УНИВЕРЗИТЕТ У БЕОГРАДУ

ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

ИМПЛЕМЕНТАЦИЈА ДИСТРИБУИРАНОГ СИСТЕМА ЗА ИГРАЊЕ ИГАРА У ОБЛАКУ

Мастер рад

Ментор:

проф. др Марко Мишић

Кандидат:

Александар Динчић

2022/3055

Београд, октобар 2025.

САДРЖАЈ

1. Увод

Развој видео игара уско је повезан са развојем рачунарског хардвера. Са порастом могућности хардвера расту визуелна и механичка комплексност игара, али и системски захтеви за њихово покретање. Ово значи да су савремени наслови ограничени на најновије играчке конзоле и рачунаре са напреднијим процесором и графичком картицом, као и већом количином радне и складишне меморије, чиме се значајно сужава скуп корисника којима су такве игре приступачне. Са друге стране, развој брзих и поузданих интернет конекција, напредних техника кодирања и преноса слике и скалабилних серверских инфраструктура омогућили су појаву првих сервиса за играње игара у облаку. При коришћењу ових система, игра се извршава на некој од серверских машина, док се слика и звук преносе до корисничког уређаја у реалном времену. Корисник интерагује са игром тако што се унете команде преносе и извршавају на серверу. Главне погодности оваквог приступа су могућност играња захтевних игара уз перформансе дефинисане понудом пружаоца сервиса, као и тренутан приступ великом асортиману игара са било ког уређаја, без потребе за инсталацијом. Овај рад се бави имплементацијом једног таквог система. Циљ је развити прототип који кориснику омогућава играње одабраних игара из веб претраживача.

Рад је сачињен од осам поглавља. Друго поглавље садржи преглед значајнијих постојећих решења, са циљем представљања одлика оваквих система. Затим ће, на основу тога, бити успостављени функционални захтеви система којег желимо да имплементирамо.

Треће поглавље представља сажети приказ добијеног решења и његову архитектуру. Такође ће бити описане технологије коришћене за његову реализацију.

Наредна поглавља детаљно описују имплементацију система, са освртом на најзначајнијим проблемима са којима смо се сусрели током имплементације. Четврто поглавље бавиће се преносом слике и звука. Биће приказане коришћене технике дохватања, кодирања и преноса до корисника, са нагласком на смањењу кашњења.

Пето поглавље посвећено је проблему преноса корисничких команди. Описаћемо начин на који се унос преноси до серверске машине и затим симулира у оквиру игре.

Шесто поглавље бави се самом корисничком сесијом, укључујући извршавање игре и управљање корисничким датотекама. Детаљно ће бити описана комуникација и проток порука између корисника, серверских компоненти и базе података, од отварања сесије до њеног затварања.

У седмом поглављу приказана је евалуација добијеног решења. Биће описан метод тестирања функционалности система, као и добијени резултати. Такође ће бити извршено мерење метрика од значаја за систем, попут кашњења преноса и времена покретања сесије.

На крају, резимираћемо рад и добијене резултате. Затим ћемо размотрити могућности за побољшање система.

2. Опис система

У овом поглављу дефинисаћемо одлике система којег желимо да имплементирамо. У сврху упознавања са развојем и кључним могућностима оваквих система, најпре ћемо представити неколико историјски значајних примера. Затим ћемо дефинисати захтеве које наш систем треба да испуњава.

2.1. Преглед постојећих сервиса

Први пример сервиса за играње игара у облаку појавио се 2003. године под именом *Game Cluster* (*G-Cluster*). Нагласак је био на игрању игара преко сет-топ бокс уређаја, а корисницима су се нудиле одабране лиценциране игре. Ове игре су биле средње комплексности, а за њихово прилагођавање платформи неопходно је било лиценцирање и измена изворног кода. Како би се постигле гарантоване перформансе платформа је сарађивала са добављачима интернет услуга, који су обезбеђивали серверске машине за извршавање игара. [1]

Следећи корак у овој сфери направили су сервиси *OnLive* и *Gaikai* 2010. године. Развој паметних уређаја и техника кодирања слике омогућио је приступ овим сервисима из веб претраживача са личних рачунара, паметних мобилних уређаја и телевизора. Понуда игара била је већа и покривала је нове, комплексније игре. Док је *OnLive* платформа нудила комплетне игре уз претплату и њихово изнајмљивање, понуда платформе *Gaikai* је била ограничена на пробне верзије игара. Оба сервиса купила је компанија *Sony* и искористила их као темељ за свој *PlayStation Now* сервис, о којем ће бити речи касније. [2]

Слика 2.1.1. Сервиси G-Cluster, OnLive и Gaikai [3][4][5]

Компанија *NVIDIA* представила је 2013. године сервис *NVIDIA GRID*, који ће касније бити преименован у *GeForce NOW*. Као водећа компанија у сфери графичких картица, на располагању имају сопствени погон напредних серверских картица, као и ранији приступ технологијама попут виртуелизације и партиционисања графичких картица. Ово се одражава на високе перформансе које сервис нуди, а снага хардвера којим корисник располаже зависи од нивоа његове претплате. Поред игара које су у понуди у оквиру сервиса, једна од иновација јесте интеграција са постојећим платформама за играње игара као што је *Steam*, што омогућава кориснику да игра било коју игру коју већ поседује на тим платформама.[6] За серверске машине користи се модификовани *Linux* оперативни систем.[7] Пренос слике и звука користи *Real-time Transport Protocol* (*RTP*), док се за кодирање слике користи стандард *H.264*. [8]

Компаније које поседују играчку конзолу као један од својих главних производа често их користе као основу за сервисе оваквог типа. Ово је случај са компанијом *Sony* и њиховим сервисом *PlayStation Now*, у чијој инфраструктури се користе блејд сервери базирани на њиховој *PlayStation* конзоли.[9] Сличан приступ користи и компанија *Microsoft*, која за свој *Xbox Cloud Gaming* сервис користи хардвер базиран на *Xbox* конзоли.[10] За коришћење оваквих сервиса обично је потребан контролер компатибилан са датом конзолом. За сву комуникацију и пренос података сервис *PlayStation Now* користи сопствени нестандардни протокол грађен преко *UDP* протокола. [8]

Последњи пример који ћемо размотрити је сервис *Stadia* компаније *Google*. Издат 2019. године, сервис је нудио одређен скуп игара са системом претплате. Сервис је имао висок ниво интеграције са *Google* производима, те је првенствено подржавао платформе попут *Chrome* и *Chromium* претраживача, или рачунара са *ChromeOS* оперативним системом. За серверску инфраструктуру, због своје распрострањености, користе се постојећи *Google* дата центри, са машинама које користе *Linux* оперативни систем и измењене *AMD* графичке картице.[11] Код овог сервиса за пренос се користи стандардни *WebRTC* протокол, са подршком за *H.264*, као и новији *VP9* стандард кодирања слике.[8] Сервис је престао са радом 2023. године.

Слика 2.1.2. Сервиси GeForce NOW, PlayStation Now и Stadia [12][13][14]

2.2. Захтеви система

Тема овог рада јесте имплементација једног једноставног дистрибуираног система за играње игара у облаку, са циљем истраживања проблема удаљеног играња игара са системског становишта. Сервис треба бити реализован у виду веб апликације којој корисник приступа из претраживача на рачунару.

Довољно је да апликација садржи само једну страницу, која служи за играње игара. Два параметара која ова страница треба да прима јесу корисничко име и игра коју корисник жели да игра. Пошто регистрација и аутентикација корисника нису од великог значаја за елементе система које истражујемо, нећемо их имплементирати у оквиру овог рада. Уместо тога, корисничко име је довољно користити као кључ при дохватању и чувању корисничких података.

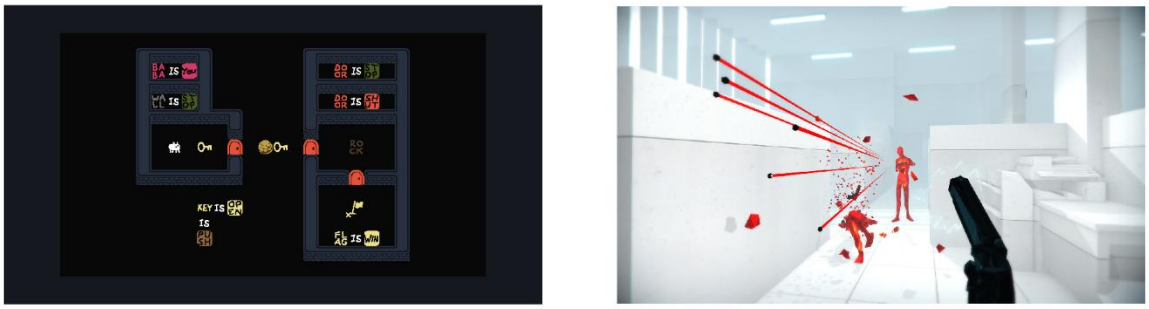
По приступу страници, игру треба покренути на некој од удаљених серверских машина и одмах започети пренос слике и звука до корисничког претраживача у реалном времену. Такође треба започети процес читања корисничког уноса, његовог преноса до серверске машине и симулације у оквиру игре. Од периферија за кориснички унос систем треба да подржава миш и тастатуру. Сесија треба да траје све док корисник не затвори прозор.

Игре које систем подржава требају бити доступне у виду самосталних извршивих апликација за одабрани оперативни систем платформе, без зависности од других играчких платформи као што је *Steam*. Извршно окружење треба бити транспарентно за игре, и није очекивано да се изворни код или датотеке игре морају прилагођавати како би се она могла играти у оквиру нашег система. За игре које ће бити у понуди нашег сервиса изабрали смо следећа два наслова:

- *Baba is You* је логичка видео игра из 2019. године. У овој игри играч се, уносом на тастатури, креће по правоугаоној мрежи. Циљ је прећи сваки од нивоа пратећи

правила, која су такође елементи мреже и којима се може манипулисати. Ова игра је изабрана због једноставног, дводимензионалног графичког приказа и ниских системских захтева, као и због своје потезне природе, што је чини толерантнијом на већа кашњења преноса и команди.

- *SUPERHOT* је акциона игра из првог лица из 2016. године. Играч користи миш и тастатуру за кретање по терену и пуцање. Циљ је елиминисати све противнике на сваком од нивоа, а играчу су на располагању одређене механике манипулације времена. Графички приказ ове игре садржи тродимензионалне елементе, што је чини захтевнијом за извршавање. Због своје акционе природе игра захтева прецизнији пренос уноса и мала кашњења. Ове особине чине ову игру меродавном за проверу могућности играња акционих и графички напреднијих игара на нашем систему.



Слика 2.2.1 Игре Baba is You и SUPERHOT [15][16]

Поред изабраних игара, систем треба бити лако проширив новим играма. Такође, за сваку од игара систем треба да чува корисников напредак. По завршетку сесије личне датотеке корисника треба сачувати како би, при покретању следеће сесије, корисник могао да настави са игром одакле је стао на претходној сесији.

Систем мора бити способан да опслужује више од једне корисничке сесије без сметњи. Важно је и да систем буде пројектован тако да буде скалабилан и да подржава једноставно проширење капацитета.

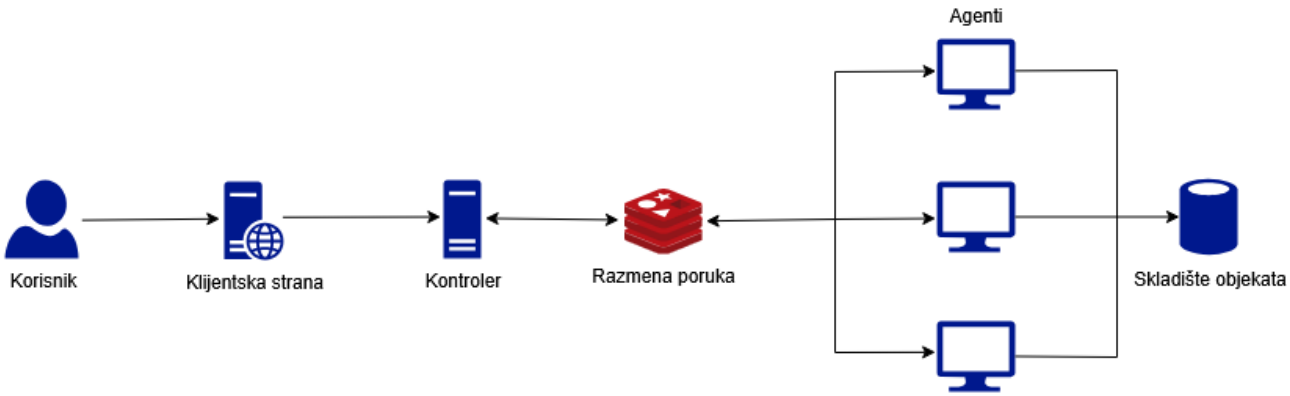
Најзначајнија метрика система за корисничко искуство јесте кашњење преноса, и од ње увелико зависи употребљивост система. Из тог разлога, приликом имплементације, неопходно је тежити што мањем кашњењу. Прихватљиво је жртвовати друге аспекте преноса, попут квалитета слике, како би се кашњење смањило на задовољавајући ниво.

Наведени захтеви описују минимални скуп функционалности које један сервис за играње игара у облаку мора да испуњава. Имплементацијом система који испуњава ове захтеве стећи ћемо разумевање начина рада најважнијих аспеката оваквих система.

3. Архитектура система

Ово поглавље садржи кратак опис нашег решења. Представићемо архитектуру система, са освртом на сваку од компоненти система, њихову улогу, као и начин повезивања између њих. Такође ће бити набројане технологије коришћене у имплементацији.

Решење је израђено у виду једног дистрибуираног система сачињеног од више независних компоненти. На слици 3.1. дата је визуелизација архитектуре система са свим његовим компонентама. Сажето излагање о свакој од приказаних компоненти биће дато у потпоглављима која следе.



Слика 3.1. Архитектура система

3.1. Клијентска страна

Клијентска страна је компонента у виду веб сервера који представља приступну тачку систему за корисника. У питању је једноставна веб апликација која нуди једну страницу, преко које корисник са унетим корисничким именом игра игру по жељи. По отварању странице, она шаље захтев контролеру за започињање сесије, а затим успоставља *peer-to-peer* везу са додељеним агентом и започиње пријем приказа игре и слање корисничког уноса.

Ова компонента је имплементирана користећи радни оквир за веб апликације *Express.js*.[17] Комуникација са агентом, као и слање уноса врши се коришћењем комуникационог протокола *WebSocket*, док се за пренос слике и звука користи протокол *WebRTC*, који омогућава слање мултимедијалног садржаја преко *peer-to-peer* везе у реалном

времену.[18][19] Оба протокола су стандардизовани у данашњим веб претраживачима. Детаљи о начину рада протокола *WebRTC* биће представљени у четвртом поглављу, док ће се о коришћењу протокола *WebSocket* за пренос команди говорити у петом поглављу.

3.2. Контролер

Контролер представља компоненту која управља сесијама и додељује агенте корисницима. По пријему захтева за почетак сесије од стране корисника, контролер шаље поруку са захтевом у ред порука, који служи као посредник између контролера и агената. Када један од агената прихвати захтев, контролер шаље кориснику одговор са информацијама о започетој сесији, како би корисник могао да успостави директну везу са агентом.

Попут клијентске стране, и ова компонента је имплементирана у виду веб сервера. За имплементацију је коришћен радни оквир *Express.js*.

3.3. Размена порука

За потребу размене порука између контролера и агената користи се база података *Redis*.[20] Као база која чува податке у радној меморији, због високих перформанси и флексибилности чест је избор за ову сврху. У понуди је мноштво апстракција за размену порука, од којих ћемо користити редове. Детаљи о коришћењу размене порука биће дати у шестом поглављу.

3.4. Агенти

Агенти представљају машине, физичке или виртуелне, на којима се извршавају игре. Ове машине најпре ишчекују захтев за покретање сесије из реда порука. По пристизању захтева, шаље се порука о прихватању захтева и агент успоставља директну комуникацију са корисником. Ако тражена игра не постоји на хард диску, довлачи се из складишта објеката, а исто се чини и са корисничким датотекама, уколико је корисник већ играо игру раније. Током трајања сесије, агент је задужен за слање приказа кориснику, као и за читање корисничког уноса и његово извођење. По завршетку сесије, агент чува корисничке датотеке у складишту објеката и ишчекује следећу сесију. Сваки агент извршава тачно једну сесију, те је број корисника који истовремено могу да играју игре ограничен бројем агената.

Агенти користе *Windows 11* оперативни систем, због највеће стопе компатибилности са данашњим играма. Управљачка логика агената је имплементирана у програмском језику *Python*, који је изабран због своје једноставности и развијених библиотека за комуникацију и рад са осталим компонентама система, као и због интеграција са неопходним системским позивима.[21] Дохватање, кодирање и пренос слике и звука извршава се користећи мултимедијалну библиотеку *GStreamer*.[22] Ова библиотека омогућава комплексну обраду слике и звука креирањем цевовода сачињених од компоненти које дефинишу сваки корак обраде. Детаљи рада ове библиотеке, као и коришћени протоколи хватања, кодирања и преноса биће приказани у четвртом поглављу.

3.5. Складиште објеката

Пошто игре и кориснички подаци представљају бинарне датотеке, за њихово чување користићемо складиште објеката. За наше потребе одабрано је складиште објеката *MinIO*.[23] Главне одлике овог складишта су брзина и скалабилност, а поред основних операција чувања и дохватања датотека, оно нуди додатне могућности попут верзионисања датотека и репликације. О начину коришћења складишта објеката детаљније ће бити речи у шестом поглављу.

Описана архитектура је модуларна и испуњава услов скалабилности. Клијентска страна и контролер су једноставни веб сервиси без стања, тако да их је могуће скалирати додавањем нових инстанци и вршењем распоређивања оптерећења. Нови агенти су одмах по покретању спремни за извршавање сесија, конзумирањем захтева из реда порука.

У наредним поглављима бавићемо се најзначајнијим проблемима са којима смо се сусрели током имплементације система. Детаљи имплементације сваке од компоненти биће приказани приликом решавања проблема релевантних за ту компоненту.

4. Пренос слике и звука

У овом поглављу бавићемо се проблемом преноса слике и звука, емитованих од игре покренуте на агенту, до корисничког веб претраживача. Проблем ћемо решавати засебно за слику и за звук, а рашчланићемо га на потпроблеме дохватања, кодирања, преноса и репродуковања. Најпре ћемо описати начин рада библиотеке *GStreamer* и протокола *WebRTC*, који представљају кључне технологије за решавање овог проблема. Затим ћемо детаљно описати решење сваког од потпроблема, са образложењем коришћених техника, формата и параметара у циљу компатибилности са веб претраживачима и минимизације кашњења.

4.1. О библиотеци *GStreamer*

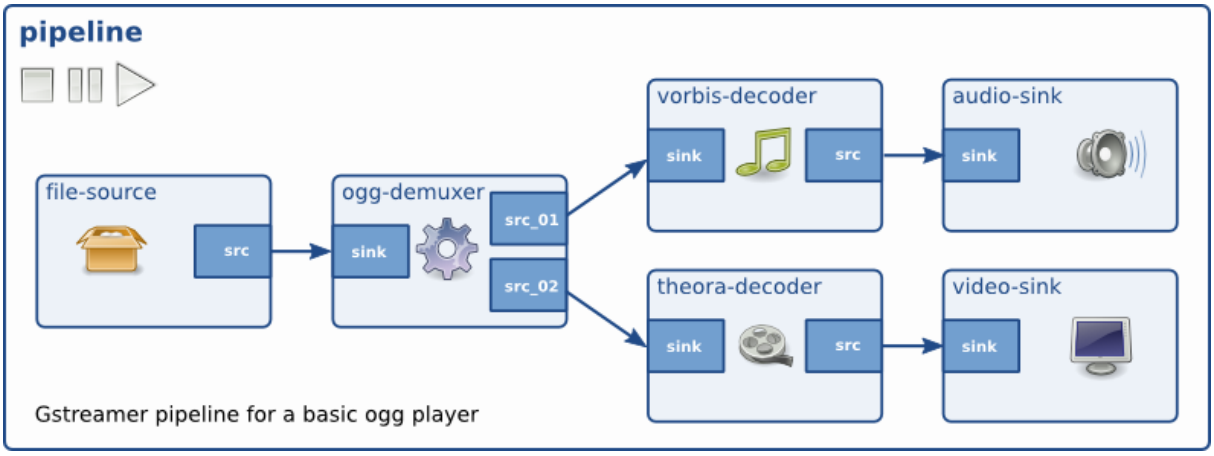
Библиотека *GStreamer* омогућава обраду мултимедијалних токова дефинисањем цевовода. Цевоводи представљају усмерене графове, где сваки чвор графа дефинише један корак у обради података. У контексту библиотеке, чворови се називају елементима (*elements*). Сваки елемент има дефинисане прикључке (*pads*), који представљају интерфејсе за повезивање између елемената и могу бити изворни (*source pads*) уколико шаљу податке, и одводни (*sink pads*) уколико их примају. Прикључци такође имају дефинисане могућности (*capabilities*), које дефинишу врсте података које могу да примају, уколико су одводни, или шаљу, уколико су изворни. Везе између елемената се успостављају преко изворног прикључка једног елемента и одводног прикључка другог, под условом да су им могућности компатибилне.

На основу своје улоге, елементи се могу класификовати у следеће групе:

- Изворни елементи (*source elements*) увозе податке у цевовод читајући их из датотека, уређаја попут камера и микрофона, или других екстерних извора података. Садрже искључиво изворне прикључке, и као такви представљају први корак обраде.
- Одводни елементи (*sink elements*) извозе податке из цевовода њиховим записивањем у датотеке, као и емитовањем преко излазних уређаја или преко

мреже. Садрже искључиво одводне прикључке, и као такви представљају последњи корак обраде.

- Филтери (*filters*), врше обраду података, и као такви чине главни део цевовода. Поседују и изворне и одводне прикључке. Користе се за операције попут додавања трансформација и ефеката, кодирања и декодирања. Од подтипова филтера треба поменути и мултиплексере (*muxers*), који спајају податке из више одводних прикључака у један изворни прикључак, као и демултиплексере (*demuxers*), који раздвајају податке из једног одводног прикључка у више изворних прикључака.



Слика 4.1. Пример цевовода у библиотеци GStreamer [24]

Библиотека сама по себи нуди велики број додатака, груписаних у оквиру званичних колекција, који садрже елементе за рад са често коришћеним форматима, протоколима, ефектима, као и улазним и излазним периферијама. Уколико је неопходно, корисник може креирати нове елементе имплементирањем понуђених интерфејса. [24]

Сви програмски позиви које библиотека нуди представљају омотач око извршног окружења библиотеке. Извршно окружење је такође могуће покренути као самостални процес, са еквивалентним могућностима у поређењу са библиотекама омотачима. Пошто у тренутку писања овог рада постоје проблеми са радом омотача у програмском језику *Python* специфично за *Windows* оперативни систем, библиотеку ћемо користити директним покретањем извршног окружења. Иако се на овај начин губи на читљивости кода и видљивости стања цевовода коришћењем омотачких класа, за потребе нашег система нема губитка у основној функционалности, тако да је приступ прихватљив. У наставку поглавља описаћемо конкретне елементе коришћене приликом имплементације система, као и начин њиховог повезивања у цевоводе.

4.2. О протоколу *WebRTC*

Намена протокола *WebRTC* јесте комуникација у реалном времену преко *peer-to-peer* везе. Овај тип везе, у сврху слања слике и звука од агента до корисника, је оптималан за наш сервис због избегавања посредника у комуникацији, што значајно доприноси смањењу кашњења. Протокол је такође подржан од стране свих данашњих веб претраживача, чиме је покривена корисничка платформа коју желимо да подржимо, али и оставља могућности за ширење на друге подржане платформе, попут мобилних уређаја.

Као *peer-to-peer* протокол, постоје два главна проблема које треба решити приликом успостављања везе између две крајње тачке. При проблем јесте међусобна спознаја крајњих тачака. Између њих је неопходно разменити намеру о успостављању везе, као и начин на који се то може извршити, користећи било који други метод комуникације.

Други проблем јесте присутност *Network Address Translation* (*NAT*) технологије у готово свим мрежама којима припадају кориснички уређаји. Ова технологија подразумева пресликавање приватне адресе уређаја у јавну адресу, уз јединствени порт, од стране рутера приликом слања одлазног саобраћаја ван мреже. Такође се врши обрнуто пресликавање приликом обраде долазног саобраћаја, а у зависности од коришћене врсте ове технологије могуће су рестрикције на основу досадашње комуникације примаоца са пошиљаоцем. Ово значи да крајња тачка мора бити свесна начина на који јој се може приступити путем јавне мреже, како би ту информацију могла доставити другој крајњој тачки.

Поступак успостављања везе у протоколу *WebRTC* прилагођен је решавању ових проблема. Први проблем решава се иницијалним поступком сигнализације (*signalling*), који представља размену информација неопходних за успостављање везе између крајњих тачака. Крајње тачке оглашавају ове информације користећи *Session Description Protocol* (*SDP*). У питању је формат поруке у виду парова кључ-вредност која садржи податке попут списка подржаних типова медија и формата, као и адреса преко којих је могућа веза са пошиљаоцем. Како се крајње тачке у овом тренутку не познају, размена ових порука између учесника се врши преко посредника који се назива сервер за сигнализацију (*signalling server*). Сама комуникација са сервером за сигнализацију одређена је специфичном имплементацијом сервера која може бити произвољна, где је најчешћи избор протокол *WebSocket*. Модел размене порука је такав да се *SDP* порука једне од крајњих тачака нуди другој као понуда (*offer*), а затим друга крајња тачка генерише своју *SDP* поруку и шаље је као одговор

(*answer*). Овом приликом одбацују се понуђени формати који нису подржани од стране оба учесника, а може се одбити и целокупна веза уколико не постоји ниједан компатибилан начин за њено остваривање. Очекивани резултат овог поступка јесте да су крајње тачке усаглашене око начина извршавања сесије, те је могуће успоставити везу коришћењем понуђених адреса.

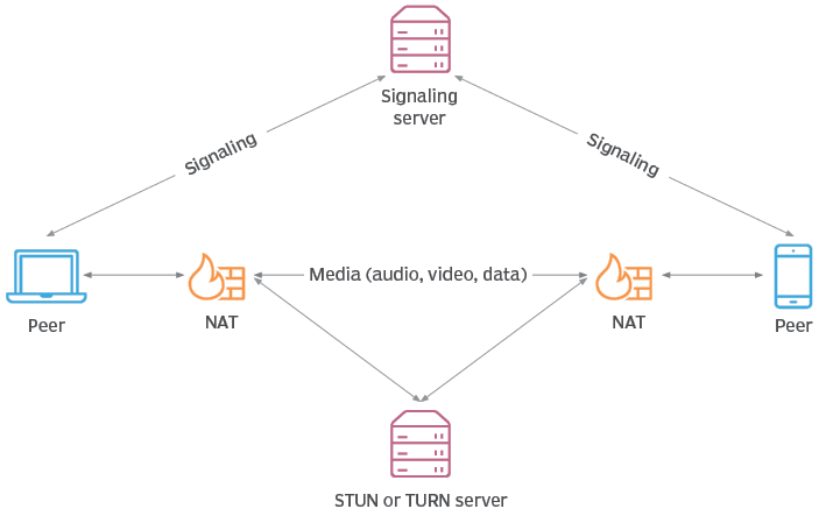
Други проблем решава се коришћењем протокола *Interactive Connectivity Establishment* (*ICE*). Намена овог протокола јесте утврђивање адреса и портова који представљају кандидате преко којих је крајња тачка доступна другој, а затим и избор кандидата са оптималном путањом. Размотрени кандидати, по опадајућој оптималности, су следећи:

- Приватна адреса и порт, који би се могли користити у случају да крајње тачке припадају истој мрежи. Тада би ово представљало оптималан начин повезивања, без изласка у јавну мрежу.
- Адреса и порт добијени користећи протокол *Session Traversal Utilities for NAT* (*STUN*). Под овим се подразумева слање захтева екстерном серверу који имплементира овај протокол. Као одговор на овај захтев, сервер враћа адресу и порт пошиљаоца, што представља мапирање које је *NAT* протокол извршио приликом слања захтева. Овако добијено мапирање валидно је и након завршеног *STUN* захтева, те се може користити за директан приступ крајњој тачки из јавне мреже и радиће са већином *NAT* верзија. У понуди су бесплатни, јавно доступни *STUN* сервери одржавани од стране компаније *Google*, и њих ћемо користити у нашем пројекту.
- Адреса и порт сервера који имплементира протокол *Traversal Using Relays around NAT* (*TURN*). Сервер који имплементира овај протокол користи се у улози посредника за сав саобраћај у случајевима кад права *peer-to-peer* конекција није могућа. Ово је случај код најрестриктивније верзије *NAT* технологије која се назива симетрични *NAT*. Код ове верзије, при слању порука ново мапирање се генерише за сваки пар пошиљаоца и примаоца, и сав долазни саобраћај који не потиче од првобитног примаоца се одбија, те се адреса и порт враћени од стране *STUN* сервера не могу користити од стране друге крајње тачке. Додавање посредника у комуникацији може значајно утицати на кашњење преноса, док одржавање самог сервера захтева незанемарљиву количину рачунарских и мрежних ресурса, те због

тога не постоје јавно доступне машине као за *STUN* протокол, и сервиси су одговорни за одржавање својих *TURN* сервера. Са друге стране, симетрични *NAT* се користи у изразито малом броју кућних мрежа, са мање од 5 процената удела, и чешћа је појава у корпоративним или мобилним мрежама. Стога ово не представља случај од великог значаја за наш систем, те нећемо користити *TURN* сервер у нашем систему, а захтеве корисника који користе симетрични *NAT* ћемо одбијати.

Генерисање кандидата врши се аутоматски на почетку сигнализације од стране протокола *WebRTC* на основу подешених адреса *STUN* и *TURN* сервера. Раније поменуте адресе и портови садржани у *SDP* поруци управо представљају откривене адресе кандидате. По завршеном поступку сигнализације, крајње тачке проверавају доступност добијених кандидата, редом по оптималности путање, и за везу користе најбољи доступни пар кандидата.

Након што је веза успостављена, омогућена је размена података, попут слике и звука, у виду произвољног броја токова, уз коришћење оптималне путање између крајњих тачака. За само слање података користи се протокол *RTP*, док се за пренос статистика о сесији, попут кашњења или количине изгубљених пакета, користи протокол *RTP Control Protocol* (*RTCP*). Коришћењем ових информација ток података аутоматски се прилагођава променљивим мрежним условима адаптирањем тренутног квалитета преноса.[25]



Слика 4.2.1. Дијаграм рада протокола WebRTC [26]

4.3. Слика

Пошто су обрада слике и звука засебне радње, најпре ћемо размотрити пренос слике, без звука. За сваки од корака у проблему приказаћемо коришћене елементе библиотеке *GStreamer*, као и програмски код за повезивање и приказ са корисничке стране.

4.3.1. Дохватање

Први корак у овом поступку јесте дохватање приказа игре. Пошто се игре извршавају на *Windows 11* оперативном систему, за дохватање слике користићемо *Windows Desktop Duplication API*. Ово је изворно подржан део графичког интерфејса *DirectX* чија је сврха пресликавање приказа у радну меморију или у меморију на графичкој картици у виду сирових оквира у *RGBA* формату. [27] Овај метод препоручен је од стране компаније *NVIDIA* као замена за њихов *Capture SDK* на овом оперативном систему. [28]

За коришћење овог интерфејса применићемо изворни елемент под називом *d3d12screencapturesrc*. Сходно интерфејсу који се интерно користи, могућности које изворни прикључак овог елемента нуди јесу сирови оквири у меморији графичке картице или у системској радној меморији.

```
d3d12screencapturesrc show-cursor=true window-handle={window_handle}
```

Код 4.3.1.1. Елемент за дохватање слике

Од параметара, елемент прима ручку прозора којег дохватамо, као и опцију приказивања курсора, коју укључујемо. Начин дохватања ручке прозора игре за потребе коришћења у овом елементу описаћемо у шестом поглављу.

Дохватање слике је континуално, и траје све до експлицитног затварања цевовода од стране корисника, или затварања циљаног прозора.

4.3.2. Кодирање

Следећи корак јесте кодирање сировог приказа у формат погодан за слање преко мреже. Од начина кодирања који су нам на располагању, постоје само два која су претраживачи у обавези да подрже у оквиру подршке за *WebRTC*, а то су *VP8* и *H.264*. [29]

Са друге стране, кодирање може бити рачунски захтеван посао, нарочито код већих резолуција и броја кадрова у секунди. Ово значи да се одређен део процесорског времена и радне меморије мора издвојити за кодирање, чиме се смањују ресурси посвећени игри, што може утицати на њене перформансе. Како машина на којој извршавамо агента поседује

графичку картицу *NVIDIA RTX 3060*, у могућности смо да искористимо технологију *NVENC* својствену *NVIDIA* графичким картицама. Ова технологија омогућава вршење операције кодирања преко графичке картице, користећи посебно хардверско језгро на картици посвећено искључиво пословима кодирања. Ових језгара може бити једно или више, у зависности од модела картице. На овај начин растерећујемо остале рачунарске ресурсе и препуштамо им извршавање игре. Од два стандарда која разматрамо, *NVENC* подржава једино *H.264*, тако да ћемо њега користити за кодирање. [30]

За кодирање овим стандардом коришћењем технологије *NVENC* употребићемо филтер елемент под називом *nvh264enc*. Могућности одводног прикључка покривају мноштво формата, а међу њима су сирови оквири смештени на меморији графичке картице, чиме смо у могућности да се повежемо на изворни прикључак елемента за дохватање без међукорака конверзије или преноса података, што значајно доприноси скраћењу трајања операције.

```
nvh264enc tune=ultra-low-latency preset=p1 rc-mode=cbr bitrate=4000 bframes=0 rc-lookahead=0 gop-size=30 zerolatency=true vbv-buffer-size=400 qp-min=25 qp-max=35
```

Код 4.3.2.1. Елемент за кодирање слике

Процес кодирања је прилагодив и елемент нуди мноштво параметара:

- Највећи значај за смањење кашњења имају параметри *preset* и *tune*. Параметар *preset* представља предефинисане шаблоне подешавања кодирања чија вредност припада опсегу од *p1* до *p7*. Мање вредности су оптимизоване за брзину, док је код већих приоритет квалитет, те бирамо најмању понуђену вредност. Параметар *tune* такође представља свеобухватно подешавање операције, с тим да је у овом случају сврха специјализација за специфичне случајеве коришћења, попут кодирања анимираних видеа или статичних слика. У нашем случају користимо вредност намењену оптимизованој обради у реалном времену.
- Проток података постављамо на константу вредност ради постизања предвидивих перформанси, уз жртвовање квалитета приликом приказа са великим бројем детаља. За вредност протока поставићемо скромнију вредност од 4 *mbit/s*. Ова вредност представља доњу границу прихватљивости за пренос видеа у резолуцији 1920x1080. Такође ћемо смањити величину бафера *Video Buffering Verifier* (*VBV*), који представља хипотетички декодер, који гарантује да ток кодираног видеа неће

преоптеретити један такав декодер и чија мања вредност смањује варијабилност и кашњење преноса, такође уз смањење квалитета.

- У оквиру стандарда за кодирање користе се технике компресије које разматрају не само досадашње оквире, него и будуће. Пример овакве технике су бидирекциони оквири (*B-frames*), међуоквири који представљају модификацију прошлог и будућег оквира, чиме се смањује количина података неопходна за кодирање. Будући оквири такође се могу разматрати за потребе прилагођавања искоришћења протока. У сценарију преноса у реалном времену коришћење ових техника нема смисла, зато што захтева чекање будућих оквира, што повећава кашњење, те ћемо ове технике угасити.
- На крају, подесићемо опсег вредности *Quantization Parameter* (*QP*). Ова вредност припада опсегу између 0 и 51, где мања вредност представља вернију слику, са мање артефакта. За наше потребе ограничићемо ову вредност између 25 и 35, што представља благи компромис квалитета у замену за брзину. Без експлицитног постављања ових параметара, коришћени шаблони постављају овај параметар на веома високе вредности, што доводи до изузетно лошег квалитета и негативног утицаја на игривост игре. [31]

Како овим елементом одређујемо формат видео преноса, могућност изворног прикључка представља једино *H.264* формат.

4.3.3. Пренос

Кодирану слику на крају треба пренети до корисничког веб претраживача и приказати је. Ово вршимо користећи протокол *WebRTC*.

За потребе преноса коришћењем овог протокола доступан је одводни елемент *webrtcsink*, и њиме завршавамо наш цевовод. Пре тога убацићемо још 2 неопходна филтер елемента. Први елемент назива се филтер могућности (*caps filter*). Намена овог филтера јесте специфицирање могућности која ће се користи између два прикључка у случајевима када постоји више компатибилних могућности. У овом случају желимо да специфицирамо формат *H.264* тока који производи елемент *nvh264enc* у *Advanced Video Coding* (*AVC*) формат. Ово је неопходно зато што *WebRTC* захтева овај формат, док се при коришћењу других формата слика не преноси. Упркос овом захтеву, могућности одводног прикључка *webrtcsink*

елемента не ограничавају формат тока, те није гарантовано његово коришћење и неопходно га је експлицитно фиксирати на овај начин.

```
video/x-h264,stream-format=avc
```

Код 4.3.3.1. Филтер могућности

Следећи филтер који ћемо искористити јесте елемент реда (*queue*). Цевовод се подразумевано извршава у оквиру једне нити. Као последица овога, најспорији елемент утиче на перформансе целог цевовода. Елемент реда раздваја делове цевовода иза изворног и одводног прикључка у две одвојене нити и представља бафер између њих. На овај начин део цевовода за обраду слике ради независно од дела за њено слање и у случају успорења код једног од делова нема блокирања другог.

```
queue max-size-buffers=1 max-size-time=0 max-size-bytes=0 leaky=downstream
```

Код 4.3.3.2. Филтер реда

Пошто у нашем случају желимо минимално кашњење, бафер треба бити што мањи, а при препуњавању треба чувати само најновије податке приказа, док се старији одбацују. На овај начин увек шаљемо најновије доступне оквире приказа и не уводимо додатно кашњење.

Сада коначно можемо додати елемент *webrtcsink*. За потребе процеса сигнализације овај елемент захтева постојање сервера за сигнализацију који испуњава специфичан понуђени интерфејс. Овакав сервер омогућава учесницима да се региструју у улози произвођача или потрошача, као и да примају обавештења о осталим учесницима, након чега могу да захтевају отварање сесије са неким од њих, чиме сервер започиње размену *SDP* порука. Уз елемент у понуди је једноставна имплементација сервера која се може користити, и она је довољна за наше потребе.

```
webrtcsink run-signalling-server=true
```

Код 4.3.3.3. Одводни WebRTC елемент

Овим је цевовод завршен, и приказ игре се дохвата у реалном времену на ефикасан начин и нуди кориснику за конзумирање. Покретање извршног окружења библиотеке за дати цевовод врши се преко команде *gst-launch-1.0*, док се везе између елемената у цевоводу означавају узвичником.

```
gst-launch-1.0.exe
d3d12screencapturesrc show-cursor=true window-handle={window_handle}
! nvh264enc tune=ultra-low-latency preset=p1 rc-mode=cbr bitrate=4000 bframes=0
rc-lookahead=0 gop-size=30 zerolatency=true vbv-buffer-size=400 qp-min=25 qp-
max=35
! video/x-h264,stream-format=avc
! queue max-size-buffers=1 max-size-time=0 max-size-bytes=0 leaky=downstream
! webrtcsink run-signalling-server=true
```

Код 4.3.3.4. Цевовод за обраду и пренос слике

4.3.4. Репродуковање

На крају је неопходно имплементирати корисничку страну везе и приказати слику у оквиру претраживача. Корисничка *HTML* страница је једноставна, са једним видео елементом који заузима целу површину странице, као и текстуалним елементом за приказ грешака или поруке о крају сесије.

За потребе повезивања са сервером за сигнализацију користићемо позиве дефинисане у интерфејсу сервера, а можемо користити постојећу *JavaScript* скрипту са омотачима око позива, која је такође доступна уз елемент. Скрипта нам омогућава да региструјемо слушаоца и дефинишемо повратне позиве за догађаје додавања и уклањања потрошача, како бисмо започели или очистили сесију, респективно.

Како смо слику кодирали у стандардизованом видео формату, ток података се може директно приказати путем *HTML* видео елемента. Ово такође чинимо у оквиру регистрованих повратних позива за догађај додавања тока, као и чишћење сесије приликом затварања тока.

Овим смо успешно доставили визуелни приказ игре до корисника. Следећи корак је приказу додати звук који емитује игра.

```
// Konfigurisanje adrese servera za signalizaciju
let video_webrtc_config = {
      meta: { name: `WebClient-Video-${Date.now()}` },
      signalingServerUrl: `${video_signalling_endpoint}`,
};

let videoSession = null;
let video_webrtc_api = new GstWebRTCAPI(video_webrtc_config);

const videoListener = {
      // Poziv prilikom dodavanja potrosaca
      producerAdded: function (producer) {
            const producerId = producer.id;

            // Provera da li je sesija vec pokrenuta
            if (videoSession) {
                  return;
            }

            // Kreiranje potrosacke sesije
            videoSession = video_webrtc_api.createConsumerSession(producerId);

            // Poziv prilikom zatvaranja toka podataka
            videoSession.addEventListener("closed", () => {
                  videoElement.pause();
                  videoElement.srcObject = null;
                  videoSession = null;
                  terminateSession();
            });
            // Poziv prilikom otvaranja toka podataka
            videoSession.addEventListener("streamsChanged", () => {
                  const streams = videoSession.streams;
                  if (streams.length > 0) {
                        // Povezivanje toka sa slikom na HTML video element
                        videoElement.srcObject = streams[0];
                        showError("");
                        videoElement.play().catch(err => {
                              showError("Video autoplay error: " + err.message);
                        });
                  }
            });
            videoSession.connect();
      },

      // Poziv prilikom uklanjanja potrosaca
      producerRemoved: function (producer) {
            if (videoSession) {
                  videoElement.pause();
                  videoElement.srcObject = null;
                  videoSession = null;
            }
            terminateSession();
      }
};
// Registracija slusaoca dogadjaja sa datim pozivima
video_webrtc_api.registerPeerListener(videoListener);
```

Код 4.3.4.1. Клијентски код за повезивање и приказ слике

4.4. Звук

Решење проблема преноса звука, укључујући редослед корака и примењене технике, слично је као код преноса слике. Стога ће ово потпоглавље имати исту структуру као претходно.

4.4.1. Дохватање

За дохватање звука који емитује игра користићемо *Windows Audio Session API* (*WASAPI*). Ово је изворно подржан интерфејс оперативног система *Windows 11*, намењен управљању аудио сесијама између апликација и улазних и излазних звучних уређаја. Преко сесија врши се пренос звучног сигнала између учесника у реалном времену. [32]

Као омотач око овог интерфејса користићемо елемент *wasapisrc*, који отвара сесију и у њој учествује као апликативна страна. Како звук који желимо да дохватимо потиче од процеса игре и намењен је излазном уређају, сесију отварамо са излазним уређајем у повратном режиму (*loopback*), где се тај уређај третира као улазни и пресреће се звук који му је намењен. Елемент такође подешавамо за рад са минималним кашњењем.

```
wasapisrc loopback=true low-latency=true
```

Код 4.4.1. Елемент за дохватање звука

Упркос широким могућностима изворног прикључка, при коришћењу повратног режима једини подржани излазни формат је сирови формат *F32LE*. Име формата означава да је сирови звучни сигнал представљен у виду реалних бројева у формату покретног зареза величине 32 бита у распореду *little-endian*. Дохватање звука је континуално и траје све до затварања цевовода.

4.4.2. Кодирање

Као и код слике, формат у којем кодирамо звук бирамо са циљем максималне компатибилности са претраживачима. Једини обавезни формат прописан од стране протокола *WebRTC* који је подржан у свим данашњим претраживачима је *OPUS*, те ћемо њега користити. [29]

Кодирање у овај формат вршимо коришћењем елемента *opusenc*. Овај елемент на свој одводни прикључак прихвата једино сирови формат *S16LE*, који означава представљање података у виду означених целих бројева величине 16 бита у распореду *little-endian*. Како ово није формат у којем елемент за дохватање емитује податке на изворном прикључку,

неопходно је извршити конверзију. Ово можемо учинити убацивањем филтер елемента *audioconvert* између елемената за дохватање и кодирање, који аутоматски врши конверзију између формата на основу могућности њихових прикључака.

```
audioconvert
```

Код 4.4.2.1. Филтер за конверзију

За вредност протока података приликом кодирања користимо 96 *kbit/s*, што код овог формата представља средину прихватљивог опсега при преносу стерео звука.

```
opusenc bitrate=96000
```

Код 4.4.2.2. Елемент за кодирање звука

Сходно својој улози, могућности на изворном прикључку овог елемента ограничени су на формат *OPUS*.

4.4.3. Пренос

Кодирани звук преносимо на истоветан начин као пренос слике, користећи протокол *WebRTC* и одводни елемент *webrtcsink*. За раздвајање логике обраде звука и логике његовог слања преко мреже, користимо елемент реда који је истоветан оном који је приказан у одељку 4.3.3.

Размотрићемо два приступа решавању проблема преноса звука. Први приступ јесте спајање слике и звука у један ток проширењем постојећег цевовода. У ову сврху елемент *webrtcsink* испуњава функционалност мултиплексера уколико му се на одводне прикључке доведу слика и звук. На овај начин додали смо звук постојећем видео преносу. Приликом дефинисања цевовода у оквиру команде за покретање извршног окружења, прикључивање више грана на мултиплексер врши се додељивањем имена мултиплексеру и коришћењем тог имена на завршетку улазних грана. У цевоводу датом у коду 4.4.3.1. мултиплексеру је дато име *sink*.

```
gst-launch-1.0.exe
webrtcsink name=sink run-signalling-server=true
d3d12screencapturesrc show-cursor=true window-handle={window_handle}
! nvh264enc tune=ultra-low-latency preset=p1 rc-mode=cbr bitrate=4000 bframes=0
rc-lookahead=0 gop-size=30 zerolatency=true vbv-buffer-size=400 qp-min=25 qp-
max=35
! video/x-h264,stream-format=avc
! queue max-size-buffers=1 max-size-time=0 max-size-bytes=0 leaky=downstream
! sink.
wasapisrc loopback=true low-latency=true
! audioconvert
! opusenc bitrate=96000
! queue max-size-buffers=1 max-size-time=0 max-size-bytes=0 leaky=downstream
! sink.
```

Код 4.4.3.1. Јединствени цевовод за обраду и пренос слике и звука

Предност овог приступа је верна синхронизација слике и звука и резултујући јединствени ток података за чије конзумирање нису неопходне никакве промене у тренутном клијентском коду. Међутим, при коришћењу се појављује приметно повећање кашњења преноса за корисника. Ово се може објаснити додатним временом неопходним за синхронизацију слике и звука. Интерно, *WebRTC* преноси слику и звук у виду два одвојена тока података, који се на клијентској страни усаглашавају и сједињују на основу временских ознака придруженим подацима у току преноса, уз додатну контролу кашњења бафером. Овај процес дефинисан је на нивоу претраживача и на њега из наше апликације не можемо утицати и прилагодити га нашим потребама. [33]

Други приступ представља одвојено слање слике и звука, користећи два независна цевовода. Раније дефинисани цевовод за обраду и слање слике остаје непромењен.

```
gst-launch-1.0.exe
wasapisrc loopback=true low-latency=true
! audioconvert
! opusenc bitrate=96000
! queue max-size-buffers=1 max-size-time=0 max-size-bytes=0 leaky=downstream
! webrtcsink run-signalling-server=true
```

Код 4.4.3.2. Издвојени цевовод за обраду и пренос слике

Због независности два тока нема синхронизације између њих, због чега не долази до додатног кашњења током преноса. Међутим, ово уводи могућност да слика и звук постану неусаглашени. Додатни недостатак је пренос преко два одвојена *WebRTC* тока са којима корисничка страна мора руковати истовремено. Како је визуелна компонента значајнија од звучне компоненте код већине игара, укључујући игре које нудимо у оквиру нашег система, негативан ефекат неусаглашености знатно је мањи од негативног ефекта кашњења, те ово

представља прихватљивију опцију у односу на повећање кашњења слике, због чега се одлучујемо за овај приступ.

4.4.4. Репродуковање

Слично начину преноса, дохватање и емитовање звука са клијентске стране извршава се као и емитовање слике, користећи исте серверске позиве. Резултујући код је истоветан коду 4.3.4.1, и једина разлика је повезивање новог тока података на *HTML* елемент за аудио уместо на елемент за видео.

Применом проверених и распрострањених техника дохватања, кодирања и преноса слике и звука успешно смо решили проблем дохватање аудиовизуелног приказа игре покренуте на агенту и његово приказивање у реалном времену на корисничком уређају, уз оптимизовано кашњење и компатибилност са свим данашњим веб претраживачима. У зависности од хардверских могућности, коришћени параметри могу се прилагодити за слање приказа вишег квалитета.

5. Пренос команди

У овом поглављу решаваћемо проблем удаљеног контролисања игре од стране корисника. Размотрићемо коришћене методе дохватања корисничког уноса са миша и тастатуре, његово паковање и пренос преко мреже до агента, и извршавање пристиглих команди у оквиру игре.

5.1. Дохватање

Дохватање корисничког уноса у оквиру веб странице на клијентској страни могуће је регистровањем повратних позива за *JavaScript* догађаје везане за интеракцију са мишем и тастатуром. Ови догађаји подржани су у свим данашњим претраживачима и покривају све релевантне случајеве, укључујући притискање и пуштање дугмета на тастатури, померање миша, притискање и пуштање дугмета на мишу и померање точкића миша.

Приликом рада са догађајима који потичу са тастатуре, могу се регистровати посебни повратни позиви за догађаје притискања и пуштања дугмета. У оба случаја једина релевантна информација је стандардни виртуелни код дугмета чије се стање променило.

Догађаји везани за дугмад на мишу такође су раздвојени на притискање и пуштање. Информација о афектованом дугмету миша представљено је целобројном вредношћу, где вредности 0, 1 и 2 представљају леви тастер, средњи тастер и десни тастер, респективно, док веће вредности означавају додатну дугмад коју миш може да поседује. Приликом догађаја померања точкића миша, померај у пикселима се представља децималном вредношћу.

Догађај померања миша емитује информацију о померају по X и Y оси. Подразумевано ове вредности представљене су као три апсолутне позиције миша након извршеног помераја, и то у односу на видљиви део странице, целокупну страницу и кориснички екран. Са друге стране, другачији формат помераја добијамо коришћењем закључаног режима (*pointer lock mode*). Приликом коришћења овог режима, показивач миша престаје да буде видљив на екрану и више није ограничен на померање у оквиру корисничког екрана, већ се може кретати неограничено, а померај се у догађајима изражава у виду релативног помераја у односу на претходни положај. Овај режим идеалан је за

коришћење код апликација код којих постоји значајан унос са миша, где свакако спада и наш случај играња игара. Такође омогућава лакшу симулацију на страни агента, без потребе за скалирањем апсолутних вредности услед потенцијалне разлике у величини корисничког екрана и симулираног приказа. [34]

Како бисмо подржали цео спектар корисничког уноса са миша и тастатуре, неопходно је обрадити све наведене догађаје. Начин чувања и преноса информација о уносу, као и релевантни исечци кода са имплементацијом повратних позива биће приказани у следећем потпоглављу.

5.2. Пренос

За пренос корисничких команди до агента користићемо протокол *WebSocket*. Ова технологија омогућава континуалну бидирекциону комуникацију између клијента и сервера путем директне везе. У случају нашег система агент представља *WebSocket* сервер, док клијентска страна започиње сесију са њим по добијању одговора о детаљима сесије од контролера, о чему ће бити више речи у шестом поглављу. Овај протокол је једноставан за коришћење, омогућава слање порука произвољног формата и подржан је у свим данашњим претраживачима.

Једноставан приступ решавању проблема преноса је слање поруке агенту у оквиру повратног позива за сваки испаљени догађај везан за кориснички унос. Иако је овакав приступ прихватљив за руковање уносом са тастатуре, чак и мали померај миша генерише изузетно велики број догађаја са инкременталним померајем, те обрада и симулација сваког од њих појединачно може довести до загушења. Стога ћемо применити другачији приступ, где се сав кориснички унос акумулира и шаље периодично као једна порука. На овај начин постижемо смањено оптерећење мреже, корисника и агента. Конкретна учесталост слања коју ћемо користити је 60*Hz*, што је довољна стопа прецизности за већину игара, осим игара са најзахтевнијом прецизношћу уноса, попут музичких игара или игара у такмичарском окружењу, где се због нестабилне природе коришћења сервиса свакако не препоручује играње у облаку.

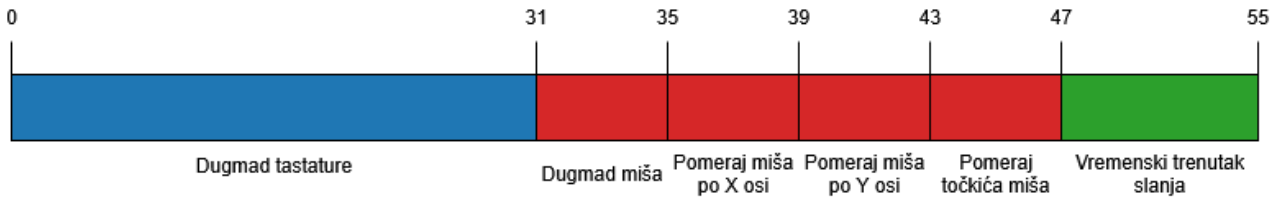
Чување и слање корисничког уноса вршимо у виду бинарне поруке. Прва 32 бајта поруке представљају унос са тастатуре, у виду бит маске где сваки бит представља једно дугме на тастатури. Виртуелни код сваког дугмета представља његову позицију у бит маски, а како кодови припадају опсегу од 0 до 255, 32 бајта је довољно за њихово чување. Јединица

на датој позицији у бит маски значи да је дугме са тим виртуелним кодом притиснуто, док нула значи да је дугме пуштено.

Унос преко дугмади на мишу такође се кодира као бит маска, а величина је 4 бајта. Начин тумачења бит маске исти је као код тастатуре, где положај у бит маски представља целобројни идентификатор дугмета миша. Иако су 3 бита довољна за пренос стања главних дугмади миша, коришћењем веће бит маске оставља се могућност проширења система тако да подржи и унос са напреднијих мишева, који могу имати и двоцифрен број дугмади.

Померај миша биће приказан као две означене целобројне вредности од по 4 бајта, које представљају укупан померај од претходног слања по X и Y оси. На исти начин кодирамо и укупни померај точкића миша.

На крају, додајемо 8 бајта у којем чувамо временски тренутак слања у виду неозначеног целог броја, кодиран као *Unix* време у милисекундама. Ову информацију користићемо за мерење кашњења извршавања команди на агентској страни. Целокупна порука за слање корисничког уноса величине је 56 бајта, што га чини компактним решењем без значајног оптерећења мреже.



Слика 5.2.1. Порука за слање корисничког уноса

У оквиру повратних позива за догађаје уноса ажурирају се релевантни делови поруке. Стање дугмади миша и тастатуре ажурира се битским операцијама, док се померај миша и точкића ажурирају сабирањем нових вредности у акумулаторе. Приликом позивања периодичне функције слања, порука се шаље агенту, а акумулатори помераја враћају на нулу. Слање вршимо једино уколико је корисник направио унос у односу на претходно слање, чиме избегавамо редундантне поруке и додатно смањујемо неопходни проток података.

```
let keys_bit_mask = new Uint8Array(32);
let mouse_buttons = 0;
let mouse_dx = 0, mouse_dy = 0;
let mouse_wheel = 0;

// Funkcija za postavljanje bita u bitskoj maski za tastaturu
function setKey(vk, down) {
    const b = vk >>> 3, bit = vk & 7;
    if (down) {
        keys_bit_mask[b] |=  (1<<bit);
    }
    else {
        keys_bit_mask[b] &= ~(1<<bit);
    }
};

// Funkcija za postavljanje bita u bitskoj maski za misa
function setMouseButton(button, down) {
    if (down) {
        mouse_buttons |= (1 << button);
    } else {
        mouse_buttons &= ~(1 << button);
    }
}

// Dogadjaji za prisitkanje i pustanje dugmeta tastature
addEventListener("keydown", e => setKey(e.keyCode, true),  {passive:true});
addEventListener("keyup",   e => setKey(e.keyCode, false), {passive:true});

// Dogadjaj za pomeranje misa
addEventListener("mousemove", e => {
    if (document.pointerLockElement === videoElement) {
            mouse_dx += e.movementX;
            mouse_dy += e.movementY;
    }
}, {passive:true});
// Dogadjaji za pritiskanje i pustanje dugmeta misa
addEventListener("mousedown", e => {
    if (document.pointerLockElement === videoElement) {
            setMouseButton(e.button, true);
            e.preventDefault();
    }
});
addEventListener("mouseup", e => {
    if (document.pointerLockElement === videoElement) {
            setMouseButton(e.button, false);
            e.preventDefault();
    }
});
// Dogadjaj za pomeranje tockica misa
addEventListener("wheel", e => {
    if (document.pointerLockElement === videoElement) {
            mouse_wheel += Math.round(e.deltaY);
            e.preventDefault();
    }
});
```

Код 5.2.1. Дохватање и паковање корисничког уноса

```
let input_packet = new Uint8Array(56);

function sendInputPacket(force) {
    // Slanje se vrsi samo ako je sesija u toku
    if (!ws || ws.readyState !== WebSocket.OPEN) {
        return;
    }

    // Podrazumevano, slanje se vrsi samo ako je kursor u zakljucanom rezimu
    // (igra je u fokusu) i ako postoje promene u odnosu na prethodno stanje
    if (!force) {
        if (document.pointerLockElement !== videoElement)  {
            return;
        }

        if (!stateChanged()) {
            return;
        }
    }

    // Postavljanje vrednosti poruke
    input_packet.set(keys_bit_mask, 0);

    let view = new DataView(input_packet.buffer);
    view.setUint32(32, mouse_buttons, true);
    view.setInt32(36, mouse_dx, true);
    view.setInt32(40, mouse_dy, true);
    view.setInt32(44, mouse_wheel, true);

    let timestamp = Date.now();
    view.setBigUint64(48, BigInt(timestamp), true);

    // Slanje poruke
    ws.send(input_packet);

    // Cuvanje prethodnog stanja i resetovanje akumulatora
    previous_input_keys.set(keys_bit_mask);
    previous_mouse_buttons = mouse_buttons;

    mouse_dx = 0;
    mouse_dy = 0;
    mouse_wheel = 0;
}

// Pokretanje periodicne funckije
setInterval(() => {
      sendInputPacket(false);
}, 1000 / 60);
```

Код 5.2.2. Слање корисничког уноса

5.3. Извршавање

Након прихватања пристигле поруке са корисничким уносом, агент врши десеријализацију поруке, уз инверзну конверзију бит маски дугмади на тастатури и мишу у одговарајуће идентификаторе. Затим се унос специфициран поруком симулира коришћењем

функције *SendInput* понуђене у оквиру интерфејса *Win32*, којег изворно нуди оперативни систем *Windows 11*. Функција производи синтетички кориснички унос са миша и тастатуре на нивоу оперативног система. Ово значи да прозор игре мора бити у фокусу из перспективе оперативног система како би игра добила унете команде, што агент обезбеђује приликом њеног покретања, о чему ће бити речи у шестом поглављу. Како је у питању функција у програмском језику *C* која је део системског интерфејса, да бисмо је користили из програмског језика *Python* неопходно је користити интерфејс за стране функције (*Foreign Function Interface*). Слично клијентској страни, агент чува претходно стање и унос се симулира само за компоненте код којих се десила промена стања. [35]

```
SendInput = ctypes.windll.user32.SendInput

# Funkcija za simulaciju unosa sa tastature
def key_event(vk, down=True):
    scan = ctypes.windll.user32.MapVirtualKeyW(vk, 0)
    flags = KEYEVENTF_SCANCODE
    if not down:
        flags |= KEYEVENTF_KEYUP

    ki = KEYBDINPUT(wVk=0, wScan=scan, dwFlags=flags, time=0, dwExtraInfo=0)
    inp = INPUT(type=INPUT_KEYBOARD, union=_INPUTunion(ki=ki))
    SendInput(1, ctypes.byref(inp), ctypes.sizeof(INPUT))

# Genericka funkcija za simulaciju dogadjaja sa misa
def mouse_event(flags, data=0):
    mi = MOUSEINPUT(0, 0, data, flags, 0, 0)
    inp = INPUT(type=INPUT_MOUSE, union=_INPUTunion(mi=mi))
    SendInput(1, ctypes.byref(inp), ctypes.sizeof(INPUT))

# Funkcija za simulaciju pomeraja misa
def mouse_move(dx, dy):
    flags = MOUSEEVENTF_MOVE
    mi = MOUSEINPUT(dx, dy, 0, flags, 0, 0)
    inp = INPUT(type=INPUT_MOUSE, union=_INPUTunion(mi=mi))
    SendInput(1, ctypes.byref(inp), ctypes.sizeof(INPUT))

# Funckija za simulaciju pomeraja tockica misa
def mouse_wheel(delta):
    mouse_event(MOUSEEVENTF_WHEEL, delta)

# Funkcije simulaciju unosa sa dugmadi misa
def mouse_left_down():  mouse_event(MOUSEEVENTF_LEFTDOWN)
def mouse_left_up():    mouse_event(MOUSEEVENTF_LEFTUP)
def mouse_right_down(): mouse_event(MOUSEEVENTF_RIGHTDOWN)
def mouse_right_up():   mouse_event(MOUSEEVENTF_RIGHTUP)
def mouse_middle_down(): mouse_event(MOUSEEVENTF_MIDDLEDOWN)
def mouse_middle_up():   mouse_event(MOUSEEVENTF_MIDDLEUP)
```

Код 5.2.3. Функције за симулацију уноса

Овим је решен проблем удаљеног контролисања игре, уз коришћење акумулирања уноса ради ефикаснијег слања и широко подржаног протокола за директну комуникацију. Могуће побољшање решења у будућности представља коришћење протокола *WebTransport* за слање корисничког уноса. За разлику од протокола *WebSocket*, који је грађен на протоколу *TCP*, *WebTransport* је грађен на протоколу *QUIC*, што резултује повећањем брзине преноса и смањењем кашњења у замену за губитак поузданости и редоследа испоруке порука. У тренутку писања овог рада, овај протокол је у развоју и још увек није широко подржан у претраживачима. [36]

6. Ток сесије

У овом поглављу детаљно ћемо приказати начин рада система и интеракције између његових компоненти током корисничке сесије. Најпре ће бити описан начин приступа систему, уз иницијални процес започињања сесије, након чега следи приказ управљања одабраном игром и корисничким датотекама од стране агента. Поглавље завршавамо описом процеса затварања сесије. У оквиру поглавља такође ће бити описан начин коришћења база података *Redis* и *MinIO* за потребе система.

6.1. Успостављање сесије

Корисник приступа систему из веб претраживача путем клијентске стране система, која нуди само једну страницу. Релативна адреса ове странице је /<*korisnik*>/<*igra*>, са улазним параметрима <*korisnik*> и <*igra*> који представљају корисничко име и одабрану игру, редом. По отварању ове странице, контролеру се шаље *HTTP* захтев за отварање сесије са истоветним параметрима.

По пристизању оваквог захтева, контролер генерише јединствени идентификатор сесије и прослеђује захтев агентима у виду *JSON* поруке која се шаље у *Redis* ред порука. Свака структура података у оквиру базе *Redis* идентификује се јединственим кључем. Док за захтеве намењене агентима постоји један заједнички ред, потврду о прихватању сесије од стране агента контролер чека на новом реду чији је кључ једнак идентификатору сесије, чиме се постиже изолованост порука између инстанци контролера и њихово једноставно конзумирање. Ово је изводљиво захваљујући могућности ове базе за динамичко креирање нових структура, као и подршци за велики број конкурентних кључева. По добијању потврде од агента, контролер је прослеђује кориснику.

```
class SessionManager {
    // Kljuc reda poruka sa zahtevima za agente
    static SESSIONS_KEY = 'sessions';

    constructor(endpoint) {
        this.endpoint = endpoint;
        this.redis = new Redis(endpoint);
    }

    // Funkcija za kreiranje sesije
    async createSession(userId, gameId) {
        // Generisanje jedinstvenog identifikatora sesije
        let sessionId = crypto.randomUUID();
        let sessionData = {
            id: sessionId,
            user: userId,
            game: gameId
        };

        // Slanje zahteva za sesijom u Redis red poruka za agente
        await this.redis.lpush(SessionManager.SESSIONS_KEY,
            JSON.stringify(sessionData));

        // Cekanje odgovora jednog od agenata
        // Kljuc Redis reda poruka je identifikator sesije
        let ack = await this.redis.blpop(sessionId, 60);

        // Prekid zahteva u slucaju isteka vremena
        // Ovo je moguce kada ne postoji slobodan agent
        if (!ack) {
            throw new Error("Session creation timed out");
        }

        // Odgovor agenta vraca se korisniku
        ack = JSON.parse(ack[1]);
        return ack;
    }
}
```

Код 6.1.1. Класа контролера за креирање сесија

Сви слободни агенти чекају захтеве контролера на заједничком реду порука. Захтев преузима тачно један агент, који затим шаље потврду о прихватању, такође у виду *JSON* поруке, на одговарајући ред. Ова потврда садржи идентификатор сесије и податке о агенту који ће опслуживати сесију, попут приступних тачака *WebSocket* сервера и сервера за сигнализацију. Подразумевано, животни век *Redis* кључа траје све док придружена структура садржи податке, али се може подесити и експлицитно време истицања, што чинимо за кључ реда који садржи одговор агента. На овај начин спречавамо препуњавање меморије редовима са необрађеним или нагло прекинутим захтевима. Агент затим ишчекује отварање *WebSocket* сесије од стране корисника, како би потврдио присутност и спремност корисника пре посвећивања ресурса преузимању и покретању игре.

```
while True:
    try:
        # Cekanje novog zahteva od kontrolera
        _, session_request = redis_client.brpop("sessions", timeout=0)
        session_data = json.loads(session_request)
        session_data = SessionData(**session_data)

        # Slanje odgovora u novi Redis red
        # Odgovor sadrzi pristupne tacke relevantne za agenta
        redis_client.lpush(f"{session_data.id}", json.dumps({
            "ws_endpoint": config.reported_ws_endpoint,
            "video_signalling_endpoint":
                config.reported_video_signalling_endpoint,
            "audio_signalling_endpoint":
                config.reported_audio_signalling_endpoint,
            "id": session_data.id
        }))
        # Postavljanje vremena isteka Redis kljuca
        redis_client.expire(f"{session_data.id}", 60)

        # Signal za potvrdu uspostavljanja sesije
        agent_state.connection_event.clear()

        # Pokretanje WebSocket servera
        ws_handler = create_ws_handle(config, agent_state, session_data)
        async with serve(ws_handler, config.ws_ip, config.ws_port) as server:
            # Cekanje na uspostavljanje WebSocket sesije od strane korisnika
            # Prekida se u slucaju prekoracenja maksimalnog trajanja
            try:
                await asyncio.wait_for(agent_state.connection_event.wait(),
                timeout=10)
            except asyncio.TimeoutError:
                server.close()

            # Cekanje na zatvaranje WebSocket servera
            await server.wait_closed()

    # U slucaju greske prilikom obrade zahteva, rad se nastavlja posle
    # odredjenog vremena
    # Ovo pokriva scenario gde je Redis baza trenutno nedostupna
    except Exception as e:
        await asyncio.sleep(10)
```

Код 6.1.2. Обрада захтева од стране агента

Корисник отвара *WebSocket* сесију по пристизању одговора од контролера, користећи адресу агента из добијеног одговора, и шаље иницијалну поруку која представља потврду спремности. Такође се врши регистрација на серверу за сигнализацију у улози потрошача на начин описан у четвртом поглављу. Овим је сесија званично успостављена и извршавање игре може да почне.

Све операције чекања порука између компоненти система временски су ограничене. По истеку времена, захтев се прекида и проглашава неуспешним. Поред ситуација где неки

од учесника неочекивано престане са радом, ово се може десити и када нема слободних агената за опслуживање сесије, и тиме спречавамо заглављивање компоненти система у таквим ситуацијама.

6.2. Покретање игре

Како би покренуо игру након успешног успостављања сесије, агент мора обезбедити присутност датотека игре на диску, као и најновију верзију корисничких датотека у случају да је корисник играо игру раније. Ове датотеке преузимају се из складишта података *MinIO*. У оквиру складишта датотеке су логички распоређене у контејнере, који играју улогу логичких дискова у фајл систему складишта. За потребе нашег система постоје два контејнера, који раздвајају игре и корисничке податке. Датотеке сваке од игара компресоване су у једну *ZIP* архиву која носи назив те игре. Корисничке датотеке компресоване су на исти начин, појединачно за сваки пар корисник-игра, и груписане по директоријумима који одговарају називу игре. Приликом приступа датотекама из складишта, задаје се путања жељене датотеке у контејнеру, те би пример захтеване путање за преузимање игре *game* био *games/game.zip*, док би путања за датотеке корисника *user* за ову игру била *saves/game/user.zip*.

Проблем који се затим јавља јесте чињеница да су назив и локација извршне датотеке у оквиру датотека игре, као и локација и формат чувања корисничких датотека на систему, у потпуној контроли креатора игре, и разликују се за сваку игру. Овај проблем ћемо решити придруживањем *JSON* датотеке са метаподацима свакој од игара. Ова датотека налази се на предефинисаној локацији у архиви сваке од игара и садржи локацију извршне датотеке релативно у односу на корени директоријум игре, као и локацију корисничких података и шаблоне које имена релевантних датотека испуњавају. На овај начин добијамо једнообразно решење где се покретање и управљање корисничким датотекама извршава на исти начин за све игре, и једини параметар представљају њени метаподаци. Поступак се сада своди на генеричке операције копирања преузетих корисничких датотека на одговарајуће место на диску, уз обавезно брисање података претходног корисника, и покретања процеса за дату извршну датотеку. На овај начин игра се доводи у стање у којем је игра била на крају последње сесије корисника, или у неиграно стање ако је корисник покреће први пут.

```
class GameManager:
    # Funkcija za pokretanje igre
    @staticmethod
    def start_game(game_root_folder: Path, metadata: GameMetadata) -> Popen:
        # Pokrece se izvrsna datoteka
        exe_path = game_root_folder / metadata.exe_location
        cwd = exe_path.parent
        return Popen([str(exe_path)], cwd=str(cwd))

    # Funkcija za ucitavanje korisnickih podataka
    @staticmethod
    def import_save(source_location: Path, metadata: GameMetadata):
        # Koreni folder korisnickih datoteka
        root = Path(metadata.save_root)

        # Brisanje trenutnih korisnickih datoteka, ukoliko postoje
        for pat in metadata.save_patterns:
            pat_root = root / pat['pattern_root']
            if not pat_root.exists():
                pat_root.mkdir(parents=True, exist_ok=True)
            for f in pat_root.glob(pat['pattern']):
                if f.is_file():
                    f.unlink()

        # Ukoliko korisnik nije igrao igru ranije, igra je vracena u
        # izvorno stanje
        if not source_location:
            return

        # Kopiranje korisnickih podataka na odgovarajucu lokaciju
        for pat in metadata.save_patterns:
            src_pat_root = source_location / pat['pattern_root']
            for f in src_pat_root.glob(pat['pattern']):
                if f.is_file():
                    rel_path = f.relative_to(source_location)
                    dest_path = root / rel_path
                    shutil.copy2(f, dest_path)

    # Funkcija za cuvanje korisnickih podataka
    @staticmethod
    def export_save(metadata: GameMetadata) -> Path:
        # Koreni folder korisnickih datoteka
        root = Path(metadata.save_root)

        # Kompresovanje korisnickih datoteka u arhivu
        # Arhiva se cuva na privremenoj lokaciji pre slanja u MinIO skladiste
        temp_folder = Path(tempfile.mkdtemp())
        zip_path = temp_folder / 'save_export.zip'
        with zipfile.ZipFile(zip_path, 'w', zipfile.ZIP_DEFLATED) as zipf:
            for pat in metadata.save_patterns:
                pat_root = root / pat['pattern_root']
                for f in pat_root.glob(pat['pattern']):
                    if f.is_file():
                        zipf.write(f, arcname=str(f.relative_to(root)))
        return zip_path
```

Код 6.2.1. Класа агента за управљање игром

```
{
    "exe_location": "SH.exe",
    "save_root": "%APPDATA%/../LocalLow/SUPERHOT_Team/SUPERHOT",
    "save_patterns": [
        {
            "pattern_root": ".",
            "pattern": "*.cfg"
        },
        {
            "pattern_root": ".",
            "pattern": "*.hot"
        }
    ]
}
```

Код 6.2.2. Метаподаци игре SUPERHOT

По покретању процеса игре, позива се системски позив за дохватање ручке прозора процеса ради прослеђивања цевоводу за дохватање слике. За дохваћени прозор такође се позива системски позив за довођење у фокус, како би успешно примао симулиране корисничке команде. Затим се покреће извршно окружење *GStreamer* коришћењем цевовода из четвртог поглавља, и након успешне сигнализације започиње се пренос приказа. Такође се започиње пренос и обрада корисничких команди преко *WebSocket* сесије на начин описан у петом поглављу.

6.3. Затварање сесије

Постоје два случаја у којима се сесија завршава на контролисан начин. Први случај представља затварање веб странице од стране корисника. Агент ово детектује када се *WebSocket* сесија са корисником затвори. На овај начин детектује се и сценарио где се веза прекида неочекивано. Други случај представља гашење процеса игре. Ово се може десити тако што корисник бира опцију за излаз у оквиру игре, али и када игра неочекивано престане са радом услед грешке у имплементацији игре. Пошто не постоји решење за асинхроно обавештавање о престанку рада процеса у програмском језику *Python*, неопходно је покренути нову нит чији је задатак периодично проверавање стања процеса игре. Уколико процес није активан, ова нит шаље сигнал главном контексту сесије за њено прекидање.

У оба случаја, агент најпре зауставља игру и извршно окружење библиотеке *GStreamer*, уколико су још увек у току. Након потврде заустављања, кориснички фајлови се архивирају у *ZIP* датотеку и шаљу у складиште објеката. За контејнер за складиштење корисничких података укључена је опција за верзионисање, што подразумева чување старих верзија датотека, уз могућност ограничења броја сачуваних верзија како би се избегло

прекомерно заузимање простора. У случају корупције корисничких података због неисправног завршавања игре, ова опција омогућава повраћај на старију верзију. На крају, треба поништити сав кориснички унос, за случај када неко од дугмади миша или тастатуре остане притиснуто у тренутку затварања сесије. Ово се врши једноставном обрадом поруке корисничког уноса са свим нулама. Након обављања ових радњи, агент је спреман за опслуживање следеће сесије и враћа се на чекање на реду порука. Игра остаје кеширана на диску агента, како би се избегло поновно преузимање приликом следећег захтева за њено играње.

```
# Telo niti za pracenje stanja procesa igre i GStreamer okruzenja
# Nit se prekida kada jedan od ovih procesa prestane sa radom
async def monitor_game_process(agent_state: AgentState):
    def is_alive(proc):
        try:
            return proc and psutil.Process(proc.pid).is_running()
        except (psutil.NoSuchProcess, psutil.AccessDenied):
            return False

    while True:
        async with agent_state.lock:
            if not all(is_alive(p) for p in [agent_state.game_proccess,
                agent_state.video_streaming_process,
                agent_state.audio_streaming_process]):
                break
        await asyncio.sleep(1.0)

# Telo niti za obradu i izvrsavanje korisnickog unosa
# Nit se prekida po zatvaranju WebSocket sesije
async def handle_messages():
    async for msg in ws:
        if isinstance(msg, bytes):
            handle_packet(msg)

tasks = [asyncio.create_task(handle_messages()),
         asyncio.create_task(monitor_game_process(agent_state))]

# Sesija traje do zavrsetka jedne od ovih niti
await asyncio.wait(tasks, return_when=asyncio.FIRST_COMPLETED)

# Zatvaranje sesije (gasenje procesa i GStreamer okruzenja,
# cuvanje korisnickih podataka, ciscenje unosa)
await finalize_session()
ws.server.close()
for task in tasks:
    if task and not task.done():
        task.cancel()
        try:
            await task
        except:
            pass
```

Код 6.3.1. Главни ток и затварање сесије агента

Оваква имплементација система покрива све функционалне захтеве специфициране у другом поглављу. Како је у питању минимална функционална имплементација, у оквиру основне функционалности постоји простор за унапређења у области поузданости, додавањем логике поновног покушавања конекције у случајевима прекида и провером интегритета датотека преузетих из складишта, као и у области безбедности, шифровањем токова комуникације и спречавањем истовремених сесија за једног корисника.

7. ЕВАЛУАЦИЈА

У овом поглављу описаћемо методе и резултате евалуације функционалности и перформанси система. Најпре ће бити описано окружење у којем је вршена евалуација. Затим ћемо описати процес ручног тестирања функционалности система, након чега ће бити приказане метрике од значаја измерене током тестирања. На крају ћемо, на основу добијених резултата, донети закључак о употребљивости нашег решења.

7.1. Опис окружења

Све компоненте серверске стране система покренуте су на истом личном рачунару, који садржи процесор *AMD Ryzen 7 6800H*, графичку картицу *NVIDIA RTX 3060*, и 16*GB* радне меморије. Рачунар је мрежним каблом прикључен на мрежу чија је брзина преузимања 250 *mbit/s*, док је брзина отпремања 25 *mbit/s*. Поред агента покренутог директно на рачунару, за потребе тестирања опслуживања конкурентних захтева покренута је и виртуелна машина са још једним агентом. Виртуелна машина је изузетно скромних карактеристика, те није меродавна за евалуацију перформанси и неће се разматрати у ту сврху.

Сва тестирања описана у наставку вршена су за две различите корисничке машине. Први корисник представља лични рачунар прикључен на исту мрежу као серверска машина, путем бежичне везе која развија брзину преузимања од 55 *mbit/s* и брзину отпремања од 25 *mbit/s*. Други корисник представља лични рачунар који приступа систему из одвојене мреже и удаљен је 2.5*km* од сервера. Ова машина прикључена је мрежним каблом на мрежу брзине 100 *mbit/s* у оба смера.

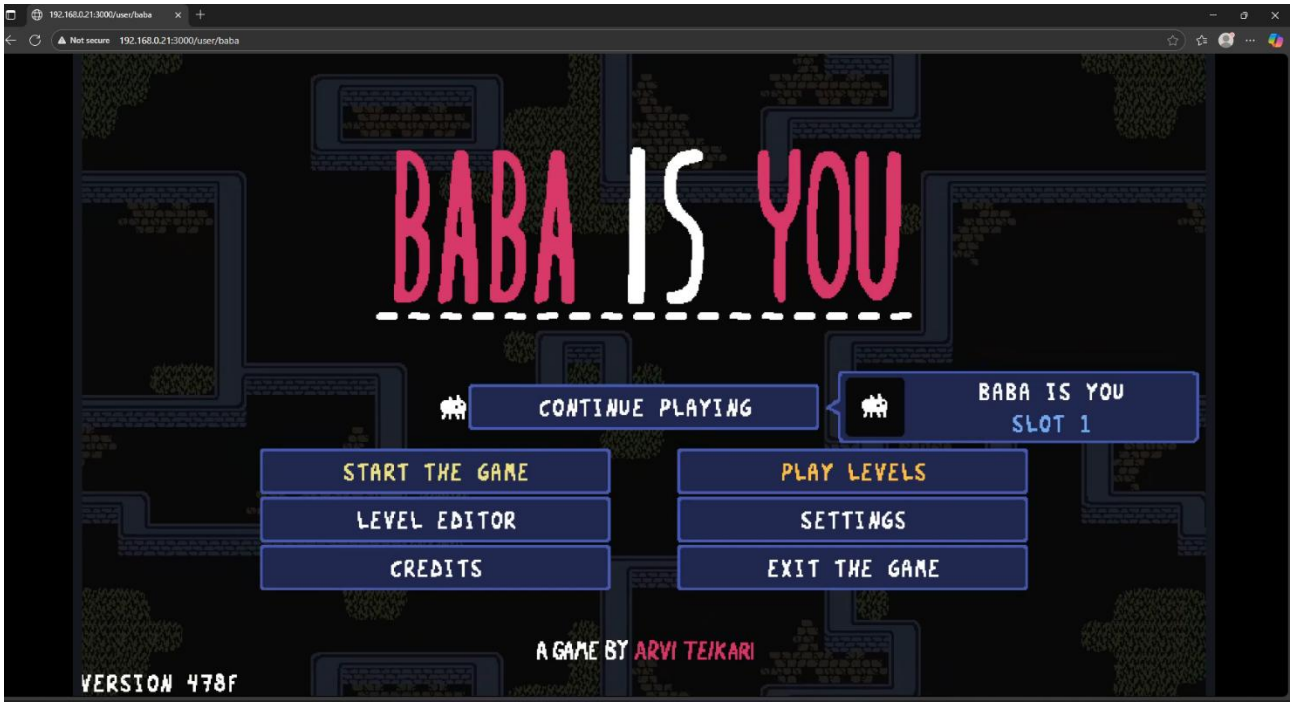
7.2. Опис тестирања функционалности

Валидација функционалности система на основу функционалних захтева обављена је ручним тестирањем. Корисник започиње тестирање приступом систему путем веб претраживача за произвољно корисничко име и игру. По отварању странице, потврђено је успешно успостављање играчке сесије и репродукција слике и звука у оквиру веб странице у реалном времену, као и извршавање корисничког уноса са миша и тастатуре у оквиру игре на

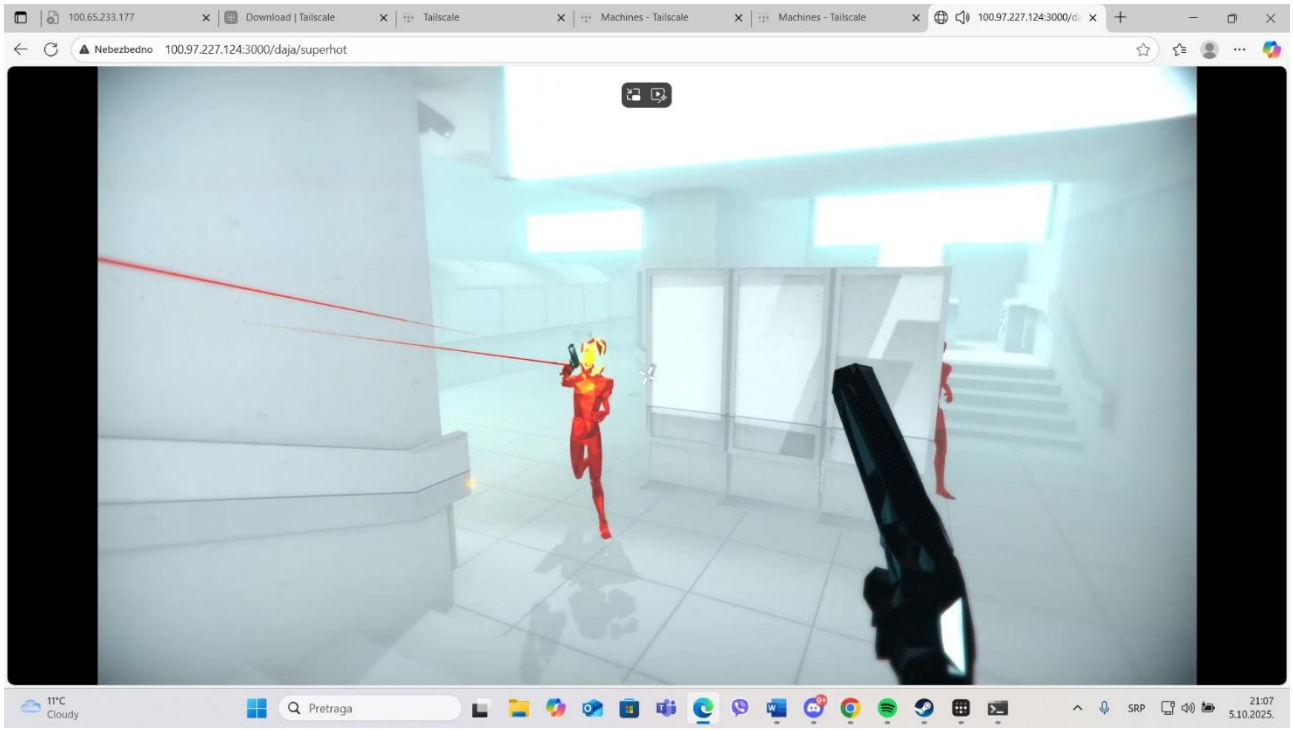
пренетом приказу. Корисник затим наставља играње игре у минималном трајању од 20 минута, чиме је потврђена континуална одрживост сесије у стабилним условима. На крају, затварањем прозора претраживача или одабиром опције за излаз у оквиру саме игре потврђује се контролисано завршавање сесије од стране система.

Поред основне функционалности играња игре, отварањем више узастопних сесија за различита корисничка имена верификована је функционалност чувања корисничких података, као и покретање игре у изворном стању приликом првог играња игре за новог корисника. На крају, успешним извршавањем две паралелне сесије из два прозора претраживача валидирана је конкурентност система и могућност потпуно изолованог опслуживања више од једне сесије.

Наведена тестирања успешно су обављена за обе игре у понуди система, као и за веб претраживаче *Edge* и *Chrome*. На овај начин потврђено је да систем испуњава све функционалне захтеве.



Слика 7.2.1. Приказ игре Baba is You у оквиру претраживача



Слика 7.2.2. Приказ игре SUPERHOT у оквиру претраживача

7.3. Мерење перформанси система

Метрике перформанси система које су најутицајније за корисничко искуство јесу кашњење и време започињања сесије, те ћемо њих разматрати. Метрике су прикупљене током функционалног тестирања система.

7.3.1. Одзив система

Због интерактивне природе система, метрику кашњења изразићемо у виду одзива. Одзив представља протекло време између уноса команде корисника и њеног ефективног извршавања у оквиру приказа игре на корисниковом екрану.

Унету команду неопходно је пренети до агента и извршити њену симулацију. Приказ игре измењен командом затим треба провести кроз *GStreamer* цевовод, пренети до корисника и приказати у оквиру видео елемента веб странице. Иако не можемо приступити извршној логици игара за потребе прецизног одређивања одзива због њихове затворене природе из угла система, можемо измерити време трајања сваког од појединачних корака преноса и обраде, те ћемо њиховим сабирањем апроксимирати просечан одзив система. Време преноса и извршавања корисничког уноса рачунамо коришћењем временског тренутка послатог у оквиру поруке са уносом. За одређивање времена обраде у оквиру *GStreamer* цевовода сама

библиотека нуди могућност периодичног исписа ове вредности у датотеку, док време преноса у оквиру *WebRTC* сесије можемо сазнати коришћењем уграђене *JavaScript* функције за штампање статистика о перформансама током трајања сесије. Збиру просечних вредности трајања ових корака на крају треба додати половину периода функције за слање акумулираног корисничког уноса, како бисмо урачунали просечно одлагање преноса уведено таквим приступом. Ова вредност је у нашем случају 8.33*ms*.

Табела 7.3.1.1. Просечан одзив измерен током тестирања

НАЗИВ ИГРЕ	КОРИСНИК	ПРЕНОС УНОСА	ОБРАДА УНОСА	РАД ЦЕВОВОДА СЛИКЕ	ПРЕНОС И ПРИКАЗ СЛИКЕ	ОДЛАГАЊЕ УНОСА	УКУПНО
Baba is You	Локални	3.64*ms*	0.34*ms*	8.77*ms*	1.70*ms*	8.33*ms*	22.78*ms*
Baba is You	Удаљени	11.53*ms*	0.37*ms*	8.02*ms*	20.32*ms*	8.33*ms*	48.57*ms*
SUPERHOT	Локални	3.37*ms*	0.40*ms*	5.35*ms*	2.76*ms*	8.33*ms*	20.21*ms*
SUPERHOT	Удаљени	15.72*ms*	0.42*ms*	4.92*ms*	9.98*ms*	8.33*ms*	39.37*ms*

Добијене вредности указују да је аспект серверске обраде слике и уноса стабилног временског трајања и оптимизован за ефикасан рад, те да кашњењу највише доприноси пренос података преко мреже. Остварене перформансе омогућавају прихватљиво корисничко искуство за обе игре, чиме смо потврдили изводљивост играња једноставних потезних игара, као и акционих игара, у оквиру система.

7.3.2. Време започињања сесије

Ова метрика одређује време протекло од приступа сајту до почетка приказа игре. У стабилним мрежним условима време започињања сесије првенствено зависи од тога да ли је игра кеширана на додељеном агенту, као и од укупне величине датотека игре и њене захтевности. Величина корисничких датотека је реда килобајта, те је трајање операција са њима занемарљиво. Резултати су приказани за обе игре, уз величине њихових датотека и времена трајања покретања за случајеве када су први пут покренуте на агенту и када су кеширане.

Табела 7.3.2.1. Време започињања сесије измерено током тестирања

НАЗИВ ИГРЕ	ВЕЛИЧИНА ДАТОТЕКА	КЕШИРАНА НА АГЕНТУ	ВРЕМЕ ЗАПОЧИЊАЊА СЕСИЈЕ
Baba is You	105*MB*	Да	2.1*s*
Baba is You	105*MB*	Не	7.5*s*
SUPERHOT	4034*MB*	Да	1.9*s*
SUPERHOT	4034*MB*	Не	32.9*s*

Иако су добијена времена чекања задовољавајућа из угла корисничког искуства, примећује се значајан пораст измерених вредности у случајевима када се игра мора преузети из складишта, сразмерно величини датотека игре. Један од начина побољшања просечног времена започињања јесте давање приоритета агентима са кешираном траженом игром приликом отварања сесије. Са друге стране, датотеке игара нису подложне честим операцијама уписа, и целокупан асортиман у понуди једног мањег сервиса овакве врсте заузимао би простор реда величине десетине терабајта, те је репликација целог складишта са играма у сваком центру података ради брзог приступа још једно од изводљивих побољшања у реалном скалираном систему.

На основу резултата тестирања и измерених метрика, закључујемо да систем испуњава захтеване функционалности за играње игара у облаку из веб претраживача, уз задовољавајуће перформансе које омогућавају његову практичну употребу. Стога је циљ овог рада успешно остварен.

8. Закључак

У овом раду описан је поступак имплементације једноставног дистрибуираног система који омогућава играње игара у облаку из веб претраживача. Разматрани су најзначајнији постојећи примери оваквих система, на основу којих су дефинисани минимални функционални захтеви које систем мора да испуњава. Успостављена је архитектура система прилагођена испуњењу ових захтева. За потребе решавања проблема обраде и преноса слике и звука коришћене су познате технологије из домена мултимедијалне обраде, имплементиране у оквиру библиотеке *GStreamer*, и описана је њихова оптимизација за рад у реалном времену, као и значај и начин рада протокола *WebRTC* за ефикасан *peer-to-peer* пренос. Објашњено је решење проблема преноса и симулације корисничког уноса коришћењем релевантних позива претраживача и оперативног система агента, као и примена периодичног слања уноса ради избегавања загушења. Описан је целокупан ток рада корисничке сесије и начин коришћења база података *Redis* и *MinIO* за испуњење захтева конкурентности и чувања корисничких података. На крају, на основу евалуације функционалности и перформанси система донет је закључак да је коришћењем примењених техника могуће реализовати практично употребљив систем, и дати су предлози за даља унапређења.

Добијени прототип може се искористити као основа за проширење у професионалан сервис, уз додавање недостајућих функционалности и повећање капацитета. Развој комерцијалних сервиса овог типа изискује изузетну количину хардверских ресурса, како би се обезбедила могућност играња савремених игара уз прихватљив пренос приказа, као и географска распрострањеност центара података услед осетљивости кашњења на удаљеност сервера од корисника. Са друге стране, прототип се може употребити и у личне сврхе, у улози приватног сервиса покренутог на кућном серверу, што постаје све изводљивија и неретко виђена пракса међу технички напредним корисницима, захваљујући напретку потрошачког хардвера и интернет конекција.

Литература

[1] Arto Ojala, Pasi Tyrväinen, „Developing Cloud Business Models: A Case Study on Cloud Gaming“, *IEEE Software (Volume: 28, Issue: 4, July-Aug. 2011)*

[2] JP Mangalindan, *Cloud gaming's history of false starts and promising reboots*, https://www.polygon.com/features/2020/10/15/21499273/cloud-gaming-history-onlive-stadia-google, 14.09.2025.

[3] Cloud Gaming, Broadmedia Corporation, https://www.broadmedia.co.jp/en/technology/cloudgame, 14.09.2025.

[4] OnLive, Logopedia, https://logos.fandom.com/wiki/OnLive, 14.09.2025.

[5] Gaikai, Logopedia, https://logos.fandom.com/wiki/Gaikai, 14.09.2025.

[6] Frequently Asked Questions for GeForce NOW, NVIDIA, https://www.nvidia.com/en-us/geforce-now/faq, 18.09.2025

[7] Details about Geforce Now infrastructure - NVIDIA Developer Forums, https://forums.developer.nvidia.com/t/details-about-geforce-now-infrastructure/237485, 18.09.2025.

[8] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, Danilo Giordano, „A network analysis on cloud gaming: Stadia, GeForce Now and PSNow“, *Network 2021, 1, 247-260*

[9] Richard Leadbetter, *Sony creates custom PS3 hardware for PlayStation Now*, https://www.eurogamer.net/digitalfoundry-2014-sony-creates-custom-ps3-for-playstation-now, 19.09.2025.

[10] Sherif Saed, *Xbox Series X hardware will power xCloud servers next year – report*, https://www.vg247.com/xbox-series-x-in-xcloud-servers-by-2021-report, 19.09.2025.

[11] Michael Larabel, *Stadia Is Google's Cloud Gaming Service Using Linux, Vulkan & A Custom AMD GPU*, https://www.phoronix.com/news/Google-Stadia-Vulkan-Linux, 20.09.2025.

[12] NVIDIA GeForce NOW, Seeklogo, https://seeklogo.com/vector-logo/457169/nvidia-geforce-now, 21.09.2025.

[13] Playstation Now, Seeklogo, https://seeklogo.com/vector-logo/459914/playstation-now, 21.09.2025.

[14] Stadia, https://stadia.google.com, 21.09.2025.

[15] Baba is You, Steam, https://store.steampowered.com/app/736260/Baba_Is_You/, 15.09.2025.

[16] SUPERHOT, Steam, https://store.steampowered.com/app/322500/SUPERHOT/, 15.09.2025.

[17] Express - Node.js web application framework, https://expressjs.com/, 22.09.2025.

[18] The WebSocket API (WebSockets), MDN, https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API, 22.09.2025.

[19] WebRTC, https://webrtc.org/, 22.09.2025.

[20] Redis – The Real-time Data Platform, https://redis.io/, 22.09.2025.

[21] Welcome to Python.org, https://www.python.org/, 22.09.2025.

[22] GStreamer: open source multimedia framework, https://gstreamer.freedesktop.org/, 22.09.2025.

[23] MinIO: S3 Compatible, Exascale Object Store for AI, https://www.min.io/, 22.09.2025.

[24] Application Development Manual, GStreamer, https://gstreamer.freedesktop.org/documentation/application-development/index.html, 23.09.2025.

[25] Sean DuBois, Claes Mogren, Alex Zhukov, WebRTC for the Curious, CC0, 2020.

[26] WebRTC (Web Real-Time Communications), TechTarget, https://www.techtarget.com/searchunifiedcommunications/definition/WebRTC-Web-Real-Time-Communications, 23.09.2025.

[27] Desktop Duplication API - Win32 apps, Microsoft Learn, https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/desktop-dup-api, 23.09.2025.

[28] NVIDA Capture SDK, NVIDIA Developer, https://developer.nvidia.com/capture-sdk, 23.09.2025.

[29] Codecs used by WebRTC, MDN, https://developer.mozilla.org/en-US/docs/Web/Media/Guides/Formats/WebRTC_codecs, 25.09.2025.

[30] NVIDIA Video Codec SDK, NVIDIA Developer, https://developer.nvidia.com/video-codec-sdk, 25.09.2025.

[31] Understanding Rate Control Modes (x264, x265, vpx), https://slhck.info/video/2017/03/01/rate-control.html, 26.09.2025.

[32] About WASAPI, Microsoft Learn, https://learn.microsoft.com/en-us/windows/win32/coreaudio/wasapi, 28.09.2025.

[33] NetEq, Chromium Google Source, https://chromium.googlesource.com/external/webrtc/+/master/modules/audio_coding/neteq/g3doc/index.md, 30.09.2025.

[34] Web APIs, MDN, https://developer.mozilla.org/en-US/docs/Web/API, 01.10.2025.

[35] SendInput function (winuser.h) - Win32 apps, Microsoft Learn, https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendinput, 02.10.2025.

[36] WebTransport, MDN, https://developer.mozilla.org/en-US/docs/Web/API/WebTransport, 02.10.2025.

СПИСАК СКРАЋЕНИЦА

AVC - *Advanced Video Coding*

ICE - *Interactive Connectivity Establishment*

NAT - *Network Address Translation*

QP - *Quantization Parameter*

QUIC - *Quick UDP Internet Connections*

RTCP - *RTP Control Protocol*

RTP - *Real-time Transport Protocol*

SDP - *Session Description Protocol*

STUN - *Session Traversal Utilities for NAT*

TCP - *Transmission Control Protocol*

TURN - *Traversal Using Relays around NAT*

UDP - *User Datagram Protocol*

VBV - *Video Buffering Verifier*

WASAPI - *Windows Audio Session API*

Списак слика и кодова

Списак табела