Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**Відеоконференції з використанням WebRTC мовою TypeScript**»

Виконав: студент 3-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Студента Молодцова Філліпа

Керівник Бабич Т.А.

магістр комп’ютерних наук, асистент

«11» травня 2021 р.

Київ – 2021

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“10” жовтня 2020 року

ЗАВДАННЯ

ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Молодцову Філліпу

1. Тема роботи **«Відеоконференції з використанням WebRTC мовою TypeScript»**, керівник роботи Бабич Трохим Анатолійович, магістр комп’ютерних наук, асистент

2. Строк подання студентом роботи 17 травня 2021

3. План роботи

Анотація

Вступ

Розділ 1. Дослідження та аналіз предметної області

1.1 Потокова трансляція відео
1.2 Технології для організації відеоконференцій у веб-застосунках
1.3 Протоколи й стандарти WebRTC

Розділ 2. Проектування та розробка системи

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 жовтня 2020			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтеня 2020 – 2 листопада 2020			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	2 листопада 2020			
4.	Написання розділів роботи	2 листопада 2020 – 01 березня 2021			
5.	Проміжний контроль виконання роботи	01 лютого 2021			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2021 – 29 березня 2021			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2021			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2021			
	Розділ 3 (проектно-рекомендаційна частина)	29 березня 2021			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня 2021 – 06 травня 2021			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	17 травня 2021			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 10 жовтня 2020 р.

Науковий керівник Бабич Трохим Анатолійович

Виконавець курсової роботи Молодцов Філліп

ЗМІСТ

Анотація

У данній роботі розглядаються особливості передачі мультимедійних даних, і аналіз підбору технології відеоконференцій. Пояснюється вибір технології WebRTC та детально описується протоколи та стандарти, які ця технологія використовує. На прикладі відеоконференційного застосунку розглядається прикладний програмний інтерфейс WebRTC та описуються особливості розробки за допомогою цієї технології динамічної вставки мультимедіа та додаткового шару шифрування.

Вступ

До пандемії COVID-19 онлайн спілкування для роботи та бізнесу було опціональним варіантом, який був значно менш пріоритетним та серйозним, ніж особиста зустріч, то зараз, у період, коли всі перебуваємо в ізоляції через пандемію COVID-19, такий різновид комунікації є бажаним та безпечним для всіх нас[1]. Безперечним є те, що для всіх нас це була досить різка зміна не лише у тому, як стали працювати, а й у цілому, як культурні та соціальні явища трансформувалися. І хоча більшість IT-гігантів створили або вдосконалили свої відеоконференційні додатки протягом 2020 року, все ж вони залишаються додатками загального призначення й досить багато бізнесів та установ мають потребу у більш специфічних варіантах відеозв'язку, напряму інтегрованих з їхніми робочими інформаційними системами. Саме тому вивчення технологій, що надають змогу розробляти засоби інтерактивної віртуальної комунікації в реальному часі, є досить актуальним у наші часи.

Метою цієї роботи є дослідити технології, що дозволяють передачу мультимедійних даних через інтернет; аналіз цих технологій щодо придатності їхнього використання для розробки відеоконференції; обґрунтувати вибір технології WebRTC як такої, що є найдоцільнішою у даному випадку й для заданих потреб; описати роботу з WebRTC.

Було виокремлено наступні завдання дослідження:

- Проаналізувати технології, які з них підійдуть найкраще для інтерактивних відеоконференцій у реальному часі.
- Обґрунтування переваг WebRTC та які саме критерії були враховані при обранні цієї технології.
- Детальний опис протоколів та стандартів, що використовуються для функціонування WebRTC.

- Побудова застосунку відеоконференцій, де користувачі, знаходячись у віртуальних кімнатах, матимуть змогу спілкуватися у реальному часі
- Введення функціоналу динамічного використання аудіо- та відеопотоків під час проведення відеоконференцій
- Введення додаткового шару шифрування за допомогою модуля WebRTC Insertable Streams

Розділ 1. Дослідження та аналіз предметної області

1.1. Потокова трансляція відео

Відеотрансляція – тривала безперервна передача відео та аудіо файлів зі сторони сервера до клієнта. На відміну від скачування мультимедійних файлів, коли повна копія файлу зберігається в постійній пам’яті пристрою клієнта й його не можна програвати допоки він не буде скачаний повністю, файл переданий потоковою службою не зберігається на пристрої. Мультимедійний файл поділяється на сегменти, обгорнуті в оболонки транспортних контейнерів, що передаються поступово із серверу до клієнта, де в інтернет-браузері (або в іншому застосунку де наявний відеопрогравач) приймається цей потік пакетів даних та інтерпретується як суцільний мультимедійний файл.

Є декілька ключових характеристик, які допоможуть зрозуміти, на що саме звертати увагу при підборі доцільного формату відеотрансляції[2]:

Величина аудиторії:

- Зв’язок точка з точкою (point-to-point) передбачає двох учасників, що можуть як споглядати так й транслювати медіафайли
- Групова передача (multicast) передбачає більше двох учасників, котрі можуть мати ролі як передавача, так й споглядача
- Широкомовна передача передбачає багатьох учасників, найчастіше більше тисячі, один з яких транслює відео-контент, а інші споглядають

Час, коли відбувається кодування медіафайлу:

- Кодування наживо
- Завчасне кодування, збереження на медіасервері

Взаємодія:

- Інтерактивні
- Не інтерактивні

Аналізуючи формат відеоконференції, треба зазначити, що вибрана технологія для її створення повинна підтримувати групову передачу, кодування медіафайлів в реальному часі та інтерактивність.

1.2. Технології для організації відеоконференцій у веб-застосунках

Критичним для формату відеоконференції є значення затримки між тим, як камера захоплює картинку або мікрофон, відповідно, звук, та відтворенням медіаконтенту на стороні співбесідника. Чим менше затримка, тим краще, але існує певний поріг приблизно в одній секунді, перехід в більші значення призведе до досить відчутного дискомфорту для користувачів при спробі вести нормальну розмову. Було розглянуто категорії технологій за підтримуваною затримкою:

- Звичайна затримка для технологій базованих на HTTP (від 18 до 45+ секунд) – до цієї категорії технології:
  - Apple HLS (HTTP Live Streaming)
  - MPEG(Moving Picture Experts Group)
  - DASH(Dynamic Adaptive Streaming over HTTP)

- Знижена затримка (від 5 секунд):
  - оптимізований HLS
  - оптимізований DASH
  - RTMP(Real Time Messaging Protocol)
- Низька затримка (нижній поріг - близько 1 секунди):
  - LL-HLS(Low Latency HLS)
  - LL-CMAF (Low Latency Common Media Application Format) для DASH

    - оптимізований RTMP
    - SRT(Secure Reliable Transport)
    - RTSP(Real Time Streaming Protocol)/RTP(Real-time Transport Protocol)
- Майже без затримки (near real-time) – єдиним представником цієї категорії є WebRTC(Web Real Time Communications) із середньою затримкою у 500 мс

Зважаючи на потребу у мінімальній затримці, у цій курсові було технологію з низькою затримкою – WebRTC.

Наступним критерієм аналізу технологій – це їхній показник розповсюдженості, бо яким би не була технологія, без її широкої підтримки мережею розповсюдження контенту (CDN) неможливе нормальне масштабування програмного продукту[3]. За графіком розповсюдженості форматів відеотрансляцій за 2019 рік (див. додаток А) було розглянуто перших чотирьох лідерів на ринку:

- HLS (45.18%)
- RTMP(33.13%)
- MPEG-DASH(6.93%)
- WebRTC(4.82%)

Спочатку було проаналізовано технологію RTMP. Пропрієтарний протокол від компанії Adobe був створений для передачі мультимедійних файлів від веб-серверу до Flash Player. Крім Flash, у ролі клієнта можна використовувати JW Player та Uppod. Компанія Adobe оголосила про те, що більше не буде підтримувати розробку Flash Player, починаючи з 1 січня 2021 року. Через це закінчується підтримка проекту Uppod. Численні CDN поступово зменшують кількість проміжних серверів на користь HLS та

MPEG-DASH. Триває тенденція переходу з RTMP на вищезгадані технології.

Технології HLS та MPEG-DASH дуже схожі за своєю манерою доставки медіа файлів, саме тому їх було порівнювано, а точніше їхні специфікації з підтримкою низької затримки LL-HLS та LL-CMAF для DASH. До появи цих нових специфікацій ці два стандарти були несумісними, тому потрібно було зберігати одне джерело у обидвох форматах, в досить схожій формі, для їхньої підтримки. У 2018 році компанія Apple випустила разом з Microsoft новий стандарт CMAF та ввела підтримку у HLS фрагментованих MP4(fMP4) транспортних контейнерів, що внаслідок робить сумісними технології HLS та MPEG-DASH[4]. Тепер достатньо мати одне джерело медіа та два різні маніфести для підтримки обох стандартів (див. схему 1). Треба зазначити, що застарілі HLS та MPEG DASH плеєри не підтримують стандарт CMAF, через неможливість використання ними фрагментованих MP4 контейнерів.

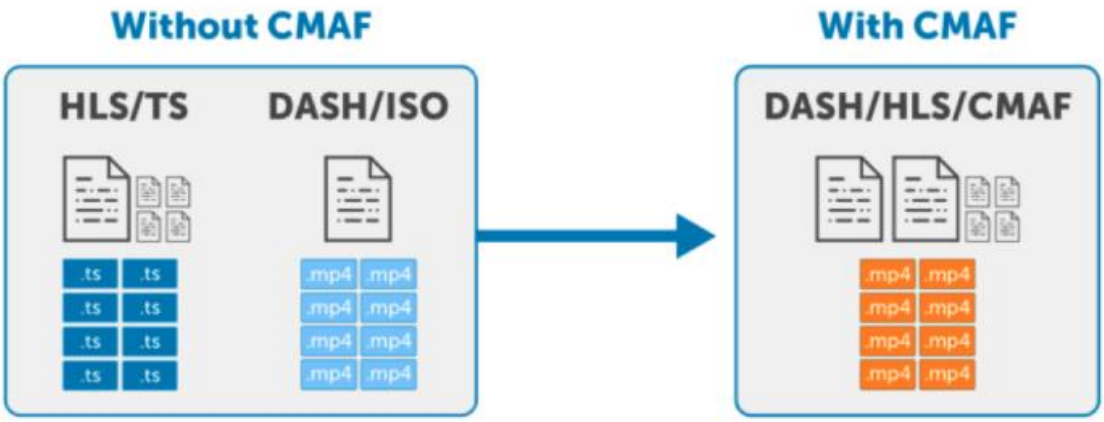


Схема 1

Враховуючи сумісність обох технологій, остаточним фактором вибору між ними стала значна популярність технології HLS(див. додаток А). Тож було прийнято рішення надалі у цій роботі розглядати два способи доставки

контенту – LL-HLS та WebRTC як технології доставки контенту, що найкраще підійдуть для розробки відеоконференцій.

Важливим фактором забезпечення якості зображення є підтримка адаптивного бітрейту (ABR, adaptive bitrate). Адаптивний бітрейт надає можливість отримати найкращу якість відео для даного підключення, тобто при передачі даних враховується як якість інтернет зв'язку, так й пропускну здатність самого девайсу, розширення екрану, завантаженість серверів[5]. Надання такої гнучкості при передачі мультимедійних даних досягається завдяки транскодуванню. Транскодування – процес декомпресії та/або декодування медіафайлу з метою його маніпуляції (траснсрейтингу або трансайзінгу). Трансрейтинг – зміна бітрейту файлу з метою підлаштування під швидкість інтернет-підключення, воно також може включати зміну кадрової частоти або розширення зображення. Трансайзінг – зміна розширення зображення заради підлаштування під різні розміри екранів. Формат HLS повністю підтримує адаптивний бітрейт. Якщо говорити про WebRTC, то адаптивний бітрейт там наявний, але наразі існує з певною особливістю: можна зіткнутися із ситуацією, коли при слабкому інтернет-зв'язку одного учасника, якість відео погіршується для всіх учасників. Обидві технології підтримують сучасні відео- та аудіо-кодеки для програвання файлів з високою якістю, але різниця в тому, що WebRTC використовує лише кодеки з відкритим вихідним кодом (VP8, VP9, H.264 – відео-кодеки; Opus, iSAC, iLBC – аудіо-кодеки).

Відносно безпеки, в для обох технологій можливо створити безпечний зв'язок. Різниця полягає лише в тому, що специфікація LL-HLS отримала від своєї попередниці можливість цифрового захисту прав (DRM, digital rights management), автентифікації за токенами, авторизації тощо. Хоча треба зазначити, поки що не всі CDN провайдери підтримують такий

функціонал в LL-HLS. До WebRTC цей функціонал також можна додати, але цим треба зайнятися самостійно.

Поки що існує міф, що з технологією WebRTC не можна масштабуватися до дійсно великих аудиторій, але це не так[6]. Найчастіше всього відбувається підключення напряму один до одного для мінімізації затримки. Щоб підтримувати аудиторію до 300 учасників, можна додати до інфраструктури медіа сервер, що зменшить навантаження на канали передачі даних. Якщо потрібно розширятися до аудиторії за поміткою 300 учасників, то трансмаксінг до HLS чи MPEG-DASH допоможе вирішити цю проблему. Трансмаксінг – процес перепакування транспортних контейнерів відеосегментів з одного формату в інший без зміни контенту мультимедійного файлу. Хоча й трансмаксінг є рішенням проблеми, воно додає помітну затримку передачі відео й є менш ефективним ніж пряме використання LL-HLS, що також дозволяє розширятися до таких масштабів за допомогою CDN провайдерів, до того ж це буде помітно дешевше.

Найважливішим критерієм для застосунку відеоконференції є підтримка щонайменшої затримки. При проведенні реальних тестів значення затримки у технології LL-HLS може досягти 2-8 с[7]. На противагу цьому, затримка при передачі даних через WebRTC досягає значень до 500 мс[8]. Такий вражаючий результат перекриває всі недоліки технології описані вище. Якщо ще раз розглянути перспективу масштабування, малоймовірно, що виникне ситуація, коли треба буде підтримувати інтерактивну відеоконференцію з аудиторією більше 300 учасників, тому було прийнято рішення вибрати WebRTC як найдоцільнішу технологію для розробки відеоконференції.

1.3. Протоколи й стандарти WebRTC

Технологія WebRTC – це проект з відкритим вихідним кодом, що являє собою колекцію стандартів, протоколів та прикладних програмних інтерфейсів (API, application programming interface), визначених робочими групами, таких як W3C(World Wide Web Consortium) та IETF(Internet Engineering Task Force). Більшість цих груп включають численних учасників, що репрезентують IT-гігантів, як от: Microsoft, Google, Mozilla, Apple, Cisco, Facebook. Кожна з цих компаній впроваджує свої версії кінцевих точок WebRTC. За означенням наданим цими робочим групами[9], кінцевою точкою WebRTC – WebRTC-браузер або WebRTC-не-браузер, що відповідає специфікації з впровадження протоколів. Браузери з підтримкою WebRTC– це програмне забезпечення, що відповідає як протокольній специфікації, так й прикладному програмному інтерфейсу для мови JavaScript (JavaScript WebRTC API). Несумісний WebRTC браузер відрізняється від браузера з підтримко WebRTC тим, що він може не мати функціоналу бібліотеки WebRTC API для JavaScript. Такі кінцеві точки WebRTC можуть також називатися WebRTC пристроями чи природніми WebRTC-додатками.

Для більшої частини свого функціонування WebRTC потребує використання транспортного протоколу UDP, але не обмежується лише ним. Протокол TCP використовується для зв’язку з сигнальним сервером через HTTP/Веб-сокети або для з’єднання з TURN або STUN сервером. Підтримується за специфікацією зв’язок через інтернет протоколи версій 4 (IPv4) та версії 6 (IPv6).

Для передачі мультимедійних даних всі кінцеві точки WebRTC повинні використувавати RTP(real-time transport protocol)/RTCP(real-time transport control protocol)[10]. Для забезпечення конфіденційності при передачі

медіапотоків використовується SRTP (Secure RTP) та DTLS(datagram transport layer security). SRTP є широко використовуваним для гарантування безпечного з'єднання завдяки шифруванню корисного навантаження RTP-пакунку, залишаючи в чистому вигляді RTP-заголовок, та перевірці походження потокових джерел. DTLS використовується для шифрування потоків даних, що не є медіа, перешкоджанню прослуховування та підміну даних; він розроблений на TLS(transport layer security), що пропонує функціонал повного асиметричного шифрування та автентифікації повідомлень. Зв'язка технологій DTLS-SRTP в основному використовується для обміну ключами та запобіганню атаки типу "людина посередині" [11].

У сучасному інтернеті майже неможливо просто напряму підключитися від однієї кінцевої точки до іншої через ряд причин[12]. По-перше, існує потреба обійти мережевий екран, що перешкоджає відкриттю нових з'єднань. По-друге, оскільки множина IPv4 незайнятих унікальних публічних адрес доволі обмежена й більшість пристроїв користуються приватними адресами, то треба знайти шлях унікально ідентифікувати кожну кінцеву точку. По-третє, певні маршрутизатори налаштовані на обмеження з'єднань кінцевих точок із зовнішнім інтернетом й для з'єднання з такими точками треба використовувати посередника. Для подолання цих перешкод технологія WebRTC використовує програмний механізм ICE (interactive connectivity establishment).

Метою використання механізму ICE – знаходження такої пари варіантів підбору адрес для кінцевих точок, що дозволить відкрити підключення між ними. Для цього процесу можуть використовуватися STUN (session traversal utilities for NAT) та TURN (traversal using relays around NAT) сервери. Перетворення мережевих адрес (NAT, Network Address Translation) –

механізм у мережах IPv4, що дає можливість змінювати приватну IP-адресу у заголовку пакунку на публічну, проходячи через маршрутизатор. STUN сервер – сервер, що на запит клієнта, відповідає його публічною інтернет адресою та інформацією про те, чи з ним можуть створити з'єднання ззовні. Якщо не дозволяється створювати підключення, використовується TURN сервер. Такий сервер працює як посередник, передаючи всю інформацію що надходить до й від кінцевої точки. Спочатку налаштовується маршрутизація так, щоб можна було підключитися до серверу з кінцевої точки; створюється підключення й говориться всім іншим точкам, що хочуть під 'єднатися, щоб вони посилали всі пакунки до цього серверу, а він вже перенаправить[13]. Кінцеві точки підбирають список кандидатів, сортовані з найкращого до найгіршого. Найчастіше кандидати будуть спілкуватися за допомогою UDP, але іноді така опція неможлива або UDP обмежений таким чином, що не можна передавати медіапотоки цим протоколом. Тоді дозволяється використання ICE-TCP кандидатів (не всі кінцеві точки WebRTC підтримують такий варіант). Процес підбору ICE кандидатів доволі довгий: потрібно перевіряти кожен варіант можливого способу з'єднання, а це означає чекати результати взаємодії зі STUN та TURN серверами. Для оптимізації було створено Trickle ICE, що робить весь цей процес паралельним й кандидати відправляються по одному або групами, залежно від того, коли вони стають доступними, не очікуючи інших кандидатів в списку. На схемі(див. додаток Б) представлено візуалізацію підключення WebRTC endpoints за допомогою механізму ICE.

Для опису та узгодження сесійних параметрів, як то використовувані кодеки аудіо- та відео-, транспортні протоколи для мультимедіа, деталі шифрування даних (наприклад, «відбитки» сертифікатів використаних при DTLS-SRTP домовленостях), ICE кандидатів тощо, використовується формат SDP (Session Description Protocol). Структура SDP – текст у форматі

UTF-8, кожен рядок з якого починається на одну літеру з латинського алфавіту, що позначає певний тип параметра згідно специфікації[14], після цього слідує знак «=», а після нього структурований текст, що містить опис того, який варіант було використано для даного типу параметра.

Розділ 2. Проектування та розробка системи

2.1. Опис складових системи та використаних технологій для розробки

Було вирішено розробити відеоконференцію як веб-застосунок. Причин цьому є декілька. По-перше, використовуючи веб-браузер, а більшість (94%) сучасних веб-браузерів є WebRTC сертифікованими, нам не треба імплементувати технологію WebRTC самим, браузер уже є валідною кінцевою точкою для створення WebRTC зв'язку рівний до рівного (одноранговий). По-друге, оскільки технологія WebRTC є такою, що постійно розвивається, то завдяки особливості сучасних браузерів оновлюватися самостійно, розробнику не треба перейматися тим, що функціонал, який увійшов у стандарт, не запрацює в якомусь WebRTC-браузері. По-третє, JavaScript WebRTC API значно спрощує й ушвидшує розробку, а також існують багато електронних джерел[15] та книг[16], що детально пояснюють як працювати з цим прикладним програмним інтерфейсом.

Застосунок складається з 2 частин: клієнтської частини та сигнального серверу. Клієнтська частина була розроблена за допомогою JavaScript-фреймворку Angular. Він є одним з найпопулярніших фреймворків для розробки клієнтської частини у веб-середовищі на підприємстві й саме тому був обраний, щоб наочно довести, що рішення з WebRTC можна вже зараз інтегрувати у серйозні бізнес-рішення. У Angular використовується мова програмування TypeScript. TypeScript є надбудовою JavaScript, яка привносить статичну типізацію та анотацію, підтримку традиційної версії об'єктно-орієнтованої парадигми, модулярності, та TypeScript повністю сумісна з JavaScript. За останні роки все більша кількість людей мігрує до TypeScript та використовує її на виробництві. Сигнальний сервер був розроблений на програмній платформі Node.js, використовуючи мову

TypeScript, фреймворк для розробки веб-серверів Express.js та обгорткою до WebSockets, Socket.IO.

JavaScript WebRTC API поділяється на три логічних модуля:

- RTCPeerConnection займається з’єднанням учасників(пірів) та перемовинами щодо того, які дані та в якому форматі будуть передаватися
- RTCDataChannel відповідає за передачу немультимедійних даних
- MediaStream – регулювання роботи мультимедійних потоків

Треба зазначити, що у цьому дослідженні на етапі розробки було зроблено спробу використати замість чистого WebRTC API обгортку над цим прикладним програмним інтерфейсом у вигляді npm-модуля(Node Package Manager, менеджер пакунків за замовчуванням для платформи Node.js) simple-peer[17]. Перевагою цього модуля є декларативний інтерфейс та зменшення кількості коду, який потрібно написати. Серед недоліків є нестабільна робота функціоналу цього модуля та досить незручний спосіб (або взагалі відсутність такого варіанту) досягти полів об’єктів заданих у WebRTC API та виклику функціоналу, що ще не було обгорнуто авторами цього пакунку. Відчувши необхідність працювати на глибшому рівні з WebRTC API, було прийнято рішення перестати користуватися цим модулем. Пакунок за своїм програмним інтерфейсом дуже схожий за концепцією з WebRTC API й тому немає сенсу детально описувати принцип роботи з ним, але під час опису наступних підрозділів буде згадано деякі ключові відмінності у роботі з цим модулем.

2.2. Налагодження мережевої взаємодії між учасниками

Для нашої відеоконференції було взято та використано концепцію віртуальних кімнат, що надаються прикладним інтерфейсом Socket.IO.

Учасники конференції з’єднуються за певним, попередньо відомим, ідентифікатором, за допомогою якого їх перенаправляють в ізольоване від інших користувачів середовище, де вони за допомогою SDP обмінюються інформацією про те, в якому саме форматі відбуватиметься їхнє спілкування. На самому початку створюється з’єднання із сигнальним сервером (див. додаток В). Робота сигнального серверу полягає в тому, що він слідкує за створенням з’єднань з ним, підключенням так званих сокетів, чекає на команди, які сокети йому надсилають, оброблює їх, а також стежить коли якесь з’єднання закінчується або його розірвано. Вказуючи назву кімнати, в яку користувач має намір додатися, зі сторони клієнта надсилається команда додати його до відповідної кімнати. На стороні сервера, отримавши таку команду, користувача додають до заданої кімнати, після того збирають список всіх інших учасників, що перебувають в даній кімнаті, та надсилають до користувача, що надіслав запит.

Отримавши список, для даного користувача створиться із кожним зі списку, крім нього, одноранговий зв’язок(див. додаток Г). Для цього спочатку ініціалізується новий екземпляр класу RTCPeerConnection із конфігурацією, де вказано посилання на STUN сервер, за допомогою якого можна буде звернутися за публічною IP-адресою користувача. Екземпляр класу RTCPeerConnection представляє собою з’єднання локального користувача з віддаленим. Це центральний об’єкт, через який йде підключення, налаштування та моніторінг зв’язку. Для зручності подальшого користування цей об’єкт разом із ідентифікаційним номером віддаленого користувача зберігатиметься в обгортці об’єкті класу VideoRTCPeer (детальніше про призначення буде описано далі), який в свою чергу зберігатиметься в пам’яті веб-браузера.

Надалі треба вказати, яким же чином потрібно реагувати на відповідні події. Вказувати функцію у відповідь на події можна одним з двох способів, що є рівноправними між собою: обробником подій або слухачем подій. Було вибрано використовувати у застосунку обробник подій. На даний момент буде розглянуто лише події, що напряму пов'язані з підключенням: icecandidate та negotiationneeded. Механізм ICE, впроваджений в браузері, вибирає контролюючого агента серед двох пірів. Цей агент остаточно вибирає пару ICE-кандидатів, через які будуть здійснені підключення. Провокується подія icecandidate, що містить кандидата чи список кандидатів, якого потрібно передати через сигнальний сервер до віддаленого підконтрольного агента, де кандидат записується до з'єднання. За одну таку сесію контролюючий агент може здійснити передачу більше одного разу. Кожного разу обидва корегують свою конфігурацію, щоб використовувати актуальну пару кандидатів. Подія negotiationneeded з'являється, коли треба почати нові перемовини щодо передачі нових потоків даних, найчастіше тут йдеться про медіа. У функції, що оброблює цю подію, треба створити пропозицію (SDP offer). Якщо за логікою роботи програми передбачено, що учасник може на початку не передавати ні аудіо, ні відео потоки, то потрібно явно створити пропозицію(див. додаток Г).

SDP offer створюється завдяки методу екземпляра RTCPeerConnection createOffer(див. додаток Г). Результат цієї функції передається методу setLocalDescription цього ж самого екземпляра. Бувають дві стадії затвердження описів SDP: поточний(затверджений) та незатверджений. Оскільки під час перемовин пропозицію може бути відхилено через несумісність форматів, то потрібно продовжити ці перемовини до повної двосторонньої узгодженості. Отже, поточний варіант відповідає реаліям того, як передаються дані між користувачами; якщо ще не було узгоджено жодного разу, то значення затвердженого варіанту дорівнює null. При

виклику setLocalDescription та setRemoteDescription, опис вимог позначується як локальний або віддалений відповідно й як такий, що очікує результатів перемовин. Щойно буде затверджено цей опис, поточний опис змінить своє значення та значення незатвердженого опису дорівнюватиме null. Після виклику методу setLocalDescription, розпочинається генерування ICE кандидатів.

Затвердивши локальний опис, його відправляють через сигнальний сервер до віддаленого користувача. Отримавши його, перевіряється чи вже було встановлено з'єднання з тим, хто надіслав пропозицію. Якщо ще не було встановлено, то відбувається процес створення екземпляру класу RTCPeerConnection та VideoRTCPeer і так само встановлюються обробники подій, описаних раніше(див. додаток Ґ). В обох випадках створюється відповідь на дану пропозицію. Процес починається з того, що викликається метод екземпляру RTCPeerConnection, який керує зв'язком того локального користувача з даним віддаленим, setRemoteDescription. Для спрощення програмного інтерфейсу WebRTC й щоб не треба було розрізняти два варіанти коду в залежності від того хто започаткував зв'язок, виклик цього метода ставить поточного користувача в роль того, хто приймає зв'язок, навіть якщо раніше було навпаки. Якщо опис містить несумісні формати або інші суперечності, з'являється потреба у новому раунді перемовин і провокується подія negotiationneeded. Якщо немає суперечок, після завершення роботи цього методу, викликається метод створення опису у відповідь(SDP answer). Далі викликається метод setLocalDescription й в кінці відправляється відповідь через сигнальний сервер. Зі сторони локального користувача (початківця зв'язку) отримується надана відповідь та встановлюється опис віддаленого користувача методом setRemoteDescription. Зв'язок успішно встановлено(див. додаток Г).

При роботі з npm-модулем simple-peer, вищеописаний процес концептуально не відрізняється, але прикладний програмний інтерфейс більш спрощений та деякі ключові рішення були приховані та стали недоступними для розробника. По-перше, тепер відбувається лише обробка подій icecandidate та negotiationneeded, а назамовні signal та connect. Створюються неявно пропозиції та відповіді на них. Готові обгортки SDP описів, які треба відправити до віддаленого користувача, отримуються у разі виявлення події signal. Спливання події connect повідомляє, що учасники успішно встановили з'єднання. По-друге, при створенні екземпляру класу модульної обгортки над RTCPeerConnection треба вказувати чи дана сторона буде ініціатором зв'язку. У WebRTC API без модульних обгорток роль сторони явно не вказується та є динамічною, оскільки використання методу екземпляру RTCPeerConnection setRemoteDescription міняє цю роль. Ці видозміни у модульній обгортці над WebRTC API спростили логіку функціоналу, яким повинен оперувати розробник, але з іншої сторони при нетривіальній роботі з технологією WebRTC це спрощення приведе до того, що дії розробника буде настільки лімітовано, що він не зможе реалізувати певний функціонал, який доступний у WebRTC API.

2.3. Робота з каналом передачі даних RTCDataChannel

Хоча в попередньому підрозділі було описано дієвий механізм WebRTC однорангового підключення учасників, з'єднання не встановлювало ніякі варіанти передачі даних. Таке з'єднання не є корисним, тому треба додати та налаштувати такі канали. RTCDataChannel надає такий функціонал.

RTCDataChannel репрезентує двосторонній канал для передачі будь-якого формату даних, але саме тут не йдеться про файли аудіо та відео. Кожен такий канал прив'язаний до певного з'єднання й таких каналів може бути

теоретично до 65,534 одиниць на одне з’єднання, хоча реальний ліміт може відрізнятися від браузера до браузера.

Процес створення такого каналу зв’язку полягає в наступному: один з учасників (у випадку даного застосунку, та сторона, що ініціалізує одноранговий зв’язок перед тим як надіслати SDP пропозицію) створює канал даних та запрошує віддаленого користувача приєднатися до новоствореного каналу(див. додаток Д). Для створення нового каналу використовується метод екземпляру RTCPeerConnection createDataChannel та передається як параметр назви, що буде асоційовано з цим каналом. Метод повертає екземпляр класу RTCDataChannel, посилання на якого потім призначено полю dataChannel екземпляру класу обгортки над VideoRTCPeer. Останнє зроблено для доступу до каналу, щоб надсилати повідомлення через нього.

Для підтримки роботи RTCDataChannel, налаштовано оброблювати ще 3 події: datachannel, open, message(див. додаток Д). При появі першої події, віддалена сторона бере посилання на канал, переданий подією, а потім призначається полю dataChannel екземпляру класу обгортки над VideoRTCPeer. Для більш симетричного (будь-яка з двох сторін може розпочати канал) підключення каналу даних, обидва учасники можуть вказати не лише назву каналу, а ще й передати завчасно домовлений ідентифікатор та вказати, що треба провести симетричні перемовини. Подія open з’являється, коли канал повністю готовий до передачі даних, а message – при появі нового повідомлення від іншої сторони. Якщо встановлення зв’язку через канал даних відбуватиметься опісля початкового узгодження підключення учасників, то подія negotiationneeded з’явиться задля проведення нового раунду узгоджень.

Відмінність при роботі з npm-модулем simple-peer, полягає в тому, що модуль-обгортка автоматично створює при підключенні один канал даних. Відпрацювання раніше згаданої події connect також повідомляє про те, що канал даних доступний для використання. А подія data, що в модулі використовується, є еквівалентною події message.

2.4. Використання MediaStreams та динамічне використання медіапотоків

Динамічне використання медіапотоків означає можливість в будь-який момент часу за наявності успішного встановленого підключення учасників мати змогу почати та закінчувати трансляцію потоків аудіо та відео. Завдяки модулю MediaStreams прикладного програмного інтерфейсу WebRTC конфігурування динамічного використання медіапотоків не є складною задачею. Особливо коли є можливість передавати потік безпосередньо з сегментами відео у HTML5 тег video без використання додаткових модулів чи плагінів. Отримати потік медіа можливо завдяки екземпляру класу window.navigator, який собою презентує стан та конфігурацію браузерного агента, що в свою чергу дає доступ до поля mediaDevices, який методами для читання дає доступ до підключених приладів вводу медіа, таких як камера чи мікрофон або механізм поширення екрану. Метод mediaDevices getUserMedia надає доступ до камери та мікрофону, а getDisplayMedia – передає, що відбувається на екрані.

Використовуючи один з цих методів нам повертається медіапотік, що є екземпляром класу MediaStream. Медіапотік складається зі списку доріжок, екземпляри класу MediaStreamTrack. Так побудована абстракція, що не важливо який формат медіа, аудіо чи відео, транслює трек, усі доріжки додаються та видаляються однаково. Процес підтримки передачі медіапотоку спирається на тому, що при зміні (додаванні/видаленні) доріжки в списку зареєстрованих в уже стабілізованому перемовинами

описі SDP, з’являється подія negotiationneeded й новими перемовинами регулюється зміна. Отже, алгоритм динамічного використання потоків у WebRTC не змінюється від того коли саме було включено/виключено доріжку. Хоча треба зазначити, що при умові, що щоразу медіа буде передаватися з самого початку, то можна явно не прописувати створення пропозиції, оскільки при додаванні доріжки з’явиться подія negotiationneeded, яка оброблюється з використанням проведенням SDP перемовин.

Зараз детально буде розглянуто лише додавання/видалення зі зв’язку відеодоріжок, оскільки вже було сказано, що механізм даної роботи з аудіо-чи відеодоріжками однаковий. У описаній системі транслюється відео захоплене з підключеної чи вбудованої відеокамери. Викликаючи метод getUserMedia екземпляру класу MediaDevices, повертається результатом потік з відеодоріжкою,встановлюємо його джерелом відео (srcObject) для HTML5 тегу video та починаємо програвати своє відео методом play(див. додаток Е). Після того додаємо до кожного наявного з даним користувачем зв’язку тільки що отриманий відеодоріжка, при тому зберігаємо результат, отриманий викликом функції, у масиві senders екземпляру кастомного класу VideoRTCPeer. Результатом виконання цих операцій є екземпляр класу RTCRtpSender, що містить інформацію про те, яким саме чином доріжка кодована та відправлена до віддаленої сторони. Його потрібно буде передати як параметр за бажання видалити доріжку зі сеансу зв’язку.

Із віддаленої сторони треба обробити подію track, що з’являється щоразу, коли до зв’язку додається нова доріжка. Якщо ще не існує екземпляру класу MediaStream, що огортає усі доріжки (аудіо та відео), що надходять від іншої сторони, то треба його створити за допомогою пустого конструктора, а потім призначити джерелом медіа в тезі video, що транслює медіадані від

іншої сторони та почати їхнє програвання. Потім треба додати відправлену доріжку до цього потоку(див. додаток Е).

Для того, щоб видалити відеодоріжкузі зв’язку, потрібно для кожного наявного з даним користувачем підключення знайти екземпляр класу RTCRtpSender, що передає шукану доріжку; передати його як параметр метода RTCPeerConnection removeTrack, а потім видалити посилання на екземпляр класу RTCRtpSender з масиву senders екземпляру кастомного класу VideoRTCPeer. Після цього треба зупинити відеодоріжку у локальному video тезі та зняти цю доріжку з відповідного потоку медіа за допомогою метода екземпляру класу MediaStream removeTrack, передаючи посилання на доріжку як параметр(див. додаток Е).

У результаті роботи з npm-модулем simple-peer було отримано незадовільні результати функціоналу динамічного використання медіапотоків. Якщо на початку зв’язку, передаючи у конструктор обгортки над RTCPeerConnection потік з медіа, то трансляція цього потоку буде успішною. На версії 9.7.2 npm-модуля, що була останньою доступною версією станом на жовтень 2020 року, коли проходили дослідження з модулем, якщо використати програмний інтерфейс для додавання/видалення доріжок у зв’язку, то з’являється подія signal з обгорткою над SDP описом, що говорила про те, що треба почати з нуля перемовини й цей процес ніколи не закінчувався. Аналогічного механізму методу RTCPeerConnection createOffer в модулі-обгортці simple-peer не існує, а також все сховано в самому модулі, тому не було можливості доступитися до самого екземпляру класу RTCPeerConnection. Під час подальшого дослідження було виявлено можливість того, щоб ініціатор зв’язку міг динамічно включати/виключати відео, але ніхто, окрім нього, не може так робити. Після додатково проведеного аналізу, було знайдено, що проблема поширена і наразі не має

рішення. Зрозумівши, що проблема присутня саме в обгортці, було вирішено перейти повністю до WebRTC API без модулів-обгорток.

2.5. Шифрування потокового медіа з використанням Insertable Streams

28 лютого 2020 року команда розробників Chrome браузеру анонсувала[18] свої плани з розробки нового модуля прикладного програмного інтерфейсу WebRTC Insertable Streams. Головним функціоналом цього API – надати доступ розробникам до даних, які проходять через потоки, щоб можна було їх обробляти специфічним чином перед тим як їх відправляти. Одним із варіантів використання є додатковий шар шифрування медіапотоків з використанням технології фонової обробки за допомогою Web Workers API. Було прийняте рішення впровадити такий функціонал у застосунок для показу можливостей цього програмного інтерфейсу, а також створення додаткового прошарку безпеки, що протидіятиме атакам типу людина-посеред. Слід зазначити, що на момент проведення досліджень, а саме у період з жовтня по грудень 2020 року, Insertable Streams API вже було включено до специфікації Chrome браузеру з виходом версії 86, але технологія все ще перебувала в експериментальному статусі та ще не підтримувалась більшістю браузерів що підтримують WebRTC.

Перед демонстрацією прикладу роботи яким чином відбувається робота з Insertable Streams API у контексті даного застосунку, треба ознайомитися з поняттям веб-працівників. Це механізм, що дозволяє певному скрипту виконувати ресурсномісткі перетворення, які би блокували на певний час головний потік виконання скрипту, що повинен оброблювати насамперед ввід користувача, у фоновому режимі. Обчислення в цих скриптах можуть бути довільного характеру, але через специфіку виконання задачі не в головному потоці, стає недоступною можливість маніпулювати DOM(document object model) веб-сторінки. Для того, щоб передавати дані

до веб-працівника існує система сповіщень: між веб-працівниками обмін сповіщеннями відбувається за допомогою метода екземпляру класу Worker postMessage, а приймаються за допомогою обробника подій onmessage.

Програмний інтерфейс Insertable Streams використовує для своєї дії відправників (екземпляр класу RTCRtpSender) та отримувачів (екземпляр класу RTCRtpReceiver). Побудова функціоналу додаткового шифрування буде базуватися саме на здобутках розглянутих в попередніх підрозділах, а механізм роботи Insertable Streams буде лише доповнюючим кроком у конвеєрі. Раніше вже було зазначено, за що відповідає клас RTCRtpSender, визначимо роль класу RTCRtpReceiver. Екземпляри цього класу відповідають за отримання та дешифрування даних MediaStreamTrack. Станом на травень 2020 цей програмний інтерфейс Insertable Streams не є затвердженим у стандарті WebRTC та не є широко впровадженим у більшості WebRTC браузерів[19], необхідно перевіряти чи даний браузерний агент підтримує цю технологію чи ні. Для цього треба перевірити чи існує у екземпляра класу RTCRtpSender метод createEncodedStreams та перевірити чи можна створити екземпляр класу ReadableStream, який буде передаватися веб-працівнику для обробки(див. додаток Є).

Для включення функціоналу, що відкриває нам програмний інтерфейс Insertable Streams, треба додати опцію у конфігурації конструктора RTCPeerConnection, яка називається encodedInsertableStreams. Для здійснення шифрування доріжок з медіа потрібно у приватному методі addTracksToConnections(див. додаток Е), перед тим як додавати до списку відправників, використати отриманий результат виклику методу addTrack - змінну sender (див. додаток Ж). Створюємо екземпляр класу TransformStream, викликаючи метод екземпляру класу RTCRtpReceiver

createEncodedStreams. Розділяється TransformStream на два потоки для читання та запису й передаються сповіщенням до веб-працівника з командою зашифрувати дані. Для дешифрування даних треба у обробнику подій track використати отримувач, що переданий разом із подією (див. додаток Ж). Так само як із відправником створюємо екземпляр класу TransformStream, викликаючи метод екземпляру класу RTCRtpReceiver. Розділяється TransformStream на два потоки для читання та запису й передаються сповіщенням до веб-працівника з командою дешифрувати дані.

Веб-працівник влаштований таким чином, що в залежності від типу команди (шифрування або дешифрування) він спеціальною функцією трансформує кадр за кадром потік, читаючи з одного потоку та записуючи в інший. Під час дослідження роботи Insertable Streams було зроблено демонстрацію, де було можливо змінити ключ шифрування під час вже стабільного підключення. Для цього було введено текстове поле, у яке користувач вводить ключ. При кожній зміні значення ключа передається сповіщенням новий ключ до веб-працівника(див. додаток Ж). Користувачі зможуть побачити та почути одне одного тільки тоді, коли співпадатимуть їхні ключі для шифрування.

Були виявлені доволі цікаві результати при тестуванні цього функціоналу. Наприклад, при приєднанні користувача з додатку, що не підтримує функціонал Insertable Streams, у кімнату де з’єднанні користувачі, додаток яких підтримує та використовує Insertable Streams, користувач побачить та почує лише шум(див. додаток З). Виявлено на момент дослідження проблему несумісності роботи шифрування в різних браузерах[20] (наприклад браузеру Chrome та Firefox). У схожій ситуації, де користувач мобільної версії шифрується, а користувач настільної версії не робить

цього, відео не проходить й шум не з’являється, хоча в ідентичному сценарії з двома користувачами настільної версії, все проходить, як й очікувалося. Це ознака того, що поки інтерфейс Insertable Streams потребує подальшого тестування, але є незаперечним той факт, що даний API виявився корисним та гнучким й незабаром ввійде у стандарт.

Висновки

Було досліджено методи інтернет-трансляції мультимедійних даних та порівняно різні технології для використання у відеоконференційному застосунку. У цій роботі було використано технологію WebRTC, адже, на час написання роботи, на думку автора, вона є найдоцільнішою для використання . Було детальніше розглянуто протоколи та стандарти, що використовуються для роботи механізмів WebRTC. На прикладі відеоконференційного застосунку було показано використання прикладного програмного інтерфейсу, що надається WebRTC браузерами. Побудовано застосунок, в якому динамічно використовуються медіапотоки та розроблений додатковий шар шифрування за допомогою Insertable Streams API. У майбутніх дослідженнях може розглядатися впровадження медіа серверу в інфраструктуру задля можливості масштабування продукту та зменшення навантаження на канали передачі даних.

Список використаних джерел

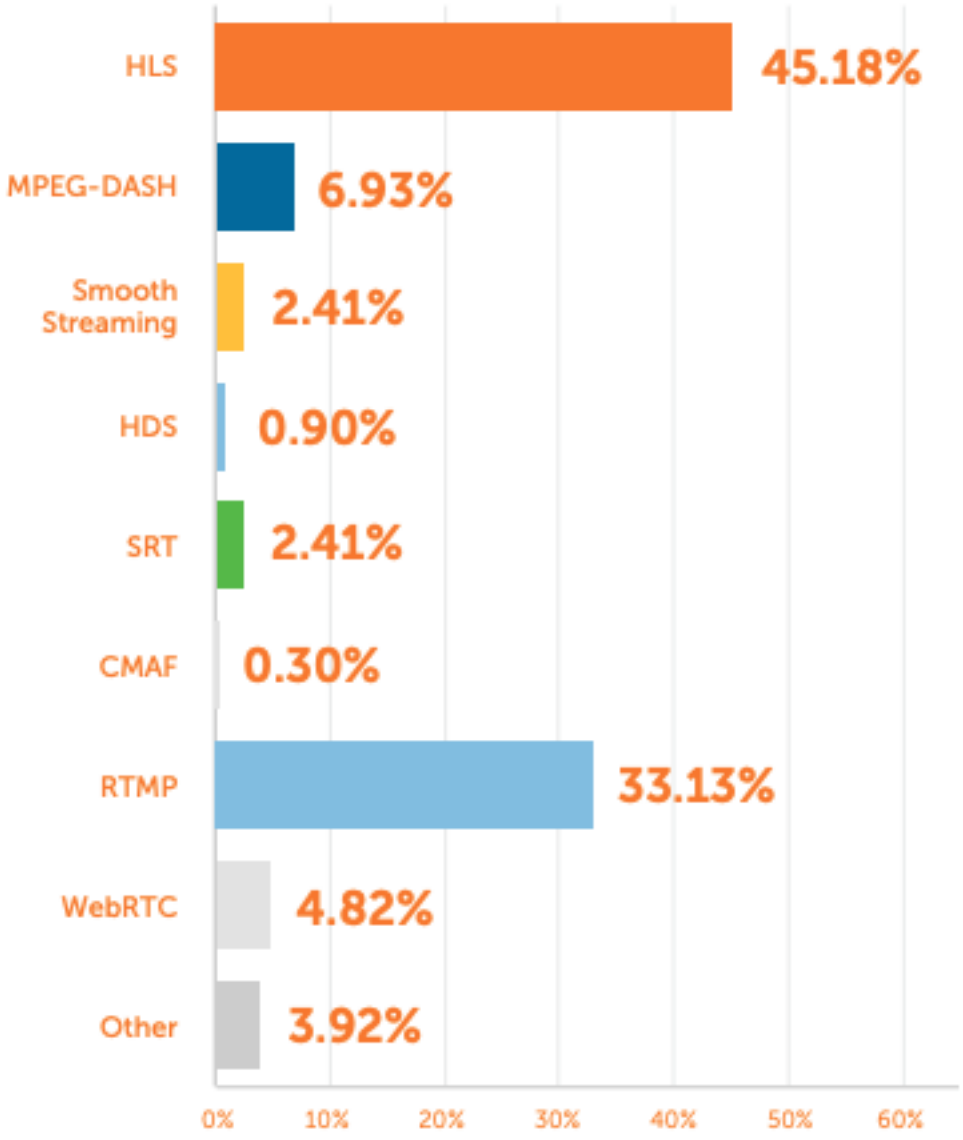
1. Саломі П. Вплив COVID-19 на технологічний сектор [Електронний ресурс] / Паул Саломі // Deloitte. – 2020. – Режим доступу до ресурсу: https://www2.deloitte.com/global/en/pages/about-deloitte/articles/covid-19/understanding-covid-19-s-impact-on-the-technology-sector-.html.
2. Апостопоулос Д. Відеотрансляції: концепти, алгоритми та системи [Електронний ресурс] / Д. Апостопоулос, В. Тан, С. Уии // HP Laboratories Palo Alto. – 2002. – Режим доступу до ресурсу: https://www.hpl.hp.com/techreports/2002/HPL-2002-260.pdf.
3. Використання CDN для відеотрансляцій наживо для поліпшення показника QoS. Дослідження кейсу: провайдер 1231 / [В. Шабрина, Д. Судіхарто, Е. Аріянто та ін.]. // Journal of Communications. – 2020. – №4.
4. Лоу В. Використання потокової передачі з наднизькою затримкою разом з CMAF [Електронний ресурс] / В. Лоу – Режим доступу до ресурсу: https://www.akamai.com/us/en/multimedia/documents/white-paper/low-latency-streaming-cmaf-whitepaper.pdf.
5. Огляд якості користувацького досвіду адаптивної потокової передачі HTTP / [М. Соуферт, С. Еггер, М. Сланіна та ін.]. // IEEE Communications Surveys & Tutorials. – 2014. – №1. – С. 469–492.
6. Робінсон Д. Мережа розповсюдження контенту: фундаменти, дизайн та еволюція / Д. Робінсон. – Хобокен: Wiley, 2017. – 171 с. – (1).
7. Знайомство з Low-Latency HLS [Електронний ресурс] // WWDC. – 2019. – Режим доступу до ресурсу: https://developer.apple.com/videos/play/wwdc2019/502/.

8. Балістрері А. Low-Latency HLS vs. WebRTC: що треба знати перед тим, як вибрати протокол [Електронний ресурс] / Анна Балістрері // Wowza Media Systems. – 2020. – Режим доступу до ресурсу: https://www.wowza.com/blog/low-latency-hls-vs-webrtc.
9. Альвестранд Х. Огляд протоколів взамодії в реальному часі для додатків, що базуються на браузерах [Електронний ресурс] / Х. Альвестранд // Network Working Group. – 2017. – Режим доступу до ресурсу: https://tools.ietf.org/html/draft-ietf-rtcweb-overview-19.
10. Перкінс С. Web Real-Time Communication (WebRTC): протоколи медіатранспорту та використання RTP [Електронний ресурс] / С. Перкінс // RTCWEB Working Group. – 2016. – Режим доступу до ресурсу: https://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage-26#section-4.1.
11. Рескорла Е. Архітектура безпеки WebRTC [Електронний ресурс] / Е. Рескорла // RTCWEB. – 2019. – Режим доступу до ресурсу: https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-20.
12. Безпека Peer-to-Peer [Електронний ресурс] // Harvard University – Режим доступу до ресурсу: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.1830&rep=rep1&type=pdf.
13. Джонстон А. Використовуючи WebRTC на підприємстві / А. Джонстон, Д. Йоакум, К. Сінх. // IEEE Communications Magazine. – 2013. – №4. – С. 48–54.
14. Хендлі М. SDP: Session Description Protocol [Електронний ресурс] / М. Хендлі // Network Working Group. – 2006. – Режим доступу до ресурсу: https://tools.ietf.org/html/rfc4566.
15. Дженнінгс К. WebRTC 1.0: спілкування в реальному часі між браузерами [Електронний ресурс] / К. Дженнінгс, Х. Бостром, Ж. Бруарой // W3C – Режим доступу до ресурсу: https://www.w3.org/TR/webrtc/.
16. Григорик І. Високопродуктивна браузерна взаємодія в мережі / Ілля Григорик., 2013.
17. npm-модуль simple-peer: документація [Електронний ресурс]. Режим доступу до ресурсу: https://www.npmjs.com/package/simple-peer.
18. Урданета Г. Намір на експеримент: WebRTC Insertable Streams [Електронний ресурс] / Гідо Урданета. – 2020. – Режим доступу до ресурсу: https://groups.google.com/a/chromium.org/g/blink-dev/c/Oy84pXDhajI/m/lu-Z0p3QAAAJ?pli=1.
19. Статус впровадження програмного інтерфейсу Insertable Streams [Електронний ресурс]. Режим доступу до ресурсу: https://www.chromestatus.com/feature/6321945865879552#details.
20. Підтримка програмного інтерфейсу Insertable Streams [Електронний ресурс]. Режим доступу до ресурсу: https://bugzilla.mozilla.org/show_bug.cgi?id=1631263.

Додатки

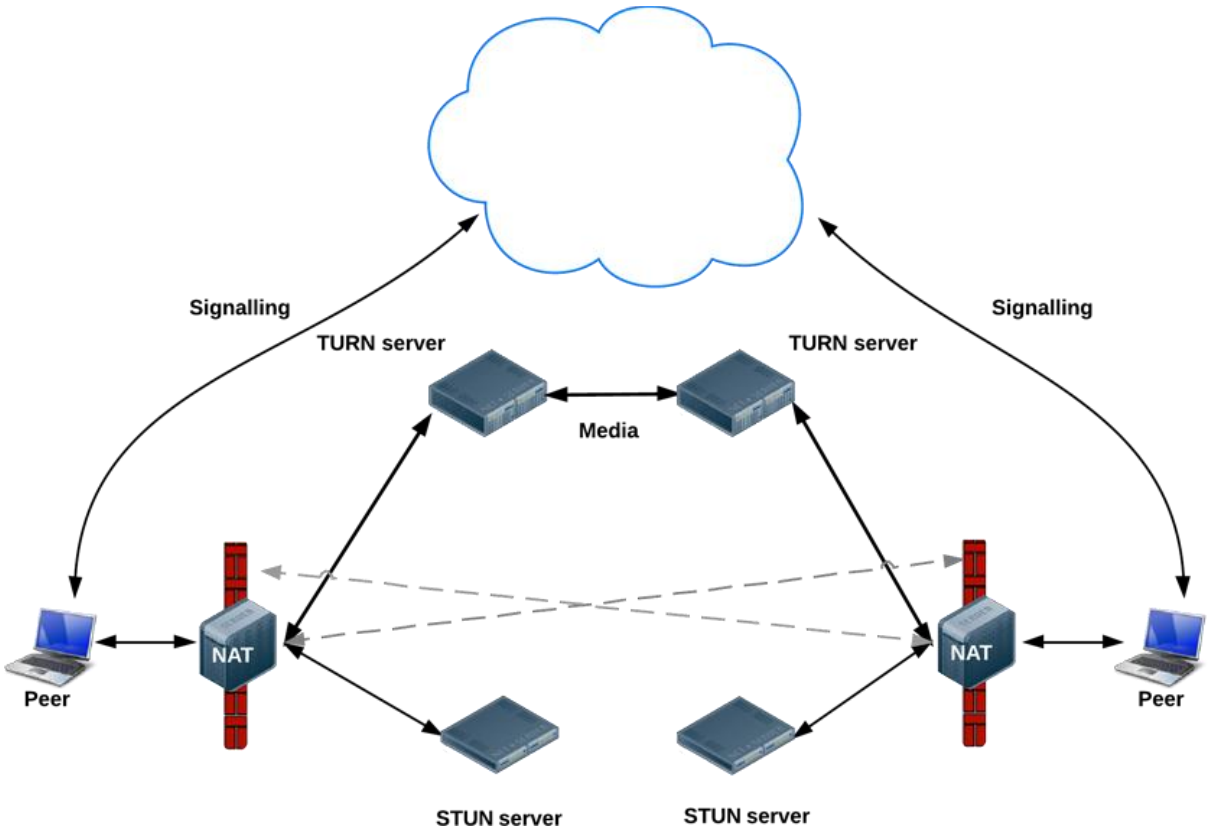
Додаток А

Графік розповсюдженості форматів відеотрансляцій на 2019 рік[3].



Додаток Б

Схема варіацій підключення двох кінцевих точок WebRTC за допомогою механізму ICE



Додаток В

Фрагмент коду, що демонструє спілкування клієнтської частини з сигнальним сервером

Клієнтська частина:

```
ngOnInit(): void {
// створюється з'єднання з сигнальним сервером
  this.webSocketService.connect(this.port);

// надсилається команда приєднання до конкретної кімнати та передається
// ідентифікатор як параметр
  this.webSocketService.emit('joinRoom', roomId);

// після підключення очікується список учасників у кімнаті, щоб з ними
// створити зв'язок
  this.webSocketService.listen('joinedUser').subscribe((participants:
Array<string>) => {

    this.initializePeerAsCaller(participants);
  });

// оброблюються пропозиції від інших учасників
  this.webSocketService.listen('madeOfferRTC').subscribe((message:
SocketDataMessage) => {

    this.answerToOffer(message);
  });

// оброблюються відповіді від інших учасників
  this.webSocketService.listen('madeAnswerRTC').subscribe((message:
SocketDataMessage) => {

    this.receivedAnswer(message);
  });

// оброблюються списки ICE-кандидатів для підключення з іншими учасниками
  this.webSocketService.listen('candidate').subscribe((message:
SocketDataMessage) => {

    this.addIceCandidate(message);
  });

// оброблюється подія виходу учасника з кімнати
  this.webSocketService.listen('userLeftRoom').subscribe((socketId: string)
=> {

    this.userLeftRoom(socketId);
  });
}
```

Серверна частина:

```
this.io.on('connection', (socket) => {
    console.log("connected %s", socket.id)
// сервер приєднує сокет до кімнати, коли надходить для цього команда
    socket.on('joinRoom', (hostId: string) => {

        socket.join(hostId, err => {
            if (!err) {

                this.io.in(hostId).clients((err, clients) => {
                    if (!err) {
                        this.io.to(socket.id).emit('joinedUser', clients)
                    }
                })
            }
        })
    })
// передача від одного учасника до другого SDP пропозиції
    socket.on('offerRTC', msg => {

        this.io.to(msg.toPeer).emit('madeOfferRTC', {
            data: msg.data,
            fromPeer: socket.id
        })
    })
// передача від одного учасника до другого відповіді на SDP пропозицію
    socket.on('answerRTC', msg => {

        this.io.to(msg.toPeer).emit('madeAnswerRTC', {
            data: msg.data,
            fromPeer: socket.id
        })
    })
// передача від одного учасника до другого ICE-кандидатів
    socket.on('candidate', msg => {
        this.io.to(msg.toPeer).emit('candidate', {
            data: msg.data,
            fromPeer: socket.id
        })
    })
// повідомлення про те, що сокет відключився
    socket.on('disconnect', () => {
        this.io.emit('userLeftRoom', socket.id)
    })
})
```

Додаток Г

Фрагмент коду, що демонструє процес обробки подій та створення SDP пропозиції, а також обробку відповіді на нашу пропозицію

```
public rtcPeerConfig = {
  iceServers: [
    {
      urls: 'stun:stun.l.google.com:19302',
    }
  ]
};

private initializePeerAsCaller(participants: Array<string>) {
// вибираємо серед учасників кімнати всіх, крім себе
  const participantsExcludingMe = participants.filter(id => id !==
this.webSocketService.mySocketId);

// створюємо пропозиції для з'єднання
  participantsExcludingMe.forEach(peerId => {
    this.initPeerToOffer(peerId);
  });
}

private initPeerToOffer(peerId: string) {
// ініціалізуємо екземпляр класу RTCPeerConnection
  const peer = new RTCPeerConnection(this.rtcPeerConfig);
// створюємо назамовну обгортку VideoRTCPeer
  const videoPeer = new VideoRTCPeer({peerId, peer});
// додаємо до списку наших підключень
  this.videoPeers.push(videoPeer);

// оброблюємо подію negotiationneeded
  this.peerOnNegotiationNeeded(videoPeer);
// оброблюємо подію icecandidate
  this.peerOnIceCandidate(videoPeer);
// створюємо пропозицію
  this.peerCreateOffer(videoPeer);
}

private peerOnNegotiationNeeded(videoPeer: VideoRTCPeer) {

  videoPeer.peer.onnegotiationneeded = ev => {
    this.peerCreateOffer(videoPeer);
  };
}

private peerOnIceCandidate(videoPeer: VideoRTCPeer) {
  const peer = videoPeer.peer;
  const peerId = videoPeer.peerId;
  peer.onicecandidate = event => {
    if (event.candidate) {
      this.webSocketService.emit('candidate', new SocketDataMessage({
        data: event.candidate,
        toPeer: peerId
      }));
    }
```

```
  };
}


private peerCreateOffer(videoPeer: VideoRTCPeer) {
// peer - екземпляр класу RTCPeerConnection
  const peer = videoPeer.peer;
// toPeer - ідентифікатор учасника на іншій стороні зв'язку
  const toPeer = videoPeer.peerId;

// створюємо пропозицію, зберігаємо як локальний опис та відправляємо через
// сигнальний сервер до іншої сторони
  peer
    .createOffer()
    .then(sdp => peer.setLocalDescription(sdp))
    .then(() => {
      console.log(`Sending offer to ${toPeer}`);
      this.webSocketService.emit('offerRTC', new SocketDataMessage({
        data: peer.localDescription,
        toPeer
      }));
    })
    .catch(err => console.error(err));
}

private receivedAnswer(message: SocketDataMessage) {

// шукаємо екземпляр класу VideoRTCPeer, який репрезентує зв'язок з
// користувачем, що надіслав нам відповідь
  const answeringPeer = this.videoPeers.find(x => x.peerId ===
message.fromPeer);
  if (answeringPeer) {
// зберігаємо у себе опис віддаленого учасника, тим самим завершуючи
// успішно з'єднання
    answeringPeer.peer
      .setRemoteDescription(message.data)
      .then(() => console.log(`Have set remote description from
${message.fromPeer}`))
      .catch(err => console.error(err));
  }
}
```

Додаток Ґ

Фрагмент коду, що демонструє процес обробки SDP пропозиції з іншої сторони та створення відповіді

```
private answerToOffer(message: SocketDataMessage) {

// аналізуємо  чи зараз треба створювати нове підключення або перемовитися
// щодо вже існуючого
  const videoPeerExisting = this.videoPeers.filter(x => x.peerId ===
message.fromPeer)[0];

// якщо не було ще створено підключення
  if (!videoPeerExisting) {

// ініціалізуємо екземпляр класу RTCPeerConnection
    const peer = new RTCPeerConnection(this.rtcPeerConfig);
// створюємо назамовну обгортку VideoRTCPeer
    const videoPeer = new VideoRTCPeer({peerId: message.fromPeer, peer});
// додаємо до списку наших підключень
    this.videoPeers.push(videoPeer);
// оброблюємо подію negotiationneeded
    this.peerOnNegotiationNeeded(videoPeer);
 // оброблюємо подію icecandidate
    this.peerOnIceCandidate(videoPeer);
// створюємо відповідь на пропозицію
    this.peerCreateAnswer(videoPeer, message.data);
  } else {
// створюємо відповідь на пропозицію
    this.peerCreateAnswer(videoPeerExisting, message.data);
  }
}

private peerCreateAnswer(videoPeer: VideoRTCPeer, remoteDescription:
RTCSessionDescription) {
// peer - екземпляр класу RTCPeerConnection
  const peer = videoPeer.peer;
// toPeer - ідентифікатор учасника на іншій стороні зв'язку
  const toPeer = videoPeer.peerId;

// зберігаємо пропозицію віддаленого учасника у себе, створюємо відповідь,
// зберігаємо цю відповідь як локальний опис та відправляємо до іншої
// сторони через сигнальний сервер
  peer
    .setRemoteDescription(remoteDescription)
    .then(() => peer.createAnswer())
    .then(sdp => peer.setLocalDescription(sdp))
    .then(() => {
      this.webSocketService.emit('answerRTC', new SocketDataMessage({
        data: peer.localDescription,
        toPeer
      }));
    });
}
```

Додаток Д

Фрагмент коду, що демонструє підключення та використання RTCDataChannel

Локальна сторона:

```
private initPeerToOffer(peerId: string) {
// . . .
  this.creatingChannel(videoPeer);
// . . .
// створюємо пропозицію
  this.peerCreateOffer(videoPeer);
}

private creatingChannel(videoPeer: VideoRTCPeer) {
// створюємо канал даних
  const channel = videoPeer.peer.createDataChannel('chat');
// зберігаємо посилання в екземплярі кастомного класу VideoRTCPeer
  videoPeer.dataChannel = channel;
// налаштування обробників подій
  this.setChannelOnOpenOnMessage(channel);
}

private setChannelOnOpenOnMessage(channel: RTCDataChannel) {
// обробник подій open
  channel.onopen = event => {
    channel.send(`Hi you! Hello from ${this.webSocketService.mySocketId}`);
  };
// обробник подій message
  channel.onmessage = event => {
    console.log(event.data);
  };
}
```

Віддалена сторона:

```
private answerToOffer(message: SocketDataMessage) {
// . . .
    this.peerOnDataChannel(videoPeer);
// . . .
}

private peerOnDataChannel(videoPeer: VideoRTCPeer) {
// обробник події появи запрошення приєднатися до каналу даних
  videoPeer.peer.ondatachannel = event => {
// посилання на канал даних
    const channel = event.channel;
// зберігаємо посилання в екземплярі кастомного класу VideoRTCPeer
    videoPeer.dataChannel = channel;
// налаштування обробників подій
    this.setChannelOnOpenOnMessage(channel);
  };
}
```

Додаток Е

Фрагмент коду, що демонструє динамічну вставку потоків відео

Додавання та прийом відео:

```
// викликається натисканням кнопки,коли користувач хоче показувати відео
streamVideo() {
// передається в результаті потік на читання від підключеної відеокамери
  navigator.mediaDevices.getUserMedia({video: true, audio: false})
    .then(stream => {
// зберігається посилання на цей потік, щоб потім можна було маніпулювати
      this.myStream = stream;
// цей потік призначується джерелом відео для тегу video
      this.myVideo.nativeElement.srcObject = stream;
// почати програвання відео
      this.myVideo.nativeElement.play();
      this.addTracksToConnections(this.myStream, this.videoPeers);
    })
    .catch(err => console.log(err));
}

private addTracksToConnections(stream: MediaStream, peerConnections:
Array<VideoRTCPeer>) {
// додаються всі доріжки, що було передано до зв'язку з усіма
користувачами,
// з якими підключені, та зберігається результат функції екземпляра
// класу RTCRtpSender, щоб потім за допомогою нього можна було видалити цю
// доріжку
  stream.getTracks().forEach(tr => {
    peerConnections.forEach(vp => {
      vp.senders.push(vp.peer.addTrack(tr));
    });
  });
}


// обробка події, коли додється доріжка до зв'язку з іншої сторони
private initPeerToOffer(peerId: string) {
// . . .
  this.peerOnTrack(videoPeer);
// . . .
// створюємо пропозицію
  this.peerCreateOffer(videoPeer);
}

private answerToOffer(message: SocketDataMessage) {
// . . .
    this.peerOnTrack(videoPeer);
// . . .
}

private peerOnTrack(videoPeer: VideoRTCPeer) {
// peer - екземпляр класу RTCPeerConnection
  const peer = videoPeer.peer;
// toPeer - ідентифікатор учасника на іншій стороні зв'язку
  const peerId = videoPeer.peerId;
```

```
// обробка події track
  peer.ontrack = event => {
// знаходимо тег video, що транслює відео користувача на іншій стороні
    const peerVideo = this.peerVideos.filter(x => x.nativeElement.id ===
peerId)[0];
// Якщо тільки перша доріжкаприйшла й ще не створено медіапотоку для їхньої
// обгортки
    if (!videoPeer.stream) {
        videoPeer.stream = new MediaStream();
// цей потік призначується джерелом відео для тегу video
        peerVideo.nativeElement.srcObject = videoPeer.stream;
// почати програвання відео
        peerVideo.nativeElement.play();
      }
// додається трек до потоку
      videoPeer.stream.addTrack(event.track);
    }
}
```

Видалення відео:

```
// викликається натисканням кнопки,коли користувач хоче перестати
// показувати відео
removeVideo() {
  this.removeTracksFromPeers(true);
// береться посилання на потік медіа, що репрезентує трансляцію користувача
  const myVideoSrc = this.myVideo.nativeElement.srcObject as MediaStream;
  this.removeTracksFromStream(myVideoSrc, this.myStream, true);
}

private removeTracksFromPeers(isVideoTrack: boolean) {
  let kind = isVideoTrack ? 'video' : 'audio';
// для кожного підключення знаходимо екземпляр класу RTCRtpSender, що
// займається передачею доріжки, що хочуть видалити зі зв'язку
  this.videoPeers.forEach(vp => {
    const sender = vp.senders.find(s => s ? s.track.kind === kind : false);
// видаляється трек зі зв'язку
    vp.peer.removeTrack(sender);
    vp.senders = vp.senders.filter(s => s.track !== null);
  });
}

private removeTracksFromStream(streamFromTag: MediaStream, streamFromVar:
MediaStream, isVideoTrack: boolean) {
  const tracks = isVideoTrack ? streamFromTag.getVideoTracks() :
streamFromTag.getAudioTracks();
  tracks.forEach(tr => {
// зупиняється доріжка та його видаляється з медіапотоку
// якщо нема більше доріжок у потоці, то видаляється посилання на цей потік
    tr.stop();
    streamFromTag.removeTrack(tr);
    if (streamFromTag.getTracks().length === 0) {
      streamFromTag = null;
      streamFromVar = null;
    }
  });
}
```

Додаток Є

Фрагмент коду, що демонструє перевірку на сумісність браузерного агенту з технологією Insertable Streams API

```
// замовчування попереджень @ts-ignore використовується через те, що дана
// технологія поки не увійшла в стандарт WebRTC
private checkCompatibility() {
  // @ts-ignore
  this.supportsInsertableStreams =
!!RTCRtpSender.prototype.createEncodedStreams;

  let supportsTransferableStreams = false;
  try {
// такий функціонал знадобиться задля передачі даних до веб-працівника
    const stream = new ReadableStream();
    // @ts-ignore
    window.postMessage(stream, '*', [stream]);
    supportsTransferableStreams = true;
  } catch (e) {
    console.error('Transferable streams are not supported.');
  }

  if (!(this.supportsInsertableStreams && supportsTransferableStreams)) {
    this.messageToUser = 'Your browser does not support Insertable Streams.
' +'This sample will not work.';
    this.messageToUser += 'Using only chrome: Try with Enable experimental
Web Platform features enabled from chrome://flags.';
  }
}
```

Додаток Ж

Фрагмент коду, що демонструє роботу з Insertable Streams API

```
public rtcPeerConfig = {
  iceServers: [
    {
      urls: 'stun:stun.l.google.com:19302',
    }
  ],
  // @ts-ignore
  encodedInsertableStreams: true
};

constructor() {
  this.worker = new Worker('../e2e-demo/e2e-demo.worker.ts', {type:
'module', name: 'E2ED worker'});
}

private addTracksToConnections(stream: MediaStream, peerConnections:
Array<VideoRTCPeer>) {
  stream.getTracks().forEach(tr => {
    peerConnections.forEach(vp => {
      const sender = vp.peer.addTrack(tr);
      this.setupSenderTransform(sender);
      vp.senders.push(sender);
    });
  });
}

private setupSenderTransform(sender: RTCRtpSender) {
  let senderStreams;
// створюємо екземпляр класу TransformStream для роботи у веб-працівнику
  // @ts-ignore
  senderStreams = sender.createEncodedStreams();

  const readableStream = senderStreams.readable;
  const writableStream = senderStreams.writable;
// надсилаємо потоки на читання та запис до веб-працівника з метою
// зашифрувати
  this.worker.postMessage({
    operation: 'encode',
    readableStream,
    writableStream,
  }, [readableStream, writableStream]);
}

private peerOnTrack(videoPeer: VideoRTCPeer) {
// . . .
  peer.ontrack = event => {
// . . .
    this.setupReceiverTransform(event.receiver);
    videoPeer.stream.addTrack(event.track);
  }
}

private setupReceiverTransform(receiver) {
  let receiverStreams;
```

```
// створюємо екземпляр класу TransformStream для роботи у веб-працівнику
  receiverStreams = receiver.createEncodedStreams();

  const readableStream = receiverStreams.readable;
  const writableStream = receiverStreams.writable;
// надсилаємо потоки на читання та запис до веб-працівника з метою
// дешифрувати
  this.worker.postMessage({
    operation: 'decode',
    readableStream,
    writableStream,
  }, [readableStream, writableStream]);
}

public cryptoKeyChange() {
  const currentCryptoKey = this.cryptoKey;
// надсилаємо конфігурацію нового ключа шифрування до веб-працівника
  this.worker.postMessage({
    operation: 'setCryptoKey',
    currentCryptoKey,
    useCryptoOffset: true,
  });
}
```

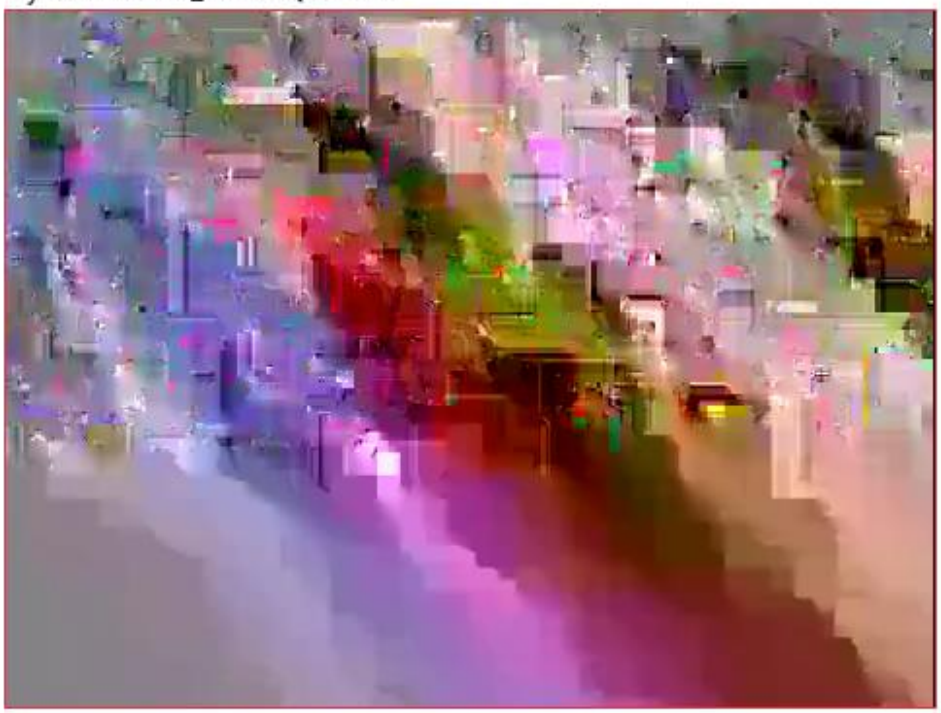
Додаток З

Знімки екрану, що демонструють роботу шифрування медіапотоків. Користувач, що не транслює медіа (з чорним екраном; id починається на Yl1s) використовує додаток, який не підтримує Insertable Streams API

My id: pIHwVu6T7pgybJrIAAAH

User id: Yl1sJAM_M1zu2QbaAAAI

My id: Yl1sJAM_M1zu2QbaAAAI



User id: plHwVu6T7pgybJrIAAAH