Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

Screen Capture API: Особливості використання та безпека у веб-застосунках

Текстова частина до курсової роботи

за спеціальністю „Комп'ютерні науки та інформаційні технології" 122

Керівник курсової роботи

Доктор технічних наук,
доцент

Глибовець А.М.

(підпис)

"____" _____ 2021 р.

Виконала студентка 4 курсу

Печура М.В.

"____" _____ 2021 р.

Київ 2021

Тема: Screen Capture API: Особливості використання та безпека у веб-застосунках

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	30.09.2020	
2.	Пошук тематичної літератури	15.11.2020	
3.	Ознайомлення з літературою	12.12.2020	
4.	Вивчення додатків із застосуванням WebRTC та Screen Capture API	08.01.2021	
5.	Планування веб-додатку	22.01.2021	
6.	Створення директорій проекту	27.01.2021	
7.	Реалізація передачі відео та аудіо медіа потоків по мережі	10.02.2021	
8.	Реалізація передачі екрану користувача по мережі	20.02.2021	
9.	Розробка функціоналу сторінок додатку	07.03.2021	
10.	Додавання стилів до сторінок	19.03.2021	
11.	Написання текстової частини	29.03.2021	
12.	Подання першої версії записки науковому керівнику	05.04.2021	
13.	Перегляд змісту роботи керівником	08.04.2021	
14.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	09.04.2021	
15.	Створення презентації	11.04.2021	
16.	Захист роботи	19.04.2021	

Студентка Печура М.В.

Керівник Глибовець А.М. “____” _____

ЗМІСТ

Перелік умовних позначень та виразів:

1. **Зам’ютити -** (від англ. mute) означає вимкнути звук іншого користувача, щоб його не було чутно. Зазвичай використовується для випадків, коли звуки з боку учасника конференції заважають головному спікеру.
2. **Підняти руку під час дзвінка -** показати іншим учасникам конференції що є бажання висловитись. Найчастіше ця функція реалізується за допомогою кнопки, після натискання котрої всі учасники бачать знак(підняту руку) біля зображення/відео/імені того, хто “підняв руку”.
3. **Використати дошку під час дзвінка -** ця функція у відео-конференції виконує наступне: в кожного учасника на головному екрані з’явиться білий прямокутник (на кшталт Paint). Користувачі можуть писати на цьому полотні за допомогою миші або тачпаду. Використовується для проведення лекцій, де потребується використання дошки.
4. **Використати кімнати -** для учасників дзвінка створюються групи, для кожної групи окремий дзвінок.
5. **Бітрейт -** швидкість проходження бітів інформації за секунду. Бітову швидкість зазвичай використовують для вимірювання ефективної швидкості передачі інформації по каналу.
6. **Затримка -** використовується у такому контексті: затримка між дією користувача та реакцією веб-програми на цю дію.
7. **Ко́дек -** пристрій або програма, здатна виконувати перетворення потоку даних або сигналу. Кодеки можуть як кодувати потік/сигнал, так і розкодовувати — для перегляду або зміни у форматі, що більше підходить для цих операцій. Кодеки часто використовуються при цифровій обробці відео й аудіо.

8. **Best-effort доставка** (з найкращими зусиллями) **-** мережева послуга, в якій мережа не надає жодних гарантій того, що дані буде доставлено або що доставка відповідає якійсь якості обслуговування.
9. **Фабричний метод -** породжуючий шаблон проектування, надає підкласам(дочірнім класам) інтерфейс для створення екземплярів деякого класу. У момент створення спадкоємці можуть визначити, який клас створювати. Іншими словами, даний шаблон делегує створення об'єктів спадкоємцям батьківського класу. Це дозволяє використовувати в коді програми не специфічні класи, а маніпулювати абстрактними об'єктами на більш високому рівні.
10. **Мультиплексування -** передача даних з багатьох каналів через один.
11. **Peer-to-peer -** позначення мереж, у яких кожен комп’ютер може виступати в ролі сервера для інших, що надає спільний доступ до файлів та периферійних пристроїв без потреби центрального сервера. Peer-to-peer connection називають також одноранговим з’єднанням.
12. **ICE trickling (ICE просочування) -** процес продовження відправки кандидатів, після початкової передачі пропозиції або відповіді віддаленому пристрою.
13. **Кастомний -** виготовлений на замовлення.
14. **SDP** (Session Description Protocol) **-** це формат для опису сесій мультимедійної комунікації, який використовується з метою оголошення або ініціювання сесії.
15. **W3C** (The World Wide Web Consortium) – консорціум Всесвітньої павутини – організація, яка розробляє і впроваджує технологічні стандарти для Всесвітньої павутини.

ВСТУП

Онлайн-зв’язок, що зможе замінити живе спілкування, є необхідним для багатьох людей. Його використовують для робочих дзвінків, розмов із друзями та проведення учбових занять. Для передачі аудіо та відео по мережі можна встановлювати різні застосунки, проте сучасні браузери здатні на peer-to-peer зв’язок: це означає, що онлайн-конференції можна проводити без встановлення додаткового програмного забезпечення.

Метою курсової роботи є дослідження технологій, призначених для передачі медіа peer-to-peer, тестування їх та створення сайту для проведення онлайн-конференцій.

Сайт розроблено за допомогою WebRTC, Screen Capture API та інших Web APIs, у поєднанні із мовою JavaScript. Контроль проекту та написання коду здійснювалось у середовищі WebStorm.

Курсова робота складається із вступу, чотирьох основних розділів, кожен з яких містить підрозділи, висновків, списку використаної літератури та додатків.

Перший розділ є оглядовим: у ньому розглядається питання проведення онлайн-конференцій, коротко описуються існуючі сервіси відео-зв’язку та формується завдання курсової роботи. У другому розділі проведений глибокий аналіз технології WebRTC, у третьому – Screen Capture API. У четвертому розділі здійснено опис розробки сайту.

РОЗДІЛ 1

1.1. Аудіо, відео конференції та peer-to-peer з'єднання у сучасному світі. Обґрунтування теми

У сучасномі світі, коли міжнародні компанії потребують встановлення зв'язку з їх співробітниками по всьому світу, а віддалена робота все більш схвалюється як керівниками, так і їх підопічними, онлайн-комунікація є вкрай важливим інструментом. В часи пандемії Covid-19 працювати дистанційно - вже не забаганка, а необхідність. Тому компанії, які випускають продукти веб-телефонії, активно працюють над якісним зв'язком, звуком та зображенням.

Говорячи про браузери, їх розвиток з роками не зупиняється: користувачеві надаються нові можливості, у тому числі ті, які надають базову функціональність комунікації без додаткового програмного забезпечення. Нові технології браузерів відриваються від звичної моделі зв'язку клієнт-сервер, що призводить до повного реінжинірингу мережевого рівня в браузері. Вони постійно вдосконалюються, мають і бета-версії, що відкриває простір для вивчення та експериментів.

Серед цих можливостей ([1]):

1. Поширення аудіо та відео

   Сучасні технології привносять у браузер повнофункціональні аудіо- та відеосистеми, які дбають про обробку аудіо- та відео потоків. Вся обробка здійснюється безпосередньо браузером, який динамічно налаштовує процесинг потоків. Досягається це за допомогою аудіо та відеосистем протоколу WebRTC ([2]).

2. Передача даних у режимі реального часу

   Використання протоколів, які призначені для толерування періодичної

втрати пакетів, дозволяє передавати інформацію по мережі з мінімальним впливом на якість вихідних даних. Перелік цих протоколів: UDP, ICE, SDP, DTLS, SCTP та SRTP.

3. Двосторонній обмін довільними даними

   Його забезпечує RTCDataChannel інтерфейс, що є подібним до WebSocket, тільки вставлюється між одноранговими мережами, та має властивості транспортування даних, які можна кастомізувати.

У даній роботі буде детально розглянуто ці та інші можливості сучасних браузерів.

1.2. Огляд існуючих сервісів відеотелефонного зв'язку

У 2021 році кожен знає про такі сервіси як Skype, Zoom та Google Meets. З того часу як почався карантин в Україні, навіть люди, які не користувались цими засобами, перейшли на дистанційну роботу/навчання. Якщо дослідити цю тему глибше, виявиться, що сервісів для веб-зв’язку достатньо, щоб вони створювали один для одного конкуренцію.

Розглянемо декілька таких програм для відеотелефонії:

Google Meets ([3]) – сервіс відеотелефонного зв'язку, розроблений компанією Google, офіційно запущений у березні 2017 року. Разом із Google Chat, він замінив Google Hangouts. У зв’язку із пандемією Covid-19, компанія вирішила надати безкоштовний доступ до програми. Це призвело до збільшення кількості користувачів в 30 разів порівняно із минулим роком.

Нижче приведено зображення, що показує яким чином виглядає дзвінок у Google meets (див. рис. 1.2.1).

Рис. 1.2.1 Інтерфейс сервісу Google Meets

Zoom Cloud Meetings

Це – розширення для браузера від додатку Zoom ([4]). Наявні розширення для Firefox, Chrome та Safari. Zoom вийшов на ринок у вересні 2012 року, проте дата виходу розширень для браузерів не відома на широкий загал.

Інтерфейс Zoom meetings (див. рис. 1.2.2) відрізняється від інтерфейсу сервісу-конкурента Google Meets.

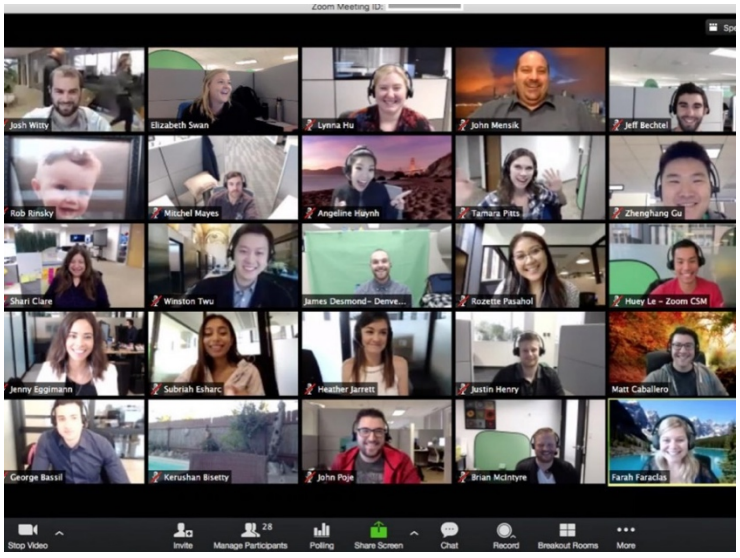

Рис.1.2.2 Скріншот дзвінка у Zoom Cloud Meetings

Функціонал розглянутих вище сервісів схожий, однак вони змагаються у кількості додаткових, цікавих можливостей для користувача.

Говорячи про стек технологій, який об’єднує згадані вище застосунки, то це – WebRTC. В контексті peer-to-peer зв’язку та передачі відео/аудіо від браузера до браузера єдиний можливий варіант для всіх розробників – це скористатись можливостями цієї технології, яку надає кожен модерний браузер.

1.3. Постановка завдання

Говорячи про загальну ідею, вона полягає у тому, щоб дослідити новий функціонал браузерів, провести огляд можливостей, протестувавши їх на практиці.

Покроково про етапи дослідження:

1. Аналіз функціональних вимог та логіки роботи існуючих сервісів відеотелефонного зв’язку.

2. Дослідження технології для передачі відео, аудіо та повідомлень між браузерами та вибір функцій для розробки власного застосунку.

3. Розробка сервісу відеозв’язку, який відтворить базові функції застосунків-аналогів, задовольняючи наступні вимоги:

- обмін відео та аудіо між учасниками конференції;
- передача повідомлень між користувачами;
- можливість налаштувати аудіо та відео;
- можливість ділитись з іншими учасниками дзвінка екраном свого пристрою.

У процесі вивчення та тестування останніх технологій браузера, до сервісу будуть додані і інші функції, в залежності від прогресу їхнього тестування та ваги бенефітів від них.

РОЗДІЛ 2

2.1. Загальні відомості про технологію WebRTC

WebRTC ([1]) (веб-спілкування в реальному часі) – це сукупність стандартів, протоколів та JavaScript API, комбінація яких робить можливим обмін аудіо, відео та іншими даними між браузерами. Замість того, щоб покладатись на сторонні плагіни або ж програмне забезпечення, WebRTC зробило комунікацію у режимі реального часу стандартною опцією, яку може використати будь-який веб-застосунок за допомогою простого JS API.

Технологія WebRTC наразі доступна більш ніж одному мільярду користувачів: нові версії браузерів Chrome та Firefox підтримують WebRTC для всіх своїх користувачів.

Архітектура WebRTC ділиться на 2 категорії ([1]):

1. Web Real-Time Communications (WEBRTC ([5])) W3C робоча група відповідає за визначення API браузера.
2. Real-Time Communication in Web-browsers (RTCWEB ([5])) IETF робоча група відповідальна за визначення протоколів, форматів даних, безпеки, та інших додаткових аспектів, які забезпечують одноранговий зв’язок у браузері.

Незважаючи на те, що основною метою WebRTC є забезпечення зв'язку в режимі реального часу між браузерами, технологія також розроблена таким чином, що може бути інтегрована із існуючими системами зв'язку: наприклад, передачею голосу через IP (VOIP), і навіть із телефонною комутованою мережею загального користування (PSTN ([16])).

Варто згадати, що WebRTC зараз знаходиться в активному стані розробки: як на рівні API браузера, так і на транспортному рівні, та рівні протоколів. Це означає, що деякі API та протоколи, згадані у цій роботі, можуть бути змінені у майбутньому.

2.2. Передача медіа та відео потоків по мережі

Щоб провести телеконференцію в браузері, йому треба мати доступ до системного апаратного забезпечення, щоб захоплювати аудіо та відео.

Однак після отримання доступу, необроблених аудіо- та відеопотоків недостатньо для роботи сервісу телефонії: кожен потік повинен бути оброблений для підвищення якості, синхронізований, а вихідний бітрейт повинен пристосовуватися до постійних коливань пропускної здатності та затримки між клієнтами. На стороні отримувача потоку, процес обробки є таким: потік має бути декодовано в режимі реального часу, тремтіння мережі та затримки мають бути відрегульовані.

WebRTC містить повнофункціональні аудіо- та відеосистеми (див. рис. 2.2.1), які полегшують завдання передачі аудіо та відео по мережі.

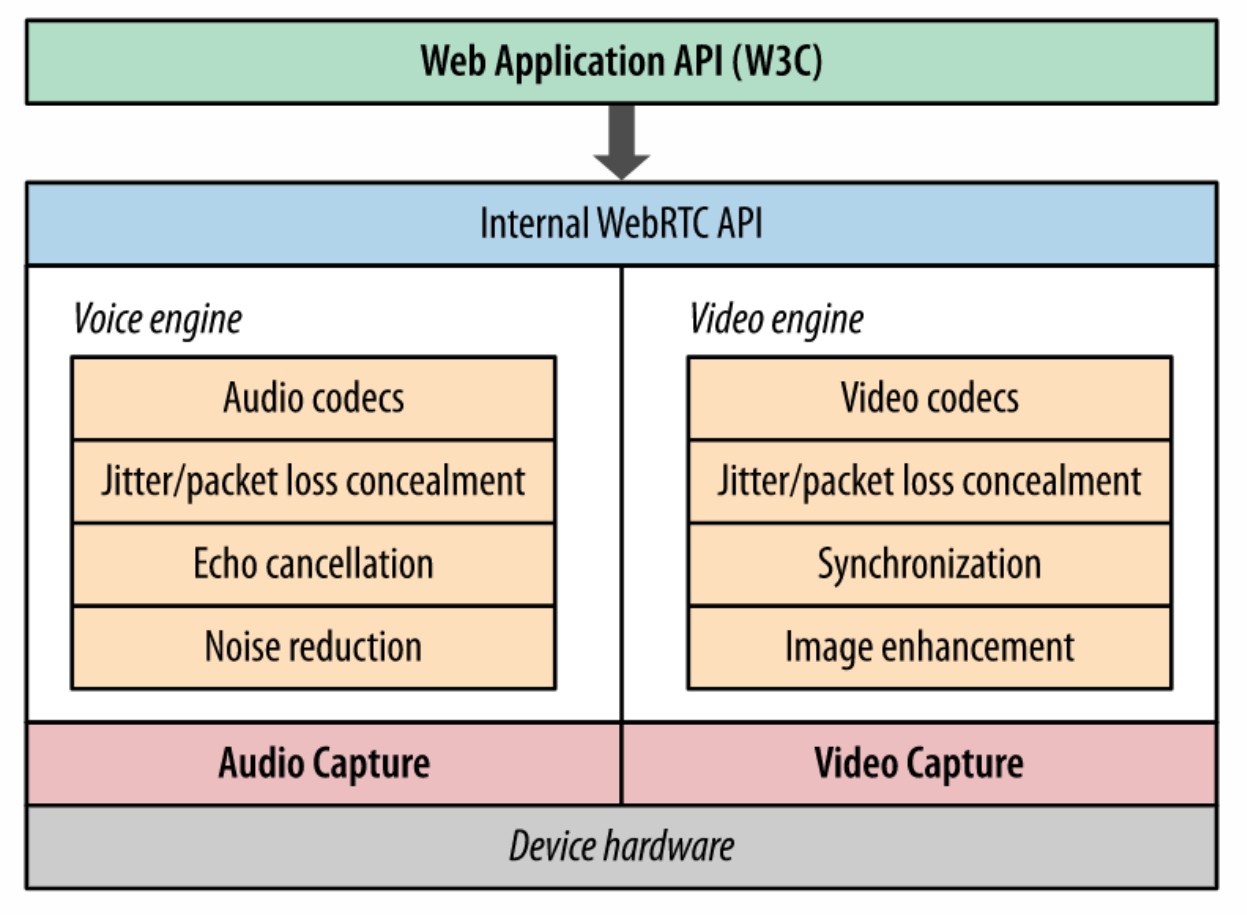

Рис. 2.2.1 [1] Аудіо та відеосистеми WebRTC

За допомогою аудіосистеми, отриманий аудіо-потік обробляється для зменшення рівня шуму та відлуння, потім автоматично кодується. Далі, спеціальний алгоритм приховування помилок прибирає негативні наслідки мерехтіння мережі та втрати пакетів. А відеосистема оптимізує якість зображення та обирає оптимальні налаштування стиснення та кодеку.

Вся обробка здійснюється безпосередньо браузером, який налаштовує її динамічно. Він враховує параметри аудіо та відео, що постійно змінюються, та стан мережі. Після того, як аудіо та відеосистеми виконали свою роботу, веб-застосунок отримує оптимізований медіа-потік, який можна

виводити на екран та передавати динамікам, виконувати подальшу обробку за допомогою HTML5 медіа-API і т.п.

Щоб отримати аудіо та відео пристрою користувача, браузер надає getUserMedia() метод, що належить АПІ MediaDevices.

Метод дозволяє: 1) запитувати доступ до мікрофона та камери у користувача та отримувати потоки, що відповідають бажаним налаштуванням аудіо та відео; 2) вказувати перелік обов'язкових та необов'язкових налаштувань відповідно до вимог до додатку.

Розглянемо простий приклад використання getUserMedia() апі-колу:

У розмітці HTML:

```
<video autoplay></video>
```

JavaScript код:

```
  var constraints = {
    audio: true,
    video: {
      mandatory: { // обов'язкові налаштування
        width: { min: 300 },
        height: { min: 170 }
      },
      optional: [ // необов'язкові налаштування
        { width: { max: 1300 }},
        { frameRate: 30 },
        { facingMode: "user" }
      ]
    }
  }
  navigator.getUserMedia(constraints, getVideo, errorHandler);
  function getVideo(stream) {
    let video = document.querySelector('video');
    video.src = window.URL.createObjectURL(stream);
```

```
  }
  function errorHandler(error) { ... }
```

2.3. Обмін аудіо-потоками

У попередньому розділі ми з'ясували, що налаштувати передачу аудіо по мережі можна досить просто: вказавши відповідний параметр `audio: true` та передавши його до методу `getUserMedia.`

Проте щоб бути певним, що у вхідному потоці справді є аудіо, рекомендується використати `getAudioTracks()` метод потоку. Цей метод повертає масив усіх аудіодоріжок, які має потік. Отже, перевіривши чи не порожній масив, ми можемо бути впевненими у тому що аудіопотік доставлений.

Перевірка може виглядати наступним чином:

```
if (stream.getAudioTracks().length > 0){
      audioCheck('user is sending audio');
} else {
      audioCheck('user is not sending audio');
}
```

Після того, як ми впевнились, що аудіо-потік є, може виникнути ситуація, що ми все одно не будемо чути співрозмовника. Можливо, він випадково вимкнув свою звукову доріжку. Кожен елемент масиву, який повертає `getAudioTracks()`, містить `enabled` властивість. Якщо вона дорівнює `true`, це означає, що звук увімкнено. Використовуючи цю властивість, можна сповіщувати користувачів додатку про вимкнені мікрофони учасників дзвінка.

2.4. Особливості передачі медіа та відео в режимі реального часу

Транспортування даних в режимі реального часу є “чутливим до часу”. Пояснюючи цей вираз, якщо один учасник конференції чекає, до прикладу, 10 хвилин на отримання відео або ж повідомлення зі сторони іншого учасника, це вже не можна вважати спілкуванням у режимі реального часу.

Відповідно, програми для потокового передавання відео та аудіо розроблені таким чином, щоб, так би мовити, толерувати періодичну втрату пакетів: аудіо- та відео кодеки заповнюють невеликі прогалини в даних, з мінімальним впливом на якість. Якщо намагатись надіслати усі пакети без втрати, отримувачу даних доведеться довго чекати. Так само у застосунках має бути імплементовано логіку для відновлення втрачених або затриманих пакетів з даними інших типів. Своєчасна доставка та мінімальна затримка пакетів є у цьому випадку важливішими за надійність.

Вимога до своєчасності доставки пакетів є основною причиною того, чому UDP протокол є найкращим варіантом для передачі даних у режимі реального часу. Він не гарантує доставки кожного повідомлення, дотримання порядку доставки пакетів, не відстежує стан мережі, не має вбудованих механізмів зворотного зв’язку від клієнта або мережі. В той час як TCP доставляє надійний і впорядкований потік даних: якщо якийсь з пакетів було втрачено, TCP буферизує пакети після нього, чекає повторної передачі цього пакету, і тільки тоді доставляє весь потік із правильним порядком пакетів.

Не дивлячись на те, що UDP здається анти-версією TCP, і його можна назвати ненадійним протоколом, UDP більше підходить для дзвінків у режимі реального часу.

Він є основою, але щоб задовольнити всі вимоги WebRTC (такі як шифрування даних користувача, контроль потоку і т.д.), браузеру потрібні

допоміжні протоколи та сервіси. Поглянемо на стек протоколів WebRTC (див.рис.2.2.2).

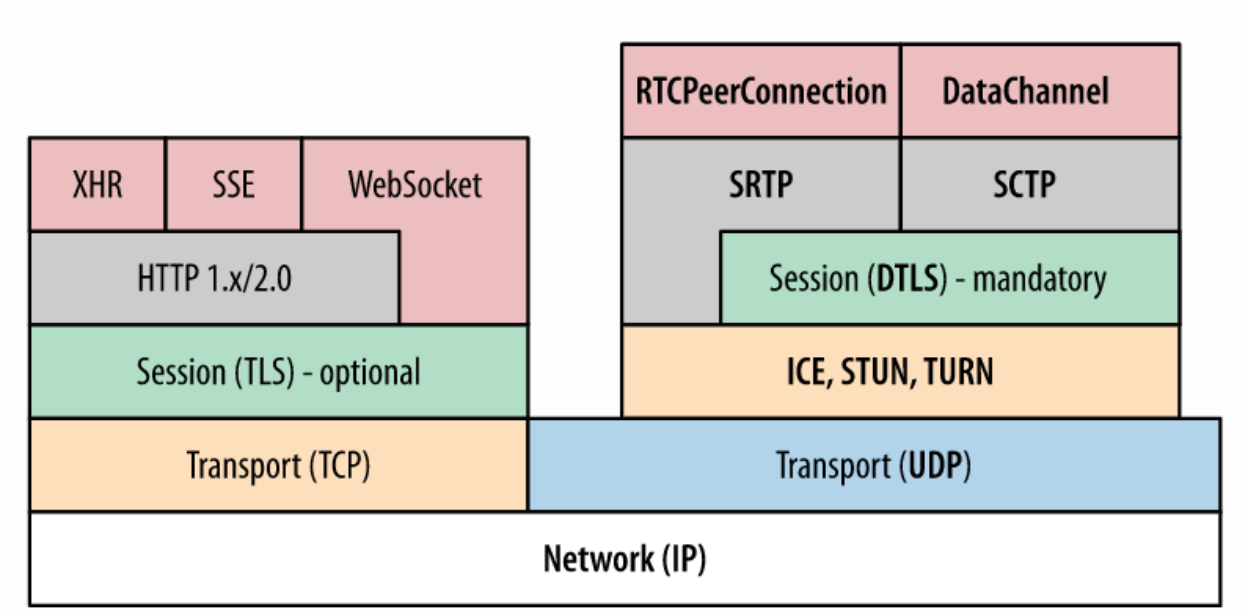

Рис. 2.2.2 [1] Протоколи WebRTC

1. ICE (Interactive Connectivity Establishment) - створення інтерактивного підключення.
    - STUN (Session Traversal Utilities for NAT) ([6]) – утиліта проходження сесій для Network Address Translation (що означає перетворення мережевих адрес).
    - TURN (Traversal Using Relays around NAT) ([7]) – обхід за допомогою передачі для NAT.

    ICE, STUN та TURN відповідають за встановлення та підтримання однорангового з'єднання через UDP.

2. SDP: Session Description Protocol ([8]) – протокол опису сесії.
3. DTLS: Datagram Transport Layer Security ([9]) – безпека транспортного рівня датаграми. DTLS використовується для захисту передачі даних між браузерами: шифрування є обов'язковою, важливою функцією WebRTC.

4. SCTP: Stream Control Transport Protocol ([10]) – протокол передачі з управлінням потоком.
5. SRTP: Secure Real-Time Transport Protocol ([11]) – безпечний протокол передачі даних у режимі реального часу.

SCTP та SRTP – це протоколи додатків, що використовуються для мультиплексування різних потоків, забезпечують перевантаження та контроль потоку, а також забезпечують частково надійну доставку.

2.5. Обмін довільними даними додатку між браузерами

2.5.1. Інтерфейс RTCDataChannel

DataChannel забезпечує двосторонній обмін довільними даними додатків між браузерами – як WebSocket, тільки у контексті однорангового з'єднання, і з можливістю налаштувати транспортування.

Перед тим, як створити DataChannel, потрібно встановити RTCPeerConnection ([12]) – об'єкт, що позначає з'єднання WebRTC між локальним комп'ютером та віддаленим пристроєм. Після встановлення RTCPeerConnection, з'єднані браузери можуть відкрити один або більше каналів для обміну даними у текстовому або бінарному форматі.

У нижче наведеному прикладі відбувається реєстрація колбеків на об'єкті DataChannel та контроль передачі даних по RTCPeerConnection:

```
let signalingChannel = new SignalingChannel();
let pc = new RTCPeerConnection(iceConfig);
let dc = pc.createDataChannel("namedChannel", {reliable: false});

handleChannel(dc);
pc.ondatachannel = handleChannel;
```

```
function handleChannel(channel) {
  channel.onerror = function(error) { ... }
  channel.onclose = function() { ... }
  channel.onopen = function(evt) {
    channel.send("DataChannel connection established!")
  }
  channel.onmessage = function(message) {
    if(message.data instanceof Blob) {
      processBlob(message.data);
    } else {
      processText(message.data);
    }
  }
}
```

DataChannel подібний до WebSocket у наступному: кожен канал викидує колбеки onerror, onclose, onopen, та onmessage, та містить поля binaryType, bufferedAmount, and protocol.

Однак, оскільки DataChannel є одноранговим та працює з більш гнучким транспортним протоколом, він пропонує ряд додаткових функцій, недоступних для WebSocket. На відміну від конструктора WebSocket, який очікує URL-адресу сервера WebSocket, DataChannel є фабричним методом для об'єкта RTCPeerConnection. Більше того, будь-який браузер може ініціювати нову сесію DataChannel. WebSocket працює поверх надійного та впорядкованого TCP-транспортування, в той час як для кожного DataChannel можна налаштувати доставку та семантику надійності.

DataChannel нашаровується поверх 3 протоколів: UDP, DTLS та SCTP.

Підсумовуючи, DataChannel може бути налаштований щоб надавати ту саму надійність та порядок доставки повідомлень, як і WebSocket. Незважаючи на це, потужність цього інструменту у тому, що він не повинен дотримуватись семантики надійності та порядку. Для кожного каналу можна

вказати його власні вимоги до доставки даних, і дані можуть передаватись безпосередньо по одноранговій мережі.

2.5.2. Налаштування з'єднання між браузерами

Незалежно від типу переданих даних - аудіо, відео чи даних додатку - два браузери, між якими встановлено з'єднання, повинні спочатку виконати повний процес пропозиція/відповідь, узгодити протоколи та порти, та успішно пройти перевірку з'єднання.

DataChannel використовує SCTP протокол. Як результат, коли браузер-ініціатор пропонує встановити зв'язок, або коли відповідь згенерована іншим браузером, обидва браузери мають оголосити параметри для SCTP асоціації у SDP рядках. Об'єкт RTCPeerConnection відповідає за створення SDP параметрів допоки один з браузерів не зареєструє DataChannel, а також створює SDP опис сесії.

Фактично, застосування може встановити peer-to-peer зв'язок для передачі виключно даних: досягається це за допомогою встановлення налаштувань для відключення передачі аудіо та відео:

```
var signalingChannel = new SignalingChannel();
var pc = new RTCPeerConnection(iceConfig);
var dc = pc.createDataChannel("namedChannel", {reliable: false});

var constraints = {
  mandatory: {
      OfferToReceiveAudio: false,
      OfferToReceiveVideo: false
  }
};
```

```
pc.createOffer(function(offer) { ... }, null, constraints);
```

У цьому прикладі встановлено медіа-налаштування для вимкнення передачі аудіо та відео. Отже, пропозиція на з'єднання генерується лише для передачі даних інших типів.

Коли SCTP параметри узгоджені між браузерами, залишається один крок перед початком обміну даними: клієнт WebRTC, який ініціює зв'язок, має надіслати DATA_CHANNEL_OPEN повідомлення, яке описує тип, надійність, протокол, та інші параметри каналу.

Після встановлення параметрів каналу, обидва браузери можуть обмінюватись даними додатку. На низькому рівні, кожен встановлений канал зв'язку є незалежним потоком SCTP: канали мультиплексовані над одною і тою самою SCTP асоціацією, що дозволяє уникати блокування між різними потоками та дозволяє одночасно доставляти кілька каналів через одну і ту ж асоціацію SCTP.

2.5.3. Налаштування порядку та надійності повідомлень

DataChannel активує однорангову передачу довільних даних додатку за допомогою сумісних з WebSocket API, що є унікальною та потужною функцією. Однак DataChannel також надає можливість гнучкої передачі даних, що дозволяє налаштувати семантику доставки кожного каналу відповідно до вимог програми та типу даних, що передаються. DataChannel може забезпечити надійну або частково надійну доставку впорядкованих та невпорядкованих повідомлень.

Налаштування каналу для впорядкованої та надійної передачі даних, звісно, є еквівалентним TCP: гарантія доставки така сама, як і у звичайного

з'єднання WebSocket. Однак DataChannel пропонує дві різні політики для налаштування часткової надійності кожного каналу:

- Частково надійна доставка з ретрансляцією: повідомлення не будуть повторно передані більше разів, ніж зазначено програмою.
- Частково надійна доставка з таймаутом: повідомлення не будуть повторно передані через певний час (у мілісекундах) програмою.

Обидві стратегії реалізовуються клієнтом WebRTC, а це означає, що все, що має зробити застосування, - це вибрати відповідну модель доставки та встановити правильні параметри на каналі. Немає необхідності керувати таймерами застосування або лічильниками ретрансляції.

2.5.4. Частково надійна доставка та розмір повідомлень

Немає ніяких перепон перед передачею великих повідомлень, проте треба звернути увагу на те, що велике повідомлення буде розбито на декілька пакетів для транспортування по мережі. Якщо хоча б один з пакетів втрачено на шляху, повідомлення не буде надіслано отримувачу. Для вирішення цієї проблеми існує дві стратегії: повторної передачі або зменшення розміру повідомлення. Найкращий підхід - це використовувати дві стратегії разом.

Оскільки частота втрати пакетів залежить від стану мережі, передбачити оптимальне налаштування для повторної передачі пакетів неможливо. Тому бажано звертати увагу саме на розмір повідомлень: ідеально, коли одне повідомлення вміщується в один пакет (розмір повідомлення має бути менше 1,150 байта).

2.6. Peer-to-peer з'єднання

Peer-to-peer з'єднання (яке ще називають P2P) є такою мережею, де група пристроїв поєднані між собою та мають однаковий рівень доступу та обов'язки щодо обробки даних. Порівняно з традиційною клієнт-сервер мережею, жоден пристрій в P2P не призначений виключно для того щоб демонструвати чи отримувати дані. Кожен браузер має ті ж права, що і інші браузери, та може використовуватись у тих самих цілях.

Щоб встановити успішне однорангове з'єднання, треба виконати ряд дій:

1. Повідомити іншого партнера про намір відкрити однорангове з'єднання, таким чином, щоб він знав про початок прослуховування вхідних пакетів.
2. Визначити потенційні шляхи маршрутизації для однорангового з'єднання по обидві сторони з'єднання і передати цю інформацію між браузерами.
3. Обмінятись необхідною інформацією про параметри медіа та дата потоків - протоколів, кодування і так далі.

Одну з цих задач виконує WebRTC: вбудований протокол ICE виконує необхідні перевірки маршрутизації та з'єднання. Однак реалізація доставки сповіщень (сигналізації) та переговорів про сесію залишаються за додатком.

2.6.1. Сигналізація та переговори про сесію

Перш ніж перевіряти підключення або узгоджувати сесію, треба з'ясувати: 1) чи доступний інший пристрій 2) чи бажає він встановити зв'язок. Фактично, необхідно надіслати пропозицію, а браузер має повернути

відповідь. Цьому процесу може завадити наступне: якщо інший браузер не слухає вхідні пакети, як повідомити його про намір з'єднання? Для цього потрібен канал сигналізації спільного користування (див.рис.2.6.1).

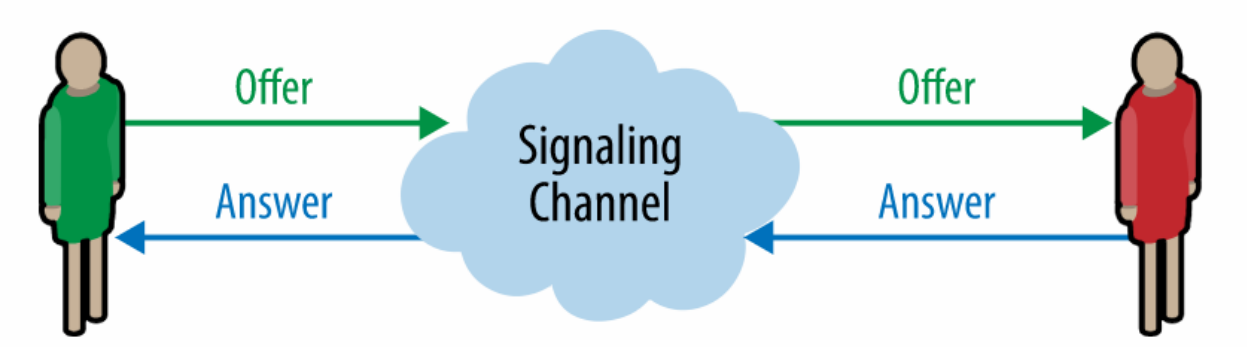

Рис. 2.6.1 [1] Спільний канал сигналізації

WebRTC перекладає відповідальність за вибір сигналізації транспорту та протоколу на застосування. Стандарт WebRTC не надає жодних рекомендацій або ж імплементації стаку сигналізації. Це забезпечує сумісність з низкою інших протоколів сигналізації існуючої інфраструктури комунікації:

- Session Initiation Protocol (SIP) ([13])
- Jingle ([14])
- ISDN User Part (ISUP) ([15]).

Застосунок WebRTC може вибрати будь-який із існуючих протоколів сигналізації та шлюзів, щоб домовитись про дзвінок або відеоконференцію з системою зв'язку: наприклад, подзвонити по телефону клієнту PSTN. Або ж можна зробити вибір у сторону імплементації власного сервісу сигналізації з самостійно налаштованим протоколом.

Сервер сигналізації може себе поводити як шлюз для існуючої мережі: тоді його обов'язком є повідомити цільовий браузер про пропозицію комунікації, і передати відповідь назад WebRTC клієнту, який ініціював комунікацію. Інший варіант - це використання власного каналу сигналізації

який може складатись з одного чи більше серверів та кастомного протоколу для передачі повідомлень.

Skype є прикладом однорангової системи із особливо налаштованою сигналізацією: аудіо- та відеозв'язок є peer-to-peer з’єднанням, але користувачі Skype повинні підключатись до сигнальних серверів Skype, які використовують власний протокол для ініціювання зв’язку однорангової мережі.

2.6.2. Session Description Protocol (SDP)

Якщо припустити, що у додатку вже імплементовано сервер сигналізації, можна переходити до першого обов’язкового кроку для ініціювання зв’язку WebRTC - надсилання згенерованої SDP пропозиції віддаленому браузеру через канал сигналізації:

```
var signalingChannel = new SignalingChannel();
var pc = new RTCPeerConnection({});

navigator.getUserMedia({ "audio": true }, getStream, errorHandler);

function getStream(stream) {
  pc.addStream(stream);

  pc.createOffer(function(offer) {
    // створення опису пропозиції
    pc.setLocalDescription(offer);
    signalingChannel.send(offer.sdp);
  });
}

function errorHandler() { ... }
```

WebRTC використовує SDP (Session Description Protocol) для опису параметрів peer-to-peer зв’язку. SDP використовується для опису “профілю сесії” - списку властивостей з’єднання:

- Типів даних для передачі (аудіо, відео та інші дані)
- Траспортів мережі
- Кодеків
- Пропускної здатності
- Інших метаданих.

Додатки що використовують WebRTC не мають працювати із SDP напряму. JSEP (JavaScript Session Establishment Protocol) абстрагує всю внутрішню систему SDP за методами, які можна викликати на RTCPeerConnection об’єкті.

2.6.3. Інтерактивне встановлення підключення – ICE

Для того, щоб встановити однорангове з’єднання, браузери мають мати можливість направляти пакети один одному. Для досягнення цього, WebRTC надає такий функціонал: кожен об’єкт зв’язку RTCPeerConnection має “агента ICE”. Цей агент відповідає за:

1. Збирання локальних IP, портових кортежів(кандидатів);
2. Проведення перевірки зв’язку між браузерами;
3. Надсилання “підтримок” (keepalives) зв’язку.

Після того, як опис сесії встановлено, локальний ICE агент автоматично починає процес виявлення всіх можливих кандидатів IP, портових кортежів для локального браузера.

1. Агент ICE надсилає запит на пошук локальних IP адрес операційній системі
2. Після цього, агент ICE надсилає запит на пошук зовнішнього STUN сервера для отримання публічних IP та порта-кортежа локального пристрою.
3. Далі, агент ICE додає TURN сервер в якості останнього кандидата для звернення. Якщо зв'язок peer-to-peer зазнає невдачі, дані будуть передані через вказаного посередника.

Коли новий кандидат(IP, порт-кортеж) знайдено, агент автоматично реєструє його з об'єктом RTCPeerConnection та сповіщує про це застосування за допомогою колбек-функції `onicecandidate`. Коли збір кандидатів завершено, цей колбек знову відпрацьовує.

Поглянемо на наступний приклад:

```
var iceParameters = {"iceServers": [
  {"url": "…"},
  {"url": "…", "username": "…", "credential": "…"}
]};

var signalingChannel = new SignalingChannel();
var pc = new RTCPeerConnection(iceParameters);

navigator.getUserMedia({ "audio": true }, getStream, errorHandler);

function getStream(stream) {
  pc.addStream(stream);

  pc.createOffer(function(offer) {
    pc.setLocalDescription(offer);
  });
}

pc.onicecandidate = function(evt) {
  if (evt.target.iceGatheringState == "complete") {
      local.createOffer(function(offer) {
        console.log("Offer with ICE candidates: " + offer.sdp);
        signalingChannel.send(offer.sdp);
      });
```

```
  }
}
```

У `iceParameters` змінній вказуємо URL до STUN та TURN серверів. У колбек-функції `onicecandidate` відбувається генерація SDP пропозиції - зі знайденими ICE кандидатами.

2.7. Безпека WebRTC

Браузери мають відповідати стандартам безпеки, що диктовані W3C, в тому числі і для WebRTC.

Контроль безпеки та конфіденційності WebRTC включає в себе:

- HTTPS – для того, щоб мати доступ до функцій WebRTC, використання HTTPS є обов'язковим
- Дозвіл на доступ до медіа – користувачі мають надавати доступ до камери, мікрофону, екрану кожному сайту, який пропонує користувачу передати дані таких типів
- Індикатори візуального використання – помітні, виразні показники мають відображати коли камера, мікрофон, або ж екран використовуються сайтом
- Анонімна інформація пристрою – інформація про пристрій залишається прихованою допоки користувач не надасть дозвіл сайту
- Захист від втрати IP – обмеження на поширення інформації про IP-адресу користувача допомагають уникнути проблем із конфіденційністю та відстеженням

Імплементація наведених вище функцій залежить від браузера, проте у більшості браузерів є приблизно однаковою.

Існує 2 основних підходи до безпеки застосунку. Безпека за допомогою: 1) непомітності; 2) проектування. Перший вид передбачає приховання механізмів розробленої системи, щоб зловмисники не могли нашкодити застосунку. Другий – навпаки, пропонує тримати систему відкритою, щоб спроби зламати її стали досвідом для розробників та допомогли вдосконалити архітектуру застосунку.

Організації, що займаються питаннями безпеки, як правило, не погоджуються із першим підходом. NIST [https://www.nist.gov/] – Національний інститут стандартів та технологій – має таку позицію[https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf]: «Безпека системи не має залежати від секретності імплементації або її компонентів». Тримання коду у таємниці не може тривати вічно, особливо від вмотивованих бажаючих зламати систему.

WebRTC також застосовує підхід проектування до безпеки своїх технологій. Браузери, які використовують WebRTC мають відкритий код та є ретельно тестованими.

Сам проект WebRTC теж має відкритий код, доповнений різними імплементаціями. Вони активно підтримуються розробниками, які постійно вдосконалюють WebRTC, включаючи напрямок безпеки.

РОЗДІЛ 3

Аналіз інтерфейсу Screen Capture API

Для написання коду для веб-додатків, браузери надають веб апі. Найчастіше, вони використовують разом із мовою JavaScript. Серед цих апі є Screen Capture API, яке є доповненням до Media Capture and Streams API.

Screen Capture API дозволяє користувачеві обрати екран або частину екрану для захоплення у якості медіапотоку. Після цього, потік можна записати або поділитись ним через мережу.

3.1. Видимі та логічні поверхні дисплея

Для Screen Capture API, поверхня дисплея є будь-яким об’єктом із контентом, який може бути вибраним для того, щоб його поширювати.
Є два типи поверхні дисплея: видимі та логічні поверхні.

Видима логічна поверхня є такою, яку повністю видно на екрані. Наприклад, вікно, вкладка або весь екран.

Логічна поверхня є частково або повністю затемненою, або ж є перекритою іншим об’єктом, повністю схованою, бути поза екраном.
Як правило, браузер надає зображення, яке якимось чином затуляє приховану частину логічної поверхні відображення. Наприклад, шляхом розмиття або заміни кольором або малюнком. Це робиться з міркувань безпеки, оскільки схований вміст може містити дані, якими користувач не хоче ділитися. Браузер може дозволити захоплення всього вмісту схованої частини вікна після отримання дозволу користувача.

3.2. Можливості Screen Capture API

Єдиний метод цього апі - `MediaDevices.getDisplayMedia()`, який просить користувача обрати екран або частину екрану для захоплення у формі `MediaStream`. Викликати метод можна на екземплярі `Media navigator.mediaDevices`:

```
captureStream = await
navigator.mediaDevices.getDisplayMedia(displayMediaOptions);
```

Почати захоплення екрану можна двома способами: у стилі async/await або Promise. В обох випадках, браузер відповідає тим, що презентує користувачу інтерфейс, у якому можна обрати частину екрану для подальшого поширення. Обидві імплементації `startCapture()` методу повертають `MediaStream`, що містить захоплене представлення дисплея.

Об'єкт так званих обмежень(constraints), переданий в `getDisplayMedia()`, є об'єктом `DisplayMediaStreamConstraints`, який використовується для налаштування потоку-результату.

Розглянемо можливі налаштування:

- cursor – вказує чи варто захоплювати курсор миші. Можливі значення цього параметра:
    1. always – курсор миші має завжди фіксуватись у згенерованому потоці.
    2. motion – курсор має бути видимим у потоці лише під час руху та, на розсуд користувача, протягом короткого часу перед переміщенням. Потім курсор видаляється з потоку.

3. never – курсор ніколи не повинен бути видимим у створеному потоці.

- logicalSurface - логічне значення, яке, якщо true, вказує, що захоплення повинне включати позаекранні ділянки джерела, якщо такі є.

3.3. Передача аудіо потоків

Браузери можуть дозволити захопити аудіо разом із відео-контентом. Доступними джерелами звуку є 1) вибране вікно, 2) вся аудіосистема комп'ютера 3) мікрофон користувача. Або комбінація всього перерахованого.

Перш ніж додавати підтримку передачі аудіопотоків у своєму додатку, треба перевірити чи підтримує браузер аудіо у потоках із захопленим екраном.

Відомо, що щоб ділитись екраном разом із аудіо, треба передати методу `getDisplayMedia()` відповідний параметр: `{audio: true}`. Проте налаштування можуть бути і більш деталізованими. Наприклад, щоб увімкнути функцію заглушення шуму та ехо, зазначимо:

```
audio: {
    echoCancellation: true,
    noiseSuppression: true
  }
```

3.4. Безпека та Screen Capture API

При передачі даних через мережу, важливо враховувати конфіденційність та наслідки для безпеки після поширення екрану користувача. Найбільшим потенційним ризиком є ті випадки, коли користувач ділиться інформацією ненавмисно, і він не бажав нею ділитись.

Наприклад, порушення конфіденційності та безпеки можуть статись, якщо користувач ділиться своїм екраном, в той час як видиме фонове вікно містить особисту інформацію, або ж менеджер паролів видно у спільному потоці. Небезпечним є і захоплення логічних поверхонь, які можуть містити інформацію, про яку користувач взагалі не знає, не кажучи вже про те, щоб бачити.

Браузери повинні затушовувати вміст, який насправді не видно на екрані, у всіх випадках крім тих, коли було надано дозвіл на спільний доступ до цього вмісту. А захоплення вмісту дисплею має бути авторизованим: перш ніж розпочати потокове передавання захопленого вмісту екрана, браузер просить користувача підтвердити запит на спільний доступ та вибрати вміст для поширення.

РОЗДІЛ 4

4.1. Аналіз технічного завдання

У сайту є лише одна група користувачів. Користувач є учасником конференції: має можливість її розпочати, до неї приєднатись, та від’єднатись від дзвінка. Розглянемо детальніше технічні вимоги користувачів до функціональності системи:

- Можливість увійти у конференцію, вказавши своє ім’я;
- Написати повідомлення іншому учаснику конференції;
- Переглянути чат;
- Подзвонити по відео-зв’язку іншому учаснику конференції;
- Поділитись екраном свого пристрою;
- Закінчити дзвінок.

Опис інтерфейсу:

Початкова сторінка сайту містить вікно для введення імені користувача. Після введення власного імені, користувач потрапляє на сторінку, де присутній перелік користувачів, які є онлайн. Після натискання на ім’я іншого користувача, з’являється чат. Натискання на кнопку «Відео» розпочинає дзвінок.

4.2. Обґрунтування алгоритму й структури програми

Алгоритм застосунку є наступним: учасник конференції вводить своє ім'я, потім обирає іншого учасника для здійснення дзвінка, та дзвонить йому. Якщо інший учасник не погоджується на дзвінок, то дзвінок не відбувається.

У застосунку можна виділити основні модулі: вхід, чат із іншим учасником конференції, дзвінок іншому учаснику конференції.

4.3. Обґрунтування вибору засобів розробки

Для розробки сайту обрано мову JavaScript – вона використовується разом із API-колами WebRTC та Screen Capture API. Ця мова є найпопулярнішим та найзручнішим способом для написання сервісу відео-зв’язку. Розміщення елементів на сторінці та створення зручного UI/UX використано HTML та CSS технології відповідно. Сервер сигналізації було написано на мові C# та з використанням .NET фреймворку.

Для реалізації сайту було обрано середовище розробки WebStorm, яке призначене для розробки засобами JavaScript, HTML та CSS. Це IDE надає можливість рефакторингу, автодоповнює код та перевіряє його на помилки. Також WebStorm дозволяє швидко переміщатись по проекту та безпосередньо по файлам, завдяки зручному відображенню структури. IDE можна легко налаштувати під власні потреби: обрати тему інтерфейсу, стиль коду, встановити плагіни. Для розробки сервера сигналізації було використано Visual Studio, яка найкраще підходить для розробки на мові C#.

4.4. Опис розробки програми

Розроблено сайт, що є сторінкою користувача, та сервер сигналізації, який керує зв’язком між пристроями та передачею повідомлень.

Проект сайту складається із HTML-сторінки, файлу CSS із стилями сторінки, та JavaScript-файлів, які відповідають за перевірку та передачу

введених даних іншому учаснику конференції. Ці файли: index.html, style.css, LoginForm.js, MessengerForm.js, SignalingChannel.js та VideoMessenger.js.

Сервер сигналізації містить файли:

- ChatMessage.cs – є об’єктом повідомлення чату,
- MessageRequest.cs – є об’єктом запиту,
- SignalingHub.cs – головний клас, що відповідає за передачу повідомлень між браузерами.

  Також варто зазначити, що файл Program.cs містить конфігурацію SSL сертифікату.

4.5. Тестування програми і результати її виконання

Вхідними даними застосунку є запити користувача на виконання певної дії, фактично, кліки на кнопки інтерфейсу. Вихідні дані – виконання відповідної функції, наприклад, захоплення екрану користувача та передача цього потоку іншому користувачу.

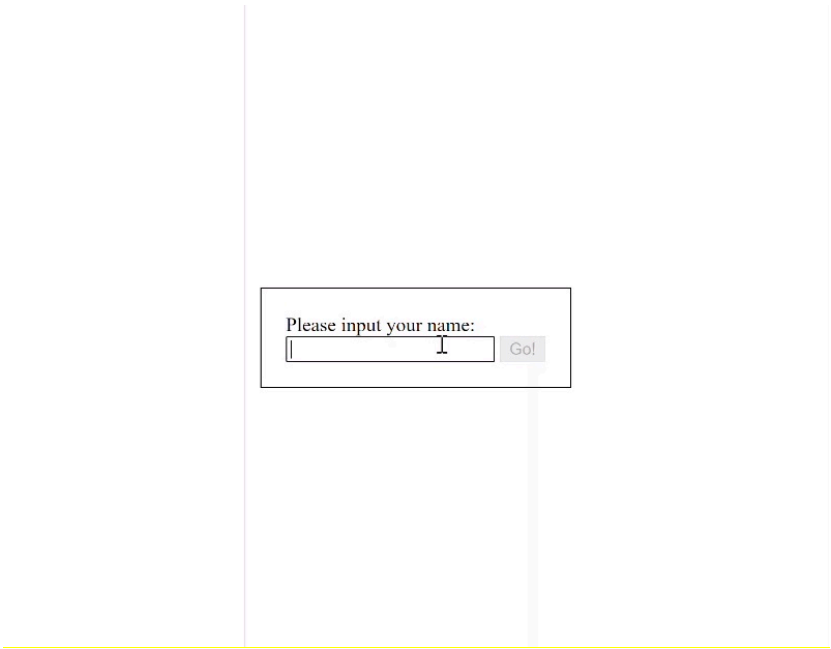

Рис. 4.5.1. Введення імені користувача

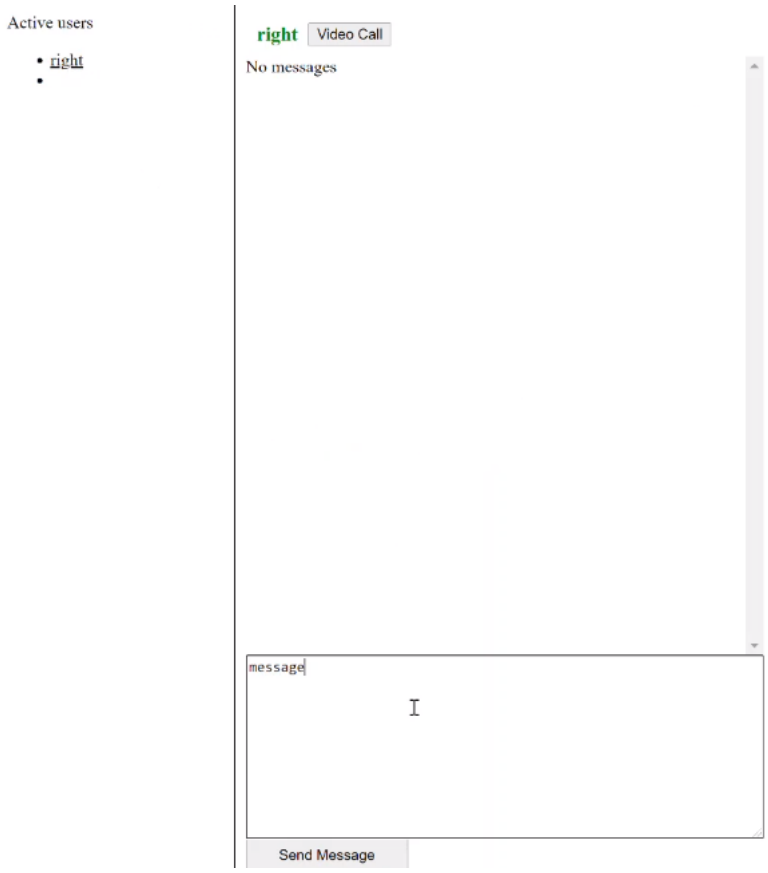

Рис. 4.5.2 Надсилання повідомлення

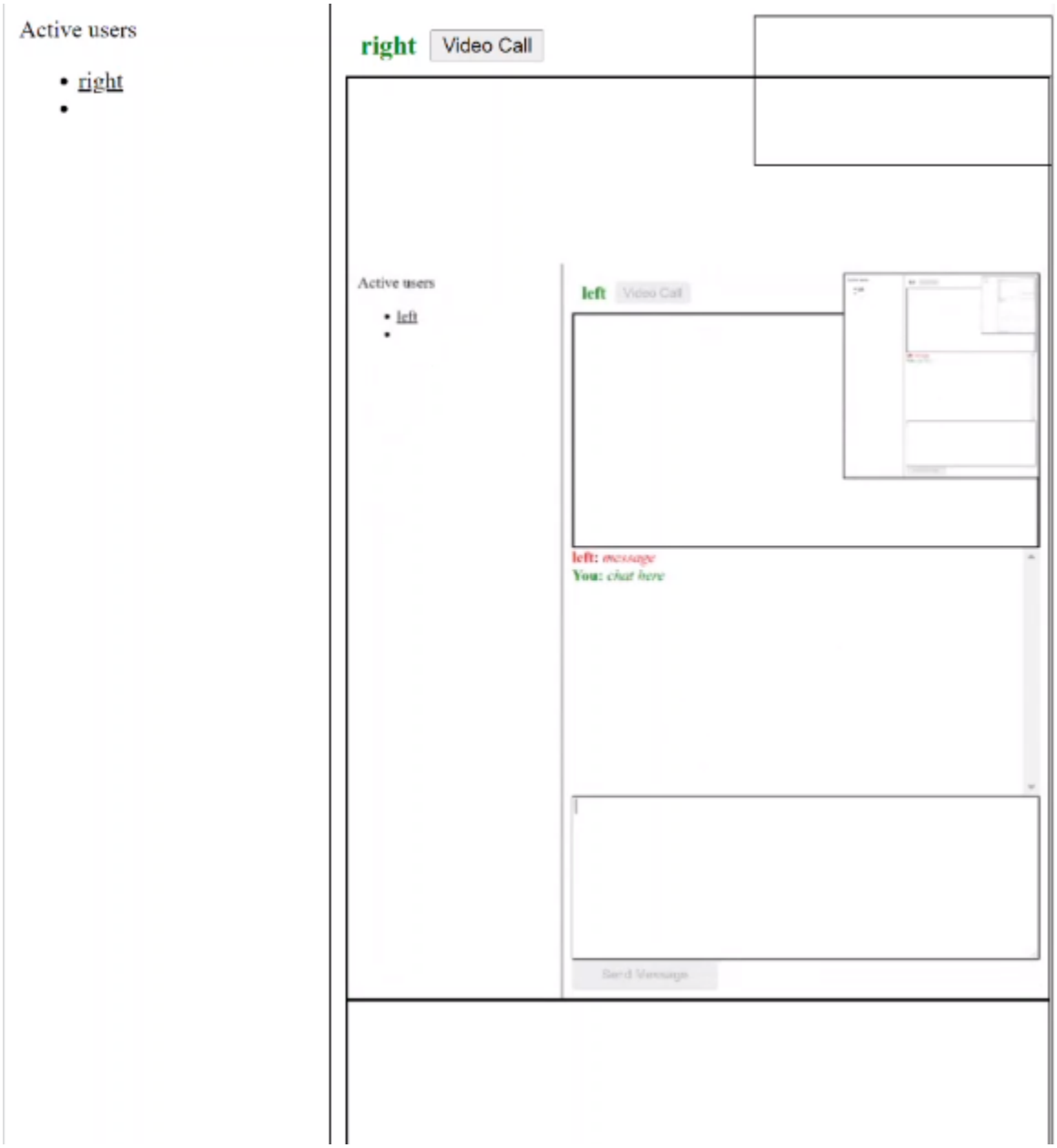

Рис. 4.5.3 Відображення екрану іншого користувача

Висновки

Отже, поставлена задача була виконана: досліджено найновіші технології для передачі відео, аудіо та інших даних від браузера до браузера. Технології протестовано та сайт для проведення онлайн-конференцій створено. Враховуючи те, що частина протестованих функцій є експериментальними, можемо зробити висновок що вони є готовими до використання, проте потребують регулярної підтримки та перевірки з боку розробника.

Обрані засоби розробки задовольнили технічні вимоги: середовище розробки надало можливість контролювати кожен процес, файл, із одного інтерфейсу – вікна середовища розробки. А поєднання мови JavaScript та Web апі-колів дозволило виконати поставлену задачу щодо реалізації сайту для створення відео-дзвінків.

Розроблений сайт є в першу чергу демонстрацією роботи розглянутих технологій, зокрема WebRTC та Screen Capture API. Присутні і напрямки для вдосконалення. Такі сервіси як Zoom Cloud Meetings та Google Meets із кожною своєю новою версією додають нові корисні можливості для користувачів застосунків, що є поштовхом для вдосконалення створеного в рамках цієї роботи сайту.

Список використаної літератури

1. Книга High Performance Browser Networking, розділ WebRTC [Електронний ресурс] https://hpbn.co/webrtc/

2. Опис технології WebRTC від MDN Web Docs [Електронний ресурс] https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API

3. Стаття із інформацією щодо функціоналу Google Meets [Електронний ресурс] https://apps.google.com/meet/how-it-works/

- Сторінка із інформацією про розширення Zoom для браузера Chrome [Електронний ресурс] https://chrome.google.com/webstore/detail/zoom/hmbjbjdpkobdjplfobhljndfdfdipjhg?hl=uk

5.Документація WebRTC від W3 [Електронний ресурс] https://www.w3.org/TR/webrtc/

6. Документація протоколу STUN [Електронний ресурс] https://tools.ietf.org/html/rfc5389

7. Документація протоколу TURN [Електронний ресурс] https://tools.ietf.org/html/rfc5766

8. Опис протоколу SDP [Електронний ресурс] https://developer.mozilla.org/en-US/docs/Glossary/SDP

9. Опис протоколу DTLS [Електронний ресурс]

https://developer.mozilla.org/en-US/docs/Glossary/DTLS

10. Опис протоколу SCTP [Електронний ресурс]

https://developer.mozilla.org/en-US/docs/Glossary/SCTP

11. Опис протоколу SRTP [Електронний ресурс]

https://developer.mozilla.org/en-US/docs/Glossary/RTP

12. Документація RTCPeerConnection API [Електронний ресурс]

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

13. Документація SIP протоколу [Електронний ресурс]

https://tools.ietf.org/html/rfc3261

14. Документація Jingle протоколу [Електронний ресурс]

https://xmpp.org/extensions/xep-0166.html

15. Опис протоколу ISUP [Електронний ресурс]

https://www.dialogic.com/webhelp/MSP1010/10.4.0/WebHelp/msp_dg/isup/introduction_to2.htm

16.Визначення PSTN мережі [Електронний ресурс]

https://www.sciencedirect.com/topics/computer-science/public-switched-telephone-network

Додаток А. Серверний код

ChatMessage файл:

```
using System;

namespace SignalingServer.Hubs
{
    public struct ChatMessage
    {
        public ChatMessage(string from, string to, string message)
        {
            this.from = from;
            this.to = to;
            this.message = message;
            this.dateTime = DateTime.UtcNow;
        }
        public string from { get; }
        public string to { get; }
        public string message { get; }
        public DateTime dateTime { get; }
    }
}
```

MessageRequest файл:

```
using System.Collections.Generic;

namespace SignalingServer.Hubs
{
    public class MessageRequest
    {
        public string type { get; set; }
       // public string username { get; set; }
        public string data { get; set; }
    }
}
```

SignalingHub файл:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.Linq;
using System.Security.Authentication;
using Microsoft.AspNetCore.SignalR;

namespace SignalingServer.Hubs
{
    public class SignalingHub : Hub
    {
        private static Dictionary<string, string> users = new Dictionary<string,
string>();
```

```
        private static Dictionary<string, Dictionary<string, List<ChatMessage>>> chats =
new Dictionary<string, Dictionary<string, List<ChatMessage>>>();
        public async override Task OnDisconnectedAsync(Exception? exception)
        {
            Console.WriteLine("OnDisconnectedAsync");
            SignalingHub.users.Remove(Context.ConnectionId);
            await this.broadcastListOfUsers();
        }
        public async override Task OnConnectedAsync()
        {
            Console.WriteLine("OnConnectedAsync");
            SignalingHub.users.Remove(Context.ConnectionId);
            await this.broadcastListOfUsers();
        }
        private async Task broadcastListOfUsers()
        {
            Console.WriteLine("broadcast list of users: ",
SignalingHub.users.Values.ToList());
            await Clients.All.SendAsync("updateUserList",
SignalingHub.users.Values.ToList());
        }
        public async Task login(string username, string password)
        {
            Console.WriteLine("login: " + username);
            if (password == "htXaSexrohWKzuF7ok6")
            {
                SignalingHub.users.Add(Context.ConnectionId, username);
                await this.broadcastListOfUsers();
            }
            else
            {
                throw new AuthenticationException("Wrong password!");
            }
        }
        public async Task send(Object request)
        {
            Console.WriteLine("send - " + request);
            await Clients.Others.SendAsync("message", request);
        }
        public async Task message(MessageRequest request)
        {
            Console.WriteLine("message - " + request.type);
            await Clients.Others.SendAsync("message", request);
        }
        public async Task textMessage(string username, string contact, string message)
        {
            ChatMessage _message = new ChatMessage(username, contact, message);
            var history = this._getChatHistory(username, contact);
            history.Add(_message);

            var addresseeHistory = this._getChatHistory(contact, username);
            addresseeHistory.Add(_message);
            await Clients.Client(Context.ConnectionId).SendAsync("chatHistory",
addresseeHistory);

            var contactId = SignalingHub.users.FirstOrDefault(x => x.Value ==
contact).Key;
            await Clients.Client(contactId).SendAsync("chatHistory", history);
```

```
        }
        public async Task chatHistory(string username, string contact)
        {
            Console.WriteLine("chatHistory: " + username + "/" + contact);
            var history = this._getChatHistory(username, contact);
            await Clients.Client(Context.ConnectionId).SendAsync("chatHistory", history);
        }
        private List<ChatMessage> _getChatHistory(string from, string to)
        {

            if (!SignalingHub.chats.ContainsKey(from))
            {
                SignalingHub.chats.Add(from, new Dictionary<string,
List<ChatMessage>>());
            }
            var allChats = SignalingHub.chats[from];
            if (!allChats.ContainsKey(to))
            {
                allChats.Add(to, new List<ChatMessage>());
            }
            return allChats[to];
        }
        public async Task initVideoCall(string username, string contact)
        {
            Console.WriteLine("initVideoCall: " + username + "/" + contact);

            var contactId = SignalingHub.users.FirstOrDefault(x => x.Value ==
contact).Key;
            await Clients.Client(contactId).SendAsync("initVideoCall", username);
        }
        public async Task declineVideoCall(string username, string contact)
        {
            Console.WriteLine("declineVideoCall: " + username + "/" + contact);

            var contactId = SignalingHub.users.FirstOrDefault(x => x.Value ==
contact).Key;
            await Clients.Client(contactId).SendAsync("videoCallDeclined", username);
        }
        public async Task videoCallAccepted(string username, string contact)
        {
            Console.WriteLine("videoCallAccepted: " + username + "/" + contact);

            var contactId = SignalingHub.users.FirstOrDefault(x => x.Value ==
contact).Key;
            await Clients.Client(contactId).SendAsync("videoCallAccepted", username);
        }
    }
}
```

Program файл:

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Security.Cryptography.X509Certificates;
```

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.Server.Kestrel.Core;
using Microsoft.AspNetCore.Server.Kestrel.Https;
using Microsoft.Extensions.Configuration;

namespace SignalingServer
{
    class Program
    {
        private static readonly Dictionary<string, string> DEFAULTS = new
Dictionary<string, string>
            {
                { "urls", "http://0.0.0.0:5012" }
            };
        static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();

            System.Threading.Thread.Sleep(-1);
        }
        public static IHostBuilder CreateHostBuilder(string[] args)
        {
            var configBuilder = new ConfigurationBuilder();
            configBuilder.AddInMemoryCollection(DEFAULTS)
                .AddJsonFile("appsettings.json");
            var configuration = configBuilder.Build();

            var hostBuilder = Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((hostingContext, appConfigBuilder) =>
                {
                    appConfigBuilder.AddConfiguration(configuration);
                })
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder
                        .UseUrls()
                        .ConfigureKestrel(serverOptions =>
                            {

serverOptions.ListenAnyIP(configuration.GetValue<int>("port"), listenOptions =>
                                {
                                    if
(configuration.GetSection("HTTPS:Enabled").Get<bool>())
                                    {
                                        try
                                        {
                                            listenOptions.UseHttps(httpsOptions =>
                                            {
                                                if
(configuration.GetSection("HTTPS:Certificate").Exists())
                                                {
                                                    if
(configuration.GetSection("HTTPS:Certificate:Path").Exists())
                                                    {
                                                        var filename =
configuration.GetSection("HTTPS:Certificate:Path").Get<string>();
```

```
                                                    if
(configuration.GetSection("HTTPS:Certificate:Password").Exists())
                                                    {
                                                        var password =
configuration.GetSection("HTTPS:Certificate:Password").Get<string>();
                                                        var certWithPassword = new
X509Certificate2(filename, password);

httpsOptions.ServerCertificate = certWithPassword;
                                                    }
                                                    var cert = new
X509Certificate2(filename);
                                                    httpsOptions.ServerCertificate =
cert;
                                                }
                                                else
                                                {
                                                    var subject =
configuration.GetSection("HTTPS:Certificate:Subject").Get<string>();
                                                    var store =
configuration.GetSection("HTTPS:Certificate:Store").Get<string>();
                                                    var location =
configuration.GetSection("HTTPS:Certificate:Location").Get<string>();
                                                    var allowInvalid =
configuration.GetSection("HTTPS:Certificate:AllowInvalid").Get<bool>();
                                                    if (Enum.TryParse(location, out
StoreLocation storeLocation))
                                                    {
                                                        var cert =
CertificateLoader.LoadFromStoreCert(subject, store, storeLocation, allowInvalid);

httpsOptions.ServerCertificate = cert;
                                                    }
                                                    else
                                                    {
                                                        var cert =
CertificateLoader.LoadFromStoreCert(subject, store, StoreLocation.CurrentUser,
allowInvalid);

httpsOptions.ServerCertificate = cert;
                                                    }
                                                }
                                            }
                                        });
                                    }
                                    catch (Exception e)
                                    {
                                        throw new Exception($"Https configuration
failed. Details: {e.Message}", e);
                                    }
                                    var protocols =
configuration.GetSection("HTTPS:Protocols").Get<string>();
                                    if (Enum.TryParse(protocols, out HttpProtocols
httpProtocols))
                                    {
                                        listenOptions.Protocols = httpProtocols;
                                    }
                                }
```

```
                                });

                            })
                        .UseStartup<Startup>();
                });


            return hostBuilder;
        }
    }
}
```

Додаток Б. Клієнтський код

index.html:

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="style.css" type="text/css" />
  <script src="lib/signalr.js"></script>
  <script src="js/SignalingChannel.js"></script>
  <script src="js/VideoMessenger.js"></script>
  <script src="js/LoginForm.js"></script>
  <script src="js/MessengerForm.js"></script>
  <title>WebRTC</title>
</head>

<body>
  <div class="login-container" id="loginContainer">
    <div class="background"></div>
    <div class="form">
      Please input your name:<br />
      <input type="text" id="txtLoginName" autofocus />
      <button id="btnLoginGo" disabled>Go!</button>
    </div>
  </div>
  <div class="container">
    <div class="user-list" id="userList">Active users
      <ul id="listUsers"></ul>
    </div>
    <div class="messenger-container" id="messengerContainer">
      <div class="contact-info" id="contactInfo">
        <div class="name" id="lblContactName">
        </div>
        <button id="btbVideoCall">Video Call</button>
      </div>
      <div class="video" id="videoArea">
        <video id="videoContainerRemote" class="remote" autoplay
          playsinline></video>
        <video id="videoContainerLocal" class="local" autoplay
          playsinline></video>
      </div>
      <div class="chat" id="chatArea">
        <div class="history" id="historyArea">
        </div>
        <div class="message-area" id="messageArea">
          <textarea class="message" id="txtMessage"></textarea>
          <button class="send-button" id="btnSendMessage" disabled>Send
Message</button>
```

```
        </div>
      </div>
    </div>
  </div>
  <script type="text/javascript">

    'use strict';

    const videoMessenger = new VideoMessenger();

    const loginForm = new LoginForm(videoMessenger);
    const messengerForm = new MessengerForm(videoMessenger);


  </script>

</body>

</html>
```

LoginForm.js:

```
class LoginForm {
    constructor(messenger) {
        this.messenger = messenger;
        this.loginContainer = document.getElementById("loginContainer");
        this.txtLoginName = document.getElementById("txtLoginName");
        this.btnLoginGo = document.getElementById("btnLoginGo");

        this.txtLoginName.onchange = (e) => {
            this._validateLogin();
        }
        this.txtLoginName.onkeypress = (e) => {
            if (this._validateLogin()) {
                if (e.key === 'Enter' || e.keyCode === 13) { // enter
                    e.preventDefault();
                    this._submitLogin();
                }
            }
        }
        this.btnLoginGo.onclick = (e) => {
            this._submitLogin();
        }
    }
    _validateLogin() {
        var valid = (/^[A-Za-z0-9\s]+$/.test(this.txtLoginName.value));
        this.btnLoginGo.disabled = !valid;
        return valid;
    }
    _submitLogin() {
        this.messenger.login(this.txtLoginName.value);
        this.loginContainer.style.display = "none";
    }
}
```

MessengerForm:

```
class MessengerForm {
    constructor(messenger) {
        console.log("MessengerForm")
        this.messenger = messenger;
        this.messenger.onuserlistupdated = list =>
this._onUserListUpdated(list);
        this.messenger.onchathistory = history =>
this._onChatHistory(history);
        this.messenger.oninitvideocall = contact =>
this._onInitVideoCall(contact);
        this.messenger.onvideocalldeclined = contact =>
this._onVideoCallDeclined(contact);
        this.messenger.onvideocallaccepted = contact =>
this._onVideoCallAccepted(contact);

        this.listUsers = document.getElementById("listUsers");
        this.listUsers.onclick = (e) => this._onContactClick(e);

        this.messengerContainer =
document.getElementById("messengerContainer");
        this.lblContactName = document.getElementById("lblContactName");
        this.contactInfo = document.getElementById("contactInfo");
        this.chatArea = document.getElementById("chatArea");
        this.historyArea = document.getElementById("historyArea");

        this.txtMessage = document.getElementById("txtMessage");
        this.txtMessage.onchange = (e) => this._onMessageChange(e);
        this.txtMessage.onkeypress = (e) => this._onMessageKeyPress(e);

        this.btnSendMessage = document.getElementById("btnSendMessage");
        this.btnSendMessage.onclick = () => this._onSendMessage();

        this.btbVideoCall = document.getElementById("btbVideoCall");
        this.btbVideoCall.onclick = (e) => this._onVideoCallClick(e);

        this.videoArea = document.getElementById("videoArea");
        this.videoContainerRemote =
document.getElementById("videoContainerRemote");
        this.videoContainerLocal =
document.getElementById("videoContainerLocal");
    }
    _onUserListUpdated(list) {
        console.log("MessengerForm on user list updated: ", list);
        const items = list.map(
            x => {
                return `<li contact="${x}" >${x}<li>`;
            }
        );
        this.listUsers.innerHTML = items.join("");
    }
    _onContactClick(e) {
```

```
        if (e.target.hasAttribute("contact")) {
            const contact = e.target.getAttribute("contact");
            this._loadContact(contact);
        }
    }
    _loadContact(contact) {
        console.log("LOAD CONTACT: " + contact);
        this._activeContact = contact;
        this.contactInfo.style.display = "flex";
        this.chatArea.style.display = "flex";
        this.lblContactName.innerText = contact;
        this.messenger.chatHistory(contact);
    }
    _onChatHistory(history) {
        console.log("MessengerForm on chat history: ", history);
        if (this._activeContact === undefined) return;
        const items = history.map(
            x => {
                const contactStyle = x.from === this._activeContact ? "item-
to" : "item-from";
                const from = x.from === this._activeContact ? x.from : "You";
                return `<div class="${contactStyle}"><b>${from}:</b>
<i>${x.message}</i></div>`;
            }
        );
        if (items.length === 0) {
            this.historyArea.innerHTML = "No messages";
        } else {
            this.historyArea.innerHTML = items.join("");
        }
    }
    _onMessageChange(e) {
        this.btnSendMessage.disabled = !this._validateMessage();
    }
    _onMessageKeyPress(e) {
        this.btnSendMessage.disabled = !this._validateMessage();
        if (!this.btnSendMessage.disabled) {
            if (e.key === 'Enter' || e.keyCode === 13) { // enter
                e.preventDefault();
                this._onSendMessage();
            }
        }
    }
    _validateMessage() {
        if (this._activeContact === undefined) return false;
        return this.txtMessage.value.trim().length > 0;
    }
    _onSendMessage() {
        this.messenger.sendTextMessage(this._activeContact,
this.txtMessage.value.trim());
        this.btnSendMessage.disabled = true;
        this.txtMessage.value = "";
    }
    _onVideoCallClick(e) {
        e.target.disabled = true;
        this.messenger.initVideoCall(this._activeContact);
    }
```

```
    _onInitVideoCall(contact) {
        console.log("_onInitVideoCall " + contact)
        if (window.confirm(`Would you like to accept call from
"${contact}"`)) {
            this._switchToVideoMode();
            this.messenger.acceptVideoCall(contact,
this.videoContainerRemote);
        } else {
            this.messenger.declineVideoCall(contact);
        }
    }
    _onVideoCallDeclined(contact) {
        this.btbVideoCall.disabled = false;
        alert("Video call was declined!");
    }
    _onVideoCallAccepted(contact) {
        console.log("_onVideoCallAccepted");
        this._loadContact(contact);
        this._switchToVideoMode();
        this.messenger.startCall(this.videoContainerLocal);
    }
    _switchToVideoMode() {
        console.log("_switchToVideoMode");
        this.chatArea.style.display = "none";
        this.videoArea.style.display = "flex";
    }
}
```

SignalingChannel.js:

```
class SignalingChannel {

    // requests
    static SEND_MESSAGE_REQUEST = "send";
    static LOGIN_REQUEST = "login";
    static INIT_VIDEO_CALL_REQUEST = "initVideoCall";
    static DECLINE_VIDEO_CALL_REQUEST = "declineVideoCall";
    static VIDEO_CALL_ACCEPTED_REQUEST = "videoCallAccepted";
    static CHAT_HISTORY_REQUEST = "chatHistory";
    static TEXT_MESSAGE_REQUEST = "textMessage";

    // responses
    static RECEIVE_MESSAGE_RESPONSE = "message";
    static UPDATE_USERS_RESPONSE = "updateUserList";
    static CHAT_HISTORY_RESPONSE = "chatHistory";
    static INIT_VIDEO_CALL_RESPONSE = "initVideoCall";
    static VIDEO_CALL_DECLINED_RESPONSE = "videoCallDeclined";
    static VIDEO_CALL_ACCEPTED_RESPONSE = "videoCallAccepted";

    constructor(url) {
        console.log("SignalingChannel: constructor " + url);
        this.connected = false;

        this._connection = new
```

```
signalR.HubConnectionBuilder().withUrl(url).build();
        this._connection.on(SignalingChannel.RECEIVE_MESSAGE_RESPONSE,
            (message) => {
                console.log("SignalingChannel: message was received ",
message);
                if (this.onmessage) {
                    this.onmessage(message);
                }
            }
        );
        this._connection.on(SignalingChannel.UPDATE_USERS_RESPONSE,
            (list) => {
                console.log("SignalingChannel: users were updated ", list);
                list = list.filter(name => name != this.username);
                if (this.onuserlistupdated) {
                    this.onuserlistupdated(list);
                }
            }
        );
        this._connection.on(SignalingChannel.CHAT_HISTORY_RESPONSE,
            (history) => {
                console.log("SignalingChannel: chat history was loaded ",
history);
                if (this.onchathistory) {
                    this.onchathistory(history);
                }
            }
        );
        this._connection.on(SignalingChannel.INIT_VIDEO_CALL_RESPONSE,
            (contact) => {
                console.log("SignalingChannel: init video call ", contact);
                if (this.oninitvideocall) {
                    this.oninitvideocall(contact);
                }
            }
        );
        this._connection.on(SignalingChannel.VIDEO_CALL_DECLINED_RESPONSE,
            (contact) => {
                console.log("SignalingChannel: video call declined",
contact);
                if (this.onvideocalldeclined) {
                    this.onvideocalldeclined(contact);
                }
            }
        );
        this._connection.on(SignalingChannel.VIDEO_CALL_ACCEPTED_RESPONSE,
            (contact) => {
                console.log("SignalingChannel: video call accepted",
contact);
                if (this.onvideocallaccepted) {
                    this.onvideocallaccepted(contact);
                }
            }
        );
        this._connection.onclose(
            (message) => {
                this.connected = false;
```

```
                console.log("SignalingChannel: connection closed", message);
            }
        );
        this._connection.start().then(
            () => {
                this.connected = true;
                console.log("SignalingChannel: connection established");
            }
        ).catch(
            () => {
                this.connected = false;
                console.log("SignalingChannel: connection failed");
            }
        );
    }
    send(message) {
        console.log("SignalingChannel: message was sent ", message);
        this._connection.invoke(
            SignalingChannel.SEND_MESSAGE_REQUEST, message
        );
    }
    login(name) {
        console.log("SignalingChannel: login " + name);
        this.username = name;
        this._connection.invoke(
            SignalingChannel.LOGIN_REQUEST, name, "htXaSexrohWKzuF7ok6");
    }
    sendTextMessage(contact, message) {
        console.log(`SignalingChannel: send text message [${this.username} /
${contact}] "${message}"`);
        this._connection.invoke(
            SignalingChannel.TEXT_MESSAGE_REQUEST, this.username, contact,
message);
    }
    chatHistory(contact) {
        console.log(`SignalingChannel: chatHistory [${this.username} /
${contact}]`);
        this._connection.invoke(
            SignalingChannel.CHAT_HISTORY_REQUEST, this.username, contact);
    }
    initVideoCall(contact) {
        console.log(`SignalingChannel: init video call [${this.username} /
${contact}]`);
        this._connection.invoke(
            SignalingChannel.INIT_VIDEO_CALL_REQUEST, this.username,
contact);
    }
    videoCallAccepted(contact) {
        console.log(`SignalingChannel: video call accepted [${this.username}
/ ${contact}]`);
        this._connection.invoke(
            SignalingChannel.VIDEO_CALL_ACCEPTED_REQUEST, this.username,
contact);
    }
    declineVideoCall(contact) {
        console.log(`SignalingChannel: decline video call [${this.username} /
${contact}]`);
```

```
        this._connection.invoke(
            SignalingChannel.DECLINE_VIDEO_CALL_REQUEST, this.username,
contact);
    }
}
```

VideoMessenger.js:

```
class VideoMessenger {

    constructor() {
        console.log("VideoMessenger: constructor ");
        this.SIGNALING_SERVER_URL = "http://localhost:5012/webrtc";
        this.ICE_SERVERS = [{ urls: 'stun:stun.example.org' }, { 'urls':
'stun:stun.l.google.com:19302' }];
        this.signalingChannel = new
SignalingChannel(this.SIGNALING_SERVER_URL);
    }
    startCall(videoContainer) {
        console.log("Sender: start call");
        const peerConnection = new RTCPeerConnection({ iceServers:
this.ICE_SERVERS });
        peerConnection.addEventListener('icecandidate', event => {
            console.log("Sender: RTCPeerConnection: on icecandidate",
peerConnection.iceConnectionState);
            if (event.candidate) {
                this.signalingChannel.send(
                    event.candidate
                );
            }
        });
        peerConnection.addEventListener('connectionstatechange', event => {
            console.log("Sender: RTCPeerConnection: on
connectionstatechange");
            if (peerConnection.connectionState === 'connected') {
                console.log("Sender: RTCPeerConnection: Peers connected!");
            }
        });
        this.signalingChannel.onmessage = async message => {
            if (message.answer) {
                console.log("Sender: SignalingChannel: answer");
                const remoteDesc = new RTCSessionDescription(message.answer);
                peerConnection.setRemoteDescription(remoteDesc);
            }
        };

        // const constraints = { audio: false, video: true };
        const constraints = {
            "video": {
                "width": 640,
                "height": 480
            },
            audio: false
        }
```

```
        navigator.mediaDevices.getDisplayMedia(constraints).then(
        //navigator.mediaDevices.getUserMedia(constraints).then(
            (localStream) => {
                console.log("Sender: localStream ");
                videoContainer.srcObject = localStream;
                localStream.getTracks().forEach(track => {
                    console.log("Sender: track in ");
                    peerConnection.addTrack(track, localStream);
                });
                peerConnection.createOffer().then(
                    offer => {
                        peerConnection.setLocalDescription(offer).then(
                            () => {
                                this.signalingChannel.send({ 'offer': offer
});
                            }
                        )
                    }
                )
            }
        ).catch(
            (error) => { console.error(error) }
        )
    }

    acceptVideoCall(contact, videoContainer) {
        console.log("Receiver: accept call " + contact);
        this.signalingChannel.videoCallAccepted(contact);

        const peerConnection = new RTCPeerConnection({ iceServers:
this.ICE_SERVERS });
        this.signalingChannel.onmessage = async message => {
            if (message.offer) {
                console.log("Receiver: SignalingChannel: offer");
                peerConnection.setRemoteDescription(new
RTCSessionDescription(message.offer));
                const answer = await peerConnection.createAnswer();
                await peerConnection.setLocalDescription(answer);
                this.signalingChannel.send({ 'answer': answer });
            } else if (message.candidate) {
                console.log("Receiver: SignalingChannel: candidate");
                try {
                    await peerConnection.addIceCandidate(message);
                } catch (e) {
                    console.error('Receiver: Error adding received ice
candidate', e);
                }
            }
        };

        peerConnection.addEventListener('connectionstatechange', event => {
            console.log("Receiver: RTCPeerConnection: on
connectionstatechange", peerConnection.iceConnectionState);
            if (peerConnection.connectionState === 'connected') {
                console.log("Receiver: RTCPeerConnection: Peers connected!");
            }
        });
```

```
        peerConnection.addEventListener('track', async (event) => {
            console.log("Receiver: track out", event);
            if (videoContainer.srcObject !== event.streams[0]) {
                console.log("Receiver: track out + ");
                videoContainer.srcObject = event.streams[0];
            }
        });
    }
    getUserList() {
        console.log("VideoMessenger: get user list ");
        return this.signalingChannel.getUserList();
    }
    login(name) {
        console.log("VideoMessenger: login " + name);
        return this.signalingChannel.login(name);
    }
    set onuserlistupdated(handler) {
        console.log("set onuserlistupdated")
        this.signalingChannel.onuserlistupdated = handler;
    }
    chatHistory(contact) {
        console.log(`VideoMessenger: chat history with ${contact}`);
        return this.signalingChannel.chatHistory(contact);
    }
    sendTextMessage(contact, message) {
        console.log(`VideoMessenger: send text message ${contact}
"${message}"`);
        return this.signalingChannel.sendTextMessage(contact, message);
    }
    set onchathistory(handler) {
        console.log("set onchathistory")
        this.signalingChannel.onchathistory = handler;
    }
    initVideoCall(contact) {
        console.log(`VideoMessenger: init video call ${contact}`);
        return this.signalingChannel.initVideoCall(contact);
    }
    set oninitvideocall(handler) {
        console.log("set oninitvideocall")
        this.signalingChannel.oninitvideocall = handler;
    }
    declineVideoCall(contact) {
        console.log(`VideoMessenger: decline video call ${contact}`);
        return this.signalingChannel.declineVideoCall(contact);
    }
    set onvideocalldeclined(handler) {
        console.log("set onvideocalldeclined")
        this.signalingChannel.onvideocalldeclined = handler;
    }
    set onvideocallaccepted(handler) {
        console.log("set onvideocallaccepted")
        this.signalingChannel.onvideocallaccepted = handler;
    }
}
```