

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования
Кафедра инженерной психологии и эргономики
Дисциплина: Компьютерные системы и сети (КСиС)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему

UDP АУДИО ЧАТ

БГУИР КР 6 - 05 - 06 12 01 029 ПЗ

Выполнил: студент группы 310901 Усов А. М.
Проверил: Болтак С.В.

Минск 2025

СОДЕРЖАНИЕ

Введение.....	3
1 Анализ предметной области.....	4
1.1 Обзор аналогов.....	4
1.2 Постановка задачи.....	7
2 Проектирование программного средства.....	8
2.1 Структура программы.....	8
2.2 Проектирование интерфейса программного средства.....	10
2.3 Проектирование функционала программного средства.....	13
3 Разработка программного средства.....	15
3.1 Архитектура программного средства.....	15
3.2 Реализация сигнального обмена с использованием <i>SignalR</i>	16
3.3 Реализация WebRTC соединения и передачи аудио.....	17
3.4 Пользовательский интерфейс.....	19
4 Тестирование программного средства.....	20
5 Руководство пользователя.....	22
5.1 Интерфейс программного средства.....	22
5.2 Порядок работы с программным средством.....	23
Заключение.....	25
Список использованных источников.....	26
Приложение А.....	27
Исходный код программы.....	27

ВВЕДЕНИЕ

Современные информационные технологии играют ключевую роль в развитии средств коммуникации. В последние годы значительно возрос интерес к приложениям, обеспечивающим обмен данными в режиме реального времени. Такие приложения позволяют людям общаться, взаимодействовать и совместно работать, независимо от их географического положения. Особенно востребованными являются голосовые чаты, позволяющие пользователям обмениваться аудиосообщениями без использования текстовых сообщений.

Для реализации голосовой связи в режиме реального времени важно обеспечить быструю и стабильную передачу аудиоданных между пользователями. Одним из наиболее подходящих для этого сетевых протоколов является *UDP (User Datagram Protocol)*. В отличие от протокола *TCP*, который обеспечивает гарантированную доставку данных за счёт механизма подтверждений и повторных передач, протокол *UDP* ориентирован на минимизацию задержек. Он отправляет пакеты без установления соединения и без подтверждений получения, что делает его оптимальным выбором для приложений, где важнее скорость передачи, чем абсолютная надёжность доставки каждого отдельного пакета.

Применение *UDP* позволяет снизить задержки и повысить плавность голосового общения, что особенно критично в случае живого разговора между пользователями. Именно поэтому протокол *UDP* широко используется в таких областях, как голосовые и видеозвонки, стриминг медиа, онлайн-игры и другие приложения, требующие быстрого обмена данными.

Разработка программного обеспечения для голосового общения с использованием *UDP* представляет собой актуальную и практическую задачу. Такие приложения находят применение в самых разных сферах — от развлечений до профессиональной деятельности. Кроме того, создание *UDP* аудио-чата требует знания сетевого программирования, основ обработки аудиосигналов и принципов работы распределённых систем.

Целью данного курсового проекта является разработка программного средства — *UDP* аудио-чата, обеспечивающего обмен аудиосообщениями между пользователями в режиме реального времени.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор аналогов

На сегодняшний день существует множество приложений для голосового общения, каждое из которых имеет свои особенности и функциональные возможности. В рамках данного обзора будут рассмотрены наиболее популярные решения, которые предоставляют пользователям возможность обмена аудиосообщениями в режиме реального времени.

В первую очередь стоит обратить внимание на приложение *Discord* [1]—одно из самых известных и распространённых средств голосового общения. Внешний вид данного приложения представлен на рисунке 1.1.

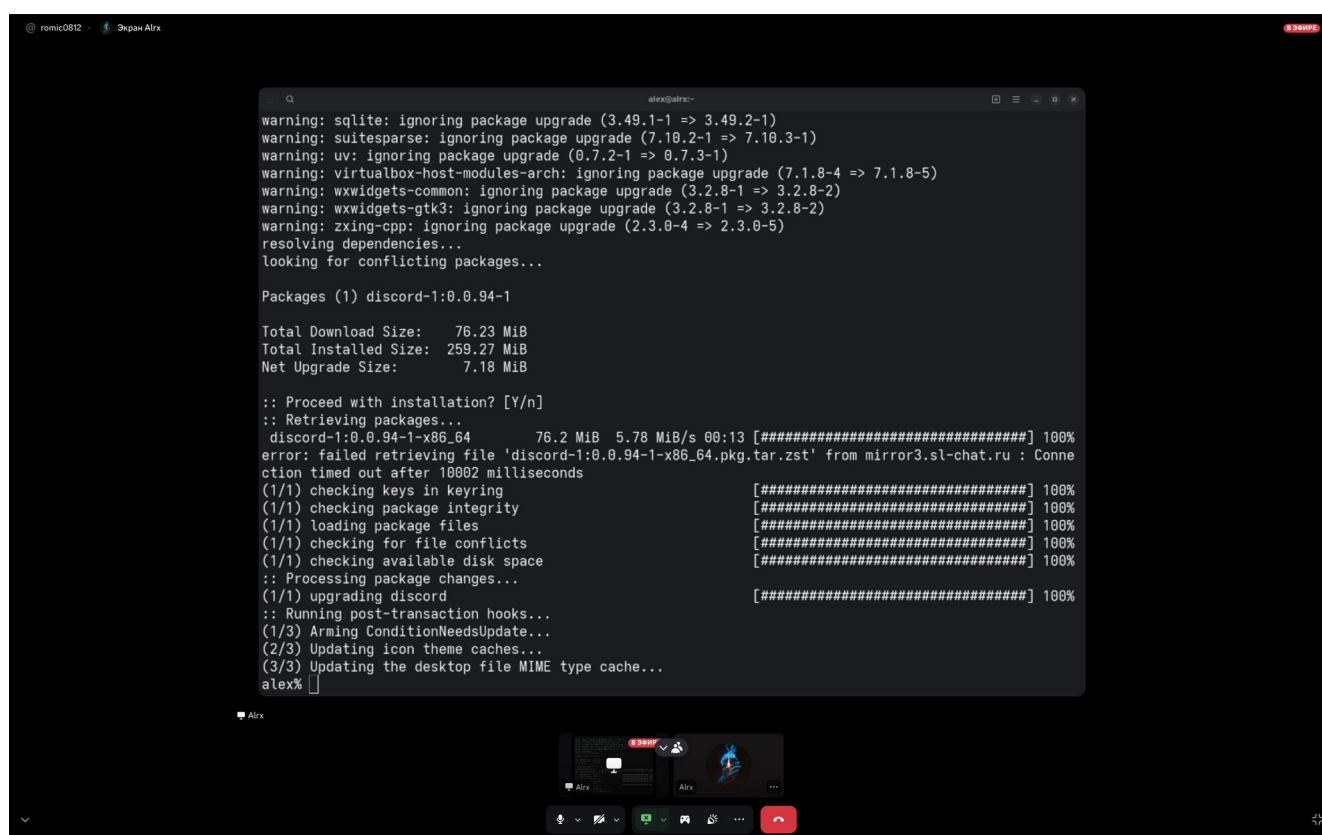


Рисунок 1.1 — приложение *Discord*

Discord изначально разрабатывался как средство голосового общения для геймеров, однако со временем его аудитория значительно расширилась. Приложение позволяет пользователям создавать собственные серверы, организовывать голосовые и текстовые каналы, обмениваться файлами и организовывать видеозвонки. Одной из ключевых особенностей *Discord* является высокая стабильность голосовой связи и возможность общения в группах с

большим количеством участников. Также присутствует функция подавления шума и автоматической регулировки громкости микрофона. Важно отметить наличие кроссплатформенности — приложение доступно на *Windows*, *macOS*, *Linux*, *Android*, *iOS* и через веб-версию.

Далее стоит рассмотреть приложение *Telegram* [2], которое также предоставляет возможность голосового общения. Внешний вид голосового звонка в *Telegram* представлен на рисунке 1.2.



Рисунок 1.2 — голосовой звонок в *Telegram*

Telegram — это популярный мессенджер, изначально предназначенный для обмена текстовыми сообщениями, однако позднее в нём появилась функция голосовых звонков. Основными преимуществами голосовых звонков в *Telegram* являются высокая скорость соединения, защищённость разговоров с применением сквозного шифрования и простота использования. Также в приложении реализованы групповые голосовые чаты, что позволяет общаться сразу с несколькими пользователями одновременно. *Telegram* активно оптимизирует

передачу аудиоданных для низкоскоростных сетей, что делает его удобным даже при ограниченной пропускной способности интернета.

Следующим аналогом является приложение *Mumble* [3]. Внешний вид клиента *Mumble* представлен на рисунке 1.3.

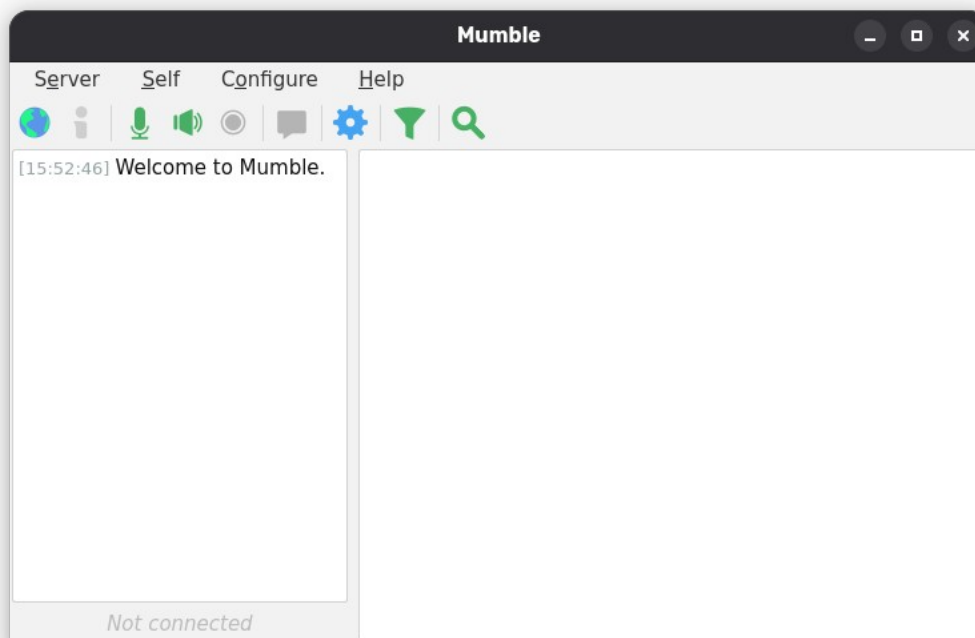


Рисунок 1.3 — приложение *Mumble*

Mumble — это бесплатное и открытое приложение для голосового общения, ориентированное на низкие задержки и высокое качество звука. Оно широко используется в игровых сообществах и командах, которым необходима надёжная голосовая связь. Одной из ключевых особенностей *Mumble* является использование кодека *Opus*, обеспечивающего отличное качество передачи речи даже при низкой скорости интернета. Кроме того, приложение поддерживает функцию позиционирования звука (*positional audio*), благодаря которой голос собеседников может звучать из разных направлений в зависимости от их положения в игровом пространстве. *Mumble* также обеспечивает высокий уровень безопасности за счёт применения шифрования.

Таким образом, рассмотренные приложения предоставляют пользователям широкий спектр возможностей для голосового общения. Однако во многих из них используются сложные архитектуры, требующие дополнительных серверов или облачных сервисов. В рамках данного курсового проекта разрабатывается более простое программное средство — *UDP* аудио-чат, обеспечивающее прямую передачу аудиоданных между пользователями с минимальными задержками и простой архитектурой.

1.2 Постановка задачи

В рамках данной курсовой работы планируется разработать *UDP* аудио-чат, обеспечивающий передачу аудиосообщений между пользователями в реальном времени. Задача заключается в создании приложения, которое будет использовать протокол *UDP* для передачи аудиоданных, обеспечивая минимальные задержки и высокое качество связи [4].

В процессе разработки будут реализованы следующие основные функции:

1. Организация голосового общения:
 - Передача аудиосообщений между двумя пользователями в реальном времени.
 - Подключение и отключение пользователей в процессе общения.
 - Определение статуса связи (например, подключение, отключение).
2. Использование протокола *UDP*:
 - Реализация передачи аудиопотока с использованием протокола *UDP* для минимизации задержек.
 - Оптимизация алгоритмов для обеспечения качества звука при возможных потерях пакетов.
3. Сигнализация через *SignalR*:
 - Реализация обмена служебной информацией, такой как *SDP* (*Session Description Protocol*) и *ICE* (*Interactive Connectivity Establishment*) кандидаты, с помощью *SignalR*.
 - Установление связи между клиентами для обмена метаданными, необходимыми для организации аудио-соединения.
4. Пользовательский интерфейс:
 - Разработка удобного и интуитивно понятного интерфейса для пользователя.
 - Простой интерфейс для начала и завершения голосового общения.
5. Технические требования:
 - Реализация клиентской части на *JavaScript* с использованием *WebRTC* для работы с аудио-потоками.
 - Использование *SignalR* на стороне сервера для обмена сигналами между пользователями.
 - Серверная часть будет реализована с использованием *C# ASP.NET* и *Razor Pages* для организации взаимодействия и управления пользователями.
6. Для разработки программного средства будут использованы следующие технологии:
 - *Frontend*: *Razor Pages*, *JavaScript*, *WebRTC*.
 - *Backend*: *C#*, *ASP.NET Core*, *SignalR*.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

2.1 Структура программы

Разрабатываемое программное средство представляет собой *UDP* аудио-чат, в котором основная функциональность реализована на стороне клиента с использованием *WebRTC*, а сервер используется только для обслуживания клиентов и обмена сигнальными сообщениями с помощью *SignalR*.

Архитектура приложения разделена на две части: клиентскую (веб-приложение) и серверную (*ASP.NET Core* сервер с *SignalR*) [5].

Клиентская часть (*JavaScript + HTML + Bootstrap*)

Клиентская часть состоит из веб-страницы с простой адаптивной вёрсткой на основе *Bootstrap* и четырёх *JavaScript*-модулей, каждый из которых выполняет отдельную функцию:

connectionHubs.js

Этот модуль обеспечивает подключение к *SignalR Hub* и обработку сигнальных сообщений. Его задачи включают:

- Подключение к серверу через *SignalR*;
- Отправку и получение сообщений о подключении и отключении пользователей;
- Передачу сигнальных данных (*SDP, ICE-кандидаты*) между клиентами для установления соединения *WebRTC*.

webRTC.js

Основной модуль, управляющий созданием и настройкой *WebRTC* соединений. В его обязанности входит:

- Создание *RTCPeerConnection*;
- Настройка обработчиков событий *WebRTC* (*onicecandidate, ontrack* и др.);
- Отправка и получение аудиопотоков между клиентами напрямую через *UDP* (в рамках *WebRTC*);
- Инициализация медиа-устройств (захват микрофона).

custom.js

Модуль, отвечающий за взаимодействие с пользовательским интерфейсом.

Он выполняет следующие функции:

- Обработка нажатий кнопок (например, подключение, отключение);
- Отображение текущего состояния соединения;
- Обновление интерфейса при изменении состояния участников чата.

utilsRTC.js

Вспомогательный модуль с утилитами для работы с *WebRTC* и медиа-данными. Этот модуль включает:

- Генерацию уникальных идентификаторов сессий;

- Функции для работы с медиа-устройствами (получение списка микрофонов и динамиков);
- Функции преобразования и проверки сигнальных данных (например, парсинг *SDP*).

Также в клиентскую часть входит *HTML*-разметка, построенная на *Bootstrap*, которая предоставляет простой и понятный пользовательский интерфейс для управления чатом (кнопки подключения, отображение участников, индикатор состояния связи).

Серверная часть (*C# / ASP.NET Core / SignalR*)

Серверная часть реализована на платформе *ASP.NET Core* и выполняет две основные задачи: раздача клиентского приложения и обеспечение сигнальной связи между клиентами.

HomeController.cs

Контроллер, отвечающий за возврат клиентской страницы пользователю.

Его функции:

- Возвращение Razor-страницы или статических файлов (*HTML, JS, CSS*);
- Выполнение базового маршрутизации (например, отображение главной страницы чата).

ConnectionHub.cs

SignalR Hub, который используется для обмена сигнальными сообщениями между клиентами. В задачи этого хаба входит:

- Приём и передача сигналов (*SDP, ICE*) между клиентами;
- Отслеживание подключения и отключения клиентов;
- Оповещение всех участников чата о появлении новых пользователей или завершении сессий.

Важно отметить, что аудиопоток между пользователями передаётся напрямую через *WebRTC (UDP)*, а сервер используется только для обмена сигнальными данными, необходимыми для установления соединения.

Взаимодействие компонентов представлено на рисунке 2.1.

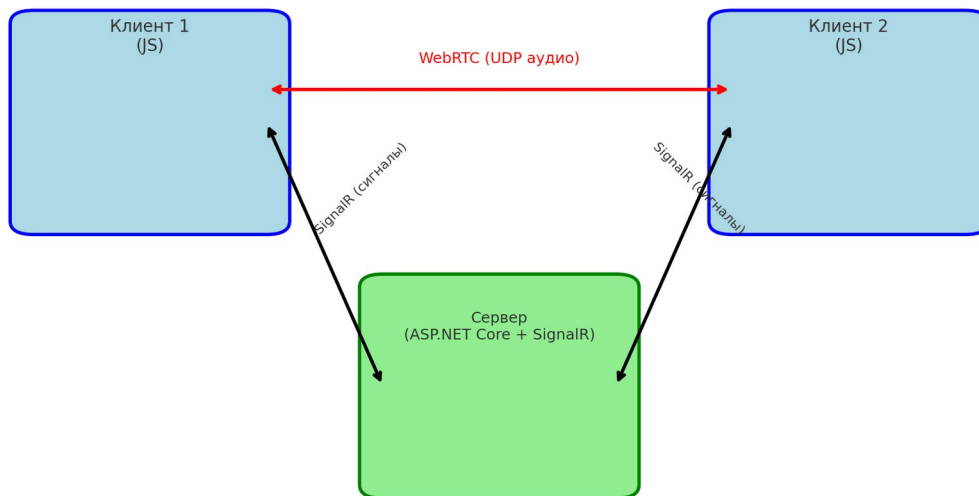


Рисунок 2.1 — Схема взаимодействия клиентов и сервера

Клиент подключается к *SignalR Hub* на сервере и обменивается сигнальной информацией (*SDP*, *ICE*-кандидаты) с другими клиентами.

После установления сигнального обмена, клиенты устанавливают прямое *peer-to-peer* соединение через *WebRTC (UDP)*, минуя сервер, и начинают передавать аудиопотоки напрямую.

Сервер остаётся посредником только на этапе установления соединения и для оповещения о событиях подключения/отключения.

2.2 Проектирование интерфейса программного средства

Интерфейс программного средства разрабатывается с учётом простоты использования и интуитивной понятности для пользователя. Внешний вид приложения будет минималистичным и функциональным, без избыточных элементов управления. Для верстки используется *CSS*-фреймворк *Bootstrap*, что позволяет обеспечить адаптивность интерфейса и современный внешний вид.

2.2.1 Главное окно

Главное окно приложения состоит из одной веб-страницы, на которой расположены основные элементы управления аудио-чатом. Пользователь при открытии сайта вводит своё имя и подключается к серверу. После подключения отображается список пользователей, находящихся в сети.

Главные элементы интерфейса:

- Поле ввода имени (*Input*);
- Кнопка подключения (*Button*);
- Список пользователей в сети (*List*);
- Кнопки управления звонком:
 - Позвонить (*Call*);
 - Принять звонок (*Answer*);
 - Завершить звонок (*Hang up*).

Макеты главного окна представлены на рисунках 2.2.

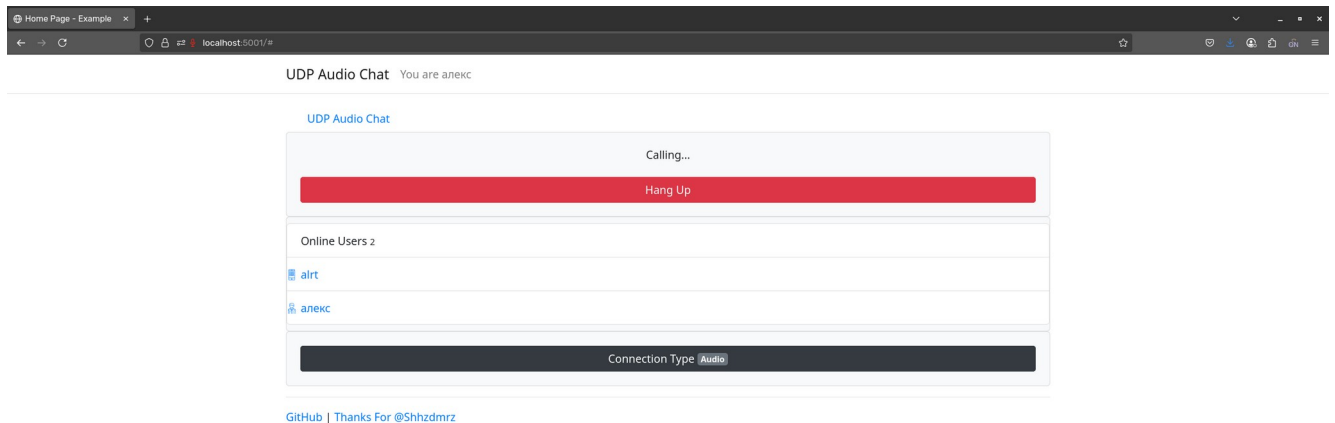


Рисунок 2.2 — Главное окно приложения

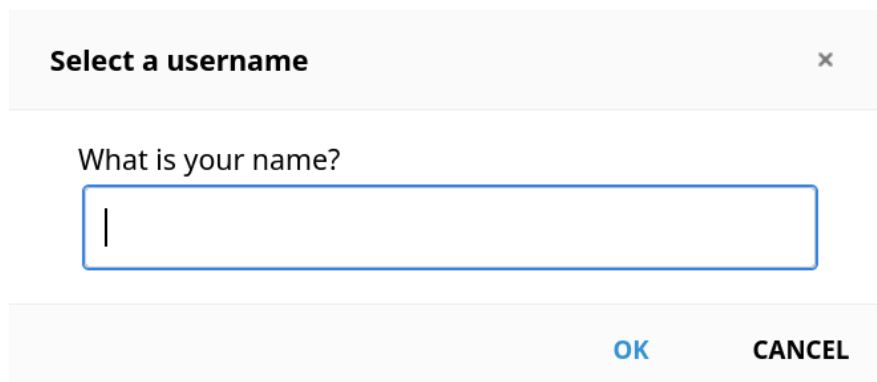
2.2.2 Элементы управления приложением

Основные элементы управления организованы следующим образом:

- Поле ввода имени — позволяет пользователю указать своё имя для идентификации в сети (представлено на рисунке 2.3);
- Кнопка подключения — отправляет имя на сервер и подключает пользователя к *SignalR Hub* (Кнопка *OK* при вводе имени);
- Список пользователей — отображает активных пользователей, доступных для звонка (представлено на рисунке 2.4);
- Позвонить выбранному пользователю (представлено на рисунке 2.6);
- Ответить на входящий вызов (представлено на рисунке 2.7);

- Завершить текущий звонок (представлено на рисунке 2.8).

При наведении на элементы управления отображаются всплывающие подсказки с кратким описанием их назначения.



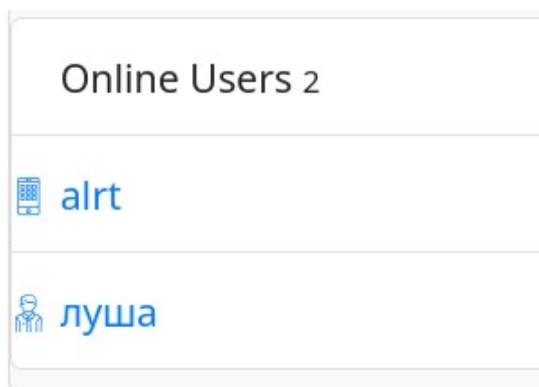
Select a username

What is your name?

|

OK CANCEL

Рисунок 2.3 — Поле ввода имени

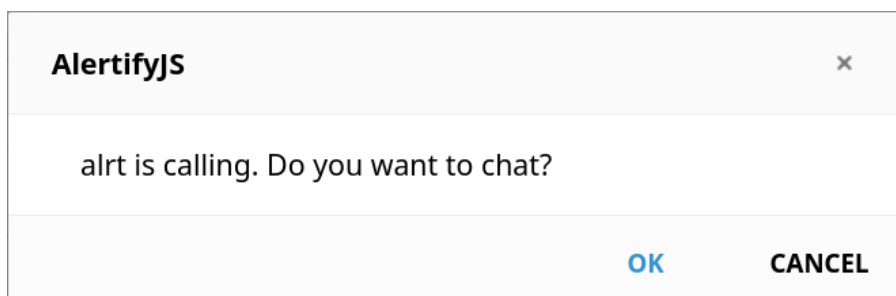


Online Users 2

alrt

луша

Рисунок 2.4 — Список активных пользователей



AlertifyJS

alrt is calling. Do you want to chat?

OK CANCEL

Рисунок 2.5 — Окно принятия звонка

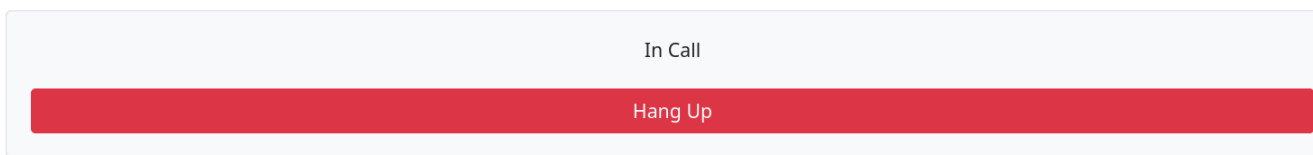


Рисунок 2.6 — Кнопка окончания вызова

2.3 Проектирование функционала программного средства

Основной функционал приложения направлен на предоставление пользователю возможности быстро и удобно связаться с другими пользователями аудио-чата.

Программное средство должно обеспечивать следующие основные функции:

- Ввод имени и подключение к серверу;
- Отображение списка пользователей в сети;
- Вызов другого пользователя;
- Принятие или отклонение входящего звонка;
- Завершение активного звонка.

2.3.1 Подключение пользователя

После ввода имени и нажатия кнопки подключения, происходит регистрация пользователя на сервере с использованием *SignalR*. Пользователь добавляется в общий список участников.

2.3.2 Совершение звонка

При выборе пользователя из списка и нажатии кнопки "Позвонить" инициируется соединение *WebRTC*. Обмен сигнальными сообщениями (*SDP* и *ICE*-кандидатами) выполняется через сервер *SignalR*. После установления соединения начинается передача аудио-потока между двумя клиентами.

2.3.3 Принятие или завершение звонка

При входящем вызове пользователю отображается уведомление с предложением принять или отклонить звонок. После нажатия соответствующей кнопки устанавливается соединение или отклоняется вызов. В любой момент пользователь может завершить активный звонок с помощью кнопки "Положить трубку".

Блок-схема основного алгоритма взаимодействия представлена на рисунке 2.7 [6].

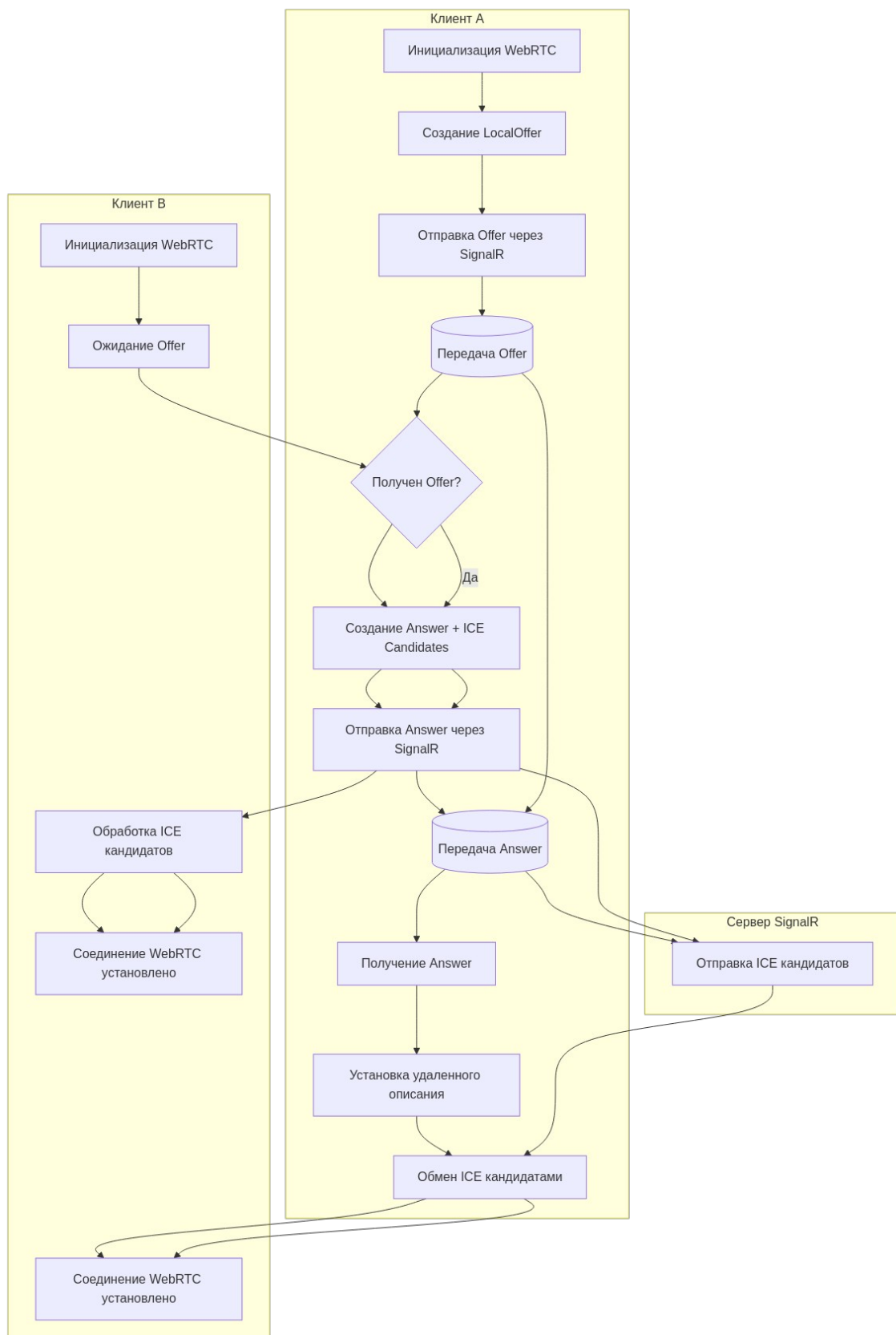


Рисунок 2.3 — Блок-схема взаимодействия пользователей

3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

В данном разделе подробно рассматривается процесс разработки программного средства UDP аудио-чата, предназначенного для организации голосового взаимодействия пользователей в режиме реального времени. Основной целью разработки является создание web-приложения, обеспечивающего установление однорангового соединения (*peer-to-peer*) между пользователями с минимальной задержкой и высоким качеством передачи звука.

Для реализации поставленных задач используются следующие современные технологии:

- *WebRTC (Web Real-Time Communication)* — для передачи аудио потока между пользователями по протоколу *UDP*;
- *SignalR* — для реализации сигнального обмена между участниками (обмен служебными данными о подключении);
- *ASP.NET Core Razor Pages* — для построения серверной части и веб-интерфейса приложения;
- *Bootstrap* — для реализации адаптивного пользовательского интерфейса.

3.1 Архитектура программного средства

Архитектура разрабатываемого программного средства основана на клиент-серверной модели с последующим установлением *P2P* соединения между клиентами. На начальном этапе пользователи подключаются к серверу, выполняющему функции сигнального обмена (*SignalR Hub*). После успешного согласования параметров соединения, между клиентами устанавливается прямой аудио-канал посредством *WebRTC*.

Архитектура состоит из следующих компонентов:

- Клиентская часть — веб-приложение, работающее в браузере пользователя. Реализует функции захвата аудио, установления соединения и взаимодействия с интерфейсом;
- Серверная часть (*ASP.NET Core*) — обеспечивает обслуживание *Razor Pages*, управление *SignalR Hub* и маршрутизацию сообщений между пользователями;
- *SignalR Hub* — реализует обмен служебными сообщениями для согласования соединения *WebRTC* (*SDP offer/answer, ICE candidates*);
- *STUN*-сервер — используется для определения внешнего *IP*-адреса клиента и облегчения прохождения *NAT* (используется публичный *STUN*-сервер *Google*).

3.2 Реализация сигнального обмена с использованием *SignalR*

Ключевым элементом серверной части разрабатываемого программного средства является *SignalR Hub* — класс *ConnectionHub*. Он обеспечивает управление пользовательскими сессиями, организацию звонков и передачу сигнальных сообщений, необходимых для установления *WebRTC* соединения [7].

Основные задачи *ConnectionHub*:

- регистрация пользователей;
- организация и завершение звонков между пользователями;
- обработка сигнальных сообщений *WebRTC* (*SDP*, *ICE*);
- поддержка списка подключенных пользователей.

Класс *ConnectionHub* реализует интерфейс *ICollectionHub*, который определяет контракты для отправки клиентам событий, таких как обновление списка пользователей или уведомления о входящих звонках.

Структура и внутренние хранилища

В *Hub* используются три коллекции (в виде списков), хранящие состояние:

- *_Users* — список подключённых пользователей;
- *_UserCalls* — список активных звонков (каждый включает участников);
- *_CallOffers* — список текущих предложений звонков (*outstanding call offers*).

Регистрация пользователя

Метод *Join(string username)* добавляет пользователя в список подключённых и уведомляет всех клиентов об изменении списка:

```
public async Task Join(string username)
{
    _Users.Add(new User
    {
        Username = username,
        ConnectionId = Context.ConnectionId
    });

    await SendUserListUpdate();
}
```

Входящий звонок

Метод *CallUser(User targetConnectionId)* инициирует вызов другому пользователю:

- проверяет, доступен ли целевой пользователь;
- создаёт "предложение звонка" (*call offer*);
- уведомляет вызываемого пользователя о входящем вызове:

```
await Clients.Client(targetConnectionId.ConnectionId).IncomingCall(callingUser);
```

Ответ на звонок

Метод *AnswerCall(bool acceptCall, User targetConnectionId)* обрабатывает принятие или отклонение вызова:

- если вызов принят, создаётся объект звонка (*UserCall*);
- оба участника уведомляются о принятии звонка;
- обновляется список пользователей (флаг *InCall*).

Завершение звонка

Метод *HangUp()* завершает активный звонок пользователя:

- уведомляет второго участника о завершении звонка;
- удаляет звонок из списка активных;
- очищает предложения звонков, исходящие от пользователя.

Передача *WebRTC* сигналов

Метод *SendSignal(string signal, string targetConnectionId)* пересылает сигнальные данные между участниками активного звонка:

```
await Clients.Client(targetConnectionId).ReceiveSignal(callingUser, signal);
```

Обновление списка пользователей

Приватный метод *SendUserListUpdate()* обновляет состояние всех пользователей и рассылает его клиентам:

```
await Clients.All.UpdateUserList(_Users);
```

3.3 Реализация WebRTC соединения и передачи аудио

Основу функциональности приложения составляет технология *WebRTC* (*Web Real-Time Communication*), которая обеспечивает передачу аудиоданных напрямую между браузерами пользователей без необходимости задействовать сервер для маршрутизации медиапоток. Взаимодействие между клиентами организуется по технологии *peer-to-peer*, а сервер используется только для начального обмена сигнальной информацией (*SDP/ICE*) [8].

Инициализация *RTCPeerConnection*

Для установления соединения создаётся объект *RTCPeerConnection*. В конфигурации задаются *STUN*-серверы, необходимые для определения внешнего *IP*-адреса пользователя и проброса *NAT*:

```
const peerConnection = new RTCPeerConnection({  
  iceServers: [{ urls: "stun:stun.l.google.com:19302" }]  
});
```

Использование общедоступного *STUN*-сервера *Google* позволяет повысить совместимость между различными сетевыми конфигурациями.

Захват аудиопотока

Для передачи аудиоданных осуществляется захват звука с микрофона пользователя с помощью *API getUserMedia*. Полученный поток добавляется в объект *RTCPeerConnection*, а также воспроизводится локально:

```
navigator.mediaDevices.getUserMedia({ audio: true })
```

```
.then(stream => {
  stream.getTracks().forEach(track => peerConnection.addTrack(track, stream));
  localAudio.srcObject = stream;
});
```

Это позволяет пользователю слышать собственный микрофонный вход (если это требуется) и подготовить поток к передаче.

Обработка ICE-кандидатов

Во время установления соединения браузеры обмениваются ICE-кандидатами — наборами сетевых адресов, по которым возможна передача данных. Каждый найденный кандидат передаётся другому пользователю с помощью *SignalR*:

```
peerConnection.onicecandidate = event => {
  if (event.candidate) {
    connection.invoke("SendSignal", user, JSON.stringify({ candidate: event.candidate }));
  }
};
```

Создание и отправка SDP-offer

Процесс установления соединения начинается с генерации *SDP-offer* — описания параметров соединения и медиапоток. Этот *offer* устанавливается как локальное описание и пересылается удалённому пользователю:

```
peerConnection.createOffer()
  .then(offer => peerConnection.setLocalDescription(offer))
  .then(() => {
    connection.invoke("SendSignal", user, JSON.stringify({ sdp: peerConnection.localDescription }));
  });
```

Обработка входящих сигналов

При получении сигнального сообщения выполняется разбор содержимого. Если это *SDP*-описание, оно устанавливается как удалённое описание. В случае получения *offer*-а — создаётся и отправляется *SDP-answer*. Если получен ICE-кандидат — он добавляется в соединение:

```
function handleSignal(user, data) {
  if (data.sdp) {
    peerConnection.setRemoteDescription(new RTCSessionDescription(data.sdp));
    if (data.sdp.type === "offer") {
      peerConnection.createAnswer()
        .then(answer => peerConnection.setLocalDescription(answer))
        .then(() => {
          connection.invoke("SendSignal", user, JSON.stringify({ sdp: peerConnection.localDescription }));
        });
    }
  } else if (data.candidate) {
    peerConnection.addIceCandidate(new RTCIceCandidate(data.candidate));
  }
}
```

Таким образом обеспечивается полный цикл установления *peer-to-peer* соединения, включая *NAT*-проброс и согласование медиапоток.

3.4 Пользовательский интерфейс

Пользовательский интерфейс приложения реализован с использованием *Bootstrap* и *Razor Pages*, что обеспечивает адаптивность и простоту использования. Интерфейс интуитивно понятен и включает основные элементы управления [9].

Основные элементы интерфейса:

- Поле ввода имени пользователя — идентификатор, используемый для отображения в списке пользователей и организации звонков;
- Кнопка подключения — инициирует подключение к *SignalR* серверу;
- Индикатор состояния подключения — отображает текущее состояние соединения с сервером и другими пользователями;
- Аудио-элемент — воспроизводит входящий аудиопоток от собеседника (скрытый элемент).

Пример интерфейса

```
<div class="row-fluid">
  <div class="span3">
    <div class="card card-body bg-light actions">
      <div id="callstatus" class="status">Idle</div>
      <button class="btn btn-danger hangup">Hang Up</button>
    </div>
    <div class="card card-body bg-light user-list">
      <ul id="usersdata" class="list-group">
        <li class="list-group-item">Online Users <small id="usersLength"></small></li>
        <li class="list-group-item user" data-cid="" data-username="">
          <a href="#">
            <div class="username"></div>
            <div class="helper" data-bind="css: $parent.getUserStatus($data)"></div>
          </a></li></ul></div>
      <div class="card card-body bg-light conn-type">
        <button type="button" class="btn btn-dark">Connection Type <span class="badge badge-secondary">Audio</span></button>
      </div>
    </div>
  </div>
</div>
```

Такой интерфейс обеспечивает минималистичный и понятный пользовательский опыт. При необходимости его можно расширить дополнительными элементами (список пользователей, кнопки вызова, индикаторы активности), сохраняя визуальное оформление за счёт возможностей *Bootstrap*.

4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

В ходе тестирования веб-приложения были выявлены некоторые ошибки и недочеты, влияющие на стабильность работы программного средства. Тестирование проводилось как в локальной сети, так и при подключении через интернет в различных браузерах (*Google Chrome*, *Mozilla Firefox*, *Microsoft Edge*).

Основные выявленные проблемы:

1. Неверное обновление списка пользователей

При отключении одного из пользователей не всегда происходило актуальное обновление списка пользователей на стороне клиента. В некоторых случаях имя отключившегося пользователя продолжало отображаться как доступное для вызова.

Причина заключалась в том, что метод *SendUserListUpdate()* на сервере вызывался не во всех случаях завершения сессии. Проблема была решена добавлением вызова обновления списка пользователей в метод *HangUp()* и *OnDisconnectedAsync()* хаба.

2. Зависание при повторном вызове после разрыва соединения

При завершении звонка и попытке немедленного начала нового вызова между теми же пользователями происходило зависание соединения — *SDP offer* не отправлялся.

Причина — объект *RTCPeerConnection* не пересоздавался после завершения звонка, и старый экземпляр содержал устаревшие *ICE*-кандидаты. Проблема решалась путём принудительного закрытия старого соединения (*peerConnection.close()*) и создания нового экземпляра перед началом нового звонка.

```
if (peerConnection) {  
    peerConnection.close();  
}  
peerConnection = new RTCPeerConnection({ iceServers: [...] });
```

3. Проблемы с доступом к микрофону

В некоторых случаях при повторном подключении к серверу браузер запрашивал доступ к микрофону повторно, либо микрофонный поток не захватывался.

Причина заключалась в том, что при отключении пользователя поток микрофона не освобождался корректно. Проблема была решена добавлением остановки всех треков после завершения звонка:

```
stream.getTracks().forEach(track => track.stop());
```

4. Некорректная обработка закрытия вкладки

При закрытии вкладки браузера во время активного вызова у второго пользователя не происходило уведомление о завершении звонка.

Для решения проблемы был добавлен обработчик события `window.onbeforeunload`, который выполнял команду разрыва соединения и отправлял уведомление о завершении вызова:

```

window.onbeforeunload = function() {
    connection.invoke("HangUp");
};

```

Основные проблемы приведены в таблице 4.1.

Таблица 4.1 — Возникшие проблемы в ходе разработки

№	Тестируемая функция	Ожидаемый результат	Фактический результат
1	Подключение нового пользователя	Пользователь успешно подключается, отображается в списке пользователей	Пользователь подключается и отображается корректно
2	Отключение пользователя	Пользователь исчезает из списка пользователей	Пользователь иногда оставался в списке (<i>ошибка устранена</i>)
3	Исходящий звонок	Оповещение целевого пользователя о входящем вызове	Оповещение отправляется корректно
4	Завершение звонка	Завершение звонка у обоих пользователей, освобождение ресурсов	Звонок завершается корректно
5	Повторный звонок после разрыва соединения	Возможность немедленного нового вызова после завершения предыдущего	Соединение зависало (<i>ошибка устранена</i>)
6	Доступ к микрофону при повторном подключении	Автоматический доступ к микрофону без повторного запроса разрешений	Поток не захватывался (<i>ошибка устранена — добавлена остановка треков</i>)
7	Закрытие вкладки браузера	Второй пользователь получает уведомление о завершении вызова	Сообщение не отправлялось (<i>ошибка устранена — добавлен onbeforeunload</i>)
8	Одновременные вызовы нескольким пользователям	Отклонение второго вызова, если пользователь уже в разговоре	Отклонение с соответствующим уведомлением
9	Отображение состояния пользователей	Отображение текущего состояния (в сети / в звонке) для всех пользователей	Состояние отображается корректно после исправления

Все выявленные ошибки были устранены на этапе тестирования. После внесения исправлений приложение демонстрирует стабильную работу, корректно устанавливает и завершает аудиосоединения между пользователями, а также надёжно синхронизирует список доступных участников.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Интерфейс программного средства

5.1.1 Главное окно

Главное окно приложения состоит из основной рабочей области и панели управления. В верхней части окна расположено поле ввода имени пользователя и кнопка подключения. Ниже находится область отображения состояния подключения и элемент воспроизведения аудиопотока. После подключения пользователь получает доступ к списку других подключенных пользователей. Список отображается в виде таблицы, напротив каждого пользователя находится кнопка для начала аудиовызова. Внешний вид главного окна приложения представлен на рисунке 5.1.



Рисунок 5.1 – Главное окно приложения

5.1.2 Окно входящего вызова

При поступлении входящего вызова на экране отображается окно уведомления с именем вызывающего пользователя и двумя кнопками: Принять вызов и Отклонить вызов. Внешний вид окна входящего вызова представлен на рисунке 5.2.

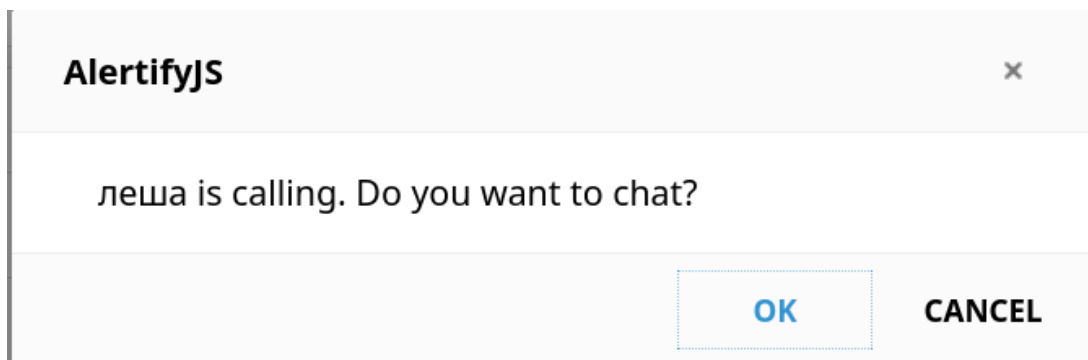


Рисунок 5.2 – Окно входящего вызова

5.1.3 Окно активного вызова

Во время активного вызова пользователь видит окно с информацией о текущем соединении и кнопкой для завершения вызова. Внешний вид окна активного вызова представлен на рисунке 5.3.

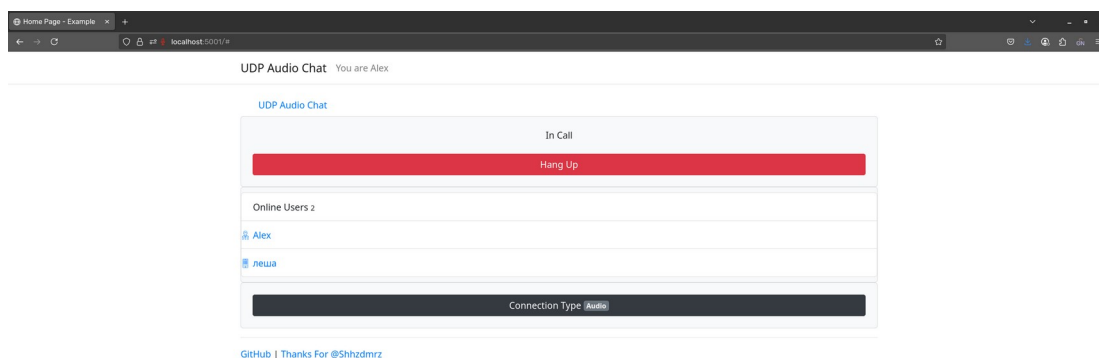


Рисунок 5.3 – Окно активного вызова

5.2 Порядок работы с программным средством

В данном разделе представлена пошаговая инструкция по работе с приложением для аудиосвязи на базе *WebRTC*.

Шаг 1. Ввод имени пользователя

После запуска приложения введите своё имя в текстовое поле, расположенное в верхней части главного окна. Это имя будет отображаться другим пользователям для идентификации при звонках.

Шаг 2. Подключение к серверу

Подключение выполняется автоматически. После успешного подключения в окне состояния отобразится сообщение «*Connection Complete*», а также появится список других пользователей, находящихся в сети.

Шаг 3. Просмотр списка пользователей

В рабочей области приложения отобразится список подключённых пользователей. Для инициации аудио вызова с определенным пользователем необходимо нажать на его имя, которое он указал при подключении.

Шаг 4. Принятие или отклонение входящего вызова

Если другой пользователь звонит вам, на экране появится окно с двумя кнопками:

- «OK» — для начала разговора;
- «Cancel» — для отказа от соединения.

Шаг 5. Проведение разговора

После принятия вызова установится аудиосоединение. Вы сможете вести голосовой разговор в реальном времени. Во время разговора на экране будет отображаться информация о текущем соединении.

Шаг 6. Завершение разговора

Чтобы завершить разговор, нажмите кнопку «*Hang Up*». Соединение будет разорвано, а вы вернётесь к списку пользователей.

Шаг 7. Отключение от сервера и выход из приложения

Для завершения работы с приложением достаточно закрыть вкладку браузера или окно приложения. При этом произойдёт автоматическое отключение от сервера.

ЗАКЛЮЧЕНИЕ

В рамках курсовой работы был разработан *UDP* аудио-чат для обмена аудиосообщениями между пользователями в реальном времени. Целью проекта было создание приложения, использующего протокол *UDP* для минимизации задержек и обеспечения высокого качества связи. В ходе разработки были решены задачи, связанные с организацией голосового общения, передачей аудиопотока через *UDP* и созданием удобного интерфейса для пользователей.

Основные достижения проекта:

- Голосовое общение: Реализована передача аудиосообщений между пользователями с возможностью подключения и отключения участников, а также управление статусами связи, что позволило обеспечить стабильную и непрерывную коммуникацию.
- Использование *UDP*: Протокол *UDP* был выбран для минимизации задержек в передаче данных, что особенно важно для приложений реального времени. Были разработаны алгоритмы, направленные на оптимизацию качества звука при возможных потерях пакетов, что обеспечило более стабильную связь.
- Сигнализация через *SignalR*: Обмен служебной информацией с помощью *SignalR* обеспечил быстрое установление связи между пользователями, а также передачу метаданных, таких как *SDP* и *ICE* кандидаты, для организации голосового соединения.
- Интерфейс: Разработан простой и интуитивно понятный пользовательский интерфейс, который позволяет легко начинать и завершать разговоры. Удобство интерфейса является важным аспектом для пользователей, не обладающих техническими знаниями.

Технически проект был реализован с использованием *JavaScript* и *WebRTC* для клиентской части, а также *C#* и *SignalR* для серверной части. Это позволило эффективно организовать работу с аудиопотоками и обеспечить надежное взаимодействие между клиентами и сервером.

В дальнейшем проект может быть улучшен с добавлением функционала видеосвязи, улучшением качества звука с помощью алгоритмов сжатия и повышения устойчивости к потере пакетов, а также обеспечением дополнительной безопасности данных. Возможность расширения функционала и интеграции с другими сервисами так же поможет улучшить приложение.

Разработка этого программного обеспечения имеет практическую ценность, так как голосовые чаты активно используются в разных областях, таких как профессиональная коммуникация, онлайн-образование и развлекательные приложения. Создание качественного и эффективного инструмента для голосового общения открывает новые возможности для улучшения связи в реальном времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] *Discord* [Электронный ресурс]. – Режим доступа: <https://discord.com/>.
- [2] *Telegram* [Электронный ресурс]. – Режим доступа: <https://t.me/>.
- [3] *Mumble* [Электронный ресурс]. – Режим доступа: <https://mumble.com/>.
- [4] Как работает JS: WebRTC и механизмы P2P-коммуникаций [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/companies/ruvds/articles/416821/>
- [5] Клиент-серверная архитектура в картинках [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/495698/>.
- [6] *Mermaid Diagram* [Электронный ресурс]. – Режим доступа: <https://mermaid.live>.
- [7] *SignalR with Microsoft* [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/aspnet/signalr/>
- [8] *WebRTC* [Электронный ресурс]. – Режим доступа: <https://webrtc.org/>
- [9] Гладкое бритье: Razor Pages для разработчиков веб-форм [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/companies/otus/articles/839220/>
- 10 Стандарт предприятия [Электронный ресурс]. – Режим доступа: СТП_2024.pdf.

ПРИЛОЖЕНИЕ А

Исходный код программы

```
HomeController.cs:
using Microsoft.AspNetCore.Mvc;

namespace SignalRCoreWebRTC.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

```
ConnectionModel.cs:
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SignalRCoreWebRTC.Models
{
    public class CallOffer
    {
        public User Caller { get; set; }
        public User Callee { get; set; }
    }

    public class User
    {
        public string Username { get; set; }
        public string ConnectionId { get; set; }
        public bool InCall { get; set; }
    }

    public class UserCall
    {
        public List<User> Users { get; set; }
    }
}
```

```

    }
}

    ConnectionHub.cs:
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR;
using SignalRCoreWebRTC.Models;

namespace SignalRCoreWebRTC.Hubs
{
    public class ConnectionHub : Hub<IConnectionHub>
    {
        private readonly List<User> _Users;
        private readonly List<UserCall> _UserCalls;
        private readonly List<CallOffer> _CallOffers;

        public ConnectionHub(List<User> users, List<UserCall> userCalls, List<CallOffer> callOffers)
        {
            _Users = users;
            _UserCalls = userCalls;
            _CallOffers = callOffers;
        }

        public async Task Join(string username)
        {
            // Add the new user
            _Users.Add(new User
            {
                Username = username,
                ConnectionId = Context.ConnectionId
            });

            // Send down the new list to all clients
            await SendUserListUpdate();
        }

        public override async Task OnDisconnectedAsync(Exception exception)

```

```

    {
        // Hang up any calls the user is in
        await HangUp(); // Gets the user from "Context" which is available in the whole
hub
    }

    // Remove the user
    _Users.RemoveAll(u => u.ConnectionId == Context.ConnectionId);

    // Send down the new user list to all clients
    await SendUserListUpdate();

    await base.OnDisconnectedAsync(exception);
}

public async Task CallUser(User targetConnectionId)
{
    var callingUser = _Users.SingleOrDefault(u => u.ConnectionId == Context.-
ConnectionId);
    var targetUser = _Users.SingleOrDefault(u => u.ConnectionId == targetConnec-
tionId.ConnectionId);

    // Make sure the person we are trying to call is still here
    if (targetUser == null)
    {
        // If not, let the caller know
        await Clients.Caller.CallDeclined(targetConnectionId, "The user you called
has left.");
        return;
    }

    // And that they aren't already in a call
    if (GetUserCall(targetUser.ConnectionId) != null)
    {
        await Clients.Caller.CallDeclined(targetConnectionId, string.Format("{0} is
already in a call.", targetUser.Username));
        return;
    }

    // They are here, so tell them someone wants to talk

```

```

        await Clients.Client(targetConnectionId.ConnectionId).IncomingCall(callingUser);

        // Create an offer
        _CallOffers.Add(new CallOffer
        {
            Caller = callingUser,
            Callee = targetUser
        });
    }

    public async Task AnswerCall(bool acceptCall, User targetConnectionId)
    {
        var callingUser = _Users.SingleOrDefault(u => u.ConnectionId == Context.ConnectionId);
        var targetUser = _Users.SingleOrDefault(u => u.ConnectionId == targetConnectionId.ConnectionId);

        // This can only happen if the server-side came down and clients were cleared,
        while the user
        // still held their browser session.
        if (callingUser == null)
        {
            return;
        }

        // Make sure the original caller has not left the page yet
        if (targetUser == null)
        {
            await Clients.Caller.CallEnded(targetConnectionId, "The other user in your call has left.");
            return;
        }

        // Send a decline message if the callee said no
        if (acceptCall == false)
        {
            await Clients.Client(targetConnectionId.ConnectionId).CallDeclined(callingUser, string.Format("{0} did not accept your call.", callingUser.Username));
            return;
        }
    }

```



```

    }

    // Make sure there is still an active offer. If there isn't, then the other use hung
    up before the Callee answered.
    var offerCount = _CallOffers.RemoveAll(c => c.Callee.ConnectionId == call-
    ingUser.ConnectionId
                                && c.Caller.ConnectionId == targetUser.ConnectionId);
    if (offerCount < 1)
    {
        await Clients.Caller.CallEnded(targetConnectionId, string.Format("{0} has
    already hung up.", targetUser.Username));
        return;
    }

    // And finally... make sure the user hasn't accepted another call already
    if (GetUserCall(targetUser.ConnectionId) != null)
    {
        // And that they aren't already in a call
        await Clients.Caller.CallDeclined(targetConnectionId, string.Format("{0}
    chose to accept someone elses call instead of yours :(", targetUser.Username));
        return;
    }

    // Remove all the other offers for the call initiator, in case they have multiple
    calls out
    _CallOffers.RemoveAll(c => c.Caller.ConnectionId == targetUser.Connec-
    tionId);

    // Create a new call to match these folks up
    _UserCalls.Add(new UserCall
    {
        Users = new List<User> { callingUser, targetUser }
    });

    // Tell the original caller that the call was accepted
    await Clients.Client(targetConnectionId.ConnectionId).CallAccepted(call-
    ingUser);

    // Update the user list, since thes two are now in a call
    await SendUserListUpdate();

```

```

    }

    public async Task HangUp()
    {
        var callingUser = _Users.SingleOrDefault(u => u.ConnectionId == Context.
ConnectionId);

        if (callingUser == null)
        {
            return;
        }

        var currentCall = GetUserCall(callingUser.ConnectionId);

        // Send a hang up message to each user in the call, if there is one
        if (currentCall != null)
        {
            foreach (var user in currentCall.Users.Where(u => u.ConnectionId != call-
ingUser.ConnectionId))
            {
                await Clients.Client(user.ConnectionId).CallEnded(callingUser, string.For-
mat("{0} has hung up.", callingUser.Username));
            }

            // Remove the call from the list if there is only one (or none) person left. This
should
            // always trigger now, but will be useful when we implement conferencing.
            currentCall.Users.RemoveAll(u => u.ConnectionId == callingUser.Connec-
tionId);
            if (currentCall.Users.Count < 2)
            {
                _UserCalls.Remove(currentCall);
            }
        }

        // Remove all offers initiating from the caller
        _CallOffers.RemoveAll(c => c.Caller.ConnectionId == callingUser.Connec-
tionId);

        await SendUserListUpdate();
    }

```

```

    }

    // WebRTC Signal Handler
    public async Task SendSignal(string signal, string targetConnectionId)
    {
        var callingUser = _Users.SingleOrDefault(u => u.ConnectionId == Context.-
ConnectionId);
        var targetUser = _Users.SingleOrDefault(u => u.ConnectionId == targetConnec-
tionId);

        // Make sure both users are valid
        if (callingUser == null || targetUser == null)
        {
            return;
        }

        // Make sure that the person sending the signal is in a call
        var userCall = GetUserCall(callingUser.ConnectionId);

        // ...and that the target is the one they are in a call with
        if (userCall != null && userCall.Users.Exists(u => u.ConnectionId == targetUs-
er.ConnectionId))
        {
            // These folks are in a call together, let's let em talk WebRTC
            await Clients.Client(targetConnectionId).ReceiveSignal(callingUser, signal);
        }
    }

    #region Private Helpers

    private async Task SendUserListUpdate()
    {
        _Users.ForEach(u => u.InCall = (GetUserCall(u.ConnectionId) != null));
        await Clients.All.UpdateUserList(_Users);
    }

    private UserCall GetUserCall(string connectionId)
    {
        var matchingCall =

```

```

        _UserCalls.SingleOrDefault(uc => uc.Users.SingleOrDefault(u => u.ConnectionId == connectionId) != null);
        return matchingCall;
    }

```

```

    #endregion
}

```

```

public interface IConnectionHub
{

```

```

    Task UpdateUserList(List<User> userList);
    Task CallAccepted(User acceptingUser);
    Task CallDeclined(User decliningUser, string reason);
    Task IncomingCall(User callingUser);
    Task ReceiveSignal(User signalingUser, string signal);
    Task CallEnded(User signalingUser, string signal);
}

```

```

}

```

Startup.cs:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using SignalRCoreWebRTC.Hubs;
using SignalRCoreWebRTC.Models;

```

```

namespace SignalRCoreWebRTC
{

```

```

    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
    }
}

```

```
public IConfiguration Configuration { get; }
```

// This method gets called by the runtime. Use this method to add services to the container.

// For more information on how to configure your application, visit <https://go.microsoft.com/fwlink/?LinkID=398940>

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddControllersWithViews();
```

//Cross-origin policy to accept request from localhost:8084.

```
services.AddCors(options =>
{
    options.AddPolicy("CorsPolicy",
        x => x.AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader());
});
```

```
services.AddSignalR();
```

```
services.AddSingleton<List<User>>();
services.AddSingleton<List<UserCall>>();
services.AddSingleton<List<CallOffer>>();
}
```

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
```

// The default HSTS value is 30 days. You may want to change this for production scenarios, see <https://aka.ms/aspnetcore-hsts>.

```

        app.UseHsts();
    }

    app.UseStaticFiles();
    app.UseFileServer();
    app.UseCors("CorsPolicy");

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        //endpoints.MapRazorPages();
        endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapHub<ConnectionHub>("/ConnectionHub", options =>
        {
            options.Transports = Microsoft.AspNetCore.Http.Connections.HttpTransportType.WebSockets;
        });
    });
}
}
}

```

Program.cs:

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace SignalRCoreWebRTC
{
    public class Program
    {
        public static void Main(string[] args)
        {

```

```

        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder
                    .UseContentRoot(Directory.GetCurrentDirectory())
                    .UseStartup<Startup>();
            });
    }
}

__Layout.cshtml:
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <title>@ViewBag.Title - Example</title>

    <link href="https://fonts.googleapis.com/css?family=Open+Sans:400,700"
rel="stylesheet" type="text/css">
    <link href="https://fonts.googleapis.com/css?family=Droid+Sans:400,700"
rel="stylesheet" type="text/css">
    <link href="~/lib/icomoon/style.css" rel="stylesheet" />
    <link href="~/lib/bootstrap/css/bootstrap.css" rel="stylesheet" />
    <link href="~/lib/alertifyjs/build/css/alertify.min.css" rel="stylesheet" />
    <link href="~/lib/alertifyjs/build/css/themes/default.min.css" rel="stylesheet" />
    <link href="~/Content/css/Custom.css" rel="stylesheet" />
</head>
<body data-mode="idle">
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">UDP Audio Chat</a>

```

```
@*<span id="loadingSpinner" class="glyphicon glyphicon-repeat fast-right-
spinner"></span>*@
```

```
<div class="collapse navbar-collapse">
  <span class="navbar-text text-right">
    You are <span id="upperUsername"></span>
  </span>
</div>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target=".navbar-collapse" aria-controls="navbarSupportedContent"
  aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
</div>
</nav>
</header>
<div class="container">
  <main role="main" class="pb-3">
    @RenderBody()
  </main>
</div>

<script src="~/lib/jquery/jquery.min.js"></script>
<script src="~/lib/bootstrap/js/bootstrap.min.js"></script>
<script src="~/lib/signalr/dist/browser/signalr.min.js"></script>
<script src="~/lib/alertifyjs/build/alertify.min.js"></script>
<script src="~/Content/js/UtilsRTC.js"></script>
<script src="~/Content/js/constWebRTC.js"></script>
<script src="~/lib/adapters/adapters.min.js"></script>
<script src="~/Content/js/connectionHub.js"></script>
<script src="~/Content/js/Custom.js"></script>
</body>
</html>
```

Index.cshtml:

```
@{
  ViewBag.Title = "Home Page";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
```

```
<!-- Invalid browser alert, and reminder to enable media things -->
```



```

<div class="container-fluid">
  <div class="row-fluid browser-warning">
    <div class="span12">
      <div class="alert alert-error">
        <h4>Your browser does not appear to support WebRTC.</h4>
        Try either the <a href="https://nightly.mozilla.org/">latest Firefox nightly
build</a>, or
                                                                    <a
href="https://www.google.com/intl/en/chrome/browser/beta.html">Google      Chrome
Beta</a> to join the fun.
      </div>
    </div>
  </div>
</div>
<!-- Top Bar -->
<nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container-fluid">
      <a class="brand pull-left" href="#">UDP Audio Chat</a>
      <span id="loadingSpinner" class="glyphicon glyphicon-repeat fast-right-spin-
ner"></span>
      <div class="nav-collapse collapse">
        <p class="navbar-text pull-right">
          You are <span id="upperUsername"></span>
        </p>
      </div>
    </div>
  </div>
</nav>
<div class="row-fluid">
  <!-- Side Bar -->
  <div class="span3">
    <!-- In Call Actions -->
    <div class="card card-body bg-light actions">
      <div id="callstatus" class="status">Idle</div>
      <button class="btn btn-danger hangup">Hang Up</button>
    </div>
    <!-- User List -->
    <div class="card card-body bg-light user-list">
      <ul id="usersdata" class="list-group">

```

```

        <li class="list-group-item">Online Users <small id="usersLength"></
small></li>
        <!-- ko foreach: Users -->
        <li class="list-group-item user" data-cid="" data-username="">
            <a href="#">
                <!-- only using an a here for bootstrap styling -->
                <div class="username"></div>
                <div class="helper" data-bind="css: $parent.getUserStatus($data)"></
div>
                    </a>
                </li>
            <!-- /ko -->
        </ul>
    </div>
    <div class="card card-body bg-light conn-type">
        <button type="button" class="btn btn-dark">Connection Type <span
class="badge badge-secondary">Audio</span></button>
    </div>
</div>
<!-- Footer -->
<hr>
<footer>
    <p class="pull-left">
        <a href="https://github.com/alrx31/audio-chat" target="_blank">GitHub</a> |
        <a href="https://github.com/Shhzdmrz/SignalRCoreWebRTC"
target="_blank">Thanks For @@Shhzdmrz</a>
    </p>
</footer>

    Custom.css:
/* loading indicator */
/* yep, I know it is off-center, but I dont want to figure out the right origin */
@-moz-keyframes rotation {
    from {
        -moz-transform: rotate(0deg);
        -moz-transform-origin: 85% 90%;
    }

    to {
        -moz-transform: rotate(359deg);

```

```

        -moz-transform-origin: 85% 90%;
    }
}

@-webkit-keyframes rotation {
    from {
        -webkit-transform: rotate(0deg);
        -webkit-transform-origin: 85% 90%;
    }

    to {
        -webkit-transform: rotate(359deg);
        -webkit-transform-origin: 85% 90%;
    }
}

[class*=" icon-"].loading-indicator {
    float: left;
    display: none;
    font-size: 24px;
    margin: 7px !important;
    color: #EC173A;
}

[class*=" icon-"].loading-indicator.on {
    display: block;
    -webkit-animation: rotation 1.5s infinite linear;
    -moz-animation: rotation 1.5s infinite linear;
}

/* browser alert */
.browser-warning {
    display: none;
}

/* user List*/
.user-list {
    padding: 9px 0;
}

```

```
.user .icon-phone-4 {
  display: none;
}

.user:hover .icon-phone-4 {
  display: inline-block;
}

.user a {
  position: relative;
  overflow: hidden;
}

.user .username {
  white-space: nowrap;
  text-overflow: ellipsis;
  overflow: hidden;
  padding-right: 16px;
}

.user .helper {
  position: absolute;
  right: 5px;
  top: 5px;
}

/* in call actions */
.actions {
  display: none;
}

[data-mode='incall'] .actions,
[data-mode='calling'] .actions {
  display: block;
}

.actions .hangup {
  width: 100%;
}
```

```

.actions .status {
  text-align: center;
  margin-bottom: 20px;
}

/* video windows */
.video {
  height: 100%;
  width: 100%;
  border: 2px solid black;
}

.cool-background {
  background: linear-gradient(135deg, #ECEDDC 25%, transparent 25%) -50px 0, linear-gradient(225deg, #ECEDDC 25%, transparent 25%) -50px 0, linear-gradient(315deg, #ECEDDC 25%, transparent 25%), linear-gradient(45deg, #ECEDDC 25%, transparent 25%);
  background-size: 100px 100px;
  background-color: #EC173A;
}

/* alertify styles */
.alertify-cover {
  background: rgba(0,0,0,.8);
}

input[type='text'] {
  padding: 10px !important;
  height: auto !important;
}

    ConnectionHub.js:
const isDebugging = true;
var hubUrl = document.location.pathname + 'ConnectionHub';
var wsconn = new signalR.HubConnectionBuilder()
  .withUrl(hubUrl, signalR.HttpTransportType.WebSockets)
  .configureLogging(signalR.LogLevel.None).build();

var peerConnectionConfig = { "iceServers": [ { "url": "stun:stun.l.google.com:19302" } ]
};
//  "iceServers": [

```

```
//      { "urls": "stun:stun.l.google.com:19302?transport=udp" },
//      { "urls": "stun:numb.viagenie.ca:3478?transport=udp" },
//      { "urls": "turn:numb.viagenie.ca:3478?transport=udp", "username":
"shahzad@fms-tech.com", "credential": "P@ssw0rdfms" },
//      { "urls": "turn:turn-testdrive.cloudapp.net:3478?transport=udp", "username":
"redmond", "credential": "redmond123" }
//  ]
//};
```

```
$(document).ready(function () {
    initializeSignalR();

    // Add click handler to users in the "Users" pane
    $(document).on('click', '.user', function () {
        console.log('calling user... ');
        // Find the target user's SignalR client id
        var targetConnectionId = $(this).attr('data-cid');

        // Make sure we are in a state where we can make a call
        if ($('#body').attr("data-mode") !== "idle") {
            alertify.error('Sorry, you are already in a call. Conferencing is not yet imple-
mented.');
```

mented.');

```
            return;
        }

        // Then make sure we aren't calling ourselves.
        if (targetConnectionId !== myConnectionId) {
            // Initiate a call
            wsconn.invoke('callUser', { "connectionId": targetConnectionId });

            // UI in calling mode
            $('#body').attr('data-mode', 'calling');
            $('#callstatus').text('Calling...');
        } else {
            alertify.error("Ah, nope. Can't call yourself.");
        }
    });

    // Add handler for the hangup button
    $('#hangup').click(function () {
```

```

    console.log('hangup....');
    // Only allow hangup if we are not idle
    //localStream.getTracks().forEach(track => track.stop());
    if ($('#body').attr("data-mode") !== "idle") {
        wsconn.invoke('hangUp');
        closeAllConnections();
        $('#body').attr('data-mode', 'idle');
        $('#callstatus').text('Idle');
    }
});
});

var webrtcConstraints = { audio: true, video: false };
var streamInfo = { applicationName: WOWZA_APPLICATION_NAME, streamName:
WOWZA_STREAM_NAME, sessionId: WOWZA_SESSION_ID_EMPTY };

var WOWZA_STREAM_NAME = null, connections = {}, localStream = null;

attachMediaStream = (e) => {
    //console.log(e);
    console.log("OnPage: called attachMediaStream");
    var partnerAudio = document.querySelector('.audio.partner');
    if (partnerAudio.srcObject !== e.stream) {
        partnerAudio.srcObject = e.stream;
        console.log("OnPage: Attached remote stream");
    }
};

const receivedCandidateSignal = (connection, partnerClientId, candidate) => {
    //console.log('candidate', candidate);
    //if (candidate) {
        console.log('WebRTC: adding full candidate');
        connection.addIceCandidate(new RTCIceCandidate(candidate), () =>
console.log("WebRTC: added candidate successfully"), () => console.log("WebRTC:
cannot add candidate"));
    //} else {
        // console.log('WebRTC: adding null candidate');
        // connection.addIceCandidate(null, () => console.log("WebRTC: added null candi-
date successfully"), () => console.log("WebRTC: cannot add null candidate"));
    //}
};

```

```

}

// Process a newly received SDP signal
const receivedSdpSignal = (connection, partnerClientId, sdp) => {
  console.log('connection: ', connection);
  console.log('sdp', sdp);
  console.log('WebRTC: called receivedSdpSignal');
  console.log('WebRTC: processing sdp signal');
  connection.setRemoteDescription(new RTCSessionDescription(sdp), () => {
    console.log('WebRTC: set Remote Description');
    if (connection.remoteDescription.type === "offer") {
      console.log('WebRTC: remote Description type offer');
      connection.addStream(localStream);
      console.log('WebRTC: added stream');
      connection.createAnswer().then((desc) => {
        console.log('WebRTC: create Answer...');
        connection.setLocalDescription(desc, () => {
          console.log('WebRTC: set Local Description...');
          console.log('connection.localDescription: ', connection.localDescription);
          //setTimeout(() => {
            sendHubSignal(JSON.stringify({ "sdp": connection.localDescription }),
partnerClientId);
          //}, 1000);
        }, errorHandler);
      }, errorHandler);
    } else if (connection.remoteDescription.type === "answer") {
      console.log('WebRTC: remote Description type answer');
    }
  }, errorHandler);
}

// Hand off a new signal from the signaler to the connection
const newSignal = (partnerClientId, data) => {
  console.log('WebRTC: called newSignal');
  //console.log('connections: ', connections);

  var signal = JSON.parse(data);
  var connection = getConnection(partnerClientId);
  //console.log("signal: ", signal);
  //console.log("signal: ", signal.sdp || signal.candidate);

```



```

//console.log("partnerClientId: ", partnerClientId);
console.log("connection: ", connection);

// Route signal based on type
if (signal.sdp) {
  console.log('WebRTC: sdp signal');
  receivedSdpSignal(connection, partnerClientId, signal.sdp);
} else if (signal.candidate) {
  console.log('WebRTC: candidate signal');
  receivedCandidateSignal(connection, partnerClientId, signal.candidate);
} else {
  console.log('WebRTC: adding null candidate');
  connection.addIceCandidate(null, () => console.log("WebRTC: added null candidate successfully"), () => console.log("WebRTC: cannot add null candidate"));
}
}

const onReadyForStream = (connection) => {
  console.log("WebRTC: called onReadyForStream");
  // The connection manager needs our stream
  //console.log("onReadyForStream connection: ", connection);
  connection.addStream(localStream);
  console.log("WebRTC: added stream");
}

const onStreamRemoved = (connection, streamId) => {
  console.log("WebRTC: onStreamRemoved -> Removing stream: ");
  //console.log("Stream: ", streamId);
  //console.log("connection: ", connection);
}

// Close the connection between myself and the given partner
const closeConnection = (partnerClientId) => {
  console.log("WebRTC: called closeConnection ");
  var connection = connections[partnerClientId];

  if (connection) {
    // Let the user know which streams are leaving
    // todo: foreach connection.remoteStreams -> onStreamRemoved(stream.id)
    onStreamRemoved(null, null);
  }
}

```

```

    // Close the connection
    connection.close();
    delete connections[partnerClientId]; // Remove the property
  }
}
// Close all of our connections
const closeAllConnections = () => {
  console.log("WebRTC: call closeAllConnections ");
  for (var connectionId in connections) {
    closeConnection(connectionId);
  }
}

const getConnection = (partnerClientId) => {
  console.log("WebRTC: called getConnection");
  if (connections[partnerClientId]) {
    console.log("WebRTC: connections partner client exist");
    return connections[partnerClientId];
  }
  else {
    console.log("WebRTC: initialize new connection");
    return initializeConnection(partnerClientId)
  }
}

const initiateOffer = (partnerClientId, stream) => {
  console.log('WebRTC: called initiateoffer: ');
  var connection = getConnection(partnerClientId); // // get a connection for the given
  partner
  //console.log('initiate Offer stream: ', stream);
  //console.log("offer connection: ", connection);
  connection.addStream(stream); // add our audio/video stream
  console.log("WebRTC: Added local stream");

  connection.createOffer().then(offer => {
    console.log('WebRTC: created Offer: ');
    console.log('WebRTC: Description after offer: ', offer);
    connection.setLocalDescription(offer).then(() => {
      console.log('WebRTC: set Local Description: ');
      console.log('connection before sending offer ', connection);
    });
  });
}

```

```

        setTimeout(() => {
            sendHubSignal(JSON.stringify({ "sdp": connection.localDescription }), part-
nerClientId);
            }, 1000);
        }).catch(err => console.error('WebRTC: Error while setting local description',
err));
    }).catch(err => console.error('WebRTC: Error while creating offer', err));

    //connection.createOffer((desc) => { // send an offer for a connection
    //  console.log('WebRTC: created Offer: ');
    //  console.log('WebRTC: Description after offer: ', JSON.stringify(desc));
    //  connection.setLocalDescription(desc, () => {
    //      console.log('WebRTC: Description after setting locally: ',
JSON.stringify(desc));
    //      console.log('WebRTC: set Local Description: ');
    //      console.log('connection.localDescription: ', JSON.stringify(connection.lo-
calDescription));
    //      sendHubSignal(JSON.stringify({ "sdp": connection.localDescription }), part-
nerClientId);
    //  });
    //}, errorHandler);
}

const callbackUserMediaSuccess = (stream) => {
    console.log("WebRTC: got media stream");
    localStream = stream;

    const audioTracks = localStream.getAudioTracks();
    if (audioTracks.length > 0) {
        console.log(`Using Audio device: ${audioTracks[0].label}`);
    }
};

const initializeUserMedia = () => {
    console.log('WebRTC: InitializeUserMedia: ');
    navigator.getUserMedia(webrtcConstraints, callbackUserMediaSuccess, errorHan-
dler);
};
// stream removed
const callbackRemoveStream = (connection, evt) => {

```

```

    console.log('WebRTC: removing remote stream from partner window');
    // Clear out the partner window
    var otherAudio = document.querySelector('.audio.partner');
    otherAudio.src = "";
}

const callbackAddStream = (connection, evt) => {
    console.log('WebRTC: called callbackAddStream');

    // Bind the remote stream to the partner window
    //var otherVideo = document.querySelector('.video.partner');
    //attachMediaStream(otherVideo, evt.stream); // from adapter.js
    attachMediaStream(evt);
}

const callbackNegotiationNeeded = (connection, evt) => {
    console.log("WebRTC: Negotiation needed...");
    //console.log("Event: ", evt);
}

const callbackIceCandidate = (evt, connection, partnerClientId) => {
    console.log("WebRTC: Ice Candidate callback");
    //console.log("evt.candidate: ", evt.candidate);
    if (evt.candidate) { // Found a new candidate
        console.log('WebRTC: new ICE candidate');
        //console.log("evt.candidate: ", evt.candidate);
        sendHubSignal(JSON.stringify({ "candidate": evt.candidate }), partnerClientId);
    } else {
        // Null candidate means we are done collecting candidates.
        console.log('WebRTC: ICE candidate gathering complete');
        sendHubSignal(JSON.stringify({ "candidate": null }), partnerClientId);
    }
}

const initializeConnection = (partnerClientId) => {
    console.log('WebRTC: Initializing connection...');
    //console.log("Received Param for connection: ", partnerClientId);

    var connection = new RTCPeerConnection(peerConnectionConfig);

```

```

    //connection.iceConnectionState = evt => console.log("WebRTC: iceConnection-
State", evt); //not triggering on edge
    //connection.iceGatheringState = evt => console.log("WebRTC: iceGatheringState",
evt); //not triggering on edge
    //connection.ondatachannel = evt => console.log("WebRTC: ondatachannel", evt);
//not triggering on edge
    //connection.oniceconnectionstatechange = evt => console.log("WebRTC: onicecon-
nectionstatechange", evt); //triggering on state change
    //connection.onicegatheringstatechange = evt => console.log("WebRTC: onicegather-
ingstatechange", evt); //triggering on state change
    //connection.onsignalingstatechange = evt => console.log("WebRTC: onsignal-
ingstatechange", evt); //triggering on state change
    //connection.ontrack = evt => console.log("WebRTC: ontrack", evt);
    connection.onicecandidate = evt => callbackIceCandidate(evt, connection, partner-
ClientId); // ICE Candidate Callback
    //connection.onnegotiationneeded = evt => callbackNegotiationNeeded(connection,
evt); // Negotiation Needed Callback
    connection.onaddstream = evt => callbackAddStream(connection, evt); // Add stream
handler callback
    connection.onremovestream = evt => callbackRemoveStream(connection, evt); // Re-
move stream handler callback

    connections[partnerClientId] = connection; // Store away the connection based on
username
    //console.log(connection);
    return connection;
}

sendHubSignal = (candidate, partnerClientId) => {
    console.log('candidate', candidate);
    console.log('SignalR: called sendhubsignal ');
    wsconn.invoke('sendSignal', candidate, partnerClientId).catch(errorHandler);
};

wsconn.onclose(e => {
    if (e) {
        console.log("SignalR: closed with error.");
        console.log(e);
    }
    else {

```

```

        console.log("Disconnected");
    }
});

// Hub Callback: Update User List
wsconn.on('updateUserList', (userList) => {
    consoleLogger('SignalR: called updateUserList' + JSON.stringify(userList));
    $('#usersLength').text(userList.length);
    $('#usersdata li.user').remove();

    $.each(userList, function (index) {
        var userIcon = "", status = "";
        if (userList[index].username === $('#upperUsername').text()) {
            myConnectionId = userList[index].connectionId;
            userIcon = 'icon-employee';
            status = 'Me';
        }

        if (!userIcon) {
            userIcon = userList[index].inCall ? 'icon-smartphone-1' : 'icon-smartphone-1';
        }
        status = userList[index].inCall ? 'In Call' : 'Available';

        var listString = '<li class="list-group-item user" data-cid=' + userList[index].con-
        nectionId + ' data-username=' + userList[index].username + '>';
        listString += '<a href="#"><div class="username"> ' + userList[index].username +
        '</div>';
        listString += '<span class="helper ' + userIcon + '" data-callstatus=' + userList[in-
        dex].inCall + '></span></a></li>';
        $('#usersdata').append(listString);
    });
});

// Hub Callback: Call Accepted
wsconn.on('callAccepted', (acceptingUser) => {
    console.log('SignalR: call accepted from: ' + JSON.stringify(acceptingUser) + '. Initi-
    ating WebRTC call and offering my stream up...');

    // Callee accepted our call, let's send them an offer with our video stream

```

```

    initiateOffer(acceptingUser.connectionId, localStream); // Will use driver email in
production
    // Set UI into call mode
    $('body').attr('data-mode', 'incall');
    $('#callstatus').text('In Call');
});

// Hub Callback: Call Declined
wsconn.on('callDeclined', (decliningUser, reason) => {
    console.log('SignalR: call declined from: ' + decliningUser.connectionId);

    // Let the user know that the callee declined to talk
    alertify.error(reason);

    // Back to an idle UI
    $('body').attr('data-mode', 'idle');
});

// Hub Callback: Incoming Call
wsconn.on('incomingCall', (callingUser) => {
    console.log('SignalR: incoming call from: ' + JSON.stringify(callingUser));

    // Ask if we want to talk
    alertify.confirm(callingUser.username + ' is calling. Do you want to chat?', function
(e) {
        if (e) {
            // I want to chat
            wsconn.invoke('AnswerCall', true, callingUser).catch(err => console.log(err));

            // So lets go into call mode on the UI
            $('body').attr('data-mode', 'incall');
            $('#callstatus').text('In Call');
        } else {
            // Go away, I don't want to chat with you
            wsconn.invoke('AnswerCall', false, callingUser).catch(err => console.log(err));
        }
    });
});

// Hub Callback: WebRTC Signal Received

```

```

wsconn.on('receiveSignal', (signalingUser, signal) => {
  //console.log('WebRTC: receive signal ');
  //console.log(signalingUser);
  //console.log('NewSignal', signal);
  newSignal(signalingUser.connectionId, signal);
});

// Hub Callback: Call Ended
wsconn.on('callEnded', (signalingUser, signal) => {
  //console.log(signalingUser);
  //console.log(signal);

  console.log('SignalR: call with ' + signalingUser.connectionId + ' has ended: ' + signal);

  // Let the user know why the server says the call is over
  alertify.error(signal);

  // Close the WebRTC connection
  closeConnection(signalingUser.connectionId);

  // Set the UI back into idle mode
  $('body').attr('data-mode', 'idle');
  $('#callstatus').text('Idle');
});

const initializeSignalR = () => {
  wsconn.start().then(() => { console.log("SignalR: Connected");
askUsername(); }).catch(err => console.log(err));
};

const setUsername = (username) => {
  consoleLogger('SignalR: setting username...');
  wsconn.invoke("Join", username).catch((err) => {
    consoleLogger(err);
    alertify.alert('<h4>Failed SignalR Connection</h4> We were not able to connect
you to the signaling server.<br/><br/>Error: ' + JSON.stringify(err));
    //viewModel.Loading(false);
  });
  //WOWZA_STREAM_NAME = username;

```



```

$("#upperUsername").text(username);
$('#div.username').text(username);
initializeUserMedia();
};

const askUsername = () => {
  consoleLogger('SignalR: Asking username...');
  alertify.prompt('Select a username', 'What is your name?', "", (evt, Username) => {
    if (Username !== "")
      setUsername(Username);
    else
      generateRandomUsername();

  }, () => {
    generateRandomUsername();
  });
};

const generateRandomUsername = () => {
  consoleLogger('SignalR: Generating random username...');
  let username = 'User ' + Math.floor((Math.random() * 10000) + 1);
  alertify.success('You really need a username, so we will call you... ' + username);
  setUsername(username);
};

const errorHandler = (error) => {
  if (error.message)
    alertify.alert('<h4>Error Occurred</h4></br>Error Info: ' + JSON.stringify(error.message));
  else
    alertify.alert('<h4>Error Occurred</h4></br>Error Info: ' + JSON.stringify(error));

  consoleLogger(error);
};

const consoleLogger = (val) => {
  if (isDebugging) {
    console.log(val);
  }
};

```

```

const WebRTC.js:
//Wowza WebRTC constants
const WEBRTC_CONSTRAINTS = { audio: true, video: false };
const ICE_SERVERS = [{ url: 'stun:numb.viagenie.ca' }, {
  url: 'turn:numb.viagenie.ca',
  username: 'shahzad@fms-tech.com',
  credential: 'P@ssw0rdfms'
}];
//const SERVER_URL = ""; //"wss://localhost.streamlock.net/webrtc-session.json"; set
it from the hub connection
const WOWZA_APPLICATION_NAME = "webrtc";
//const WOWZA_STREAM_NAME = ""; //"myStream"; set it from the user name
const WOWZA_SESSION_ID_EMPTY = "[empty]";

const STATUS_OK = 200;
const STATUS_APPLICATION_FAILURE = 500;
const STATUS_ERROR_STARTING_APPLICATION = 501;
const STATUS_ERROR_STREAM_NOT_RUNNING = 502;
const STATUS_STREAMNAME_INUSE = 503;
const STATUS_STREAM_NOT_READY = 504;
const STATUS_ERROR_CREATE_SDP_OFFER = 505;
const STATUS_ERROR_CREATING_RTP_STREAM = 506;
const STATUS_WEBRTC_SESSION_NOT_FOUND = 507;
const STATUS_ERROR_DECODING_SDP_DATA = 508;
const STATUS_ERROR_SESSIONID_NOT_SPECIFIED = 509;

const CODEC_AUDIO_UNKNOWN = -1;
const CODEC_AUDIO_PCM_BE = 0x00;
const CODEC_AUDIO_PCM_SWF = 0x01;
const CODEC_AUDIO_AC3 = 0x01; //TODO steal this slot
const CODEC_AUDIO_MP3 = 0x02;
const CODEC_AUDIO_PCM_LE = 0x03;
const CODEC_AUDIO_NELLYMOSER_16MONO = 0x04;
const CODEC_AUDIO_NELLYMOSER_8MONO = 0x05;
const CODEC_AUDIO_NELLYMOSER = 0x06;
const CODEC_AUDIO_G711_ALAW = 0x07;
const CODEC_AUDIO_G711_MULAW = 0x08;
const CODEC_AUDIO_RESERVED = 0x09;
const CODEC_AUDIO_VORBIS = 0x09; //TODO steal this slot
const CODEC_AUDIO_AAC = 0x0a;

```

```

const CODEC_AUDIO_SPEEX = 0x0b;
const CODEC_AUDIO_OPUS = 0x0c;
const CODEC_AUDIO_MP3_8 = 0x0f;
window.RTCPeerConnection = window.RTCPeerConnection || window.mozRTCPeer-
Connection || window.webkitRTCPeerConnection;
window.RTCIceCandidate = window.RTCIceCandidate || window.mozRTCIceCandi-
date || window.webkitRTCIceCandidate;
window.RTCSessionDescription = window.RTCSessionDescription || window.-
mozRTCSessionDescription || window.webkitRTCSessionDescription;
    utilsRTC.js:
var isEdge = navigator.userAgent.indexOf('Edge') !== -1 && (!!navigator.msSave-
OrOpenBlob || !!navigator.msSaveBlob);
var isOpera = !!window.opera || navigator.userAgent.indexOf(' OPR/') >= 0;
var isFirefox = typeof window.InstallTrigger !== 'undefined';
var isSafari = /^(?!chrome|android).*safari/i.test(navigator.userAgent);
var isChrome = !!window.chrome && !isOpera;
var isIE = typeof document !== 'undefined' && !!document.documentMode && !
isEdge;
var edgeVersion = isEdge ? navigator.userAgent.split('Edge/')[1].replace(".", "") : "";
    libman.json:
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.4.1",
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "library": "twitter-bootstrap@4.4.1",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "unpkg",
      "library": "@microsoft/signalr@3.1.0",
      "destination": "wwwroot/lib/signalr/"
    },
    {
      "provider": "jsdelivr",
      "library": "icomoon@1.0.0",

```

```

    "destination": "wwwroot/lib/icomoon/"
  },
  {
    "library": "adapterjs@0.15.4",
    "destination": "wwwroot/lib/adapter/"
  },
  {
    "provider": "unpkg",
    "library": "alertifyjs@1.13.1",
    "destination": "wwwroot/lib/alertifyjs/"
  }
]
}

  launchSettings.json:
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:13481",
      "sslPort": 44389
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "SignalRCoreWebRTC": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```