatūrai ir jāpārvalda, tāpēc ir ieteicams savlaicīgi padomāt par vispārinātu pavedienu kontroli, tik īpaši, kad servera programmatūrai ir jābeidz darbu.

Klienta programatūras pusē var pārsvarā iztikt ar vienu pavedienu, kurš bezgalīgi ieciklējas TCP vai UDP savienojuma datu saņemšanā. Saņemot RTP protokola straumes datus, OMX player tos automātiski dekodē un kadru saturu uzzīmē uz Linux kadru bufera. Šeit var rasties sarežģītības ar TSControlNode programmatūras grafiskās saskarnes paplašinājumiem. Ja ir paredzēts izveidot speciālas pogas ar kurām kontrolēt monitoru pieslēgumus, tad TSDisplayNode programmatūras kadri ātri pārrakstīts pāri tām. To atrisināt var vienīgi kaut kādā veidā ierobežot TSDisplayNode izmantoto Raspberry Pi ierīces ekrāna laukumu, lai atstātu vietu pogām no TSControlNode programmatūras.

Kopsummā šo papildus programmatūru realizēt nebūs viegli, ja ir jāņem prātā citu signālu apstrādi un citas grafiskās saskarnes funkcionalitātes klienta programmatūras pusē. Šāds projekts pats par sevi noteikti sastādītu veselu bakalaura vai pat maģistra darba apjomu.

5.3. Infiniviz atjaunināšana

Autors bija cerējis izmantot lielu daļu pirmkoda no Infiniviz praktiskā projekta, tik īpaši video signālu apstrādes kodu. Klienta video apstrādes kodu patiešām var izmantot arī maģistra darba praktiskajam projektam, bet pats VirtualBox kadru izveides kods nestrādā, kā vajadzēja. Piemēram, tā kā Infiniviz tiešā veidā izmanto COM vai XPCOM izsaukumus, tad rodas grūtības, kad jāatjauno VirtualBox versija. Tas izmaina funkciju un datu struktūru, kas attiecīgi sabojā Infiniviz pirmkodu un ir vajadzīgi papildus labojumi.

Liela daļa laiks tika veltīts, lai nokompilētu Infiniviz risinājumu uz autora servera iekārtas, kurā bija daudz jaunākās grafiskās ierīces (GTX 1060) un jaunāka VirtualBox versija. Bija nepieciešamība salabot pirmkodu vairākās vietās, lai vismaz nokompilētu programmatūru vai kadru kodēšanas dinamiskās bibliotēkas bez kļūdu izmešanas. Pat ar salabotu pirmkodu, autors pamanīja, ka kaut kādā brīdi programmatūras darbība tiek sagrauta, jo tā ir nepareizi izmantojusi operētājsistēmas atmiņas apgabalu, kad ir veiktas NVENC kodēšanas funkcijas. Šo kļūmi autors vairāk nevarēja salabot, jo nebija nekāda pieredzes un dokumentācijas izmainīt dziļas NVidia kadru kodēšanas apakš izsaukumu pirmkodu.

Viss Infiniviz praktiskais projekts arī nebija labi dokumentēts un pirmkods netika daudz komentēts. Tas pagrūtināja tā izpratni un atrast vietas, kuras ir jāpielabo, kad tika veikts VirtualBox atjauninājums. Projekta struktūra ar failu izvietojumu arī nebija labi saprotama. Risinājuma pirmkoda faili bija grupēti zem apakšmapēm vienā līmenī ar apakšmapēm, kuras apraksta programmatūras sākumu. Kompilācijas nosacījumu faili attiecīgi bija citās mapēs un beigās sanāk, ka nav pilnībā skaidrs, kuras pirmkoda apakšmapes ir izmantotas kādai programmatūrai (dažas bija izmantotas vairākām). Autors būtu vēlējies sadalīt pirmkodu pa koka-veida struktūru, kur katrai programmatūrai ir savi unikālie pirmkoda faili un koplietotie faili (starp dažādām programmatūrām) ir ielikt atsevišķā bibliotēkas mapē.

Šis viss liecina par to, ka Infiniviz risinājumu vajadzētu pareizi atjaunināt ar jaunāku VirtualBox versiju, kā arī izveidot labu dokumentāciju un pirmkoda komentārus. Atjaunināšanas gaitā arī var pievienot citu studentu eksperimentus ar monitora sienu sistēmām (piemēram, autora praktiskā darba skārienjūtīgo signālu apstrādi uz VirtualBox).

REZULTĀTI

Autors ir veiksmīgi veicis detalizētu apskati VirtualBox izmantošanai un tā potenciālām skārienjūtīgo monitoru sienas sistēmas izmantošanai. Tika aprakstīti iepriekšējie mēģinājumi, kur ar Infiniviz projektu ir apskatīts VirtualBox parasto monitoru sienas risinājums, Lauras Ķirsones darbs par skārienjūtīgajām sienām un autora paša bakalaura darbs par to signālu apstrādi un atdarināšanu uz dažādām operētājsistēmām. Papildus arī tiek novērtēti šie iepriekšējie mēģinājumi ar jaunu pieejamu informāciju no VirtualBox praktiskajiem eksperimentiem.

Darba ietveros ir aprakstīta VirtualBox uzstādīšana, tās programmatūras izstrādes komplekta izmantošana un pašu programmatūras klašu, objektu pareiza pielietošana. Ar šo informāciju ir iespējams dziļāk izprast VirtualBox programmatūru un kā izveidot cieši saistītus risinājumus ar virtuālām mašīnām. Autors bija veicis vairākus programmatūras izstrādes komplekta testus, lai iegūtu papildus informāciju par sīkām niansēm, kuras ir jāievēro, lai veiksmīgi uzprogrammētu VirtualBox paplašinājumus.

Kā galveno praktisko daļu maģistra darbam, autors ir izveidojis klienta-servera programmatūru, kura pārraida skāriena signālus no skārienjūtīgajiem ekrāniem uz serveri, kur tie tiek simulēti uz virtuālām mašīnām. Skārienu ievākšana no ekrāna ierīcēm ir ļoti vienkāršs princips, kas bija daudzkārt apskatīts Ķirsones un autora bakalaura darbos. Galvenais ieguldījums, kas līdz šim nebija izdarīts, ir simulēt saņemtos skārienus caur VirtualBox programmatūras izstrādes komplektu. Tas attiecīgi simulē skārienus uz jebkādas viesa virtuālās operētājsistēmas.

Ar autora praktisko projektu arī ir demonstrēts, ka Raspberry Pi ierīces ir diezgan pielāgojamas ierīces un uz tām var attīstīt vairākas citas funkcijas monitora sienas izveidei. Autors skārienu pārraides demonstrācijai bija saslēdzis kopā mazu skārienjūtīgu LCD ekrānu ar Raspberry Pi ierīci, kura caur bezvadu internetu varēja kontrolēt viesa darbvirsmas saturu.

Pēc praktiskā darba izstrādes, autors arī ir aprakstījis vairākus nākotnes attīstības soļus, kurus var izvirzīt, kā tiešu turpinājumu šim maģistra darbam vai arī kā noderīgu informācijas avotu. Daži no attīstības soļiem arī dod nelielu ideju, kā detalizēti izpildīt soļus un agros brīdinājumus par iespējamiem sarežģījumiem, ar kuriem var sastapties.

SECINĀJUMI

Infiniviz risinājums ir stipri novecojis, jo tā bibliotēkas balstās uz vecām fiziskām ierīcēm (piemēram, GTX 6. paaudze NVENC kodētājam). Projekta pirmkods arī nav labi komentēts vai dokumentēts, kas rada grūtības to atjaunināt. Infiniviz programmatūra *DisplayWallServer* izlaiž VirtualBox ieteikto C COM GLUE bibliotēku, lai strādātu ar COM/XPCOM tiešā veidā. Tas attiecīgi rada papildus risku, ka risinājums tiks stipri sabojāts ar jaunākām VirtualBox versijām.

Vērtējot Infiniviz H.264 kadru kodēšanas pieeju, autors pamanīja, ka liela daļa NVidia grafikas kartes ir ierobežotas kodēšanas pavedienu izveidei (pārsvarā var strādāt tikai divi vienlaicīgi pavedieni, kuri kodē kadrus ar GTX grafikas kartēm). Aizstājot šo pieeju ar *x264* bibliotēku, kas izmanto galveno datora procesoru, noņemtu šādus ierobežojumus un uzlabotu mērogojamību (var izmantot servera procesoru paaudzes kā Intel Xeon).

Skārienu notikumu ievākšana no ekrāna ierīces var atšķirties starp dažādiem ekrāniem, bet ir vismaz garantēts, ka notikums satur X, Y koordinātas un pirksta nospieduma stāvokli. Liela daļa skārienjūtīgu ekrānu arī atbalsta vairāku-skārienu režīmu ar iedalītiem identifikatoriem (ABS_MT_SLOT, ABS_MT_TRACKING_ID). Diemžēl katram ekrānam var būt savs pikseļu blīvums, kas attiecīgi izmaina maksimālās X, Y tīrās vērtības (tās neatbilst operētājsistēmas izšķirtspējai) un tās vajag zināt pirms programmatūra tiek palaista, kura tās nolasa.

Praktiskā projekta risinājums arī ir izveidojis spēcīgus pamatus, kā pareizi lietot VirtualBox saskarnes objektus un ietekmēt virtuālās mašīnas. No šī projekta ir iespējams tālāk attīstīt citus bakalaura vai pat maģistra darbus ar sarežģītākiem VirtualBox pielietojumiem. Viens no pielietojumiem būtu izveidot video straumēšanu uz skārienjūtīgajiem ekrāniem, lai attēlotos servera viesa operētājsistēmas darbvirsmas grafiskais saturs. Šī video straumēšana jeb programmatūra *TSDisplayNode* nebija izstrādāta, jo autoram nepietika laiks. Tās izstrāde āri ir novērtēta kā ļoti sarežģīta, jo tās video signālu apstrādei ir vienlaicīgi jāstrādā ar citu signālu apstrādi. Klienta ierīču pusē šai programmatūrai arī ir jāmēģina netraucēt *TSControlNode* ekrāna skārienu apstrādes programmatūra, ja tai ir pielikta papildus grafiskā saskarne.

Esošā risinājuma servera programmatūra *TSControlServer* tikai strādā uz Linux operētājsistēmas. Lai tā strādātu uz, piemēram, Windows Server operētājsistēmas, tad ir jāveic nelieli kodu labojumi, kur Linux-specifiski funkciju izsaukumi ir aizstāti ar citas operētājsistēmas izsaukumiem.

IZMANTOTIE AVOTI UN LITERATŪRA

1. T. Lācis, ***SKĀRIENJŪTĪGU AUGSTAS IZŠĶIRTSPĒJAS MONITORU SIENAS IZSTRĀDE***, Latvijas Universitāte, Datorikas fakultāte, 2017.
2. R. Bundulis, G. Arnicans, ***Architectural and Technological Issues in The Field of Multiple Monitor Display Technologies***, Caplinskas, Albertas, et al. (eds.) Frontiers in Artificial Intelligence and Applications. *Databases and Information Systems VII: Selected Papers from the Tenth International Baltic Conference*, DB&IS 2012, Vol. 249, IOS Press, 2013, 317-329
3. R. Bundulis, G. Arnicans, ***Concept of virtual machine based high resolution display wall***. Information, Electronic and Electrical Engineering (AIEEE), 2014 IEEE 2nd Workshop on Advances in, pp. 1-6. IEEE, 2014
4. R. Bundulis, G. Arnicans. ***Use of H. 264 real-time video encoding to reduce display wall system bandwidth consumption***, Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in, pp. 1-6. IEEE, 2015
5. R. Bundulis and G. Arnicans, ***Conclusions from the Evaluation of Virtual Machine Based High Resolution Display Wall System***, International Baltic Conference on Databases and Information Systems. pp. 211-225. Springer International Publishing, 2016
6. R. Bundulis and G. Arnicans. ***Infiniviz – Virtual Machine Based High-Resolution Display Wall System***, Arnicans, Guntis, et al. (eds.) Frontiers in Artificial Intelligence and Applications. *Databases and Information Systems IX*, Vol. 291, IOS Press, 2016, 225-238
7. L. Ķirsone. ***Skārienjūtīgu monitoru izmantošana augstas izšķirtspējas monitoru sienā***, Datorikas Fakultāte, Latvijas Universitāte, 2016
8. VideoLAN / x264 Gitlab [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://code.videolan.org/videolan/x264
9. Video Encode and Decode GPU Support Matrix | Nvidia Developer [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://developer.nvidia.com/video-encode-decode-gpu-support-matrix
10. H.264 Pro Recorder | Blackmagic Design [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://www.blackmagicdesign.com/products/h264prorecorder

11. Github – popcornmix/omxplayer: omxplayer [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://github.com/popcornmix/omxplayer
12. RTP: A Transport Protocol for Real-Time Applications [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://tools.ietf.org/html/rfc3550
13. Linux *input* interfeisa dokumentācija [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://www.kernel.org/doc/Documentation/input/input.txt
14. Ubuntu Manpage: evtest - Input device event monitor and query tool [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: http://manpages.ubuntu.com/manpages/precise/man1/evtest.1.html
15. Github – kergoth/tslib: tslib [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://github.com/kergoth/tslib
16. VirtualBox Main API: IMouse Interface Reference [tiešsaiste] – [atsauce: 2019.04.21]. Pieejams: https://www.virtualbox.org/sdkref/interface_i_mouse.html
17. CMake [tiešsaiste] – [atsauce: 2019.04.29]. Pieejams: https://cmake.org/
18. Boost.Asio – 1.63.0 [tiešsaiste] – [atsauce: 2019.04.29]. Pieejams: http://www.boost.org/doc/libs/1_63_0/doc/html/boost_asio.html
19. Autora *lu_df_ts_project* GitHub repozitorijs [tiešsaiste] – [atsauce: 2019.04.29]. Pieejams: https://github.com/azeroc/lu_df_ts_project
20. USB 3.0 hub fails to pass through USB 2.0 devices (ID 2109:3431) Issue #64 raspberrypi/firmware [tiešsaiste] – [atsauce: 2019.04.29]. Pieejams: https://github.com/raspberrypi/firmware/issues/64
21. InjectTouchInput function (Windows) [tiešsaiste] – [atsauce: 2019.04.29]. Pieejams: https://msdn.microsoft.com/en-us/library/windows/desktop/hh802881(v=vs.85).aspx
22. S.T. King, P.M. Chen. ***SubVirt: implementing malware with virtual machines,*** 2006 IEEE Symposoim on Security and Privacy (S&P'06), pp. 14-327. IEEE Conferences, 2006.
23. VirtualBox 5.2 [tiešsaiste] – [atsauce: 2019.04.29]. Pieejams:

https://www.virtualbox.org/wiki/Download_Old_Builds_5_2
24. OS Boxes VirtualBox Images [tiešsaiste] – [atsauce: 2019.04.30]. Pieejams: https://www.osboxes.org/virtualbox-images/
25. Download a Windows 10 virtual machine [tiešsaiste] – [atsauce: 2019.04.30]. Pieejams: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines
26. Installing the VirtualBox Guest Additions [tiešsaiste] – [atsauce: 2019.04.30]. Pieejams: https://docs.oracle.com/cd/E36500_01/E36502/html/qs-guest-additions.html
27. The Component Object Model | Microsoft Docs [tiešsaiste] – [atsauce: 2019.05.01]. Pieejams: https://docs.microsoft.com/en-us/windows/desktop/com/the-component-object-model
28. XPCOM – Mozilla | MDN [tiešsaiste] – [atsauce: 2019.05.01]. Pieejams: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM
29. VirtualBox Main API: Class list [tiešsaiste] – [atsauce: 2019.05.01]. Pieejams: https://www.virtualbox.org/sdkref/annotated.html
30. VirtualBox Programming Guide and Reference [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://download.virtualbox.org/virtualbox/SDKRef.pdf
31. Autora pirmkods TSControlPanel [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://github.com/azeroc/lu_df_tscontrolpanel
32. CUDA LLVM Compiler [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://developer.nvidia.com/cuda-llvm-compiler
33. About CUDA [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://developer.nvidia.com/about-cuda
34. NVIDIA VIDEO CODEC SDK [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://developer.nvidia.com/nvidia-video-codec-sdk
35. ITU-T, ***ITU-T H.264 (V12) (04/2017)***, ITU-T Study Group 16, ITU-T, 2017.
36. VisualGDB [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://visualgdb.com/
37. Decklink models [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://www.blackmagicdesign.com/products/decklink/models
38. Intel QuickSync Video [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams:

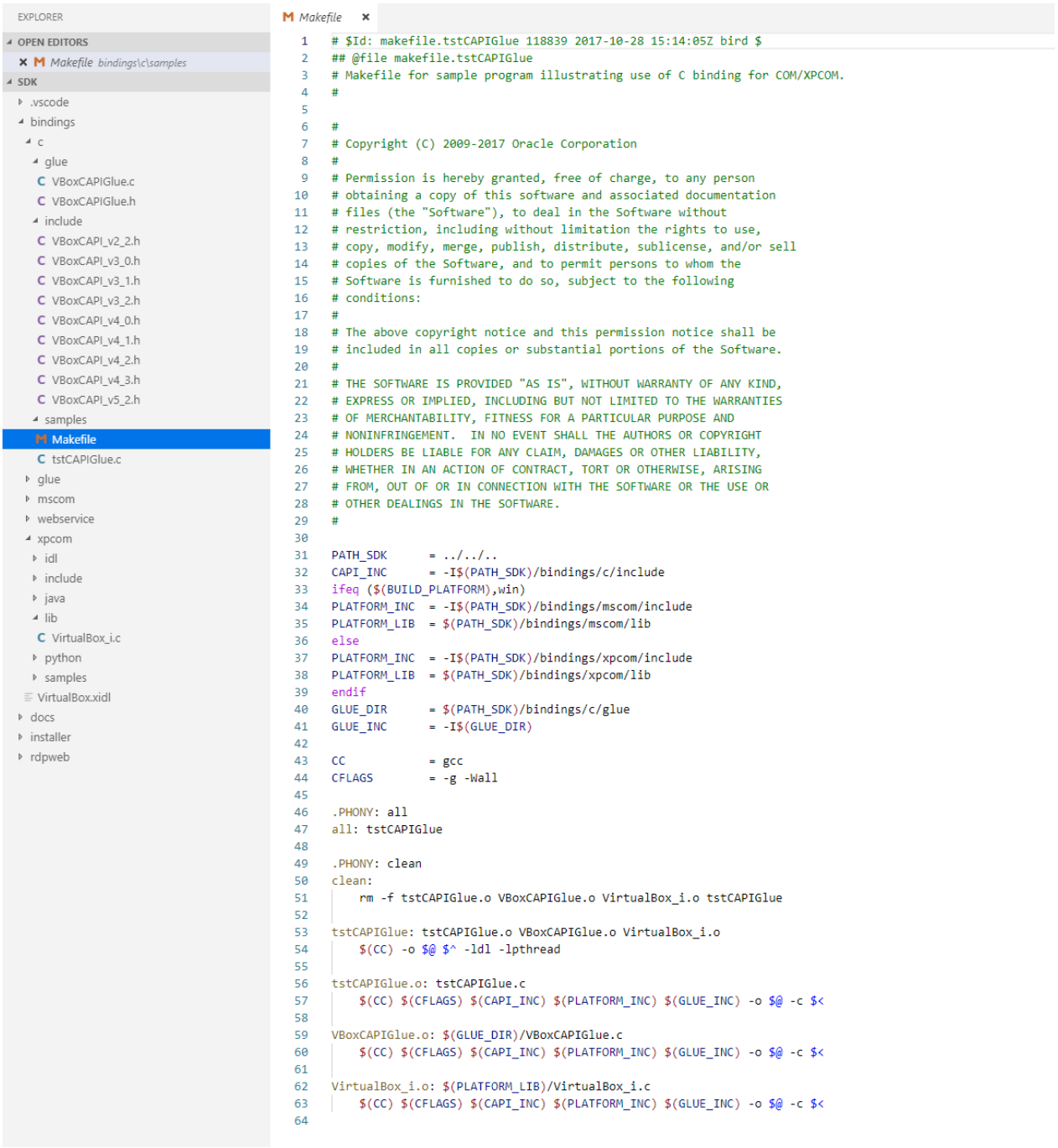
https://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html

39. The Frame Buffer Device [tiešsaiste] – [atsauce: 2019.05.02]. Pieejams: https://www.kernel.org/doc/Documentation/fb/framebuffer.txt
40. Raspbian [tiešsaiste] – [atsauce: 2019.05.04]. Pieejams: https://www.raspberrypi.org/downloads/raspbian/
41. Endianness [tiešsaiste] – [atsauce: 2019.05.04]. Pieejams: https://en.wikipedia.org/wiki/Endianness
42. IEEE and The Open Group, ***POSIX.1-2017***, The Open Group Technical Standard Base Specifications, Issue 7, 2017.
43. pthread_create(3) – Linux manual page [tiešsaiste] – [atsauce: 2019.05.04]. Pieejams: http://man7.org/linux/man-pages/man3/pthread_create.3.html
44. VirtualBox Main API: IEvent Interface Reference [tiešsaiste] – [atsauce: 2019.05.04]. Pieejams: https://www.virtualbox.org/sdkref/interface_i_event.html
45. "Raspbbery Pi" elektroniskie komponenti un detaļas tiešsaistē | LEMONA [tiešsaiste] – [atsauce: 2019.05.05]. Pieejams: https://www.lemona.lv/?sn.q=Raspberry%20PI
46. Raspberry Pi 3B+ Specs and Benchmarks [tiešsaiste] – [atsauce: 2019.05.05]. Pieejams: https://www.raspberrypi.org/magpi/raspberry-pi-3bplus-specs-benchmarks/
47. Visual Studio [tiešsaiste] – [atsauce: 2019.05.05]. Pieejams: https://visualstudio.microsoft.com/downloads/
48. Visual Studio Code [tiešsaiste] – [atsauce: 2019.05.05]. Pieejams: https://code.visualstudio.com/
49. Why is epoll faster than select | StackOverflow [tiešsaiste] – [atsauce: 2019.05.05]. Pieejams: https://stackoverflow.com/a/17355702

PIELIKUMI

1. pielikums:

VirtualBox SDK COM GLUE piemēra kompilēšana



2. pielikums:

Infiniviz DisplayWall un RaspberryPiClient CMake iestatījumi

```
# [DisplayWallServer]
add_executable(DisplayWallServer
	src/DisplayWallServer/AMF/AMFDeserializer.cpp
	src/DisplayWallServer/AMF/AMFObject.cpp
	src/DisplayWallServer/AMF/AMFSerializer.cpp
	src/DisplayWallServer/AMF/AMFValue.cpp
	src/DisplayWallServer/DisplayWallServer.cpp
	src/DisplayWallServer/Main.cpp
	src/DisplayWallServer/Profiler.cpp
	src/DisplayWallServer/RawH264FileTransport.cpp
	src/DisplayWallServer/RtmpSocket.cpp
	src/DisplayWallServer/RtmpTransport.cpp
	src/DisplayWallServer/RtpH264Socket.cpp
	src/DisplayWallServer/RtpH264Transport.cpp
)
target_include_directories(DisplayWallServer PRIVATE
	${CMAKE_SOURCE_DIR}/dep/libyuv/include
)
target_link_libraries(DisplayWallServer dl pthread X11 stdc++fs)
target_compile_definitions(DisplayWallServer PRIVATE NDEBUG RELEASE)
target_compile_options(DisplayWallServer PRIVATE -ggdb -ffunction-sections -O0 -std=c++14)
set_target_properties(DisplayWallServer PROPERTIES PREFIX "")

# [RaspberryPiClient]
FIND_PACKAGE ( Threads REQUIRED )
add_executable(RaspberryPiClient
	Main.cpp
	Profiler.cpp
	RtpOverUdpVideoParser.cpp
	Rtp/RtpH264Packet.cpp
	Rtp/RtpMJPEGPacket.cpp
	Posix/ClosableDescriptor.cpp
	Omx/Component.cpp
	Omx/ComponentPort.cpp
	JPEG/Huffman.cpp
)
target_include_directories(RaspberryPiClient PRIVATE
	/opt/vc/include/
)
target_link_libraries(RaspberryPiClient ${CMAKE_THREAD_LIBS_INIT}
/opt/vc/lib/libbcm_host.so /opt/vc/lib/libvcos.so /opt/vc/lib/libopenmaxil.so rt)
target_compile_definitions(RaspberryPiClient PRIVATE NDEBUG RELEASE)
target_compile_options(RaspberryPiClient PRIVATE -ggdb -ffunction-sections -O3 -std=c++11)
set_target_properties(RaspberryPiClient PROPERTIES PREFIX "")
```

3. pielikums:

Skāriena signāla struktūras pirmkods touch_sig.h

```
#pragma once
#include "common.h"

// Common touch signal struct
#define TOUCH_SIG_STRFMT "%u %u %hhu %d %hhd\n"
struct touch_sig {
	uint32_t x;
	uint32_t y;
	uint8_t slot;
	int32_t tracking_id;
	int8_t touch_state; // -1: doesnt change state / still touching,
	                    //  0: touch release, 1: initial touch
};

// Modify given buf to hold touch_sig fields in string format delimited by spaces
// and terminated with \0
// Returns 0 if struct write succeeded, -1 if encoding error,
//         1 if buf/maxsize was too small
static int serialize_touch_sig(struct touch_sig tsig, char* bufptr, size_t maxsize) {
	int ret = snprintf(bufptr, maxsize, TOUCH_SIG_STRFMT,
	                   tsig.x, tsig.y, tsig.slot, tsig.tracking_id, tsig.touch_state);
	if (ret < 0) {
		return -1;
	}
	else if (ret < maxsize) {
		return 0;
	}
	else {
		return 1;
	}
}

// Deserialize touch sig string fields into touch_sig struct
// Returns 0 if deserialization succeeded, -1 if input failure,
//         1 if couldn't scan all touch_sig fields
static int deserialize_touch_sig(struct touch_sig* tsig, char* bufptr) {
	int ret = sscanf(bufptr, TOUCH_SIG_STRFMT,
	                 &(tsig->x), &(tsig->y),
	                 &(tsig->slot), &(tsig->tracking_id), &(tsig->touch_state));
	if (ret < 0) {
		return -1;
	}
	else if (ret != 5) {
		return 1;
	}
	else {
		return 0;
	}
}
```

4. pielikums:

Praktiskā projekta CMakeSettings.json fails

```
{
  "configurations": [
    {
      "name": "RPi-Debug-TSControlNode",
      "generator": "Unix Makefiles",
      "remoteMachineName": "1570323091;192.168.1.112 (username=pi, port=22,
                                                      authentication=Password)",
      "configurationType": "Debug",
      "remoteCMakeListsRoot": "/home/pi/projects/TSControlPanel/source",
      "cmakeExecutable": "cmake",
      "buildRoot": "${env.LOCALAPPDATA}\\CMakeBuilds\\TSControlPanel\\build",
      "installRoot": "${env.LOCALAPPDATA}\\CMakeBuilds\\TSControlPanel\\install",
      "remoteBuildRoot": "/home/pi/projects/TSControlPanel/build",
      "remoteInstallRoot": "/home/pi/projects/TSControlPanel/install",
      "remoteCopySources": true,
      "remoteCopySourcesOutputVerbosity": "Normal",
      "remoteCopySourcesConcurrentCopies": "10",
      "remoteCopySourcesMethod": "rsync",
      "remoteCopySourcesExclusionList": [
        ".vs",
        ".git"
      ],
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "gcc-arm" ],
      "intelliSenseMode": "linux-gcc-arm"
    },
    {
      "name": "RPi-Debug-TSControlServer",
      "generator": "Unix Makefiles",
      "remoteMachineName": "576969687;192.168.1.125 (username=azeroc, port=22,
                                                      authentication=PrivateKey)",
      "configurationType": "Debug",
      "remoteCMakeListsRoot": "/home/azeroc/projects/TSControlPanel/source",
      "cmakeExecutable": "cmake",
      "buildRoot": "${env.LOCALAPPDATA}\\CMakeBuilds\\TSControlPanel\\build",
      "installRoot": "${env.LOCALAPPDATA}\\CMakeBuilds\\TSControlPanel\\install",
      "remoteBuildRoot": "/home/azeroc/projects/TSControlPanel/build",
      "remoteInstallRoot": "/home/azeroc/projects/TSControlPanel/install",
      "remoteCopySources": true,
      "remoteCopySourcesOutputVerbosity": "Normal",
      "remoteCopySourcesConcurrentCopies": "10",
      "remoteCopySourcesMethod": "rsync",
      "remoteCopySourcesExclusionList": [
        ".vs",
        ".git"
      ],
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "gcc-arm" ],
      "intelliSenseMode": "linux-gcc-arm"
    }
  ]
}
```

5. pielikums:

Praktiskā projekta programmatūru CMakeLists.txt saturs

```
cmake_minimum_required (VERSION 3.8)

# Add source to this project's executable.
add_executable(TSControlNode
	main.c
	framebuffer.c
	touch_io.c
)
target_link_libraries(TSControlNode PRIVATE ts)

# === C GLUE API example ===
add_executable(tstCAPIGlue
	vbox_api/samples/tstCAPIGlue.c

	# VBox API
	vbox_api/glue/VBoxCAPIGlue.c
	vbox_api/xpcom/lib/VirtualBox_i.c
)
target_include_directories(tstCAPIGlue
	PRIVATE ${CMAKE_SOURCE_DIR}/TSControlServer/vbox_api/glue
	PRIVATE ${CMAKE_SOURCE_DIR}/TSControlServer/vbox_api/include
	PRIVATE ${CMAKE_SOURCE_DIR}/TSControlServer/vbox_api/xpcom/include
)
target_link_libraries(tstCAPIGlue dl ${CMAKE_THREAD_LIBS_INIT})

# Add source to this project's executable.
add_executable(TSControlServer
	main.c
	server.c
	vbox.c
	vbox_event.c
	vbox_mouse.c

	# VBox API
	vbox_api/glue/VBoxCAPIGlue.c
	vbox_api/xpcom/lib/VirtualBox_i.c
)
target_include_directories(TSControlServer
	PRIVATE ${CMAKE_SOURCE_DIR}/TSControlServer/vbox_api/glue
	PRIVATE ${CMAKE_SOURCE_DIR}/TSControlServer/vbox_api/include
	PRIVATE ${CMAKE_SOURCE_DIR}/TSControlServer/vbox_api/xpcom/include
)
target_link_libraries(TSControlServer m dl ${CMAKE_THREAD_LIBS_INIT})

# === Group targets ===
add_custom_target(server_targets)
add_dependencies(server_targets tstCAPIGlue TSControlServer)
```

DOKUMENTĀRĀ LAPA

Maģistra darbs "Skārienjūtīgas saskarnes izveide operatīvai sistēmai darbināmai uz virtuālās mašīnas Virtualbox" izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 2019.05.16.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: **Tomass Lācis** _____

(paraksts un datums)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Vadītājs: **profesors Dr. sc. comp. Guntis Arnicāns** _____

(paraksts un datums)

Darbs iesniegts **maģistrantūras sekretariātā** _____.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: _____

(Metodiķes paraksts)

Recenzents: _____

(Akad. amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)