|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  **У НОВОМ САДУ** |  |
| --- | --- | --- |

Огњен Шврака

**Развој апликације за групни позив примјеном WebRTC технологије**

Дипломски рад

- Основне академске студије -

Нови Сад, 2024

|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ ⚫ **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  21000 НОВИ САД, Трг Доситеја Обрадовића 6 | Број: |
|  |
| **ЗАДАТАК ЗА ЗАВРШНИ РАД** | Датум: |
|  |

*(Податке уноси предметни наставник - ментор)*

|  |  |  |  |
| --- | --- | --- | --- |
| Студијски програм: | Рачунарство и аутоматика | | |
| Студент: | Огњен Шврака | Број индекса: | RA 181/2019 |
| Степен и врста студија: | ОСНОВНЕ АКАДЕМСКЕ СТУДИЈЕ | | |
| Област: | Електротехничко и рачунарско инжењерство | | |
| Ментор: | др Милан Стојков, доцент | | |
| НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:   * проблем – тема рада; * начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; | | | |

**НАСЛОВ ЗАВРШНОГ РАДА:**

|  |
| --- |
| Развој апликације за групни позив примјеном WebRTC технологије |

**ТЕКСТ ЗАДАТКА:**

|  |
| --- |
| Упознати се са WebRTC технологијом са посебним освртом на апликације за видео позиве. Специфицирати и имплементирати прототип апликације за видео позиве. Документовати решење. |

|  |  |
| --- | --- |
| Руководилац студијског програма: | Ментор рада: |
|  |  |

|  |
| --- |
| Примерак за:  - Студента;  - Ментора |

# КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

| Редни број, **РБР**: |  |
| --- | --- |
| Идентификациони број, **ИБР**: |  |
| Тип документације, **ТД**: | монографска публикација |
| Тип записа, **ТЗ**: | текстуални штампани документ |
| Врста рада, **ВР**: | дипломски рад |
| Аутор, **АУ**: | Огњен Шврака |
| Ментор, **МН**: | др Милан Стојков, доцент |
| Наслов рада, **НР**: | Развој апликације за групни позив примјеном WebRTC технологије |
| Језик публикације, **ЈП**: | српски |
| Језик извода, **ЈИ**: | српски / енглески |
| Земља публиковања, **ЗП**: | Србија |
| Уже географско подручје, **УГП**: | Војводина |
| Година, **ГО**: | 2024 |
| Издавач, **ИЗ**: | ауторски репринт |
| Место и адреса, **МА**: | Нови Сад, Факултет техничких наука, Трг Доситеја Обрадовића 6 |
| Физички опис рада, **ФО**: | бр. Поглавља 7 / страница 63 / цитата / табела / слика 26 / графикона / прилога / листинга 19 |
| Научна област, **НО**: | Електротехничко и рачунарско инжењерство |
| Научна дисциплина, **НД**: | Софтверско инжењерство |
| Предметна одредница /  кључне речи, **ПО**: | WebRTC, peer-to-peer, видео репродукција у реалном времену |
| **УДК** |  |
| Чува се, **ЧУ**: | Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад |
| Важна напомена, **ВН**: |  |
| Извод, **ИЗ**: | Рад описује функционисање WebRTC технологије кроз коришћење ICE и STUN протокола, као и улогу TURN сервера. Приказана је архитектура апликације за видео позиве и објашњени су слојеви у клијентском и серверском делу апликације са посебним фокусом на коришћење WebRTC-а |
| Датум прихватања теме, **ДП**: |  |
| Датум одбране, **ДО**: |  |
| Чланови комисије, **КО**: |  |
| председник | др Горан Сладић, ред. проф, ФТН Нови Сад |
| члан | др Мирослав Зарић, ванред. проф. ФТН Нови Сад |
| ментор | др Милан Стојков, доцент, ФТН Нови Сад |
| Потпис ментора | |

# KEY WORDS DOCUMENTATION

| Accession number, **ANO**: |  |
| --- | --- |
| Identification number, **INO**: |  |
| Document type, **DT**: | monographic publication |
| Type of record, **TR**: | textual material |
| Contents code, **CC**: | bachelor thesis |
| Author, **AU**: | Ognjen Švraka |
| Mentor, **MN**: | Milan Stojkov, assistant professor, PhD |
| Title, **TI**: | Development of application for group call using WebRTC technologies |
| Language of text, **LT**: | Serbian |
| Language of abstract, **LA**: | Serbian / English |
| Country of publication, **CP**: | Serbia |
| Locality of publication, **LP**: | Vojvodina |
| Publication year, **PY**: | 2024 |
| Publisher, **PB**: | author’s reprint |
| Publication place, **PP**: | Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6 |
| Physical description, **PD**: |  |
| Scientific field, **SF**: | Electrical and Computer Engineering |
| Scientific discipline, **SD**: | Software Engineering |
| Subject / Keywords, **S/KW**: | WebRTC, peer-to-peer, realtime video streaming |
| **UDC** | No. of chapters 7 / pages 63 / quotes / tables / pictures 26 / graphs / appendix / listings 19 |
| Holding data, **HD**: | Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad |
| Note, **N**: |  |
| Abstract, **AB**: | The paper describes the functioning of WebRTC technology through the use of ICE and STUN protocols, as well as the role of the TURN server. The architecture of the video calling application is presented and the layers in the client and server part of the application are explained with a special focus on the use of WebRTC. |
| Accepted by sci. Board on, **ASB**: |  |
| Defended on, **DE**: |  |
| Defense board, **DB**: |  |
| president | Goran Sladić, PhD, full prof. FTN Novi Sad |
| member | Miroslav Zarić, PhD, adj. prof. FTN Novi Sad |
| mentor | Milan Stojkov, PhD, assist. prof. FTN Novi Sad |
| Mentor's signature | |

[КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА 5](#_Toc176534671)

[KEY WORDS DOCUMENTATION 7](#_Toc176534672)

[1 Увод 11](#_Toc176534673)

[2 WebRTC 13](#_Toc176534674)

[2.1 getUserMedia интерфејс 14](#_Toc176534675)

[2.2 getDisplayMedia интерфејс 15](#_Toc176534676)

[2.3 RTCDataChannel интерфејс 16](#_Toc176534677)

[2.4 RTCPeerConnection интерфејс 17](#_Toc176534678)

[2.5 Peer-to-peer 18](#_Toc176534679)

[2.6 *Signaling* процес 20](#_Toc176534680)

[2.7 WebRTC протоколи 20](#_Toc176534681)

[2.7.1 Interactive Connectivity Establishment 21](#_Toc176534682)

[2.7.2 Session Traversal Utilities for NAT 22](#_Toc176534683)

[2.7.3 Traversal Using Relays around NAT 23](#_Toc176534684)

[2.7.4 Session Description Protocol 24](#_Toc176534685)

[3 Апликација за видео позиве 27](#_Toc176534686)

[3.1 Дијаграм архитектуре 27](#_Toc176534687)

[3.2 Дијаграм случајева коришћења 28](#_Toc176534688)

[3.3 Дијаграм архитектуре клијентског дијела апликације 29](#_Toc176534689)

[4 Имплементација 31](#_Toc176534690)

[4.1 Серверски дио апликације 31](#_Toc176534691)

[4.2 Клијентски дио апликације 38](#_Toc176534692)

[4.2.1 *Redux Store* 38](#_Toc176534693)

[4.2.2 *MediaAccess Store* 40](#_Toc176534694)

[4.2.3 *WebSocket Store* 43](#_Toc176534695)

[4.2.4 *Chat Store* 44](#_Toc176534696)

[4.2.5 *RtcLogic Store* 45](#_Toc176534697)

[4.3 TURN сервер 50](#_Toc176534698)

[4.4 Финална апликација 52](#_Toc176534699)

[5 Закључак 57](#_Toc176534700)

[6 Литература 59](#_Toc176534701)

[7 Биографија 63](#_Toc176534702)

# 1 Увод

Од најранијих година рачунарства постојала је идеја о комуникацији између рачунара. Ако се узме у обзир да огроман број људи свакодневно користи друштвене мреже и комуницира са осталим људима, било путем позива, видео позива или дописивања, може се рећи да је комуникација између рачунара кључна област у свијету програмирања. У последњих двадесет година, појавило се мноштво апликација које су омогућавале групне видео позиве, као што су *Skype, Zoom, Google meet* итд. Највећи тест за све њих је сигурно било вријеме пандемије COVID-19, када су се читав образовни сектор и сви послови пребацили на комуникацију путем интернета.

Овакве апликације могу бити врло захтјевне, у зависности од тога шта се жели постићи, колико много људи може бити у позиву, да ли се жели опција дијељења екрана, квалитет снимка итд. Као оптимално рјешење за већину тих проблема може се издвојити *peer-to-peer* мрежа, тј. директна комуникација између два рачунара, без сервера посредника. Међутим *peer-to-peer* комуникација уводи додатне компликације. Као рјешење за те компликације, појављује се технологија WebRTC која знатно олакшава прављење *peer-to-peer* апликација и програмеру даје могућност да се *peer-to-peer-*омбави на већем степену апстракције. Овај рад се бави WebRTC технологијом, начином на који се она користи и протоколима које он користи, што је објашњено кроз апликацију за групне позиве. Апликација садржи функционалности видео позива, дијељења екрана, дописивања и по потреби гашења микрофона или камере. Уз помоћ WebRTC технологије, корисници апликације комуницирају директним путем, тј. *peer-to-peer* уколико је то могуће, док се у супротном између корисника користи сервер посредник.

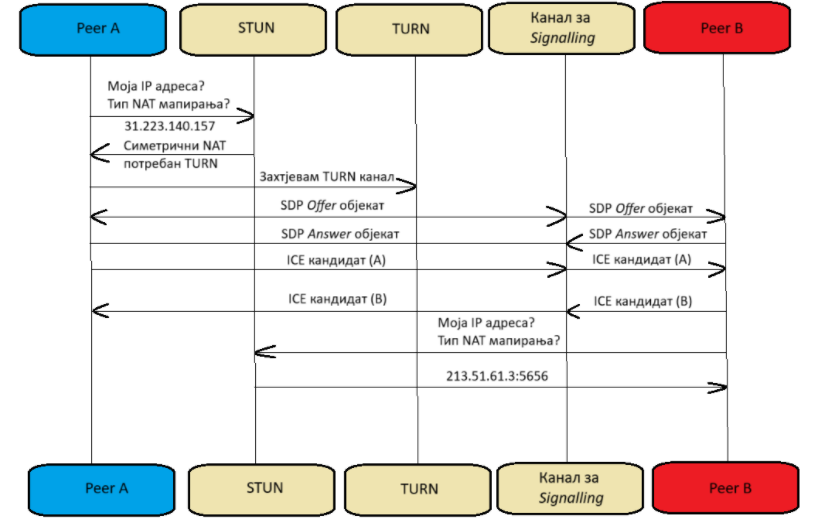
Рад је конципиран тако да се у другом поглављу бави теоријским основама за развијање WebRTC апликације. У то спадају основе о *peer-to-peer* начину комуникације*,* WebRTC технологији и протоколима које користи. Након тога фокус је пребачен на саму апликацију. У трећем поглављу, кроз модел се може стећи шира слика како изгледа инфраструктура развијане апликације и које функционалности она има. Након тога у четвртом поглављу су приказани и објашњени значајнији дијелови кода на клијентском и серверском дијелу као и презентација изгледа финалне апликације. У последњем, петом поглављу, сумирани су доприноси рада и наведени даљи правци развоја.

# 2 WebRTC

*Web Real-Time Communication* (WebRTC) је технологија која је трансформисала начин на који веб претраживачи комуницирају. Представља пројекат отвореног кода који обједињује више различитих протокола и технологија, те у свим модерним претраживачима нуди интерфејсе (енгл. Application Programming Interface - API) за *peer-to-peer* комуникацију у реалном времену [31]. Омогућава директну равноправну комуникацију без потребе за посредницима или додатним софтвером, чиме се повећава приватност и смањује кашњење. Поред тога, WebRTC омогућава различитим индустријама, укључујући здравство, образовање и забаву, да интегришу висококвалитетну аудио и видео комуникацију директно у своје апликације. Обзиром на његову широку употребу и подршку, WebRTC постаје кључна технологија у дигиталном добу.

Данас се многе апликације ослањају на WebRTC. Неке од њих су *Google Meet, Facebook Messenger, Discord* и *Microsoft Teams*. Међутим, постоје и апликације које не служе за видео позиве, а успјеле су да искористе моћ WebRTC-а, на пример, игрица *Haxball* [1,2].

На слици 2.1 је приказан основни ток WebRTC апликације. На крајевима се налазе клијенти и редослијед операција до успостављања конекције. Сваки *peer* добавља информацију од STUN сервера о томе путем које јавне IP (енгл. *Internet Protocol*) адресе се њему може приступити и који тип рутера је у питању. За *peer* A на слици се може видјети да се налази иза рутера са симетричним NAT мапирањем. Због тога мора да се обрати и TURN серверу да би добио нови комуникациони канал преко ког ће му се моћи приступити. Даље, преко серверске стране коју власник система мора да омогући (на слици *Signal Channel*), *peer-*овишаљу SDP и ICE објекте. Све то ће бити детаљније објашњено у наредним поглављима.



Слика 2.1 основни ток WebRTC апликације

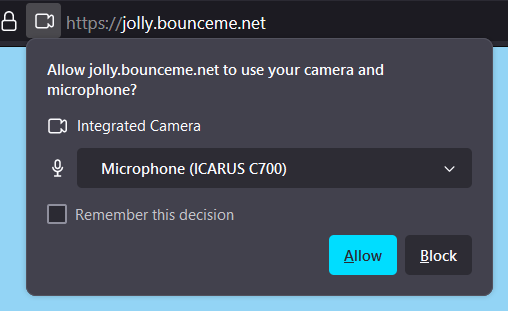
Постоје четири интерфејса (енгл. Application Programming Interface - API) која чине WebRTC и омогућавају комуникацију у реалном времену:

1. *getUserMedia*
2. *getDisplayMedia*
3. *RTCDataChannel*
4. *RTCPeerConnection*

## 2.1 getUserMedia интерфејс

У веб програмирању, појављивала се потреба за све чешћим приступом камери и микрофону кроз апликацију и постојало је пар рјешења за тај проблем. Једно од њих је и APIgetUserMedia. getUserMediaпредставља методу интерфејса MediaDevices, која кориснику даје избор да дозволи веб претраживачу да приступи уређајима као што су камера и микрофон [5,6].

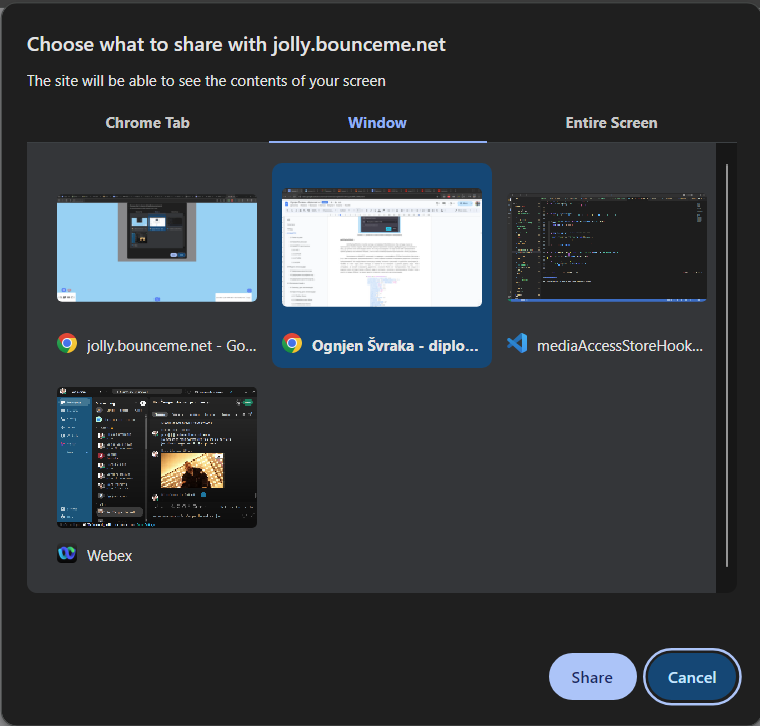
getUserMediaметода прима параметре на основу којих се одређује да ли се приступа само камери или само микрофону, или обоје. На слици 2.2 је приказано како веб претраживач тражи дозволу од корисника за приступ уређајима.



Слика 2.2 дозвола за приступање камери и микрофону

## 2.2 getDisplayMedia интерфејс

getDisplayMediaје такође метода интерфејса MediaDevices. Ова метода служи за приступање снимку екрана. Она такође прима параметре којима се одређује дозвола претраживачу за приступ звуку или видео запису [7]. На слици 2.3 се види на који начин веб претраживач тражи дозволу за снимање екрана, и исто тако за избор екрана који корисник жели да дијели.



Слика 2.3 избор екрана за дијељење

## 2.3 RTCDataChannel интерфејс

RTCDataChannelје задужен за размјењивање података у реалном времену који нису аудиовизуелни. Што значи да се преко њега шаљу текстуалне поруке и фајлови. Постоје многи начини на које се може постићи иста ствар, као што су *WebSocket-*и, али RTCDataChannelнам даје могућност да искористимо моћ *peer-to-peer* мреже и смањимо кашњење [9]. Креира се једноставно помоћу методе createDataChannel*,* којој се треба прослиједити име лабеле канала. Главне двије методе су sendза слање поруке и onmessageза пријем поруке [8]. На листингу 2.4 је приказано креирање канала, слање и пријем поруке.

1 const pc = new RTCPeerConnection();

2 const dc = pc.createDataChannel('my channel');

3

4 dc.send('Hello world');

5

6 dc.onmessage = (event) => {

7 console.log(`received: ${event.data}`);

8 };

Листинг 2.4 креирање канала за размјену порука*,* слање и пријем поруке

## 2.4 RTCPeerConnection интерфејс

Све везано за WebRTCконекцију је садржано у интерфејсу RTCPeerConnection. Он представља конекцију између локалног рачунара и удаљеног рачунара [3]. Такође он нам нуди разне методе, о којима ће се говорити у даљем дијелу рада. Такав интерфејс је могуће направити директно у конзоли било ког претраживача. На слици 2.5 видимо како изгледа празан објекат када се испише у конзолу у претраживачу. Клијент мора имати по један објекат за сваког клијента с којим жели да комуницира. Поља која се могу издвојити као битнија су сљедећа:

* localDescription, remoteDescription – овдје објекат чува вриједности Session Description Protocol (SDP) објеката
* connectionState – чува информацију о томе у ком стању је тренутна комуникација са тим *peer*-ом
* onicecandidate – чува функцију у којој је описано шта објекат да ради када изгенерише Interactive Connectivity Establishment (ICE) кандидата
* ondatachannel – чува функцију у којој је описано шта објекат да ради када други *peer* креира комуникациони канал; у тој функцији се дефинише на који начин се прима порука



Слика 2.5 примјер RTCPeerConnectionобјекта

## 2.5 Peer-to-peer

Комуникација код WebRTC апликација није класична клијент/сервер, већ се обавља између равноправних рачунара (енгл. *peer-to-peer*). *Peer-to-peer* представља архитектуру дистрибуираних система гдје не постоји сервер који је задужен за процесирање података, већ је све подијељено између равноправних уређаја (енгл. *peer*). Сваки *peer* је једнак у смислу да не постоји неки од њих који има већу привилегију да добије податке. Сви они заједно повезани чине *peer-to-peer* мрежу. Битно је истаћи да за разлику од клијент/сервер архитектуре, сви учесници у комуникацији могу да буду и корисници и произвођачи података. *Peer-to-peer* архитектура може да има примјену у разним типовима апликација. Неки од њих су видео репродукција и међусобно дијељење фајлова. *Peer* такође може да има улогу обучавања неуронских мрежа у дистрибуираним системима. Прва популарна *peer-to-peer* апликација је *Napster*.

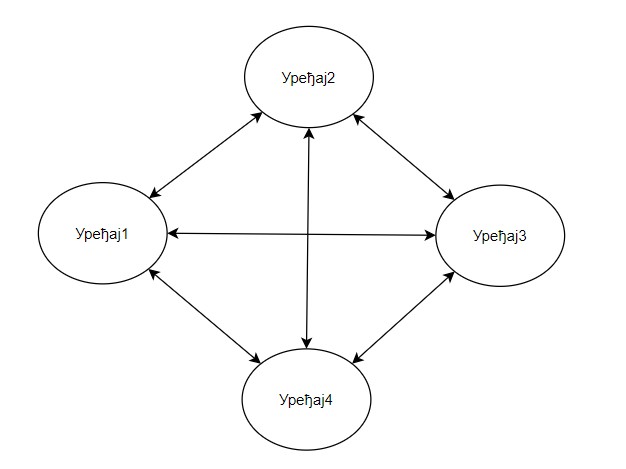
Предности *Peer-to-peer* архитектуре у односу на класичну клијент/сервер [4]:

* Смањени трошкови пропусног опсега - не постоји потреба за сервером који ће испоручивати податке
* Смањено кашњење - нема сервера који би успоравао комуникацију
* Безбедност - будући да је директна комуникација, корисници не морају вјеровати серверу да ће се користити шифровање
* Скалабилност - док у клијент/сервер моделу додавање нових учесника захтјева скалирање и изискује својеврсне проблеме, у *peer-to-peer* архитектури додавање *peer-*ова може да побољша перформансе, у зависности од типа апликације
* Робустност - отказивање једног *peer*-ане захтјева пад читавог система

Међутим, како у свијету програмирања не постоји савршена ствар, тако и *peer-to-peer* има своје изазове, неки од њих су [4]:

* Загушење мреже - уколико више *peer-*овау исто вријеме шаље већу количину података, може доћи до загушења, јер ће сви остали *peer-*овибити преплављени подацима
* Синхронизација - пошто нема компонентe која ће имати ауторитет над осталим компонентама, доста је теже да се читава мрежа доведе у једно исто стање, јер то може изискивати времена док се *peer-*овиусагласе
* Интегритет података - ако се чувају подаци у мрежи, очување интегритета постаје проблем, јер су сви подаци разбацани по читавој мрежи

*Peer-to-peer* не захтјева да сви уређаји буду међусобно повезани, али WebRTC апликацију је најједноставније развијати тим приступом. На слици 2.1.1 види се приказ топологије мреже коришћене у WebRTC апликацији.



Слика 2.1.1 приказ *peer-to-peer* система

## 2.6 *Signaling* процес

До почетка размјењивања порука и аудио-видео садржаја, комуникација не може да иде путем WebRTC-а. Корисници морају да обезбједе начин да два рачунара размјене потребне информације за *peer* конекцију. Тај процес се назива *signaling*. Добра ствар у вези WebRTC-а, је што он не обавезује кориснике шта треба да користе за *signaling*. У *Mozzila mdn web* документацији чак шаљиво пише да се може користити голуб писмоноша. Ипак, у пракси се најчешће користи *signaling server* у комбинацији са *WebSocket-*има[10,11]*.*

Неке од порука које се размјењују у *signaling* процесу су*:*

1. Session Description Protocol (SDP) објекти
2. Interactive Connectivity Establishment (ICE) кандидати
3. Поруке везане за отварање/затварање комуникационог канала и за манипулисање грешкама

Прије него што се започне процес, потребно је извршити одређене акције да би процес био могућ:

1. Сваки *peer* мора да направи RTCPeerConnection објекат
2. Сваки *peer* мора да има механизам за обраду icecandidate догађаја
3. Сваки *peer* мора да има механизам за обраду trackдогађаја
4. Иницијатор позива мора да обезбиједи јединствени токен којим ће се идентификовати позив, и послати исти другом кориснику
5. На основу токена сваки *peer* се повезује са *signaling* сервером

## 2.7 WebRTC протоколи

WebRTC је настао на групи протокола. Сваки од тих протокола има своју улогу у проналаску најбољег начина за комуникацију два *peer-*а*.* Ти протоколи су [12]:

1. Interactive Connectivity Establishment (ICE)
2. Session Traversal Utilities for NAT (STUN)
3. Traversal Using Relays around NAT (TURN)
4. Session Description Protocol (SDP)

### 2.7.1 Interactive Connectivity Establishment

У већини случајева директна веза између два рачунара није могућа (ако се изузме случај да рачунар има јавну Internet Protocol (IP) адресу и да два рачунара нису у истој мрежи)*.* За рјешавање тог проблема, WebRTC користи Interactive Connectivity Establishment (ICE) да одреди оптимални начин за повезивање два уређаја [13]. Један кандидат представља једну путању којом је могуће доћи до уређаја. Постоји неколико типова кандидата [14]:

* Кандидат домаћина - ово су заправо локалне IP адресе уређаја, и оне су дио мреже које не захтјевају Network Address Translation (NAT) заобилазак
* Кандидат рефлексивног сервера - представља јавну IP адресу која је добављена помоћу Session Traversal Utilities for NAT (STUN) сервера, о чему ће бити говора у наставку
* Релејски кандидати - представља информацију о томе како приступити уређају преко Traversal Using Relays around NAT (TURN) сервера

WebRTC у раду са ICEкандидатима има процес прикупљања истих који се састоји из три фазе [33]:

* прикупљање - ова фаза подразумијева прикупљање свих кандидата из различитих извора (горе споменути)
* упаривање и провјера - кандидати оба уређаја се покушавају спојити и тестира се њихова конекција
* селекција - ова фаза подразумијева избор најбољег пара кандидата, на основу кашњења и доступности, који ће се користити за комуникацију

Параметри које ICEузима у обзир су: приоритизација директне везе (избјегавање TURNсервера), брзина избора кандидата (потражња и избор не би требали трајати предуго) и избор TURNсервера на основу географске локације.

У овом примјеру на слици 2.7.1.1 може се видјети који су то битни параметри. Овај ICE кандидат носи информацију о IP адреси 10.51.5.8 на порту 54754 преко User Datagram Protocol (UDP) протокола [15].

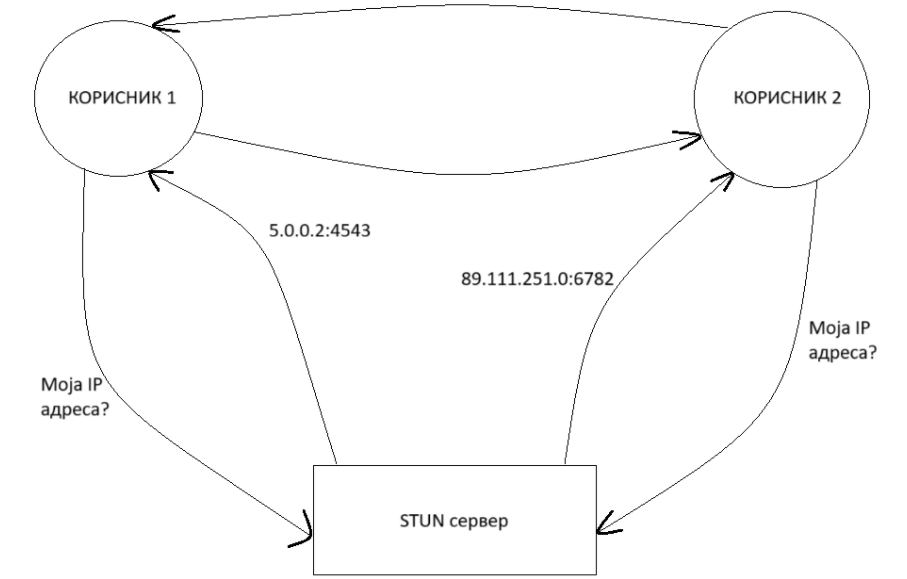


Слика 2.7.1.1 примјер RTCIceCandidateобјекта

### 2.7.2 Session Traversal Utilities for NAT

Ово је протокол који је старији и од WebRTC-а и од ICE*-*а*.* Да би се говорило о његовој улози, мора се споменути Network Address Translation (NAT) мапирање. NAT мапирање је механизам којим се одређује како се екстерна IP адреса мапира на интерну [16]. STUN протокол се користи у WebRTC-у да се уређај обрати STUN серверу како би добио информацију о томе са којом јавном IP адресом ступа у комуникацију са интернетом. STUN је кључан за *peer-to-peer* системе. Када корисник добије ту информацију, он може да је пошаље другом кориснику како би могли да комуницирају директно [17].

У примјеру на слици 2.7.2.1 види се да рутер који има јавну IP адресу, просљеђује све пакете који стижу на порт 4543, ка кориснику у његовој мрежи на порт 4543. STUN омогућава да корисник добије информацију о адреси и порту рутера који му пропушта пакете. Кориснику је та информација корисна, јер је може послати другим корисницима како би му могли директно слати поруке. Да би се то постигло, потребно је обратити се STUN серверу, чију адресу треба прослиједити RTCPeerConnection објекту.



Слика 2.7.2.1 Комуникација са STUN сервером

### 2.7.3 Traversal Using Relays around NAT

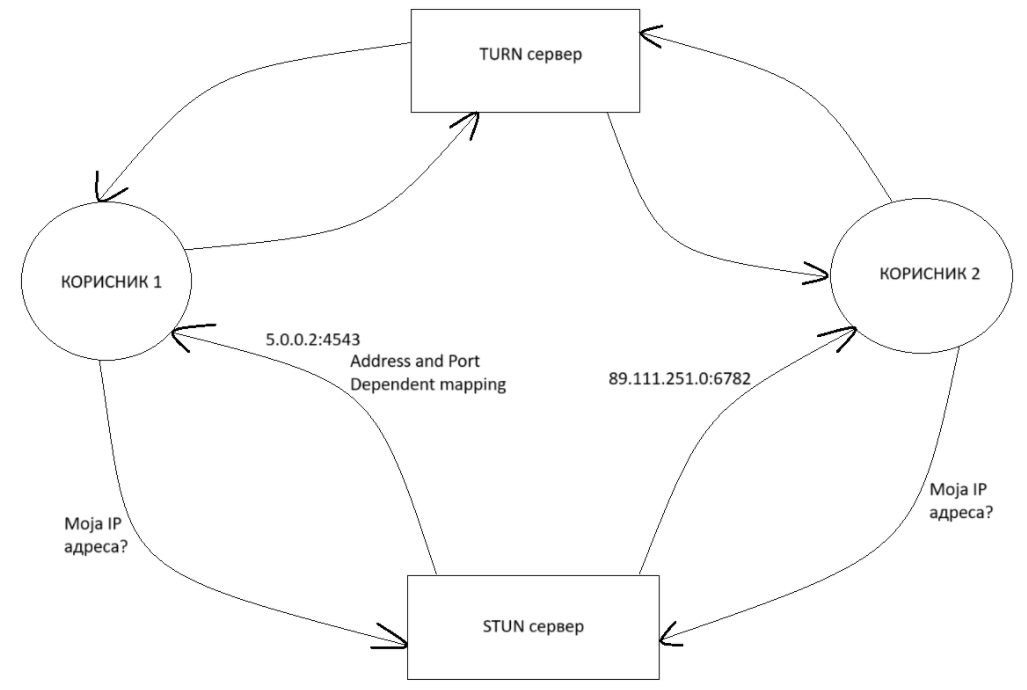
У неким случајевима комуникација не може ићи током који је описан у поглављу изнад. Разлог за то је што рутери могу да имају различите типове мапирања. Тај проблем се рјешава тако што у комуникацију улази TURN сервер, који ће служити само као преносник података [18].

Постоје три типа NAT мапирања [19]:

1. *Endpoint-Independent* - једно мапирање се користи за сваког пошиљаоца, тј. пакети послати са друге адресе и другог порта на мапирану путању, успјешно ће стићи на дестинацију. Ово је најбољи случај за WebRTC, а да би позив био остварен, бар једна страна мора имати овакво мапирање
2. *Address dependent* - за сваку нову адресу, прави се ново мапирање
3. *Address and Port Dependent Mapping* - за сваку нову адресу и нови порт, прави се ново мапирање

На основу ових типова мапирања, може се закључити да ће директна комуникација између два *peer-*абити могућа само ако је мапирање првог типа. STUN сервер ће открити на коју адресу и порт треба слати пакете за *peer* 1, и то је довољно да *peer* 2успјешно ступи у везу. Међутим ако је било који други тип мапирања, потребан је TURN сервер. У том случају, TURN сервер служи као прокси*,* како би сва комуникација прошла кроз њега, и успјешно стигла до одредишта.

На слици 2.7.3.1 је приказано како тече комуникација када је у њу укључен TURN сервер. Корисници прво добијају информацију од STUN сервера о томе која је њихова јавна IP адреса и који је тип NAT мапирања, и даље у зависности од тог у комуникацију се укључује и TURN сервер.

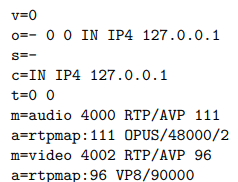


Слика 2.7.3.1 Комуникација са TURN и STUN серверима

### 2.7.4 Session Description Protocol

Session Description Protocol (SDP) представља протокол који служи за опис медија. Он служи како би се уређаји могли споразумјети које медије “разумију” и које користе. Имплементиран је као скуп парова кључ вриједност, гдје је кључ увијек један карактер [20].

На слици 2.7.4.1 је приказано како изгледа SDP објекат.



Слика 2.7.4.1 Изглед *SDP* објекта

Седам најбитнијих кључева и њихове вриједности су [21]:

* *v* - верзија, вриједност би требала бити 0
* *o* - *origin*, садржи идентификатор за преговарање
* *s* - назив сесије, вриједност би требала бити -
* *t* - *timing*, овај параметар даје почетно и крајње вријеме. Ако су подешени на 0 0, корисници нису ограничени временом
* *m* - опис медија
* *a* - *attribute*, текстуална вриједност. Доста различитих вриједности се шаље овим параметром, једна од њих је IP адреса и порт који ће се кориситити за Real-Time Transport Control Protocol (RTCP)
* *c* - *connection line*, овдје се налази вриједност IP адресе преко које би се требала одвијати комуникација, али пошто се користи ICEпротокол, тај параметар ће бити игнорисан.

У WebRTC-у SDPсе прави преко објекта RTCPeerConnection помоћу методa createOffer() и createAnswer()*.*

# 3 Апликација за видео позиве

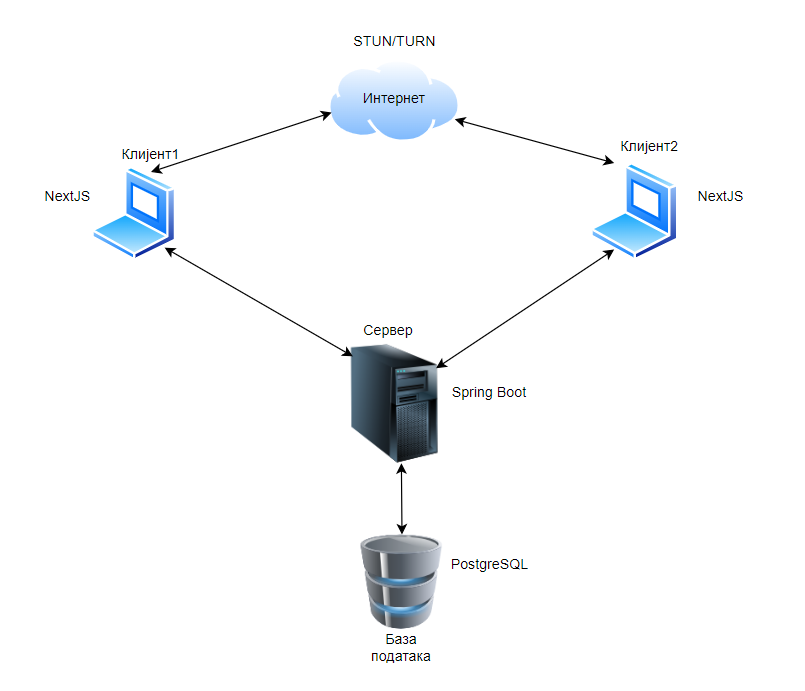
Циљ апликације која је описана у овом раду јесте да уз помоћ WebRTC технологије корисницима омогући *peer-to-peer* везу гдје год је то могуће и да уз што мање кашњења групи корисника да могућност видео позива. Њене главне функционалности су:

* укључивање више људи у позив
* видео и аудио комуникација
* дијељење екрана
* укључивање/искључивање микрофона и камере
* дописивање (енгл. chat)

Да би се разумјело како ради WebRTC апликација, потребно је видјети како све компоненте комуницирају међусобно, и како изгледа архитектура такве апликације. У наставку ће бити приказан дијаграм архитектуре система, комуникација и дијаграм клијентског дијела апликације.

## 3.1 Дијаграм архитектуре

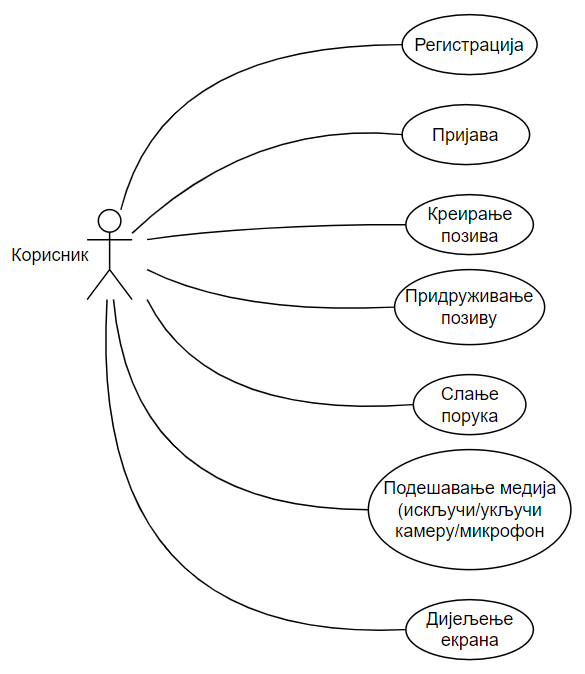
На слици 3.1.1 је приказан дијаграм који објашњава да је у читаву архитектуру укључен сервер на којем је серверска апликација написана у *Spring boot* радном оквиру [24]. Сви подаци везани за позиве и за учеснике позива се чувају у *PostgreSQL* бази података [22]. Са серверскомапликацијом комуницирају клијентске апликације на корисничким уређајима. Те апликације су написане у *NextJS* радном оквиру [23]. Да би се остварила комуникација морају постојати STUNи TURNсервери. У овом случају STUN сервер се налази на адреси stun:stun.l.google.comа TURNна адреси 89.40.8.248.



Слика 3.1.1 архитектура апликације

## 3.2 Дијаграм случајева коришћења

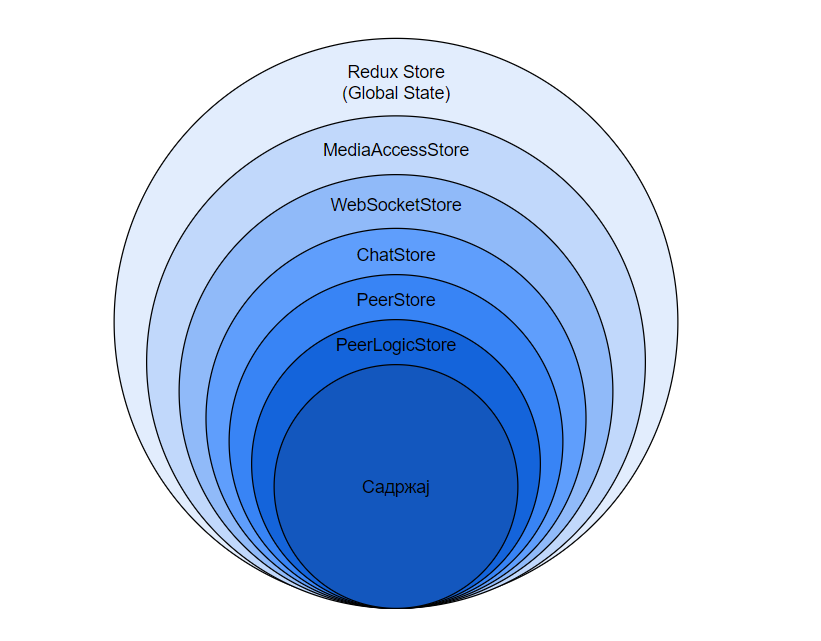
Дијаграм случајева коришћења служи да се види на које све начине корисник интерагује са апликацијом. На слици 3.2.1 види се да постоји само једна улога корисника, те да су његове основне активности: регистрација, пријава, креирање позива, придруживање позиву, слање порука, подешавање медија, дијељење екрана.



Слика 3.2.1 дијаграм случајева коришћења

## 3.3 Дијаграм архитектуре клијентског дијела апликације

На слици 3.3.1 се види да је клијентски дио апликације подијељен по слојевима. Искориштен је тај приступ јер су унутрашњи слојеви зависни од спољашњих. У наставку слиједи кратак опис сваког слоја, а у поглављу имплементације апликације је детаљнији.



Слика 3.3.1 архитектура фронтенда по слојевима

***ReduxStore*** - слој који представља управљање глобалним стањем апликације, у њему се чувају подаци о учесницима у позиву, позицијама на екрану, те лични подаци и идентификатор позива

***MediaAccessStore*** - слој у ком се чувају објекти који су везани за медије, као што су камера и микрофон

***WebSocketStore*** - у овом слоју се налази *socket* конекција која се користи у дубљим слојевима

***ChatStore*** - овај слој чува све текстуалне поруке које се шаљу

***PeerStore*** - у овом слоју се налази објекат RTCPeerConnection; пошто свака конекција ка кориснику у позиву мора имати свој објекат, постоји онолико слојева колико има учесника у позиву

***PeerLogicStore*** - овај слој издваја сву логику везану за RTCPeerConnection; овај дио је могао да се споји или са слојем изнад или испод, али је издвојен због прегледности кода.

**Садржај** - под овим се мисли на видљиви дио апликације.

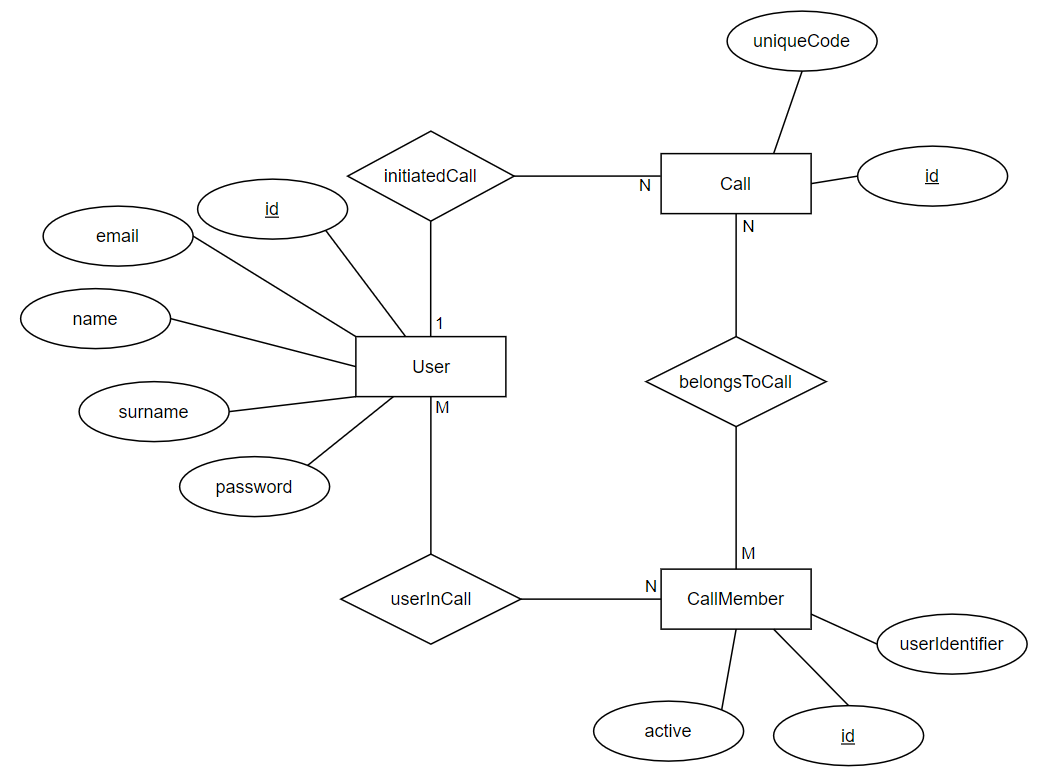
# 4 Имплементација

Наредно поглавље је посвећено конкретној имплементацији апликације која користи WebRTC. Поглавље ће бити подијељено у четири дијела. Први дио ће бити фокусиран на серверски дио апликације, док ће други дио бити фокусиран на клијентски дио, коришћење WebRTC-a и поруке које се размјењују. У трећем дијелу ће се говорити о конфигурацији сопственог TURN сервера, а у четвртом дијелу биће презентована апликација.

## 4.1 Серверски дио апликације

Серверска апликација је имплементирана у радном оквиру *Spring Boot.* Већ је споменуто да за WebRTC није битан серверски дио, па су улоге серверског дијела у овој апликацији: *регистрација, пријава на систем, креирање позива* тј*. добијање идентификатора позива, придруживање позиву* и *излазак из позива*.

На слици 4.1.1 је приказан ЕР модел класа које се налазе на серверском дијелу апликације. Модел приказује три кључне класе за разумијевање пројекта и свака од њих ће бити посебно објашњена у овом поглављу.



Слика 4.1.1 ЕР модел модела апликације на серверској страни

Једна од класа је User која је приказан на листингу 4.1.2. Userпредставља модел који служи за ауторизацију и аутентификацију, те је то такође модел који ће се везати за позив. Он у себи садржи основне информације о кориснику: мејл, име, презиме и лозинку.

Листинг 4.1.2 модел *User*

1 public class User {

2

3 @Id

4 @GeneratedValue(strategy = GenerationType.AUTO)

5 private long id;

6

7 private String email;

8

9 private String name;

10

11 private String surname;

12

13 private String password;

14 }

Модел Call је приказан на листингу 4.1.3. Он садржи информације о позиву. Те информације су јединсвтени идентификатор, у овом случају Universally Unique Identifier (UUID)и креатор позива. Веза између креатора позива и самог позива јесте “више на један”, тј. више позива може бити креирано од стране једног корисника.

1 @Entity(name = "calls")

2 public class Call {

3

4 @Id

5 @GeneratedValue(strategy = GenerationType.AUTO)

6 private Long id;

7

8 private String uniqueCode;

9

10 @ManyToOne

11 private User initiator;

12 }

Листинг 4.1.3 модел *Call*

Модел CallMember је приказан на листингу 4.1.4. Он спаја позив и учесника у позиву. Веза овог модела са моделом Call је више ка једном, тј. више различитих учесника може бити у једном позиву. Веза са моделом User је такође више ка једном, тј. један корисник није везан само за један позив, већ може бити у више различитих њих. Поље userIdentifierје имплементациони детаљ. Та вриједност је Universally Unique Identifier (UUID) и служи да разликује сваког учесника у позиву. UserIdentifierсе исто тако користи као јединствена адреса за *WebSocket* на коју се корисник претплати. Вриједност isActiveслужи као индикатор да ли је корисник изашао из позива.

1 @Entity(name = "call\_members")

2 public class CallMember {

3

4 @Id

5 @GeneratedValue(strategy = GenerationType.AUTO)

6 private Long id;

7

8 @ManyToOne

9 private Call call;

10

11 @ManyToOne(fetch = FetchType.EAGER)

12 private User user;

13

14 private String userIdentifier;

15

16 private boolean active;

17 }

Листинг 4.1.4 модел *CallMember*

Листинг 4.1.5 приказује битну класу за комуникацију два корисника а то је SdpDto. Садржај објекта ове класе се не чува нигдје у бази, већ само служи за пренос података од једног корисника до другог. У овом случају, серверска страна служи само као веза, она ништа не ради са подацима.

Листинг 4.1.5 класа *SdpDto*

1 public record SdpDto(MemberDto memberDto,

2 String sdp,

3 String destination,

4 String requestType) {

5 }

За SdpDtoје искориштен Java *record*. Описи свих чланова слиједе у наставку.

* memberDto- садржи податке о пошиљаоцу (идентификатор, мејл, име, презиме)
* sdp- чува серијализован JavaScript Object Notation (JSON) садржај поруке која се шаље
* destination- идентификатор примаоца поруке
* requestType - инфомрација о ком типу поруке се ради, може да има сљедеће вриједност: JOIN\_NOTIFICATION*,* SENDING\_OFFER*,* SENDING\_ANSWER*,* SENDING\_ICE*,* END\_CALL

Да би се боље разумјела улога серверског дијела, потребно је упознати се са кључним Representational State Transfer (REST) крајњим тачкама помоћу којих се комуницира са њим.

* *POST api/call* - креира нови позив и враћа идентификатор позива и идентификатор креатора позива
* *POST api/call/{callId}* - придружује улогованог корисника позиву који има идентификатор прослијеђен у путањи. Враћа листу свих учесника у позиву, са њиховим подацима (име, презиме, мејл, идентификатор) и идентификатор корисника који се придружује
* *POST api/cal/{callId}/exit* - улогованог корисника искључује из позива са прослијеђеним идентификатором
* *POST api/sdp* - служи за међусобну комуникацију два корисника. Прима SdpDto објекат.

На листингу 4.1.6 види се како изгледа креирање позива. Позивом методе initiateCall, серверска страна генерише идентификатор позива и чува у бази података објекат класе Call који је везан за улогованог корисника. Даље се поново генерише идентификатор али овог пута се користи за идентификацију корисника. У базу података чува се објекат класе CallMember, и враћа кориснику идентификатор позива, лични идентификатор и личне податке.

1 @Override

2 public CreatedCallDto initiateCall() {

3 String callIdentifier =

4 identifierGenerator.generate();

5 var user = userUtils.getLoggedUser();

6 var call = Call.builder()

7 .initiator(user)

8 .uniqueCode(callIdentifier)

9 .build();

10 callRepository.save(call);

11

12 String callerIdentifier =

13 identifierGenerator.generate();

14 CallMember callMember = CallMember.builder()

15 .call(call)

16 .user(user)

17 .userIdentifier(callerIdentifier)

18 .active(true)

19 .build();

20 callMembersRepository.save(callMember);

21

22 return new CreatedCallDto(callIdentifier,

23 callerIdentifier, user.getName(), user.getSurname(),

24 user.getEmail());

25 }

Листинг 4.1.6 метода за прављење позива

На листингу 4.1.7 је приказано како изгледа придруживање позиву. Позивом методе joinCall*,* серверска страна ће провјерити да ли позив са прослијеђеним идентификатором постоји, уколико не постоји вратиће грешку. Уколико постоји, добавиће из базе података све кориснике који су и даље у позиву и заједно са њиховим идентификаторима и личним подацима, вратити изгенерисани идентификатор улогованог корисника.

1 @Override

2 public JoinedCallDto joinCall(String callId) {

3 Call call = callRepository.findByUniqueCode(callId)

4 .orElseThrow(() -> new

5 CustomBadRequestException("Call does not exist"));

6

7 List<CallMember> members =

8 callMembersRepository.

9 getAllByCallAndActiveIsTrue(call);

10 var user = userUtils.getLoggedUser();

11 String memberId = identifierGenerator.generate();

12 CallMember callMember = CallMember.builder()

13 .user(user)

14 .call(call)

15 .userIdentifier(memberId)

16 .active(true)

17 .build();

18 callMembersRepository.save(callMember);

19 return new JoinedCallDto(callId, memberId,

20 user.getName(), user.getSurname(),

21 user.getEmail(),

22 members.stream()

23 .map((member) -> MemberDto.builder()

24 .email(member.getUser().getEmail())

25 .name(member.getUser().getName()) 26 .surname(member.getUser().getSurname())

27 .memberId(member.getUserIdentifier())

28 .build()).collect(Collectors.toList()));

29 }

Листинг 4.1.7 метода за придруживање позиву

На листингу 4.1.8 је приказано како изгледа излазак из позива. Позивом методе exitCall, серверска страна ће прво провјерити да ли постоји позив, уколико не постоји, вратиће грешку. Даље ће провјерити да ли улоговани корисник припада том позиву, ако не припада, такође ће вратити грешку. Уколико је креатор позива, тај који прекида позив, сви учесници ће изаћи из позива, а у супротном само тај корисник. Даље се шаље обавјештење путем *WebSocket-*асвим учесницима о напуштању позива, како би се на клијентском дијелу могле одрадити одговарајуће акције [28].

1 @Override

2 public void exitCall(String callId) {

3 Call call = callRepository.findByUniqueCode(callId)

4 .orElseThrow(() -> new

5 CustomBadRequestException("Call does not exist"));

6 List<CallMember> callMembers =

7 callMembersRepository.getAllByCallAndActiveIsTrue(call);

8 User user = userUtils.getLoggedUser();

9 CallMember callMember =

10 callMembersRepository.findByCallAndUser(call, user)

11 .orElseThrow(() -> new

12 CustomBadRequestException("Call does not exist"));

13 if(call.getInitiator().equals(user)) {

14 callMembers.forEach(cm -> cm.setActive(false));

15 callMembersRepository.deleteAll(callMembers);

16 } else {

17 callMember.setActive(false);

18 callMembersRepository.save(callMember);

19 }

20 notifyMembersAboutExitingCall(callMembers, user,

21 callMember);

22

23 }

Листинг 4.1.8 метода за напуштање позива

## 4.2 Клијентски дио апликације

Највећи дио логике и сва комуникација налази се у клијентском дијелу. То је зато што се, ако се заиста жели остваривање *peer-to-peer* комуникације, не може превише ослањати на серверску комуникацију, већ на клијентски дио. У поглављу дијаграм архитектуре клијентског дијела апликације, угрубо је представљена архитектура клијентског дијела у виду слојева зависности, па ће у наставку бити детаљнији опис сваког од тих слојева.

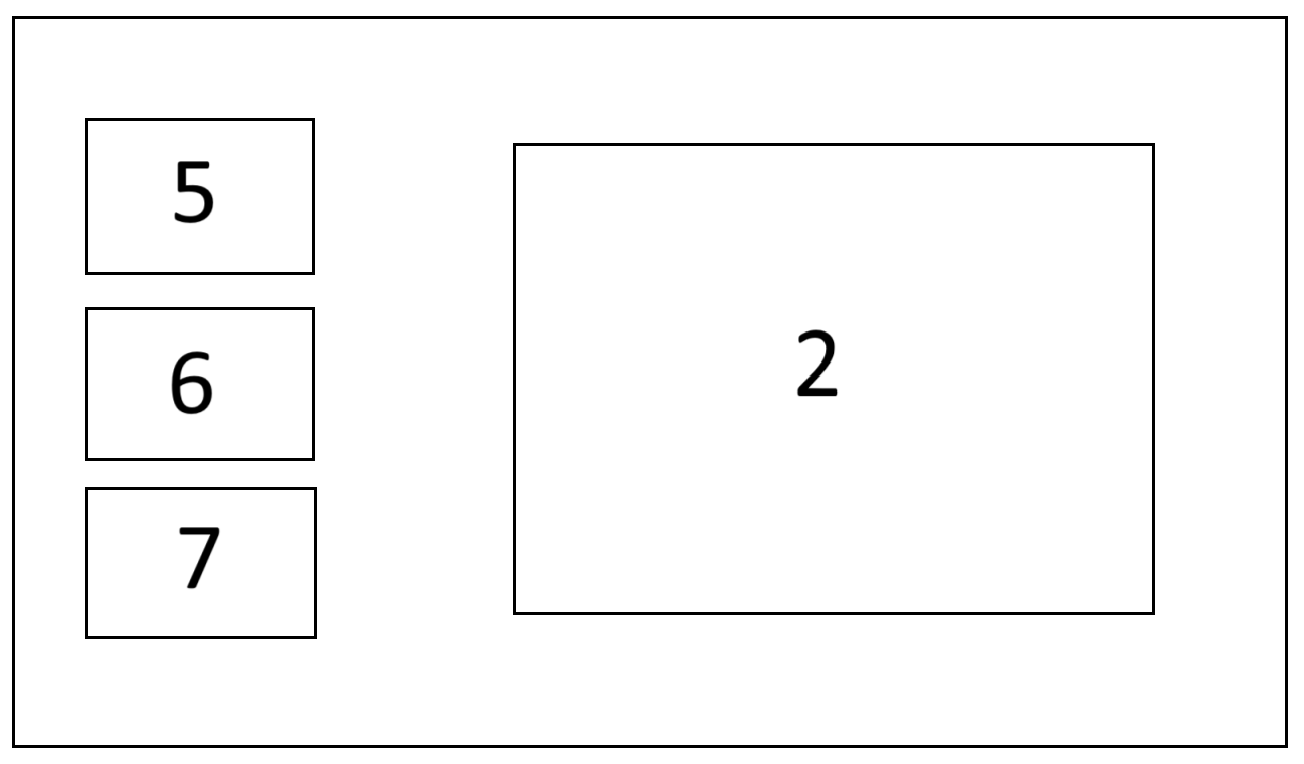
### 4.2.1 *Redux Store*

*Redux* је Javascript библиотека која је намијењена за манипулисање подацима на нивоу читаве апликације. У овој апликацији се користи *Redux Toolkit* [25]који је само имплементација *Redux-*а аглавна идеја је да олакша подешавање *Redux-*аи смањи количину кода. Подаци су подијељени у одсјечке и пододсјечке. На слици 4.2.1.1 је приказано како у овој апликацији изгледају одсјечци и подосјечци у току позива.



Слика 4.2.1.1 Изглед *redux slice-*a *call*

У овој апликацији, *Redux* је задужен за чување свих битних података везаних за учеснике у позиву и за сам позив. Главни и једини одсјечак је callкоји обједињује све податке. У оквиру њега чувају се лични подаци о кориснику (идентификатор, име, презиме, мејл) и идентификатор позива. Такође, у одсјечку call чувају се подаци о остаилим учесницима у позиву. Поље members садржи листу свих учесника са њиховим подацима (име, презиме, мејл, идентификатор и стање њиховог rtcPeerConnectionобјекта). У пољу membersPositions чувају се позиције тих учесника на екрану. На слици 4.2.1.2 је приказано како су организоване позиције учесника у позиву, без корисника апликације. Ове позиције се такође могу прочитати на слици 4.2.1.1 као вриједности поља membersPositions. Уколико би позив имао више од пет учесника, шести учесник би имао вриједност 8 која говори да је учесник у позиву, и да се његов видео и звучни запис и даље прима, али да се не приказује. Имена тих корисника су приказана са стране, те се њихов видео запис може укључити било када у току позива.



Слика 4.2.1.2 приказ распореда учесника позива на екрану

### 4.2.2 *MediaAccess Store*

Овај слој је имплементиран помоћу *ReactContext*-а [26]. *Context* даје могућност да се у компонентама вриједностима приступа без да се прослијеђују као props објекти.

У том контексту се чувају референце камере и микрофона корисника који користи апликацију. Разлог зашто је овај слој обухватио готово читаву апликацију, јесте зато што је потребно да у сваком слоју испод постоји приступ камери и микрофону чије вриједности треба слати *peer-*овима*.*

На листингу 4.2.2.1 је приказан *hook* који је задужен за приступање камери и микрофону корисника. Кориснику ће бити отворен прозор на којем ће тражити од њега дозволу за притупање камери и микрофону. Те вриједности ће сачувати као посебне и обједињене као MediaStreamобјекат. Битно је напоменути да комуникација неће бити остварена ако нема приступ ни камери ни микрофону, због начина како је WebRTC имплементиран (неће бити креирани ICEкандидати уколико нема ниједнога медија).

1 export const useCreateMediaStream = () => {

2 const context =

3 useContext(MediaAccessStoreContext);

4 return async (constraints: MediaConstraints) => 5 {

6 await navigator.mediaDevices

7 .getUserMedia(constraints)

8 .then((stream) => {

9 const videoTrack =

10 stream.getVideoTracks()[0];

11 const audioTrack =

12 stream.getAudioTracks()[0];

13 context.setVideoTrack(videoTrack);

14 context.setAudioTrack(audioTrack);

15 context.setMediaStream(stream);

16 context.setConstraints(constraints);

17 })

18 .catch((error) => {

19 console.error('Error accessing media

20 devices.', error);

21 });

22 };

23 };

Листинг 4.2.2.1 *hook* за приступање камери и микрофону

На листингу 4.2.2.2 је приказан *hook* који је намијењен за дијељење екрана. Он је по коду веома сличан *hook-*уна листингу 4.2.2.1. Разликује се по томе што другачијим параметрима методе getDisplayMediaдобија се могућност избора који екран корисник жели дијелити. Такође потребно је узети само видео траку и замијенити постојећу. У дубљим слојевима, детектоваће се та промјена, и замијениће се трака која се шаље *peer*-овима.

1 export function useAddScreenShare() {

2 const { setMediaStream, mediaStream,

3 setVideoTrack } =

4 useContext(MediaAccessStoreContext);

5

6 return async () => {

7 const screenCaptureStream = await

8 navigator.mediaDevices.getDisplayMedia({

9 video: {

10 displaySurface: 'browser',

11 },

12 });

13 const recentVideoTrack =

14 mediaStream?.getVideoTracks()[0];

15 if (recentVideoTrack) {

16 mediaStream?.removeTrack(recentVideoTrack);

17

18 mediaStream

19 .addTrack(screenCaptureStream

20 .getVideoTracks()[0]);

21 setMediaStream(mediaStream);

21 }

22

23 setVideoTrack(screenCaptureStream

24 .getVideoTracks()[0]);

25 return screenCaptureStream;

26 };

27 }

Листинг 4.2.2.2 *hook* за дијељење екрана

### 4.2.3 *WebSocket Store*

Овај слој је такође имплементиран као *ReactContext*. Он је задужен за чување *WebSocket* конекције. Нема потребе да се оптерећује мрежа и сервесрка страна *WebSocket* конекцијама, па се овим слојем постиже то да сви *peer-*овидијеле исту конекцију. За пријем порука преко *WebSocket*-акористи се библиотека *StompJS* [27]*.* На листингу 4.2.3.1 је приказано да се при првом учитавању апликације отвара *WebSocket* конекција и прави Stompобјекат. Чува се конекција и мијења вриједност индикатора да је конекција креирана.

1 export function WsStoreProvider({ children }: {

2 children?: ReactNode }) {

3 const [stompClient, setStompClient] =

4 useState<CompatClient | undefined>(undefined);

5 const [isConnected, setIsConnected] = useState(false);

6

7 let value: WsStoreState = {

8 stomp: stompClient,

9 isOpen: isConnected,

10 };

11 useEffect(() => {

12 const stomp = Stomp.over(function () {

13 return new WebSocket(`ws://localhost/api/ws`);

14 });

15 stomp.connect(

16 {},

17 () => {

18 setStompClient(stomp);

19 setIsConnected(true);

20 },

21 () => {

22 console.error('Error connecting to Stomp

23 server:');

24 }

25 );

26 stomp.debug = function () {};

27 return () => {

28 stomp.deactivate();

29 };

30 }, []);

31

32 return <WsStoreContext.Provider

33 value={value}>{children}</WsStoreContext.Provider>;

34 }

Листинг 4.2.3.1 креирање *WebSocket* конекције

### 4.2.4 *Chat Store*

*ChatStore* слој је такође имплементиран помоћу *ReactContext-*а. Његова функција је да чува текстуалне поруке које се шаљу у позиву. Овај слој мора бити изнад слоја гдје се брине о комуникацији, јер поруке могу доћи од било којег *peer-*а, тако да се морају обухватити сви *peer-*ови*.* Поруке се не чувају нигдје на серверској страни и нигдје не перзистују након што се позив заврши. Све поруке се чувају у контексту у промјенљивој messages. На листингу 4.2.4.1 се види *hook* који се позива да се порука која је стигла од неког *peer-*адода у листу свих порука.

1 export function useAddMessage() {

2 const context =

3 useContext(ChatStoreContext);

4 return (messageData: Message) => {

5 context.setMessages((prevMessages) => {

6 return [...prevMessages,

7 messageData];

8 });

9 };

10 }

Слика 4.2.4.1 додавање поруке у листу свих порука

На листингу 4.2.4.2 се види *hook* који се позива да се порука корисника убаци у листу свих порука и у *state* који чува поруку која се треба послати осталим учесницима у позиву. У слоју за WebRTC логику ће се детектовати промјена у том *state-*у, и порука ће бити послата сваком *peer-*упонаособ.

Слика 4.2.4.2 додавање корисникове поруке у листу свих порука

1 export function useAddMyMessage() {

2 const context = useContext(ChatStoreContext);

3 const myId = useMyId();

4 const myName = useMyName();

5 const mySurname = useMySurname();

6 const myEmail = useMyEmail();

7 return (message: string) => {

8 const messages = context.messages;

9 context.setMessages([

10 ...messages,

11 {

12 member: {

13 email: myEmail,

14 memberId: myId,

15 name: myName,

16 surname: mySurname,

17 rtcState: RtcState.SDP\_FULLFILED,

18 },

19 message: message,

20 time: getCurrentTime(),

21 },

22 ]);

23 };

24 }

### 4.2.5 *RtcLogic Store*

Изнад овог слоја заправо има још један слој, и он је задужен само за чување RTCPeerConnectionобјекта, али је његова имплементација тривијална и служи само због лакше организације кода. *RtcLogicStore* је слој у ком се налази сва логика везана за WebRTC. Због тога што сваки учесник у позиву мора имати своју конекцију тј. свој RTCPeerConnectionобјекат, за сваког учесника постоји један овакав слој.

На листингу 4.2.5.1 је приказана претплата на *WebSocket* конекцију на адресу */user/${myId}/queue/private*, гдје је myIdидентификатор корисника. На ту адресу ће стизати све поруке, и оне које нису везане за тај *peer* и које јесу. Зато треба направити провјеру да ли је порука која је стигла, за тај *peer* у чијем слоју се провјерава, и за одговарајући тип поруке која се шаље, одрадити одговарајућу акцију.

Слика 4.2.5.1 примање порука са сервера

1 useEffect(() => {

2 if (stomp && myId) {

3 stomp.subscribe(`/user/${myId}/queue/private`, 4 async function (message) {

5 const msg = JSON.parse(message.body) as

6 SdpPayload;

7

8 if(checkMessage(msg,RequestType.SENDING\_OFFER)) {

9 await setOfferAndSendAnswer(peerMember,

10 JSON.parse(msg.sdp) as RTCSessionDescriptionInit);

11 } else if (checkMessage(msg,

12 RequestType.SENDING\_ANSWER)) {

13 await setAnswer(peerMember,

14 JSON.parse(msg.sdp) as RTCSessionDescriptionInit);

15 } else if (checkMessage(msg,

16 RequestType.SENDING\_ICE)) {

17 const candidate = JSON.parse(msg.sdp) as 18 RTCIceCandidate;

19 addIceCandidate(candidate);

20 } else if (checkMessage(msg,

21 RequestType.END\_CALL)) {

22 deleteMember(peerMember);

23 }

24 });

25 }

26 }, [myId]);

На листингу 4.2.5.2 је приказан *hook* којим се прави и шаље offer. Након што се направи offer, потребно је да се дода као локални опис RTCPeerConnectionобјекта. Када се пошаље серијализовани offer*,* корисник коме се шаље ће примити поруку као што је приказано на листингу 4.2.5.1.

Листинг 4.2.5.2 креирање *offer-*аи слање истог другом кориснику

1 export function useCreateAndSendOffer() {

2 const { rtcPeerConnection, setRtcPeerConnection } 3 = useContext(PeerStoreContext);

4 const sendSdp = useSendSdp();

5

6 return async (member: CallMember) => {

7 if (rtcPeerConnection) {

8 const offer = await

9 rtcPeerConnection.createOffer();

10

11

12 await rtcPeerConnection.setLocalDescription(offer);

13 setRtcPeerConnection(rtcPeerConnection);

14

15 await sendSdp(member.memberId,

16 RequestType.SENDING\_OFFER, JSON.stringify(offer));

17 }

18 };

19 }

На листингу 4.2.5.3 је приказан *hook* који прима offerкоји је послат *hook-*омса листинга 4.2.5.2, додаје се тај offerкао удаљени опис RTCPeerConnectionобјекта, креира answer, додаје као локални опис и шаље истом кориснику који је послао offer. И offerи answer, су заправо SDP објекти, и по својој структури се не разликују. Корисник коме је послат answerће примити поруку као што је приказано на листингу 4.2.5.1 и додати га као удаљени опис. Након што се размјене и offerи answer, објекти су спремни да генеришу и шаљу ICEкандидате.

Листинг 4.2.5.3 креирање *answer-*аи додавање удаљеног описа

1 export function useSetOfferAndSendAnswer() {

2 const { rtcPeerConnection, setRtcPeerConnection } = 3 useContext(PeerStoreContext);

4 const sendSdp = useSendSdp();

5 const changeMember = useChangeMemberInStore();

6

7 return async (member: CallMember, offer:

8 RTCSessionDescriptionInit) => {

9 if (rtcPeerConnection) {

10 await

11 rtcPeerConnection.setRemoteDescription(offer);

12 const answer = await

13 rtcPeerConnection.createAnswer();

14 await

15 rtcPeerConnection.setLocalDescription(answer);

16 await sendSdp(member.memberId,

17 RequestType.SENDING\_ANSWER, JSON.stringify(answer));

18 setRtcPeerConnection(rtcPeerConnection);

19 changeMember(member, RtcState.SDP\_FULLFILED);

20 }

21 };

22 }

На листингу 4.2.5.4 је приказан код који дефинише шта објекат треба да ради када нађе ICEкандидата. Сви пронађени ICEкандидати се шаљу преко серверске стране, кориснику за који је везан конкретно тај RTCPeerConnection објекат. Корисник ће примити поруку као што је приказано на листингу 4.2.5.1, и додати тај ICEкандидат у RTCPeerConnection објекат, што се може видјети у коду приказаном на листингу 4.2.5.5.

Листинг 4.2.5.4 слање ICEкандидата

1 export function useAddOnSignalingChange() {

2 const { rtcPeerConnection, setRtcPeerConnection } 3 = useContext(PeerStoreContext);

4 const sendSdp = useSendSdp();

5 return async (peerMember: CallMember) => {

6 if (rtcPeerConnection) {

7 rtcPeerConnection.onicecandidate = async

8 (event) => {

9 if (event.candidate) {

10 const a = async () => {

11 await sendSdp(peerMember.memberId,

12 RequestType.SENDING\_ICE,

13 JSON.stringify(event.candidate));

14 };

15 setTimeout(a, 1000);

16 }

17 };

18 setRtcPeerConnection(rtcPeerConnection);

19 }

20 };

21 }

1 export function useAddIdeCandidate() {

2 const { rtcPeerConnection, setRtcPeerConnection } 3 = useContext(PeerStoreContext);

4 return async (iceCandidate: RTCIceCandidate) => {

5 if (rtcPeerConnection &&

6 rtcPeerConnection.remoteDescription != null) {

7 try {

8 setRtcPeerConnection(rtcPeerConnection);

9

10 await

11 rtcPeerConnection.addIceCandidate(iceCandidate);

12 } catch (e) {

13 console.error('Error adding received ice

14 candidate', e);

15 }

16 }

17 };

18 }

Листинг 4.2.5.5 примање ICEкандидата

## 4.3 TURN сервер

У ранијим поглављима било је ријечи о TURN серверима и о њиховом значају за WebRTC апликације. Постоје сервери на интернету који се могу користити. Већина њих се плаћа, али постоје и неки који дају бесплатне пакете у оквиру неког коришћења. Најчешће ограничење је по мјесечном протоку или по трајању. За ову апликацију искоришћен је бесплатни пројекат отвореног кода *Coturn* [29]. Он омогућава да корисници сами подесе свој сопствени TURN сервер. Само је потребан рачунар са јавном IP адресом, како би му остали уређаји могли приступити. Предности коришћења сопственог TURN сервера су [32]:

* могућност конфигурације - оптимизација перформанси, конфигурација сигурносних параметара и прилагођавање специфичним ситуацијама
* безбедност података - подаци пролазе кроз сервер што гарантује да неће бити злоупотребљени
* цијена - због конфигурабилности може се имати бољи увид у потрошњу и самим тим смањити трошкове
* скалабилност – постоји могућност да се сервер подеси на било којој географској локацији у зависности од потреба корисника

Неки од недостатака подешавања TURN сервера су [32]:

* одржавање - одлучивање на тај корак уводи додатну одговорност одржавања тог сервера
* безбедносни ризик - потенцијалне грешке отварају могућност рањивости апликације
* цијена при великом броју корисника - при великом броју корисника цијена оваквог сервера може изаћи више него рјешења са тржишта

На листингу 4.3.1 се види како изгледа конфигурација TURN сервера помоћu *Coturn-*a*.* Довољан је само један такав конфигурациони фајл, преко којег се могу подесити сви неопходни параметри [30]. Ти параметри су:

* realm - овај параметар служи како би се доменски могли одвојити кориснци који користе сервер. Користи се у комбинацији са корисничким именом и лозинком
* server-name - име сервера које се користи при OAuth аутентификацији (у овом конкретном случају неће бити потребно)
* fingerprint - параметар који говори да ће се при размјени TURN порука свака од њих потписати
* listening-ip - коју IP адресу ће TURN сервер слушати (0.0.0.0 значи да ће слушати све адресе)
* external-ip - IP адреса сервера
* listening-port - на ком порту ће сервер слушати захтјеве
* min/max-port - минимална и максимална граница за UDP портове које сервер може да додијели
* log-file и verbose - омогућава исписивање свих акција које се дешавају
* user - дефинисање креденцијала за пријаву на сервер
* lt-cred-mech - класичнимеханизам пријаве који је неопходан за WebRTC

1 # TURN server name and realm

2 realm=DOMAIN

3 server-name=turnserver

4

5 # Use fingerprint in TURN message

6 fingerprint

7

8 # IPs the TURN server listens to

9 listening-ip=0.0.0.0

10

11 # External IP-Address of the TURN server

12 external-ip=IP\_ADDRESS

13

14 # Main listening port

15 listening-port=3478

16

17 # Further ports that are open forcommunication

18 min-port=10000

19 max-port=20000

20

21 # Log file path

22 log-file=/var/log/turnserver.log

23

24 # Enable verbose logging

25 verbose

-

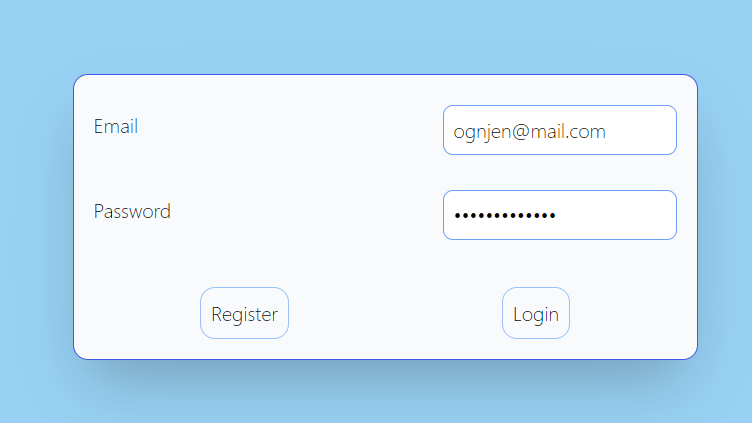
Sa

Листинг 4.3.1 конфигурација TURN сервера

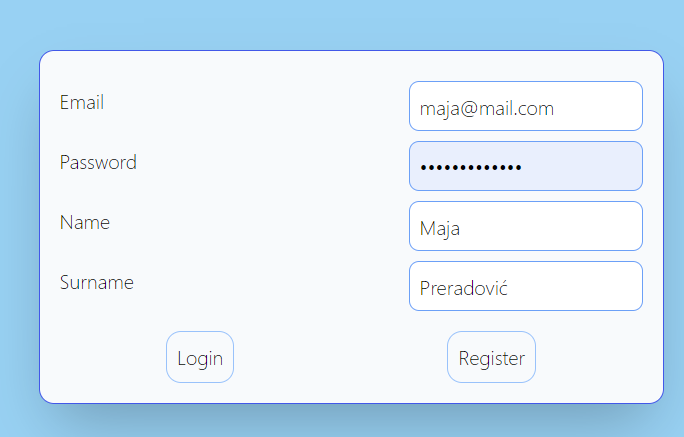
## 4.4 Финална апликација

У овом поглављу ће бити приказано како изгледа финална апликација. Кроз један позив у ком ће бити неколико учесника, биће приказане све функционалности које апликација нуди.

Корисник коришћење апликације почиње класичном пријавом са мејлом и лозинком, што је приказано на слици 4.4.1. Уколико корисник нема направљен налог, може да се региструје, што је приказано на слици 4.4.2. Име и презиме које корисник упише на регистрацији, писаће осталим корисницима док комуницирају са њим.

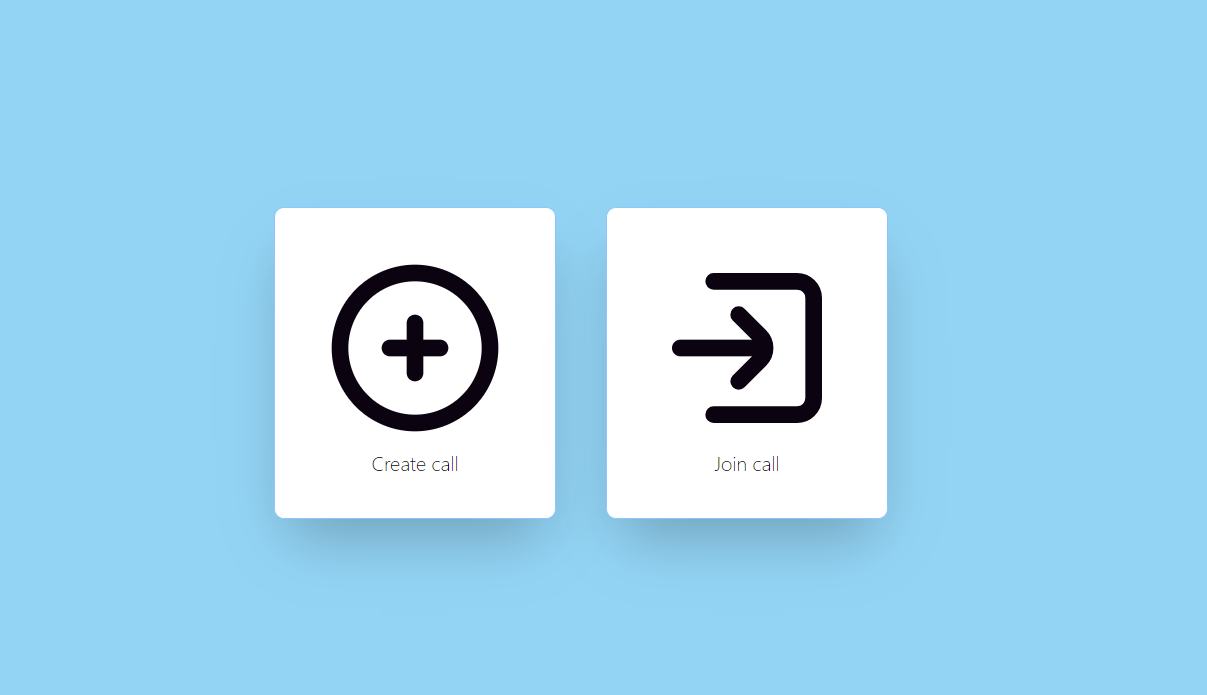


Слика 4.4.1 пријава корисника

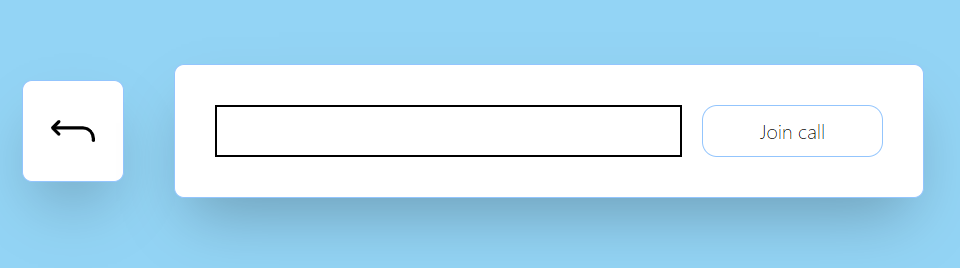


Слика 4.4.2 регистрација корисника

Улоговани корисник, на главној страници, има двије опције, да буде иницијатор позива или да се придружи неком позиву. На слици 4.4.3 се види како изгледа избор тих опција. На слици 4.4.4 је приказано како корисник треба да унесе идентификатор позива којем жели да се прикључи.

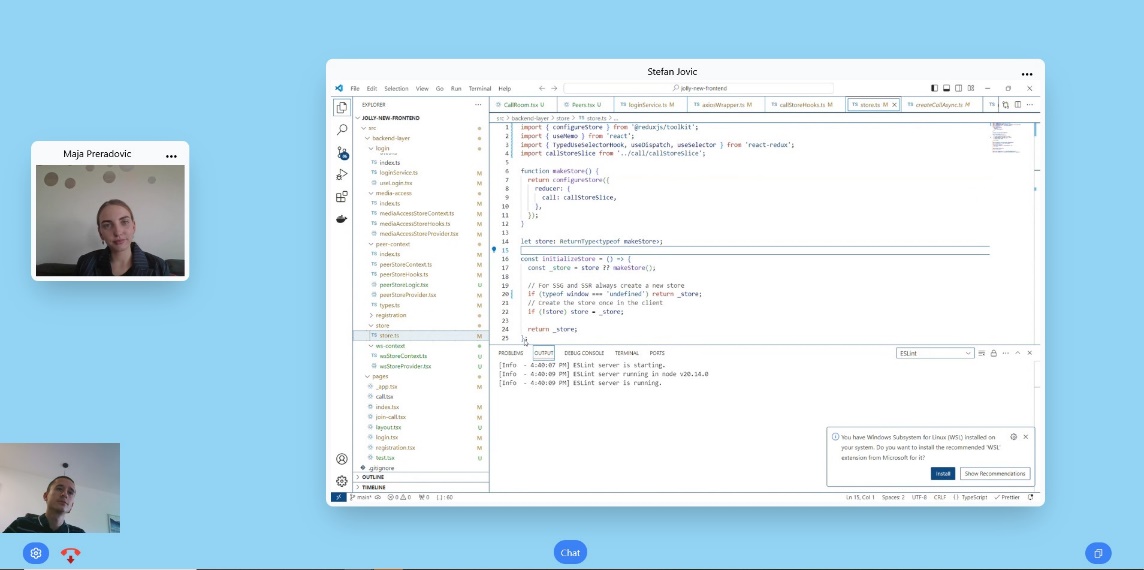


Слика 4.4.3 избор креирања или придруживања позиву



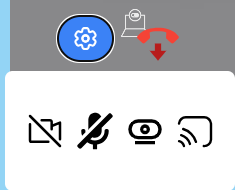
Слика 4.4.4 унос идентификатора позива

Слика 4.4.5 приказује како изгледа групни позив од 3 корисника. Учесници позива су распоређени тако што је један од њих у крупном плану, док је други распоређен у једној колони са лијеве стране. У лијевом доњем углу се налази и приказ камере оног који користи апликацију.



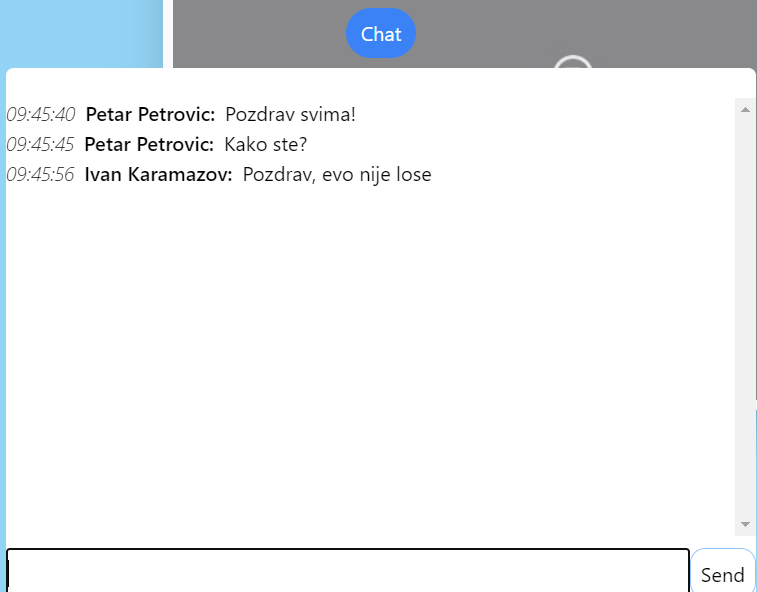
Слика 4.4.5 распоред учесника у групном видео позиву

Осим приказа видео снимака, корисник има 5 опција које се налазе са доње и десне стране. На слици 4.4.6 се види које додатне опције корисник добија кликом на зупчаник у лијевом доњем углу. Доступне опције су укључивање и искључивање камере, укључивање и искључивање микрофона, те двије опције пребацивања на дијељење екрана и враћање на снимак камере. Поред зупчаника, црвена слушалица служи за излазак из позива.



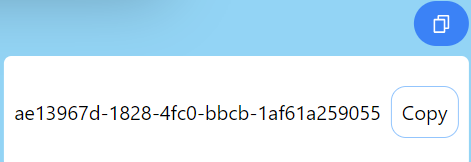
Слика 4.4.6 додатне опције

На слици 4.4.7 је приказан изглед прозора гдје се налазе текстуалне поруке. У истом прозору се и шаљу поруке. За сваку поруку постоји информација ко је шаље и тачно вријеме када је послана порука. О примљеној поруци корисник добија и обавјештење у десном горњем углу.



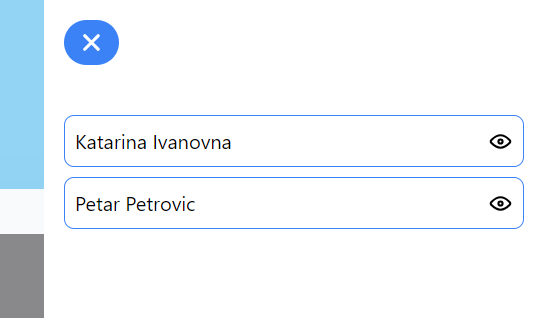
Слика 4.4.7 изглед чета

Корисник када креира позив, идентификатор креираног позива ће се појавити у десном доњем углу. Било када у току позиву може се приступити том идентификатору. Кликом на дугме *copy* идентификатор се смјешта у *clipboard,* и спреман је за слање другом кориснику.



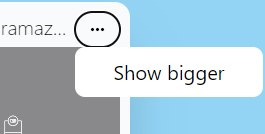
Слика 4.4.8 идентификатор позива

Већ је споменуто да не могу сви учесници у позиву бити приказани, тј не може њихов видео снимак бити приказан. До тих учесника може се доћи кликом на дугме у десном горњем углу. У склопу тог дугмета налази се број који говори колико има таквих учесника. Кликом на то дугме, отвара се прозор са листом имена тих корисника. Сваки сакривени корисник може се приказати кликом на иконицу ока. Приказивањем сакривеног корисника, један од приказаних ће прећи у сакривене, а изабрани корисник ће заузети његово мјесто. Битно је рећи да иако се сакривени корисници не виде, њихов звук је и даље присутан. На слици 4.4.9 је приказана листа сакривених учесника.

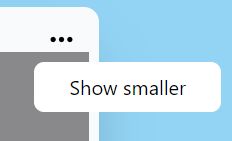


Слика 4.4.9 приказ сакривених учесника

На слици 4.4.5 може се видјети да је један учесник увијек у фокусу, тј. налази се на средини екрана. Међутим у сваком моменту, учесници који нису у фокусу, могу се ставити у фокус, рецимо ако тај учесник дијели екран. Та опција се добија кликом на три тачкице које се налазе у десном горњем углу снимка учесника, поред имена. На слици 4.4.10 је приказана опција фокусирања учесника, док је на слици 4.4.11 приказана опција избацивања учесника у фокус. Уколико се фокусирани корисник избацује из фокуса, аутоматски ће један од нефокусираних заузети његово мјесто.



Слика 4.4.10 фокусирање учесника



Слика 4.4.11 избацивање учесника из фокуса

# 5 Закључак

У првом дијелу овог рада је приказано како WebRTC функционише. Приказане су поруке и начин на који WebRTC користи ICE протокол. Објашњена је суштина NAT-а и како WebRTC користи STUN протокол како би открио на који начин комуницирати са крајњим корисницима. Међутим, често STUN неће бити довољан због појединих имплентација рутера. Опис тог проблема и рјешење помоћу TURN сервера такође има своје мјесто у овом поглављу. Такође у овом поглављу постоји опис SDP-а, који служи да крајњи корисници могу размијенити информације о медијима које ће користити у комуникацији.

У сљедећем поглављу је приказана шира слика читаве апликације. Дијаграм архитектуре апликације садржи све компоненте које чине апликацију и њихову комуникацију. Дијаграм случајева коришћења објашњава које све акције корисник може да обави на апликацији. Пошто је највећи дио логике на клијентском дијелу апликације, приказан је дијаграм архитектуре клијентског дијела кроз слојеве, гдје сваки слој има своју функцију и зависи од слојева који се налазе изнад њега.

Поглавље имплементације је подијељено на два дијела, серверски и клијентски дио. Серверски дио садржи листинге кода који креира нови позив и јединствени идентификатор помоћу којег се приступа позиву. Такође, приказује код који убацује корисника у позив. Клијентски дио је организован по поглављима. Па тако постоји поглавље које приказује *Redux* и скуп свих података битних за позив, поглавље које приказује како се приступа камери, микрофону, дијељењу екрана, итд. Посебно поглавље заузима опис управљања *WebSocket* конекцијом, која је битна за слање информација потребних за остваривање позива. Такође постоји опис како функционише слање текстуалних порука у позиву и гдје се оне чувају. Најбитнији дио овог рада се налази у поглављу које садржи листинге кода како се заправо користи WebRTC. Ту постоје прикази како се креирају SDPofferи answer*,* ICEкандидати, остваривање конекције итд. Пошто је већ речено да WebRTC у доста случајева захтјева TURN сервер, дио имплементације садржи опис како је могуће направити сопствени TURN сервер и користити га у својој апликацији.

WebRTC је технологија која је заправо често присутна у разним апликацијама а да тога корисници нису ни свјесни. Доста модерних апликација које имају функцију позива, у својој позадини користе WebRTC. Он на неки начин демистификује *peer-to-peer* комуникацију и даје могућност програмеру да направи занимљиве апликације као што су апликације за групне позиве. Сигурно његова највећа предност је једноставност коришћења и добра апстракција кода који се заправо бави остваривањем конекције, проналаском најбоље путање за комуникацију итд. Потенцијално његова мана би могла бити безбједносне природе. Ако узмемо случај да се корисников рачунар налази иза рутера који ће пропустити саобраћај занемарујући да ли долази са исте адресе и порта, одакле је послат први пакет, корисников рачунар може да буде угрожен уколико се преко WebRTC-а неко домогне те информације.

Овај рад је конципиран тако да садржи опис како WebRTC функционише, те апликације која га користи. Даљи развој апликације би ишао у смјеру коришћења позива без пријаве и регистрације на систем, уљепшавању изгледа апликације те побољшању перформанси у смислу да може радити са неколико стотина корисника.

# 6 Литература

[1] 8 powerful applications built using WebRTC (приступ 2024-05-30)

<https://www.unitedworldtelecom.com/learn/webrtc-applications/>

[2] Haxball About (приступ 2024-05-30)

<https://www.haxball.com/about>

[3] RTCPeerConnection documentation (приступ 2024-05-15)

<https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection>

[4] What are P2P networks exactly? (приступ 2024-05-16)

<https://medium.com/@asafkozovsky/what-are-p2p-networks-exactly-77284fe3b8a3>

[5] MediaDevices: getUserMedia() method documentation

(приступ 2024-05-30)

<https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia>

[6] Caputre audio and video in HTML5 (приступ 2024-05-30)

<https://web.dev/articles/getusermedia-intro>

[7] How to record the user’s screen (приступ 2024-05-30)

<https://web.dev/patterns/media/screen-record>

[8] RTCDataChannel documentation (приступ 2024-05-30)

<https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel>

[9] RTCDataChannel - WebRTC Explained (приступ 2024-05-30)

<https://www.onsip.com/voip-resources/voip-fundamentals/rtcdatachannels>

[10] The signaling server (приступ 2024-05-30)

<https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling>

[11] WebRTC for the Curious – Signaling (приступ 2024-05-30)

<https://webrtcforthecurious.com/docs/02-signaling/>

[12] Introduction to WebRTC protocols (приступ 2024-05-30)

<https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols>

[13] WebRTC for the Curious – Connecting (приступ 2024-05-30)

<https://webrtcforthecurious.com/docs/03-connecting/>

[14] Ultimate Guide to STUN, TURN, and ICE Servers in WebRTC (приступ 2024-05-23)

<https://www.ecosmob.com/stun-turn-ice-servers-in-webrtc/>

[15] RTCIceCandidate documentation (приступ 2024-05-30)

<https://developer.mozilla.org/en-US/docs/Web/API/RTCIceCandidate>

[16] What is NAT – Comptia (приступ 2024-05-30)

<https://www.comptia.org/content/guides/what-is-network-address-translation>

[17] WebRTC for Curious - Connecting – STUN (приступ 2024-05-30)

<https://webrtcforthecurious.com/docs/03-connecting/>

[18] TURN Server (приступ 2024-05-23)

<https://getstream.io/glossary/turn-server/>

[19] WebRTC for the Curious - Connecting - NAT Mapping

(приступ 2024-05-30)

<https://webrtcforthecurious.com/docs/03-connecting/>

[20] WebRTC for the Curious - Signaling - What is Session Description Protocol (SDP) (приступ 2024-05-30)

<https://webrtcforthecurious.com/docs/02-signaling/#what-is-the-session-description-protocol-sdp>

[21] WebRTCHacks - SDP anatomy (приступ 2024-05-30)

<https://webrtchacks.com/sdp-anatomy/>

[22] PostgreSQL (приступ 2024-05-30)

<https://www.postgresql.org/>

[23] NextJS (приступ 2024-05-14)

<https://nextjs.org/>

[24] Spring Boot (приступ 2024-05-30)

<https://spring.io/projects/spring-boot>

[25] Redux toolkit (приступ 2024-05-30)

<https://redux-toolkit.js.org/>

[26] React context (приступ 2024-05-30)

<https://react.dev/reference/react/createContext>

[27] StompJS (приступ 2024-05-30)

<https://github.com/stomp-js/stompjs>

[28] Spring WebSockets: Send Messages to a Specific User

(приступ 2024-05-30)

<https://www.baeldung.com/spring-websockets-send-message-to-user>

[29] Coturn TURN server (приступ 2024-05-27)

<https://github.com/coturn/coturn>

[30] Coturn flags (приступ 2024-05-27)

<https://github.com/coturn/coturn/wiki/turnserver>

[31] WebRTC (приступ 2024-05-30)

<https://webrtc.org/>

[32] TURN Server Costs: A complete Guide (приступ 2024-08-05)

<https://dev.to/alakkadshaw/turn-server-costs-a-complete-guide-1c4b>

[33] WebRTC for the Curious – Connecting – ICE (приступ 2024-08-08)

<https://webrtcforthecurious.com/docs/03-connecting/#ice>

# 7 Биографија

Огњен Шврака је рођен 11. маја 2000. године у Бањалуци. Основну школу “Бранко Ћопић” у Прњавору, завршио је 2015. године, а 2019. године завршава гимназију, информатички смјер, такође у Прњавору. Своје даље школовање наставља на Факултету техничких наука у Новом Саду. Положио је све испите предвиђене планом и програмом са просјечном оцјеном 8.19. Након школовања на факултету, враћа се у Бањалуку.