Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Курило Максима Васильовича*
(ПІБ)

академічної групи *122М-19-1*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

на тему: *Інформаційна технологія удосконалення процесу передачі файлів бездротовим способом між пристроями з різними операційними системами на базі технології WebRTC*

_____ *М.В. Курило*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	Проф. Мещеряков Л.І.			
економічний	Доц. Касьяненко Л.В.			

Рецензент				

Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)
_____ І.М. Удовик
(підпис) (прізвище, ініціали)
« » _____ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122М-19-1* *Курило Максиму Васильовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Інформаційна технологія удосконалення процесу передачі файлів бездротовим способом між пристроями з різними операційними системами на базі технології WebRTC*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 р. № 888-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес бездротової передачі файлів між пристроями з різними операційними системами.

Предмет досліджень – методи бездротової передачі файлів між пристроями з різними операційними системами.

Мета роботи – підвищення ефективності передачі файлів між пристроями з різними операційними системами.

Методи дослідження – базуються на основних принципах системного аналізу, функціонального аналізу, теорії баз даних. Також були використані методи емпіричного узагальнення на основі даних.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна – дістала подальшого розвитку технологія передачі файлів за допомогою WebRTC.

Практична цінність полягає в тому, що створений в результаті роботи додаток можна використовувати у повсякденному житті для передачі файлів з одного пристрою на інший незалежно від їх операційної системи.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі.	15.09.2020-30.09.2020
Дослідження існуючих підходів до створення додатків з підтримкою різних операційних систем, а також методів передачі даних за допомогою WebRTC.	01.10.2020-05.11.2020
Створення додатку для передачі файлів між пристроями з різними операційними системами на базі технології WebRTC.	06.11.2020-07.12.2020

Завдання видав _____ *Мещеряков Л.І.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Курило М.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *15.09.2020 р.*

Термін подання кваліфікаційної роботи до ЕК 10.12.2020

РЕФЕРАТ

Пояснювальна записка: 90 с., 33 рис., 10 табл., 3 дод., 70 джерел.

Об'єкт дослідження: процес бездротової передачі файлів між пристроями з різними операційними системами.

Предмет дослідження: методи бездротової передачі файлів між пристроями з різними операційними системами.

Мета магістерської роботи: підвищення ефективності передачі файлів між пристроями з різними операційними системами.

Методи дослідження. Методи дослідження базуються на основних принципах системного аналізу, функціонального аналізу, теорії баз даних. Також були використані методи емпіричного узагальнення на основі даних.

Наукова новизна: дістала подальшого розвитку технологія передачі файлів за допомогою WebRTC.

Практична цінність полягає в тому, що створений в результаті роботи додаток можна використовувати у повсякденному житті для передачі файлів з одного пристрою на інший незалежно від їх операційної системи.

У розділі «Економіка» проведено розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПО і тривалості його розробки.

Список ключових слів: багатоплатформовий, бездротова передача файлів, графічний інтерфейс, клієнт-серверна архітектура, peer-to-peer, WebRTC, JavaScript, Firebase.

ABSTRACT

Explanatory note: 90 p., 33 fig., 10 tables, 3 applications, 70 sources.

Object of research: process of wireless files transfer between devices with different operating systems.

Subject of research: methods of wireless files transfer between devices with different operating systems.

Purpose of Master's thesis: improving the efficiency of files transfer between devices with different operating systems.

Research methods. Research methods are based on main principles of system analysis, functional analysis and the theory of databases. Also data-based empirical generalization methods were used.

Originality of research is in consists of improvements of the file transfer technology through WebRTC.

Practical value of the results consists of the developed app which can be used in everyday life for files transfer from one device to another wirelessly and independently from their operating systems.

In the Economics section the complexity, costs and duration of software development were calculated.

Keywords: cross-platform, wireless file transfer, graphical interface, client-server architecture, peer-to-peer, WebRTC, JavaScript, Firebase.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД –	База даних
ПЗ –	Програмне забезпечення
ПК –	Персональний комп’ютер
ОС –	Операційна система
API –	Application Programming Interface
DOM –	Document Object Model
HTML –	HyperText Markup Language
HTTP –	HyperText Transfer Protocol
JSON –	JavaScript Object Notation
NAT –	Network Address Translation
NFC –	Near Field Communication
P2P –	Peer to peer
SCTP –	Stream Control Transmission Protocol
TCP –	Transmission Control Protocol
UDP –	User Datagram Protocol
WebRTC –	Web Real Time Communications

ЗМІСТ

ВСТУП

Актуальність дослідження. За останнє десятиліття ми стали свідками як фізичні носії, на кшталт компакт-дисків та флеш-носіїв були майже витіснені бездротовими способами передачі файлів.

Це було досягнуто за рахунок повсюдного розвитку мережі інтернет та інших бездротових технологій. Поділитися файлом тепер можна через хмарне сховище, або навіть без нього. Великі компанії серед яких Apple, Microsoft, Huawei розробляють рішення, що дозволяють передавати дані прямо з одного пристрою на інший без проміжного сервера. Наприклад, Apple AirDrop, Windows Nearby Share, Huawei Share та інші.

Однак кожне з цих рішень провідних корпорацій працює лише в окремій екосистемі. В даній роботі планується розробити рішення, що буде націлене на використання на пристроях під керуванням різних операційних систем.

Мета дослідження полягає в підвищення ефективності передачі файлів між пристроями з різними операційними системами.

Завдання дослідження. Для досягнення поставленої мети в роботі сформульовані та вирішені такі завдання:

1. Визначити існуючі технології бездротової передачі даних;
2. Визначити архітектуру додатку;
3. Розробити багатоплатформовий додаток для передачі файлів;
4. Провести дослідження ефективності роботи даного додатку.

Об’єкт дослідження: процес бездротової передачі файлів між пристроями з різними операційними системами.

Предмет дослідження: методи бездротової передачі файлів між пристроями з різними операційними системами.

Методи дослідження: Методи дослідження базуються на основних принципах системного аналізу, функціонального аналізу, теорії баз даних. Також були використані методи емпіричного узагальнення на основі даних.

Наукова новизна: дістала подальшого розвитку технологія передачі файлів за допомогою WebRTC.

Практичне значення: полягає в тому, що створений в результаті роботи додаток можна використовувати у повсякденному житті для передачі файлів з одного пристрою на інший незалежно від їх операційної системи.

Особистий внесок автора:

1. Наукові результати роботи отримані автором самостійно;
2. Вибір методів досліджень і технологій реалізацій;
3. Реалізація передачі файлів за допомогою технології WebRTC;
4. Розробка теоретичної частини роботи, в якій досліджені і систематизовані знання про передачу файлів за допомогою технології WebRTC;
5. Оцінка отриманих результатів.

Структура і обсяг роботи. Робота складається з вступу, чотирьох розділів і висновків. Містить 90 сторінок, в тому числі 60 сторінок тексту основної частини з 33 рисунками та 10 таблицями, списку використаних джерел з 70 найменуваннями на 6 сторінках та 3 додатки на 16 сторінках.

РОЗДІЛ 1
БЕЗДРОТОВА ПЕРЕДАЧА ДАНИХ МІЖ ПРИСТРОЯМИ ПІД КЕРУВАННЯМ РІЗНИХ ОПЕРАЦІЙНИХ СИСТЕМ

1.1. Бездротова передача даних

В період з 2010 – 2020рр. суттєво виросла кількість рішень та технологій, що дозволять передавати файли бездротовим шляхом. Ще наприкінці нульових компакт-диски та USB-флеш-накопичувачі вважалися невід'ємною частиною нашого життя, а зараз майже повністю витіснені бездротовими.

До переваг бездротової передачі даних можна віднести зручність користування таким рішенням. Щоб відправити дані з одного пристрою на інший не потрібно мати при собі ніяких фізичних каналів для передачі (на кшталт кабеля з USB-роз'ємами). Для того, щоб відправити дані іншому пристрою достатньо декілька кліків у графічному інтерфейсі додатку.

Також ще однією перевагою бездротової передачі є сумісність з багатьма пристроями. Користувачеві не потрібно мати декілька кабелів з роз'ємами під кожен конкретний пристрій.

До недоліків бездротової передачі даних перед фізичними носіями є відносно невелика швидкість передачі даних. Сучасні стандарти, наприклад, Thunderbolt 3 можуть забезпечити швидкість передачі до 5 ГБ/с. Бездротова ж передача обмежена пропускною здатністю технологій. Наприклад, локальні мережі здатні забезпечити до 1 Гбіт/с (0.125 ГБ/с). Технологія Bluetooth 5 – 2 Мбіт/с, NFC – 424 Кбіт/с.

Однак дійсно важливе значення швидкість має лише при передачі надзвичайно великих обсягів даних і коли час передачі має критичну важливість. Але в повсякденному житті при передачі невеликих порцій файлів, швидкості локальної мережі навіть з пропускною спроможністю в 300 Мбіт/с вистачить для комфортного користування.

Великі компанії вкладають значні гроші в розвиток додатків, що дозволяють передавати дані прямо з одного пристрою на інший. Водночас рішення корпорацій зазвичай працюють лише на продуктах цих корпорацій. Наприклад, технологія AirDrop працює лише на продуктах компанії Apple під керуванням операційних систем iOS та MacOS. Схоже рішення від Huawei так само підтримується лише пристроями цієї компанії.

Таким чином, на даний момент є потреба у розробці додатку, що підтримував би багато платформ та дозволяв би користувачам легко і швидко передавати дані між пристроями. Однак для підтримки багатьох пристроїв одразу дуже важливо обрати правильну архітектуру додатку та дотримуватися всіх важливих вимог до їх створення.

1.2. Проблеми підтримки багатьох пристроїв

В світі є мільярди комп’ютерів різного типу з різними операційними системами. Персональні комп’ютери, ноутбуки, планшети, смартфони, тощо під керуванням MacOS, Windows, Linux, iOS або Android. Існує безліч варіацій які потрібно враховувати та підтримувати.

Передача файлів – не виняток. Розроблений в ході цієї роботи додаток має виконувати свої функції на будь-якому з названих вище типі пристроїв під керуванням будь-якої із вище зазначених ОС.

1.2.1. Підтримка різних типів пристроїв

Різні пристрою можуть мати різні інтерфейси передачі інформації, або не мати таких взагалі. Таким чином при побудові рішення, що буде підтримуватись на якомога більшій кількості пристроїв потрібно вибрати ті інтерфейси, які є в наявності у всіх класів пристроїв.

Наприклад, переважна більшість сучасних смартфонів мають Bluetooth та Wi-Fi. Тим же переліком інтерфейсів володіють і ноутбуки. Однак більшість персональних комп'ютерів цих інтерфейсів не мають.

Таким чином технологію передачі даних не можна будувати поверх тих технологій, які не підтримуються всіма категоріями комп'ютерів, що нас цікавлять. З табл. 1.1 Можна зробити висновок, що при побудові такої технології не варто використовувати Wi-Fi, Bluetooth або NFC, оскільки вони притаманні лише певній кількості класів пристроїв.

Таблиця 1.1

Приклад підтримки технологій передачі даних, що можуть використовуватися різними типами пристроїв

Тип пристрою	Wi-Fi	Bluetooth	NFC	Internet
Смартфон	+	+	+	+
Планшет	+	+		+
Ноутбук	+	+		+
Персональний комп'ютер				+

Найбільш універсальним рішенням буде використання більш високорівневих технологій, таких як Інтернет. Тут потрібно зробити уточнення, що це більш високорівнева абстракція, ніж просто технології, що описані поруч з нею. Так, наприклад, смартфон може бути під'єднаним до мережі за допомогою WiFi, а ПК – за допомогою Ethernet. Але передати дані з одного пристрою на інший вийде, оскільки WiFi та Ethernet слугують лише засобами для підключення до мережі.

1.2.2. Підтримка різних операційних систем

Ще одним важливим фактором є підтримка не тільки физичних пристроїв, а й різних операційних систем, що можуть керувати цими пристроями.

Для кожної операційної системи притаманні власні середовища розробки та фреймворки. Наприклад, для Windows доволі поширеною є фреймворк .NET та мова програмування C#. Для MacOS та iOS – Cocoa Framework та мова програмування Swift.

Фінальний програмний продукт має підтримуватись декількома операційними системами. Для цього можна розробити універсальний алгоритм роботи програми, який буде втілено за допомогою різних мов програмування під різні операційні системи.

Переваги такого методу в тому, що додаток буде оптимізовано під кожну конкретну ОС. Швидкість виконання програми буде максимальною, що позитивно вплине на досвід користування цим додатком. Також оновлення фреймворку зможуть дати можливості для збільшення функціоналу та оптимізації процесу підтримки.

Недоліками є значний обсяг розробки, масштабування та підтримки. Навіть якщо є загальний алгоритм роботи, його потрібно втілити для всіх ОС, що мають підтримуватись. Також, якщо планується розширення певного функціоналу, він має бути доданий до всіх існуючих рішень. Те ж стосується й виправлення помилок ПЗ. Таким чином, зростає ціна кінцевого продукту та його підтримки.

1.2.3. Підтримка різних користувальницьких інтерфейсів

Оскільки додаток повинен мати графічний інтерфейс, з яким буде взаємодіяти користувач, то він також має бути адаптованим під різні типи пристроїв.

Наприклад, смартфон має витягнутий по вертикалі екран та відносно невелику його площу. На такому екрані за один раз повинна бути відносно невелика кількість елементів інтерфейсу і вони не мають бути ні занадто дрібними, щоб ними було зручно користуватись, ні занадто великими, щоб не займали завелику площу.

Повна протилежність – екран ноутбуку або монітор персонального комп'ютера. Вони зазвичай витягнуті по ширині, мають велику площу та роздільну здатність. На таких екранах вміщується великий обсяг контенту, а елементи інтерфейсу можна робити дрібнішими. На рис. 1.1 наведено приклад різниці між інтерфейсами для різних типів пристроїв.

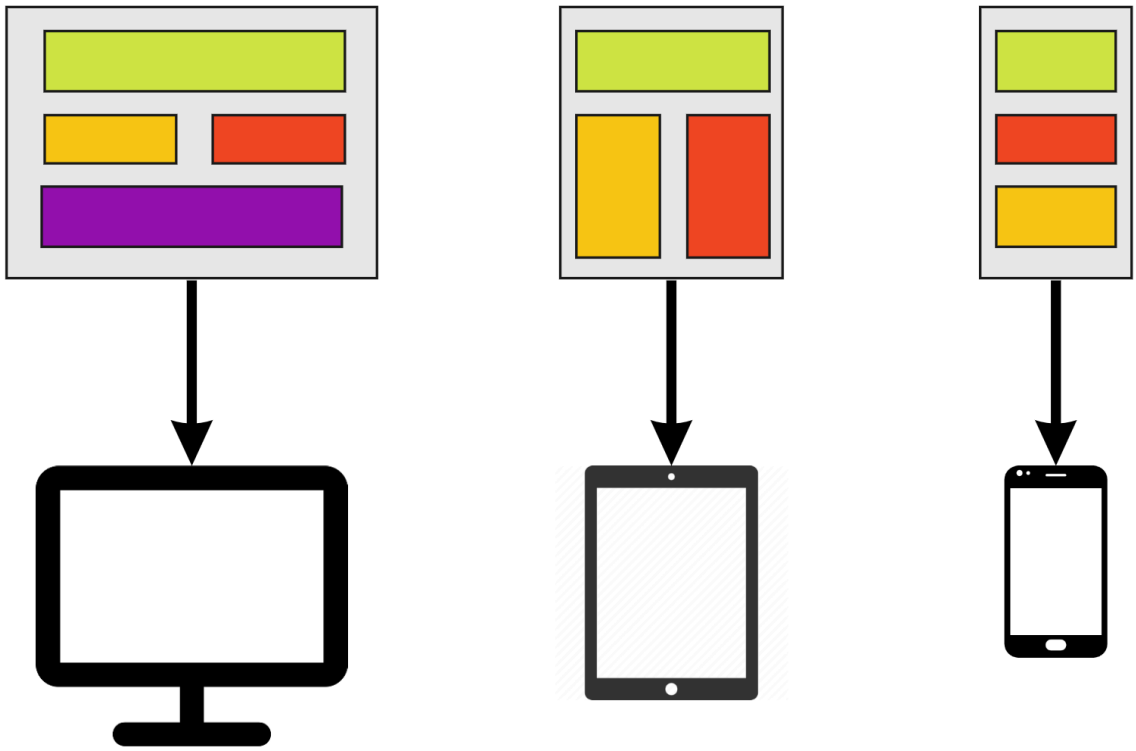

Рис. 1.1. Різниця між інтерфейсами для ПК, планшетами і смартфонами

Розмітка сторінки сильно відрізняється у різних типів пристроїв. Таким чином для кожного типу пристроїв потрібно підготувати власний макет розташування елементів та продумати всю навігацію в додатку незалежно одне від одного.

1.2.4. Багатоплатформні фреймворки та мови програмування

Існують мови програмування та фреймворки, що одразу мають на меті розробку додатків для багатьох платформ. Однією з таких мов є скриптова мова JavaScript.

JavaScript було розроблено в 1995 році як заміна Java-апплетам для веб-браузерів. Ця скриптова мова дозволяє взаємодіяти з розміткою, контентом та стилями сторінки, таким чином дозволяючи робити анімацію та складну логіку роботи додатку на стороні клієнта прямо в браузері.

JavaScript поєднує в собі функціональний та об'єктно-орієнтований стилі програмування та має схожий на мову програмування C синтаксис. Однак є і багато відмінностей від того ж C:

- анонімні функції;
- автоматична збірка сміття;
- функції є об'єктами першого класу (їх можна передавати як параметри або назначати змінним);
- та інше.

З появою в 2010 році програмної платформи Node.js JavaScript перестав бути виключно браузерною мовою, а тепер міг бути і на серверній стороні. Node.js дав значний поштовх до розвитку фреймворків для JavaScript, що значно полегшили створення веб-додатків, а також дозволили розширити функціонал цієї мови, а також і супутніх технологій – HTML та CSS.

Наразі JavaScript є найпопулярнішою мовою в світі для створення веб-додатків, а також є сильним конкурентом при створенні нативних додатків для різних операційних систем. Саме тому найбільш правильним рішенням для розробки буде використати саме JavaScript, оскільки він дає підтримку якомога більшого спектру пристроїв та технологій.

JavaScript та веб-програмування в цілому допоможе також вирішити і проблему дизайну для різних пристроїв. Для цього є спеціальна парадигма – адаптивний дизайн. Адаптивний дизайн (рис 1.2) передбачає, що один додаток

пристосовується до всіх типів пристроїв. Тобто залежно від ширини та висоти екрану, інтерфейс додатку сам підлаштовується таким чином, щоб ним було зручно користувались.

Рис. 1.2. Адаптивний дизайн

Трудомісткість самого адаптивного інтерфейсу значно вище, ніж для звичайного. Однак перевагою такого підходу полягає в тому, що одна розмітка для всіх типів пристроїв значно економить час розробки, оскільки не потрібно робити окремі додатки для кожного окремого пристрою.

1.3. Передача файлів через мережу

Як було констатовано в підрозділі 1.2. найбільш поширеною технологією передачі даних є передача їх через мережу. Отже потрібно обрати архітектуру, що якнайкраще буде підходити для передачі файлів між різними пристроями.

Найбільш поширеною архітектурою для передачі даних в веб-додатках є клієнт-серверна архітектура. Клієнт-сервер – мережева архітектура в яких мережеве навантаження розподілені між постачальником послуг (сервером) та замовником (клієнтом) (рис 1.3). Фактично клієнт та сервер – це програмне

забезпечення. Зазвичай ці програми розташовані на різних обчислювальних машинах, та взаємодіють між собою через мережу за допомогою мережевих протоколів, але можуть також і бути розташовані на одній машині (наприклад, при локальній розробці).

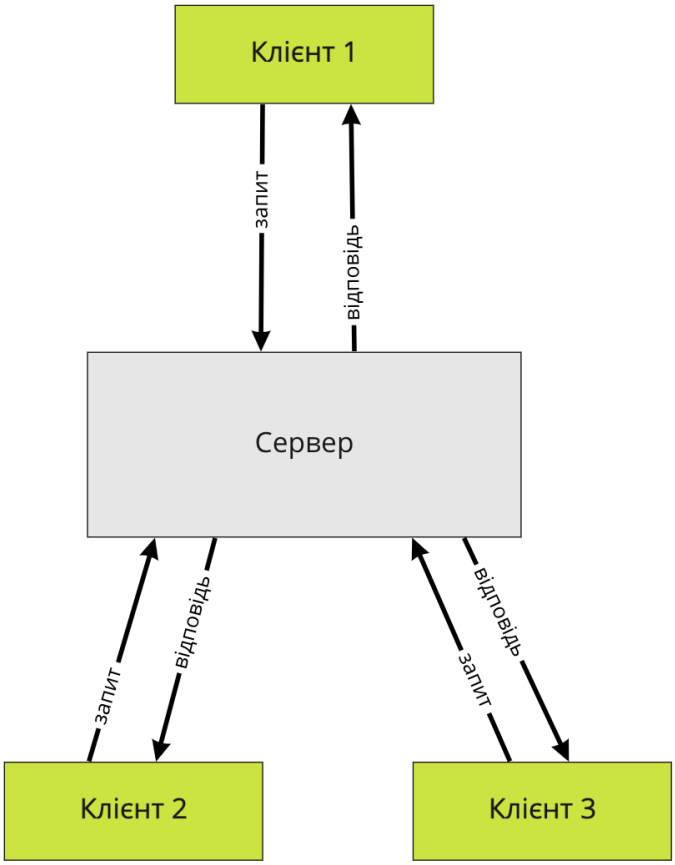

Рис. 1.3. Клієнт-серверна архітектура

Сервер очікує запит від клієнта та надає свої послуги у вигляді даних, роботи з базою даних, завантаження файлів або будь-які інші сервісні функції. Оскільки для одного серверу може бути багато клієнтів, сервер зазвичай встановлюють на спеціально виділеній обчислювальній машині, налаштованій таким чином, щоб оптимально опрацьовувати багато запитів від клієнтів.

Однак все одно сервер має досить обмежені ресурси, а провайдери зазвичай обмежують трафік, який може обробляти кожен сервер. Згідно з чистої клієнт-серверної при обміні одного файлу між двома користувачами, всі дані будуть йти через сервер. Якщо при невеликому розмірі файлу та невеликій

кількості користувачів це не критично, то якщо передавати файли в декілька гігабайт, то сервер може стикнутися з надлишковим навантаженням.

Таким чином, саме дані файлу найкраще передавати між двома користувачами напряму без проміжного пункту у вигляді серверу. Для цих цілей підходить тип архітектури peer-to-peer.

1.4 Архітектура передачі даних peer-to-peer

Для передачі файлів з одного пристрою на інший без проміжних серверів, що зберігали б контент, якнайкраще підходить ідея архітектури peer-to-peer.

Peer-to-peer – це архітектура передачі даних, що основана на рівноправності всіх учасників мережі (рис. 1.4). В мережі відсутні окремі сервери, так що кожен учасник одночасно є як клієнтом, так і сервером.

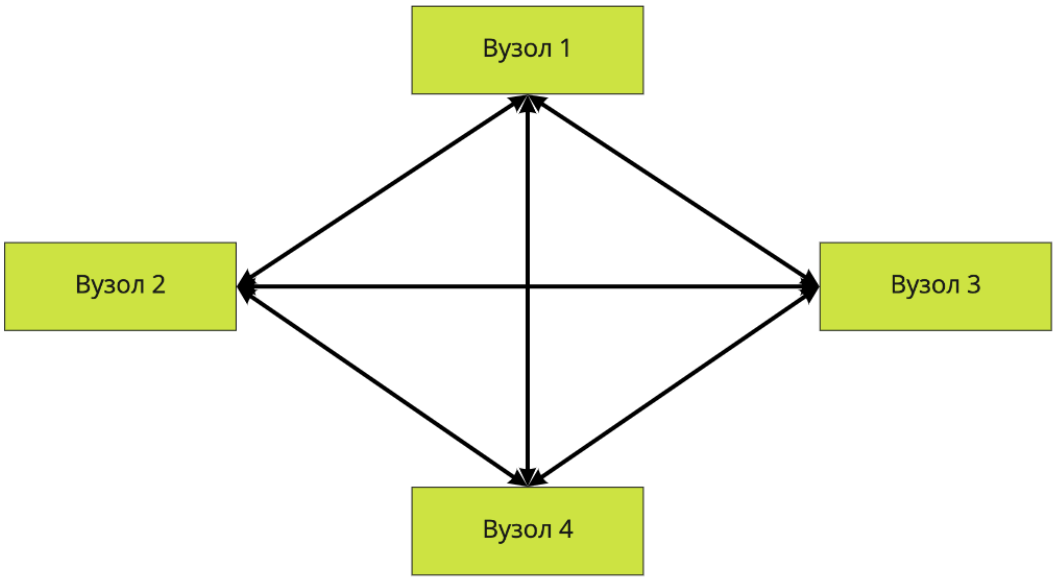

Рис. 1.4. Схема однорангової мережі

1.4.1. Сервер для координації роботи учасників

Крім чистих P2P-мереж існують і гібридні мережі, в яких є сервер для координації дій учасників, пошуку або надання інформації про інших учасників мережі, тощо. Такий тип мережі поєднує швидкість децентралізованих мереж та надійність централізованих.

Розглянемо принцип роботи такої мережі на прикладі протоколу передачі BitTorrent.

Протокол BitTorrent складається з наступного: існує невеликий торрент-файл, що містить інформацію про трекер, що стежить за файлом та список сегментів з яких складається файл. Для сегмента зберігається 160-бітний SHA-1 відповідного сегмента файлу (зазвичай розмір сегмента коливається від 64 до 512 кб). Таким чином, завантаживши файл можна порахувати хеші від нього та визначити чи все завантажилось без помилок.

Крім торрент-файла також є ще трекер – сервер, що стежить за станом вузлів, що мають відношення до передачі файлу. Клієнт, що отримав торент-файл підключається до трекеру та дізнається про список тих вузлів, в кого вже є певні сегменти цього файлу. Дізнавшись адресу вузла, у якого можна щось завантажити, клієнт звертається до нього напряму та завантажує необхідні дані. Тож в цьому випадку трекер виступає лише в якості координатора процесу.

На рис 1.5 графічно проілюстровано як клієнти взаємодіють з трекером, щоб отримати інформацію про знаходження потрібної їм частини файлу.

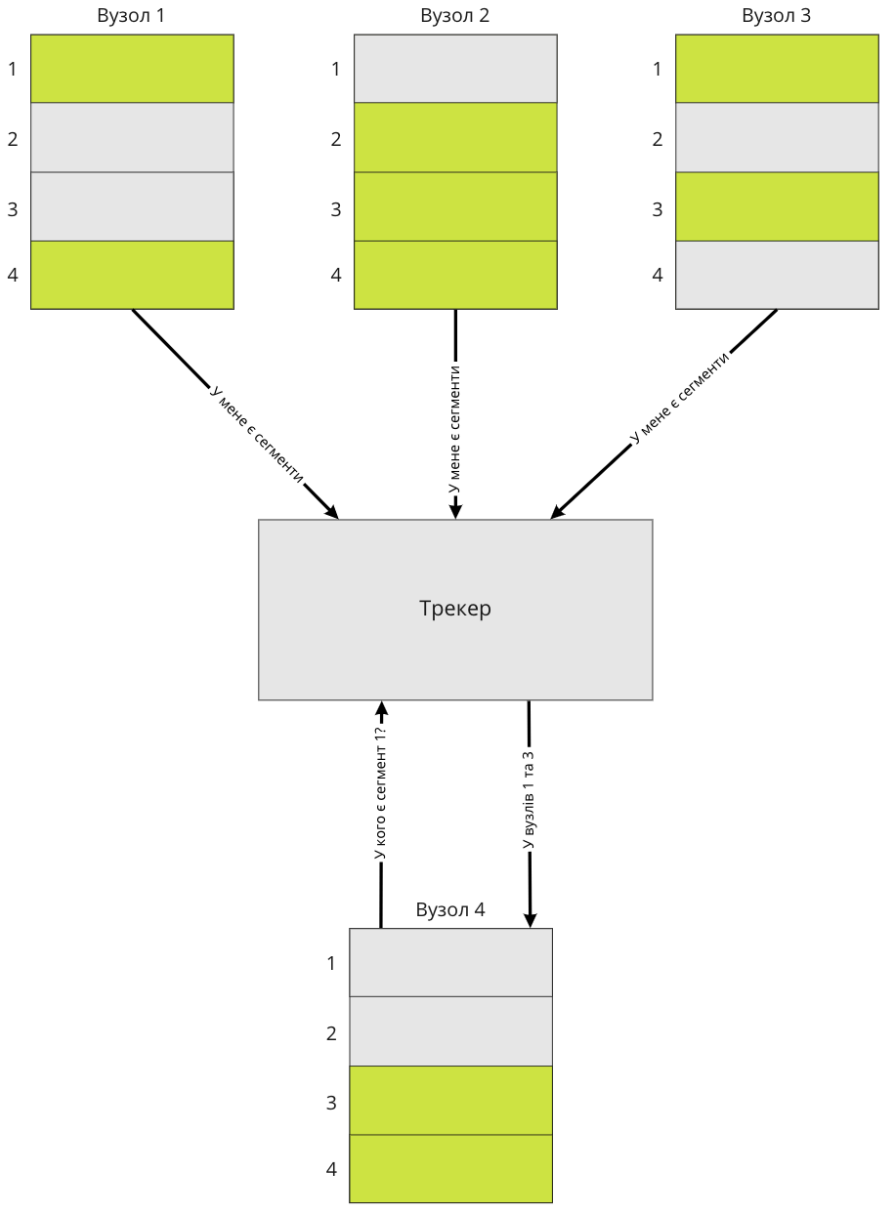

Рис. 1.5. Схема роботи протоколу BitTorrent

Таким чином, протокол не є централізованим (немає ніяких виділених сховищ з даними), але вирішує проблему координації роботи учасників.

Слабким місцем такої системи можна назвати сам трекер: він завжди повинен бути в мережі та бути готовим координувати роботу великої кількості учасників.

1.4.2. Архітектура peer-to-peer у WebRTC

Організувати peer-to-peer комунікацію можна власноруч, для цього потрібно лише дізнатися IP адреси вузлів, налагодити зв'язок між ними, зробити систему передачі файлу і, власне, передати файл. Однак можна скористатися готовим рішенням, що дає технологія WebRTC.

Для WebRTC архітектура peer-to-peer зроблена схожим чином з протоколом BitTorrent. Але координуючих серверів тут декілька і всі мають певну роль в синхронізації дій учасників.

По-перше, для того щоб дізнатися про підключення нових учасників та отримання від них інформації про їхнє знаходження в мережі є сервер з веб-сокетами (рис. 1.6). Цей сервер є аналогом трекера, з описаного вище протоколу BitTorrent. Його задача – передавати дані, що надходять від одного вузла до інших. Таким чином всі вузли дізнаються, коли до сесії приєднується новий вузол та отримують інформацію про нього.

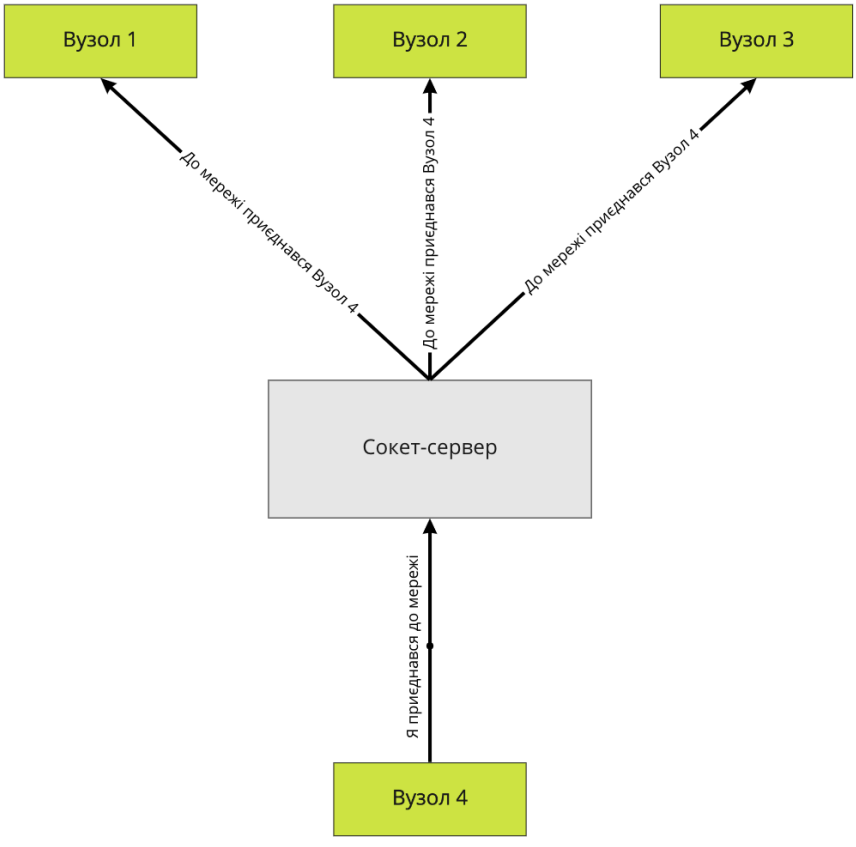

Рис. 1.6. Схема взаємодії вузлів із сокет-сервером

По-друге, для нормальної роботи WebRTC не лише в локальній мережі, а й за її межами потрібні STUN/TURN сервери (докладніше про них в Розділі 2). Завдання цих серверів – повідомляти вузлу його зовнішню IP-адресу та порт. Клієнт отримавши ці дані, може розіслати їх іншим учасникам мережі, щоб вони в свою чергу могли підключитися до нього (рис. 1.7).

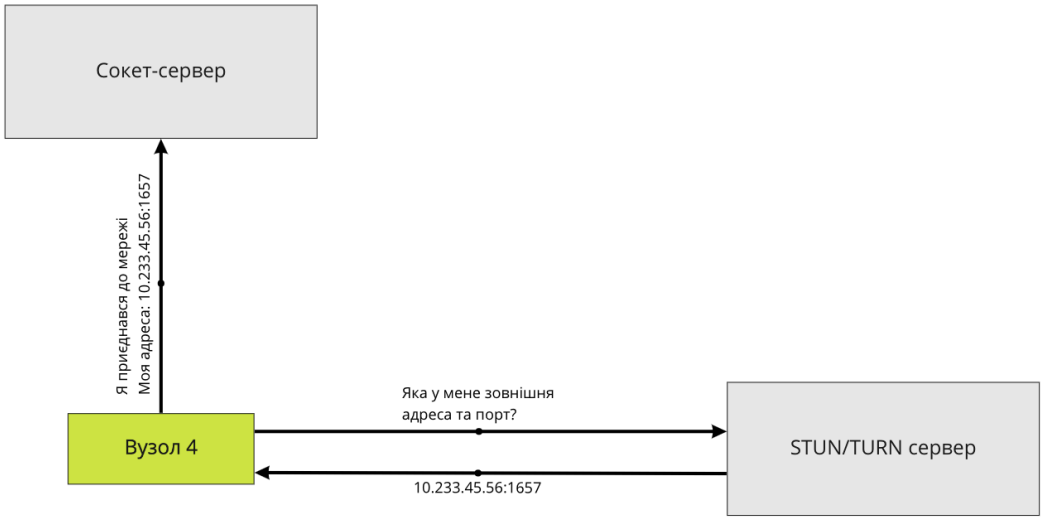

Рис. 1.7. Схема взаємодії вузла із STUN/TURN сервером

1.5. Висновки до першого розділу

При вивченні особливостей створення багатоплатформових додатків було виділено три основні проблеми:

1. Підтримка різних фізичних пристроїв;
2. Підтримка різних операційних систем;
3. Підтримка різних користувальницьких інтерфейсів.

Для вирішення цих проблем було обрано використання скриптової мови JavaScript, що підтримується багатьма платформами та має широку підтримку серед сучасних фреймворків для веб-розробки.

Для забезпечення оптимального користувальницького досвіду додатком було прийнято рішення використовувати парадигму адаптивного дизайну. Таким чином, буде зроблено один дизайн в рамках одного додатку, що буде

підлаштовуватись під будь-який тип пристроїв: персональний комп’ютер, ноутбук, планшет або смартфон.

В якості архітектури додатку в цілому, була обрана клієнт-серверна архітектура, а для безпосередньої передачі файлів – peer-to-peer. Таким чином поєднається простота та стабільність клієнт-серверної архітектури, а також забезпечиться мінімальне навантаження на сервер, через те, що найбільш вибаглива до ресурсів передача файлів буде здійснюватися між користувачами напряму.

Для забезпечення роботи архітектури peer-to-peer було обрано використання технології WebRTC, оскільки вона сумісна з JavaScript та дозволяє не витрачати час на створення власного способу передачі даних між двома користувачами. Це значно спростить розробку, а також забезпечить стабільність в роботі додатка в цілому, оскільки ця технологія підтримується усіма сучасними браузерами та операційними системами.

РОЗДІЛ 2
ТЕХНОЛОГІЯ ПЕРЕДАЧІ ФАЙЛІВ ЗА ДОПОМОГОЮ WEBRTC

2.1. Принцип роботи WebRTC

WebRTC – це технологія, що дозволяє веб-додаткам та сайтам захоплювати та вибірково передавати аудіо та/або відео медіа-потоки, а також обмінюватись довільними даними між браузерами без обов'язкового використання посередників. Набір стандартів, що включає в себе технологія WebRTC дозволяє обмінюватись даними та проводити пірингові телеконференції без необхідності користувачу встановлювати плагіни або будь-яке стороннє програмне забезпечення.

2.1.1. Дескриптор сесії (SDP)

У різних комп'ютерів завжди будуть різні камери, мікрофони та інше обладнання. Існує безліч параметрів, які потрібно синхронізувати для передачі даних між вузлами. WebRTC робить це автоматично, та створює унікальний об'єкт – дескриптор сесії (SDP).

Локальний дескриптор (Offer SDP) сесії необхідно передати іншому вузлу, з яким ще не встановлено зв'язок. Для цього використовується сторонній сигнальний механізм (як правило за допомогою веб-сокетів). Після отримання чужого дескриптору, вузол генерує власний дескриптор (Answer SDP), спираючись на отримані дані, та відправляє його у якості відповіді через той же сторонній механізм передачі.

Після того як вузли обмінялись дескрипторами, вони мають встановити і локальний, і чужий дескриптор для WebRTC. Це необхідно для того, щоб знайти компроміс між «побажаннями» обох сторін. Наприклад, Вузол 1 підтримує кодек A та B, а Вузол 2 – кодеки B та C. Таким чином, буде обрано кодек, що підтримується обома сторонами – кодек B (рис. 2.1).

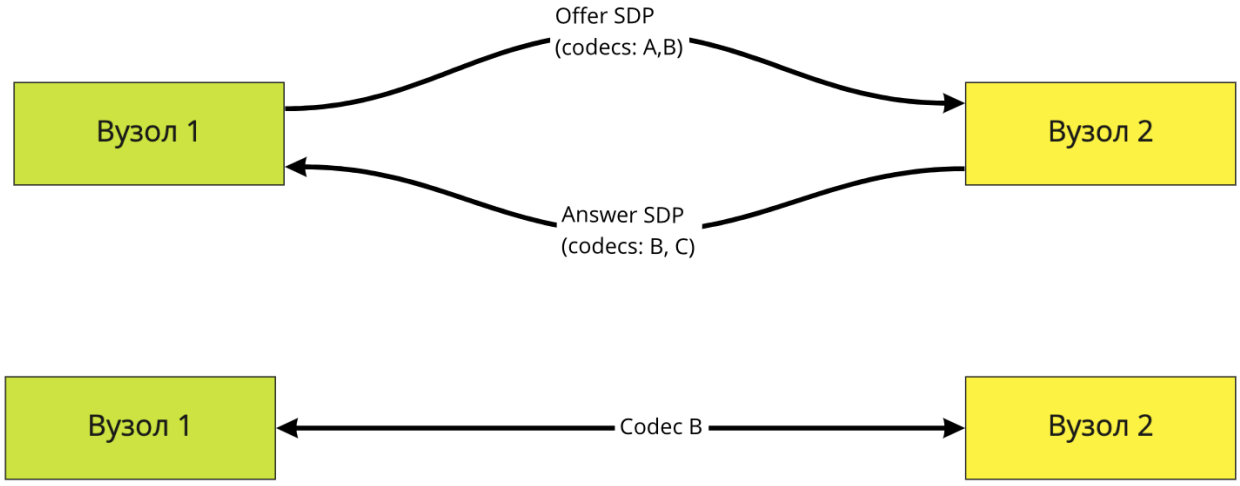

Рис. 2.1. Вибір кодеку між двома вузами

2.1.2. Кандидати (Ice candidate)

Дескриптор сесії допомагає налаштувати лише логічне з’єднання, але поки що немає шляхів для передачі даних між двома вузлами. Для цього існують інші об’єкти – Ice Candidate, які так само генеруються за допомогою WebRTC і які також треба переслати через сигнальний механізм іншому вузлу.

На відміну від дескриптора сесії, який містив інформацію про налаштування пристроїв вузлів, Ice Candidate містять інформацію про знаходження вузла в мережі.

Ще одна відмінність об’єктів кандидатів від дескриптору сесії в тому, що у себе потрібно встановлювати тільки чужий Ice Candidate, оскільки його не можна редагувати.

Кандидатів може бути декілька, оскільки пристрої мають декілька IP адрес – внутрішні та глобальний.

2.1.3. STUN та TURN сервери

Для того, щоб вузол міг дізнатися свою адресу не лише в локальній мережі, а й за межами NAT (Network Address Translation), потрібен STUN сервер.

STUN сервер – це просто деякий сервер в інтернеті, що повертає адресу відправника, тобто вузла що знаходиться за маршрутизатором. Розглянемо детально принцип взаємодії STUN серверу з клієнтом.

Візьмемо деяку таблицю NAT, що існує на маршрутизаторі користувача. Вона містить інформацію про адресу та порт вузла всередині мережі (Internal IP та Internal Port) та за її межами (External IP та External Port). На початку ця таблиця буде порожньою (табл. 2.1).

Таблиця 2.1

Порожня таблиця NAT маршрутизатора

Internal IP	Internal Port	External IP	External Port

Вузол хоче відправити пакет на STUN-сервер. Наведені адреси були використані для прикладу, в реальному пакеті замість них знаходяться справжні адреси STUN-серверу та вузла. Частина заголовку пакету наведена в табл. 2.2.

Таблиця 2.2

Заголовок пакету від вузла

Src IP	Src Port	Dest IP	Dest Port
192.168.0.200	35777	12.62.100.200	6000

В заголовку пакета є 4 колонки, що нас цікавлять:

1. IP адреса вузла, що робить запит – Src IP;
2. Порт вузла – Src Port;
3. IP адреса STUN серверу – Dest IP;

4. Порт STUN серверу – Dest Port.

Вузол відправляє даний пакет на маршрутизатор. Той отримавши цей пакет, підміняє адресу відправника з локальної на зовнішню (табл. 2.3). Це робиться для того, що локальні IP адреси та порти діють лише всередині локальної мережі. І щоб вузол зміг отримати відповідь від STUN-серверу, йому треба вказати зовнішню адресу маршрутизатора. В даному прикладі, зовнішня адреса маршрутизатора – 10.50.200.5.

Таблиця 2.3

Заголовок пакету від вузла з підміною в адресі та порту

Src IP	Src Port	Dest IP	Dest Port
10.50.200.5	**888**	12.62.100.200	6000

Але для того, щоб вузол зміг потім отримати відповідь цього недостатньо. Маршрутизатор також вносить в таблицю NAT відповідність внутрішньої адреси вузла до його зовнішньої адреси (табл. 2.4).

Таблиця 2.4

Таблиця NAT маршрутизатора із записом про вузол

Internal IP	Internal Port	External IP	External Port
192.168.0.200	35777	10.50.200.5	888

Пакет із підміненою адресою відправляється на STUN-сервер. Той, отримавши його бачить, що він надійшов з вузла 10.50.200.5:888. Отже на цю адресу він і відправляє свою відповідь. На разі ми беремо до уваги лише заголовки пакетів. Адреса вузла 10.50.200.5:888 буде відправлена в тілі пакету, до якого ми повернемося пізніше. Таким чином STUN-сервер відправляє наступний пакет-відповідь, наведений у табл. 2.5.

Таблиця 2.5

Заголовок пакету від STUN-серверу

Src IP	Src Port	Dest IP	Dest Port
12.62.100.200	6000	10.50.200.5	888

Коли цей пакет приходить до маршрутизатора, він звертається до своєї таблиці NAT щоб встановити відповідність зовнішньої адреси вузла внутрішній. Якщо така адреса існує, то вона підміняється та локальну (табл. 2.6) та пакет відправляється до свого кінцевого адресату – вузла.

Таблиця 2.6

Заголовок пакету від STUN-серверу з підміненою локальною адресою

Src IP	Src Port	Dest IP	Dest Port
12.62.100.200	6000	**192.168.0.200**	**35777**

Таким чином, пакет доходить до вузла, в тілі якого знаходиться інформація про його місцезнаходження в глобальній мережі – адреса маршрутизатора 10.50.200.5 та відкритий порт 888. Тепер за цією адресою до вузла можуть звертатися інші користувачі мережі, оскільки в таблиці NAT було створено відповідність внутрішньої та зовнішньої адреси вузла.

TURN сервер – це покращений STUN. Він може виконувати функції STUN серверу, але має також і додаткову функцію – ретранслятора даних. Такий сервер використовується якщо, наприклад, використовується різновид NAT із захистом від «фальсифікацій». Наприклад, в таблиці NAT зберігається ще 2 параметри – адреса та порт віддаленого вузла. Пакет із зовнішньої мережі може дістатися відправника тільки якщо адреса та порт джерела співпадають. Таким чином таблиці буде зберігатися адреса та порт STUN сервера, а коли роутер отримає пакет від WebRTC вузла, його буде відкинуто, оскільки не співпадають адреси.

Саме в цьому випадку стає в нагоді TURN сервер, який і ретранслює необхідні дані.

2.2. Протокол передачі даних SCTP

Ще однією важливою особливістю технології WebRTC є протокол передачі даних транспортного рівня. Ця технологія використовує не звичні TCP або UDP, а розроблений ще в 2002 році протокол SCTP (Stream Control Transmission Protocol).

Такий вибір розробників WebRTC зумовлено тим особливою системою передачі даних, яку пропонує SCTP.

Наприклад, якби у WebRTC використовувався транспортний протокол TCP, то дані гарантовано передавалися б кінцевому вузлу та надходили саме в тій послідовності, що й були відправлені. Однак стає питання швидкості, тому що TCP може створювати значні затримки, які будуть лише накопичуватись. А оскільки основне призначення WebRTC – передача потокових медіа-даних для відео та аудіо конференцій, то їх проведення з протоколом TCP було б надзвичайно некомфортним для користувачів.

Якщо WebRTC передавав би дані через UDP, то вони надходили би до кінцевого вузла дуже швидко, але є велика вірогідність того, що вони втратяться або прийдуть не в тому порядку. Тобто неможливо контролювати цілісність даних, а тому на приймаючій стороні довелося б щось з цим робити.

Саме тому для передачі даних у WebRTC було обрано протокол SCTP, що дозволяє контролювати те як дані передаються, але в той же час не створює значної затримки при передачі потокових даних.

2.2.1. Безпечне встановлення з’єднання

Для того, щоб створити з’єднання (для протоколу SCTP використовується поняття «асоціація») між двома вузлами в протоколах TCP та SCTP, потрібно пройти процедуру підтвердження пакетів. Наприклад, в протоколі TCP такий механізм має назву «триетапне рукостискання» (рис. 2.2). Клієнт відправляє серверу пакет SYN, а у відповідь отримує SYN-ACK. Клієнт,

щоб підтвердити отримання пакету SYN-ACK відправляє на сервер пакет ACK. Після цього з’єднання вважається встановленим і сторони готові до передачі даних.

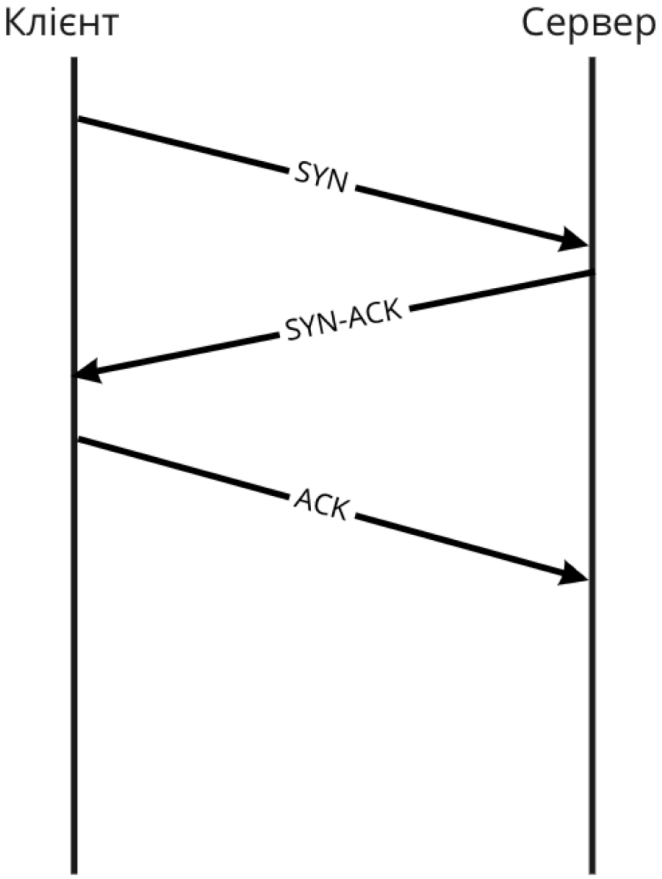

Рис. 2.2. Встановлення з’єднання в протоколі TCP

В такому механізмі встановлення з’єднання в протоколі TCP є потенційна вразливість. Якщо в пакеті SYN підмінити IP-адресу відправника на фальшиву, сервер відправить SYN-ACK на цю підмінену адресу. Небезпечно це тим, що порушник може згенерувати необмежену кількість пакетів та відправляти їх. Сервер витрачає певну кількість ресурсів на обробку кожного пакету. І якщо їх буде забагато, сервер може бути перенавантаженим та перестати обробляти нові запити. Такий вид атаки називається Denial of Service, або скорочено DoS.

В протоколі SCTP це було передбачено і встановлення асоціації відбувається за допомогою чотирьохетапного рукостискання. Асоціація

розпочинається з того, що клієнт відправляє на сервер пакет INIT. Сервер відповідає пакетом INIT-ACK, в який поміщає унікальний ідентифікатор сесії. Клієнт отримує цей пакет із ідентифікатором, та відправляє на сервер пакет COOKIE-ECHO, в якому міститься цей самий ключ. Таким чином, сервер точно може знати, що запит про асоціацію не було підроблено і він є реальним. Перевіривши ключ, сервер виділяє ресурси для нового для нового з'єднання і відправляє клієнту пакет COOKIE-ACK, що підтверджує успішне створення нової асоціації (рис. 2.3).

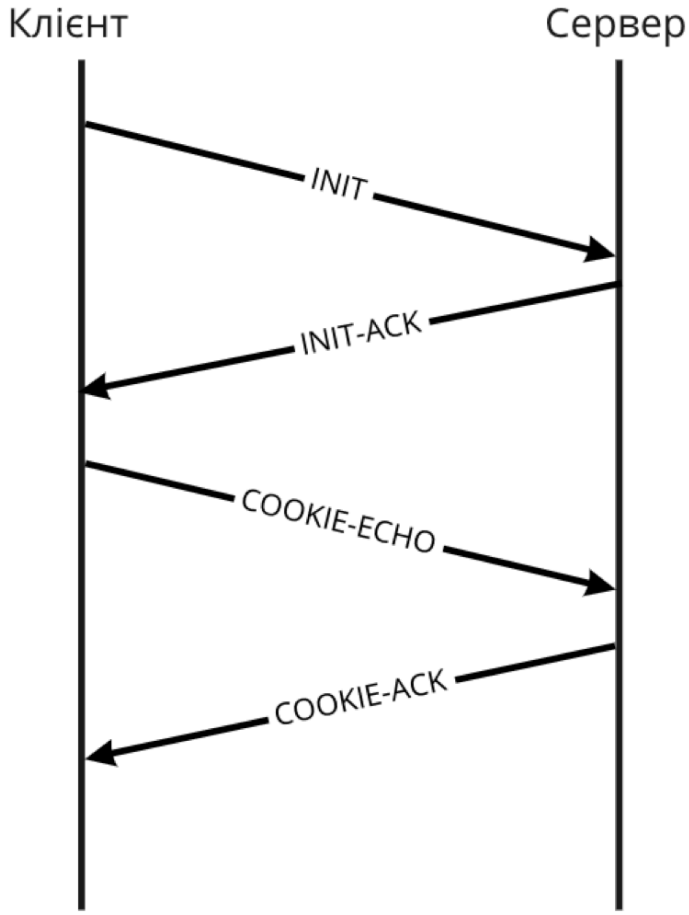

Рис. 2.3. Схема встановлення асоціації в протоколі SCTP

2.2.2. Поетапне завершення передачі даних

Завершення з'єднання також є важливим етапом комунікації між вузлами. В протоколі SCTP процес завершення асоціації також є відмінним від протоколу TCP.

В протоколі TCP завершення асоціації відбувається шляхом передачі пакету FIN з обох сторін з’єднання. Про успішне отримання такого пакету вони відповідають пакетом ACK (рис. 2.4).

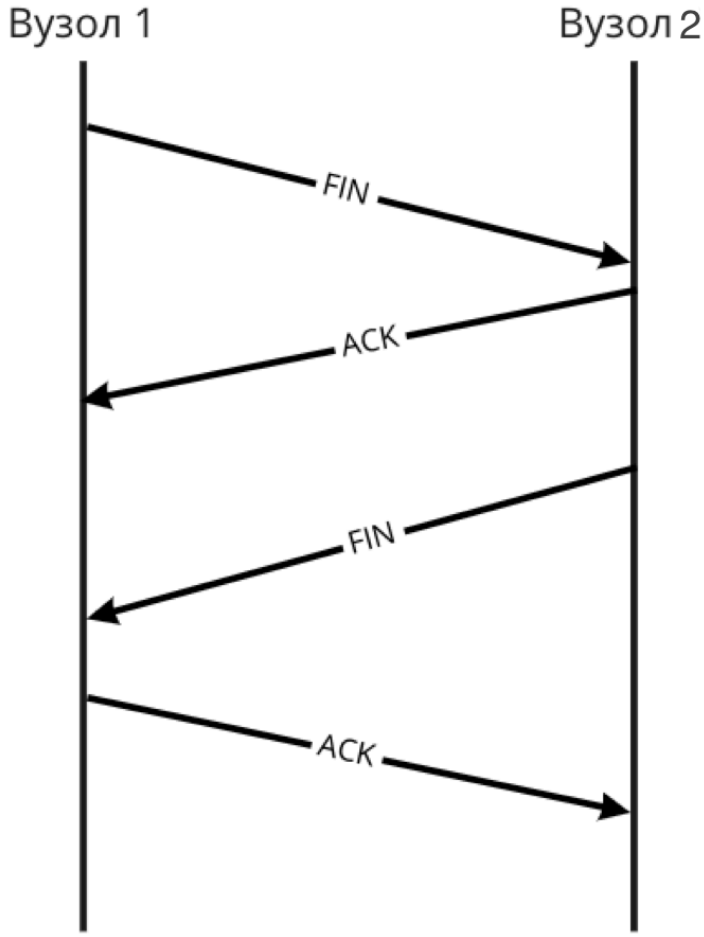

Рис. 2.4. Схема послідовності завершення з’єднання в протоколі TCP

В такій схемі процесу завершення з’єднання між двома вузлами є стан, коли один з вузлів вже завершив передачу даних, а інший ні. Це називається частковим закриттям з’єднання. Такий стан можна отримати, коли вузол завершив передавати свої дані, але при цьому може приймати дані від іншого вузла. І цей перехідний стан триває до тих пір, поки інший вузол також не завершить з’єднання.

Протокол SCTP має інакшу послідовність завершення асоціації (рис. 2.5) і часткове закриття з’єднання в ньому виключене. Щоб закрити асоціацію, будь-якому вузлу достатньо відправити пакет SHUTDOWN. При цьому будь-

яка передача даних має припинитися і тепер можна обмінюватись лише пакетами, що підтверджують прийняття отриманих раніше даних. На відповідь пакету SHUTDOWN інший вузол відправляє SHUTDOWN-ACK, сигналізуючи про підтвердження завершення асоціації. Після цього вузол-ініціатор завершення відправляє SHUTDOWN-COMPLETION, що означає повне завершення асоціації.

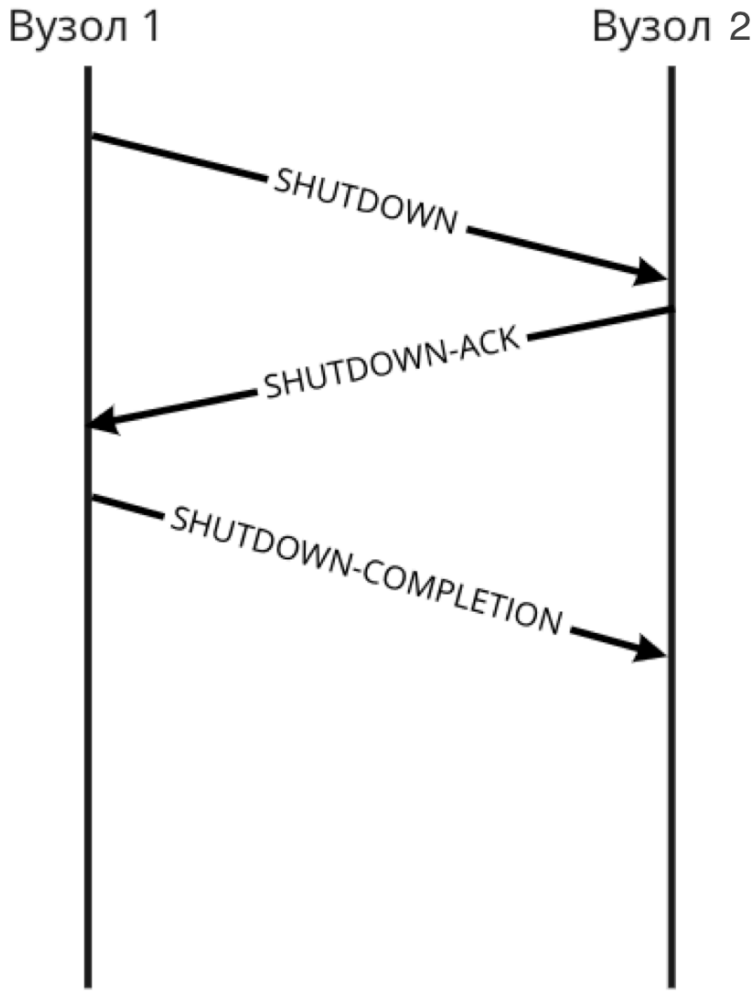

Рис. 2.5. Схема послідовності завершення з’єднання в протоколі SCTP

2.2.3. Потоки

Протокол SCTP також цікавий тим, що підтримує передачу даних в декілька потоків, на відміну від TCP. В TCP є лише керування послідовністю байт, тобто дані, що були відправлені в певному порядку, в тому ж порядку і будуть отримані кінцевим вузлом. Оскільки протокол IP може змінити

послідовність пакетів для цього вони поміщуються в буфер, та видаються, коли послідовність правильна. Це створює додаткову часову затримку. Ще більшу затримку створює те, що пакет може бути втрачено і тоді TCP відправить їх повторно.

SCTP працює одразу з цілими повідомленнями. До того ж, повідомлень може бути одночасно декілька і вони будуть передаватися різними потоками в рамках однієї SCTP-асоціації (рис. 2.6).

SCTP може передавати дані між вузлами одночасно в декілька потоків повідомлень (рис. 2.4). На відміну від TCP, SCTP обробляє цілі повідомлення, а не звичайні байти інформації.

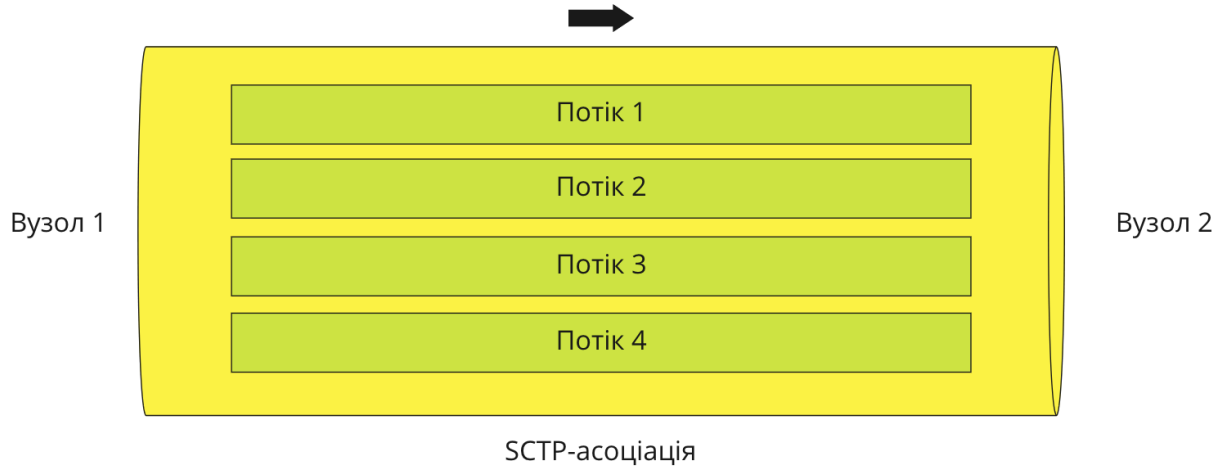

Рис. 2.6. Декілька потоків повідомлень в SCTP-асоціації

Таким чином SCTP може передавати дані в декілька потоків повідомлень. Наприклад, ми передаємо декілька файлів від одного вузла до іншого. Можна відкрити для кожного файлу декілька TCP-з'єднань, а можна передати всі файли в рамках однієї SCTP-асоціації.

2.2.4. Переваги та недоліки

Крім вищезгаданих особливостей протокол SCTP також має ряд переваг перед протоколами TCP або UDP:

1. Підтримка декількох мережевих інтерфейсів в рамках однієї асоціації (рис. 2.7). Наприклад, є два вузли, що мають підключення через Ethernet та безпровідне підключення через Wi-Fi (802.11), отже вони також мають і декілька IP-адрес. І SCTP-асоціація передбачає такий випадок, на відміну від TCP, де для кожного інтерфейсу довелося б мати окреме з'єднання;

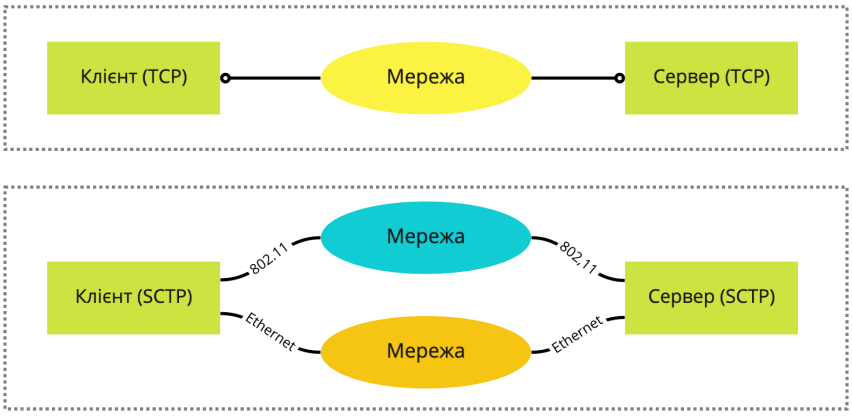

Рис. 2.7. Схема множинних інтерфейсів протоколу SCTP

2. Передача даних в декілька потоків в рамках однієї асоціації, що економить час;
3. Покращені механізми встановлення та розриву асоціації, а також валідації отриманих даних.

Недоліки SCTP:

1. Набагато більший обсяг службового трафіку в порівнянні з TCP або UDP. Це потребує більшої пропускної спроможності каналу.

2.3 WebRTC для передачі файлів

WebRTC найбільшого поширення набула для відео та звукових дзвінків. Але це лише частина можливостей, що дає ця технологія. Вона встановлює peer-to-peer з'єднання, що може використовуватись для будь-яких цілей, наприклад, передачі файлів.

2.3.1. Створення каналу передачі даних

WebRTC дозволяє створювати мережеві канали між вузлами як тільки з'єднання було встановлено. Ці канали можуть використовуватись для передачі будь-яких типів даних. Інтерфейс RTCDataChannel у JavaScript представляє ці канали.

Такий канал можна створити шляхом виклику методу createDataChannel у об'єкта peerConnection. Кожен канал повинен мати свою назву, яка задається на етапі створення цього каналу.

Для контролю за успішним створенням каналу потрібно впровадити метод onopen. Таким чином після успішного створення каналу спрацює код всередині функції.

Для прослуховування події отримання повідомлення від іншого вузла, потрібно впровадити метод onmessage.

Приклад створення каналу передачі даних:

```
const channel = peerConnection.createDataChannel('Test Channel');
channel.onopen = () => {
    // код після відкриття каналу
}
channel.onmessage = event => {
    // код після отримання повідомлення
}
```

Якщо одна сторона створює такий канал передачі даних, то інша має впровадити прослуховування події ondatachannel для того, щоб далі можна було взаємодіяти з цим каналом:

```
peerConnection.onDataChannel = event => {
    const { channel } = event;
    // код після встановлення каналу для передачі даних
}
```

Відправляти повідомлення можна методом send об’єкту channel:

```
channel.send(‘Test message’);
```

2.3.2. Передача файлу

Для передачі файлу потрібно спочатку його зчитати. Оскільки WebRTC функціонує всередині браузеру, то можна використати елемент input в мові HTML:

```
<input type=”file” id=”select-file-input” />
```

Для відстеження події вибору файлу потрібно впровадити метод onchange для DOM-об’єкту в JavaScript:

```
document.getElementById(‘select-file-input’).onchange(event => {
    const  file = event.target.files[0];  // дані про обраний файл
});
```

Після встановлення з’єднання між вузлами та вибору файлу на стороні передавача , потрібно отримати потік даних до даного файлу та відправити його:

```
const arrayBuffer  = file.arrayBuffer();
channel.send(arrayBuffer);
```

Для файлів менше 256 КБ така реалізація підходить. Однак якщо відправляти великі файли таким чином, то виникне помилка завеликого буферу. Це відбувається через імплементацію браузерами протоколу SCTP, про який ішла мова у підрозділі 2.2. Для того, щоб передавати файли більше ніж 256 КБ, їх потрібно розбивати на шматочки (chunks). Для індикації завершення передачі всього файлу відправляється відповідне повідомлення:

```
const CHUNK_SIZE = 65535; // 64 КБ
const arrayBuffer  = file.arrayBuffer();
for (let i = 0; i < arrayBuffer.byteLength; i += CHUNK_SIZE) {
    channel.send(arrayBuffer.slice(i, i + CHUNK_SIZE));
}
channel.send(‘EOF’)
```

На стороні приймача, всі шматочки файлу зберігаються. В кінці передачі файлу створюється об’єкт типу Blob, у який записуються всі збережені записуються дані. Потім цей об’єкт можна зберегти як файл до файлової системи користувача:

```
const recievedBuffers = [];
channel.onmessage = event => {
    const { data } = event; // отримуємо посилання на потік
    if (data !== ‘EOF’) {
        recievedBuffers.push(data)
    } else {
        const blob = new Blob([recievedBuffers]);
        // далі цей об’єкт можна завантажити
    }
}
```

2.4. Висновки до другого розділу

WebRTC – це технологія, що дозволяє веб-додаткам та сайтам захоплювати та вибірково передавати аудіо та/або відео медіа-потоки, а також обмінюватись довільними даними між браузерами без обов’язкового використання посередників.

Набір стандартів, що включає в себе технологія WebRTC дозволяє обмінюватись даними та проводити пірингові телеконференції без необхідності користувачу встановлювати плагіни або будь-яке стороннє програмне забезпечення.

І хоча основне призначення WebRTC – відео та аудіоконференції, ця технологія може використовуватись для передачі файлів будь-яких розмірів між вузлами.

Для роботи WebRTC потрібен STUN-сервер, що дасть змогу користувачам з різних мереж дізнатися про свою адресу в глобальній мережі.

Також запит на STUN-сервер створить запис у таблиці NAT маршрутизатора, таким чином інші вузли зможуть звертатися до вузла напряму.

Для передачі файлів через WebRTC необхідно встановити з’єднання з сокет-сервером, що буде транслювати повідомлення від одного вузла до іншого, згенерувати ICE Candidate для кожного з користувачів та відправити його одне одному. Після цього відкрити потік даних та правильно налаштувати розмір буферу даних через який передається файл. При завершені передачі файлу зберегти його на стороні клієнта.

РОЗДІЛ 3
ПРАКТИЧНЕ ВТІЛЕННЯ ПЕРЕДАЧІ ФАЙЛІВ ЗА ДОПОМОГОЮ WEBRTC

3.1. Принцип роботи додатку

В результаті дослідження в розділах 1 та 2 було вирішено використовувати клієнт-серверну архітектуру для функціонування самого додатку, а також архітектуру peer-to-peer для передачі файлу безпосередньо між двома вузлами. Для клієнтської частини було прийняте рішення використовувати в якості основної мови для розробки багатоплатформового додатку скриптову мову JavaScript.

Незалежно від архітектури та основних методів функціонування та розробки, потрібно було продумати принцип роботи додатку і його інтерфейс. Адже саме з цими двома речами буде взаємодіяти кінцевий користувач, що значно вплине на сприйняття додатку в цілому.

Додаток має кілька ключових концепцій, які використовуються для розмежування доступу користувачів та оптимізації досвіду користування додатком.

3.1.1. Користувачі

Основна ідея додатку в тому, що один користувач може відправити файл з одного пристрою на інший бездротовим шляхом. Кожен пристрій, на якому відкрито додаток заноситься в систему як окремий користувач. Таким чином основна взаємодія відбувається за принципом користувач – користувач.

Коли на пристрої відкривається додаток, пристрою (або для кожної вкладки браузеру, якщо використовується веб-версія додатку) автоматично присвоюється унікальний ідентифікатор, та випадковим шляхом генерується назва. Присвоєні дані діють лише під час однієї сесії користування. Тобто коли

додаток закривається, ці дані видаляються, а при повторному відкритті додатку генеруються заново.

Користувачі відображаються в списку, з їхніми назвами та картинками (назви пристроїв генеруються на базі назв тварин, тому кожній назві існує відповідна картинка) (рис. 3.1).

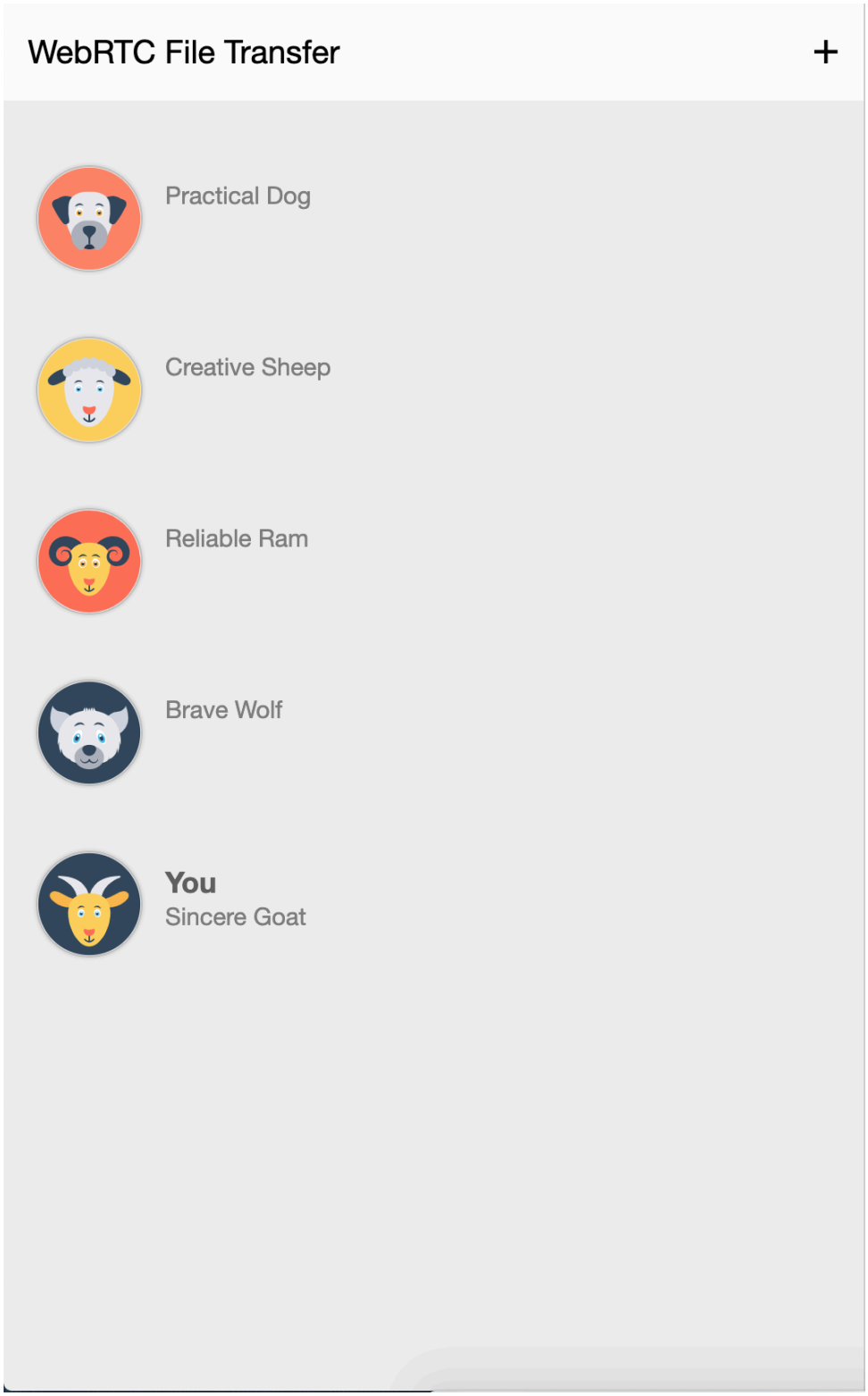

Рис.3.1. Приклад згенерованих назв пристроїв

Така процедура генерації назви та унікального ідентифікатора пристрою була зроблена для того, щоб виключити процес реєстрації для кожного окремого пристрою. Немає необхідності в тому, щоб витрачати час користувача на введення його даних для кожного окремого пристрою, оскільки завдяки згенерованим унікальним назвам, можна точно ідентифікувати де який пристрій.

3.1.2. Кімнати

Концепція користувачів дозволяє відокремлювати один пристрій від іншого за допомогою назви та унікального ідентифікатора. Однак потрібен ще спосіб, щоб відокремити одну групу користувачів від іншої. Припустимо, що додатком одночасно користується 50 різних користувачів з 2 пристроями у кожного. Таким чином в додатку буде відображатись 100 пристроїв одночасно і користувачам буде важко знайти свої пристрої, щоб передати на них файл.

Саме для цього в додатку існує концепція кімнат. Вони потрібні для розмежування пристроїв одного користувача від інших. Коли користувач заходить в додаток, він одразу ж додається в певну кімнату. Працюють кімнати за двома принципами: на основі локальної мережі, та за посиланням.

На основі локальної мережі принцип роботи кімнат наступний. Коли користувач заходить в додаток, він одразу ж додається до згенерованої кімнати. Якщо інший користувач, що також зайшов в додаток знаходиться в одній локальній мережі з попереднім користувачем, його буде додано в ту ж саму кімнату. Таким спосіб якнайкраще підходить для повсякденного використання додатку, коли користувач має поруч обидва пристрої та передає дані з одного на інший. Це дуже швидко та легко, не витрачає час на неважливі користувачеві абстракції у вигляді кімнат, які за таким сценарієм йому можуть і не знадобитися.

Однак спосіб на основі локальної мережі не підходить для компаній, в чиїх офісах можуть бути десятки чи навіть сотні пристроїв, що знаходяться в одній локальній мережі. В таких умовах потрібні окремі кімнати, які будуть створені для кожного конкретного користувача. Саме для цього випадку були створені кімнати за посиланням.

Для того, щоб створити власну кімнату користувач має на одному пристрої натиснути на кнопку створення кімнати. Додаток автоматично згенерує нову кімнату, додасть в неї поточний пристрій та запропонує скопіювати посилання або відсканувати QR-код (рис. 3.2).

Рис 3.2. Приклад QR-коду та посилання для входу в кімнату

Все що потрібно користувачеві – відправити посилання на щойно створену кімнату на інший пристрій, або відсканувати з нього запропонований QR-код (звісно, якщо на пристрої є камера та функція розпізнавання QR-кодів) та перейти за цим посиланням і приєднатися до цієї кімнати.

3.1.3. Повідомлення

Концепція користувачів слугує для розділення кожного окремого пристрою. Концепція кімнат слугує для розмежування користувачів по локальній мережі або за посиланням. Але основна ідея додатку – саме передача файлів і потрібна ще одна концепція для розмежування саме процесу передачі файлів. Саме для цього було розроблено систему повідомлень.

При передачі файлу, користувач має обрати іншого користувача, а потім обрати сам файл для передачі. Після цих дій, файл не буде одразу ж відправлено користувачеві з міркувань безпеки. Для того, щоб відправка файлу відбулася, з приймаючої сторони потрібно отримати схвалення на прийняття даних. До того ж користувачі на етапі вибору файлу не знають про місце знаходження одне одного в мережі, тому їм потрібно отримати цю інформацію.

Отже коли було обрано користувача та файл для передачі, відправник генерує повідомлення про наміри передати дані. Це повідомлення відправляється обраному користувачеві та містить в собі дані про відправника та інформацію про файл – його розмір та назву. Приймаюча сторона отримує це повідомлення та має дві опції: прийняти та відхилити (рис. 3.3).

Якщо приймаюча сторона погоджується прийняти файл, їй спочатку потрібно вказати відправнику своє місце знаходження в мережі. Таким чином приймаюча сторона генерує свій ICE Candidate, в якому вказано IP адреса та порт. Ці дані надсилаються відправнику який, отримавши їх, розпочинає передачу файлу на вказану IP адресу та порт.

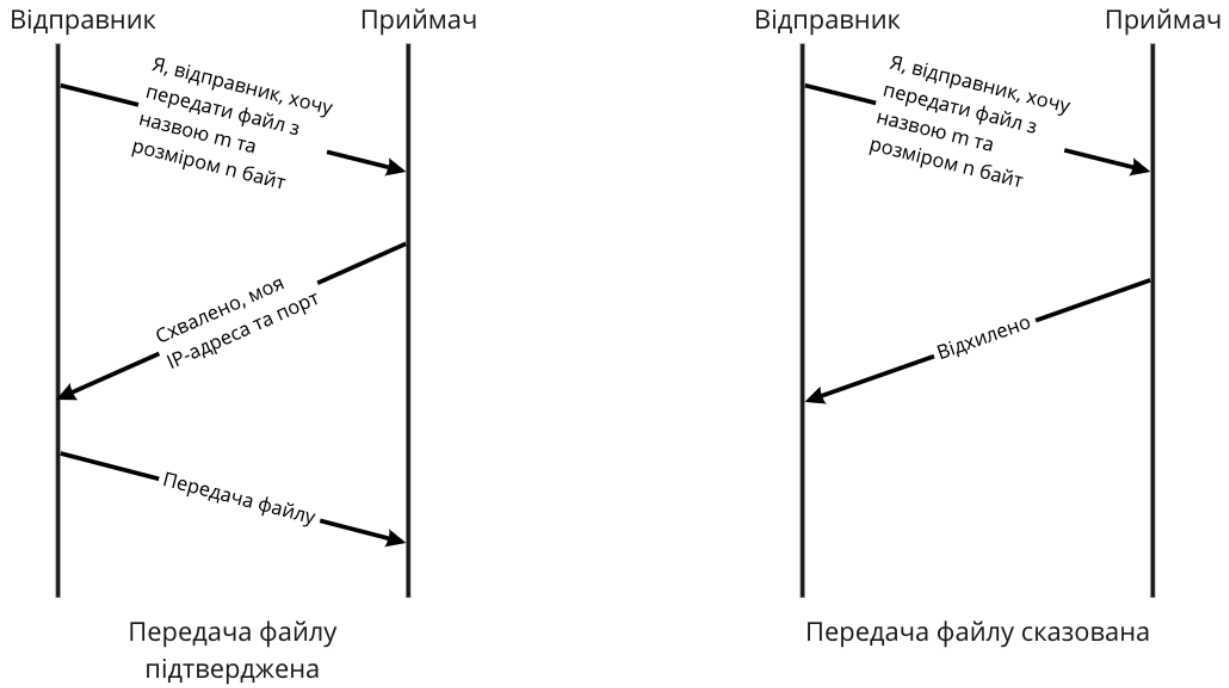

Рис. 3.3. Схема обміну повідомленнями

Якщо приймаюча сторона відхиляє прийняття файлу, відправнику у відповідь надсилається відповідне повідомлення та процес передачі файлу не розпочинається.

3.1.4 Графічний інтерфейс

Графічний інтерфейс – найголовніша частина додатку, з якою, власне, взаємодіє користувач. Він впливає на загальний досвід користування додатком, тож має бути розроблено таким чином, щоб не містити в собі зайвих графічних елементів, а весь існуючий функціонал був зрозумілим та доступним.

На основному екрані додатку, за замочуванням для нього відкривається нова кімната. В ній він бачить тільки тих, хто знаходиться з ним в одній локальній мережі. Також на основному екрані доступна назва додатку, а також кнопка для того, щоб створити нову кімнату (рис. 3.4).

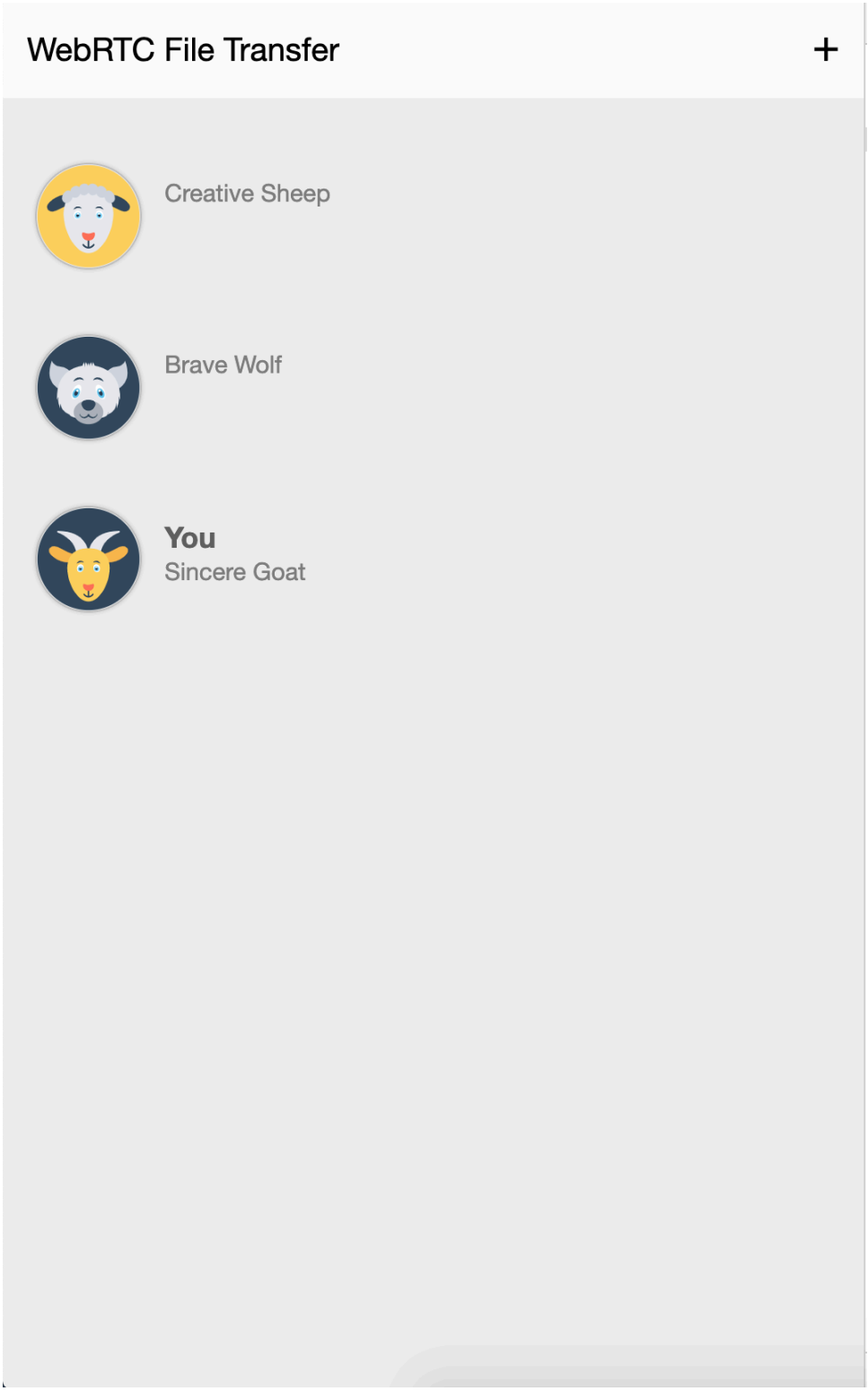

Рис. 3.4. Інтерфейс додатку

По мірі того, як в кімнату заходять нові користувачі, вони відображаються у списку. Для того, щоб передати обраному користувачу файл, потрібно на нього натиснути. Відкриється системне діалогове вікно, яке дозволить обрати файл для передачі (рис. 3.5).

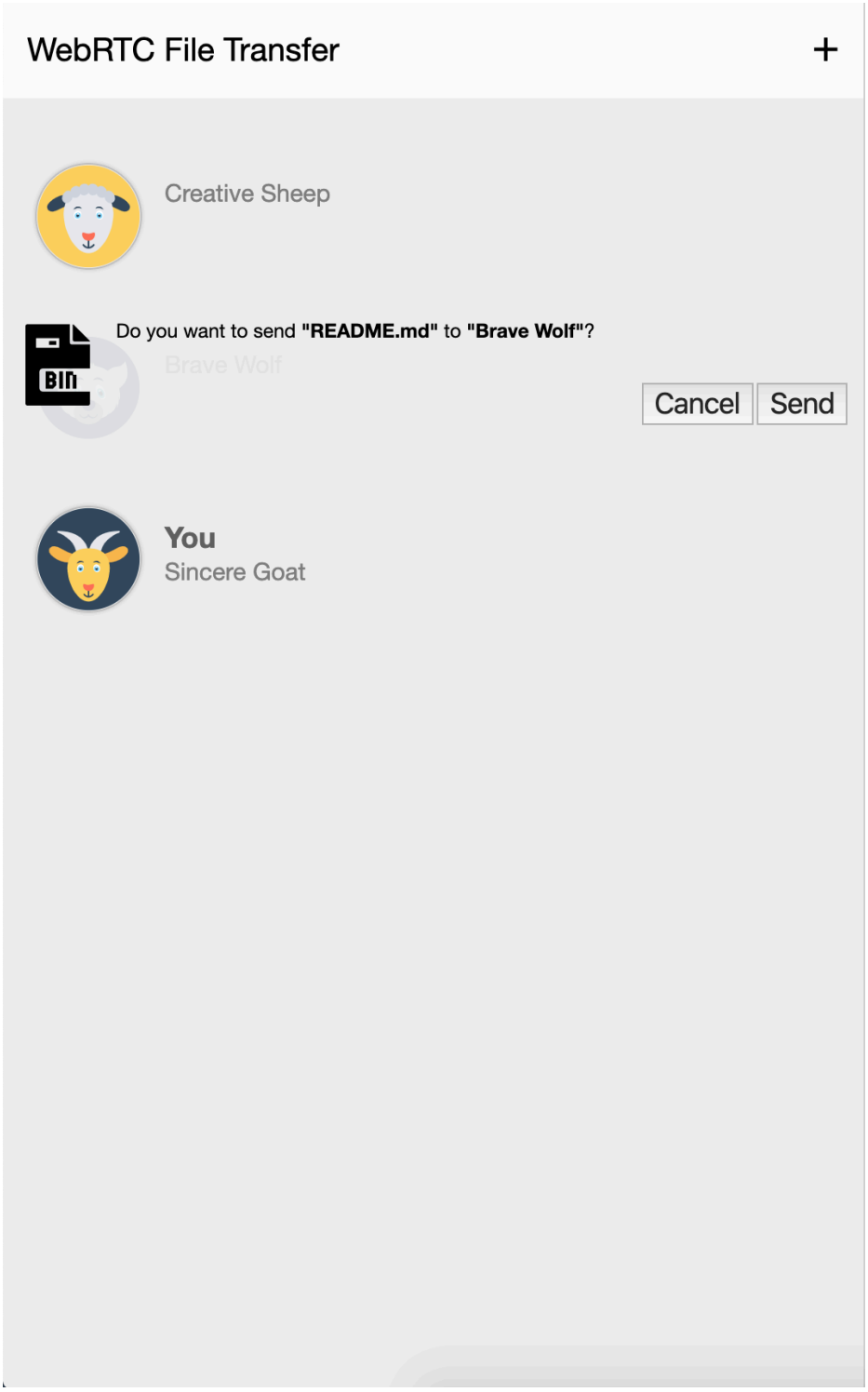

Рис. 3.5. Діалогове вікно для вибору файлу для відправки

Після того, як файл було обрано, іншому користувачу відправляється повідомлення про передачу файлу. У нього з'являється діалогове вікно з пропозицією прийняти файл (рис. 3.6). Якщо користувач приймає файл, він завантажується в буфер. Після завантаження, його можна зберегти у файловій системі комп'ютера.

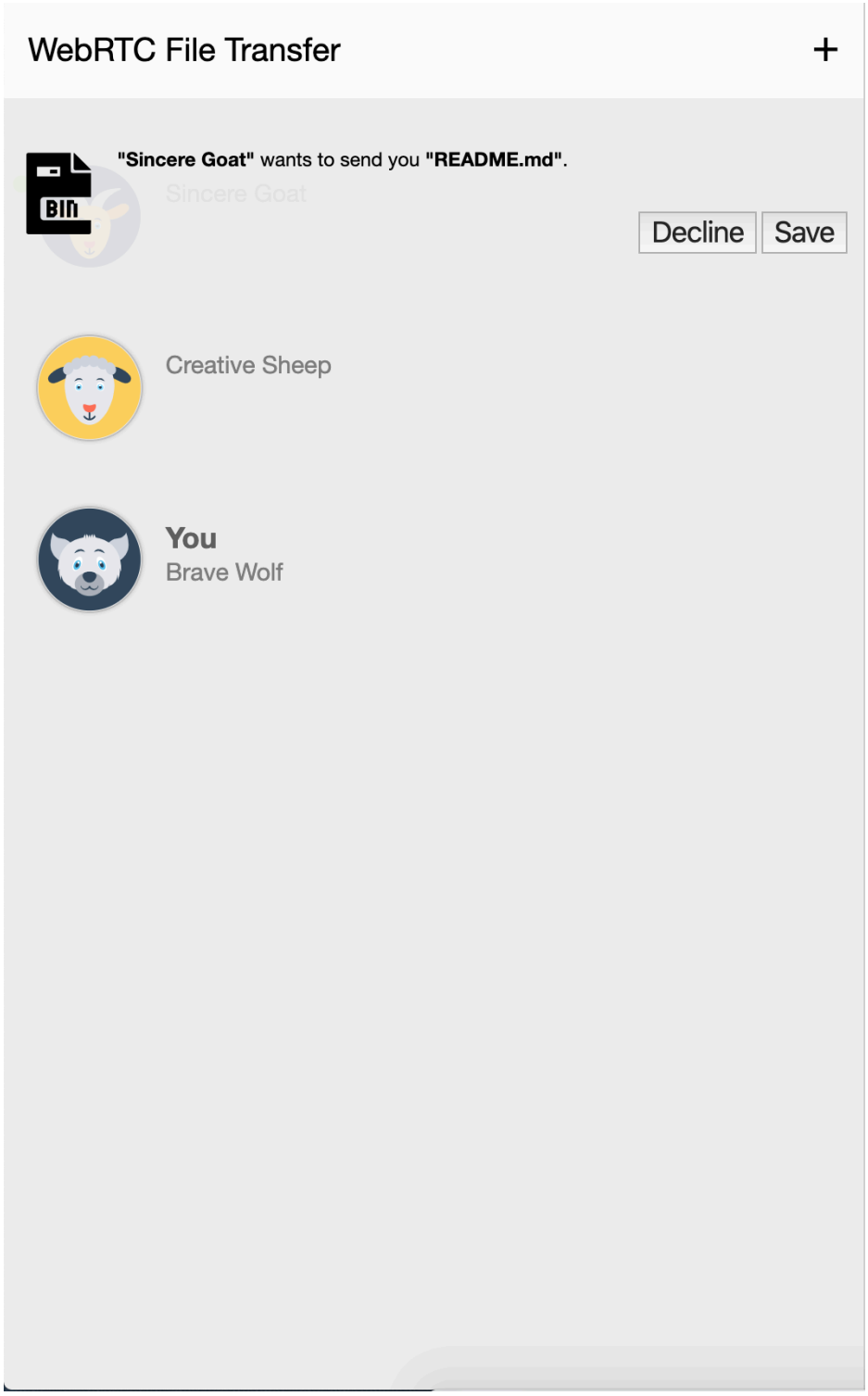

Рис 3.6. Діалогове вікно для прийняття файлу

Створити нову кімнату можна натиснувши на відповідну кнопку в верхньому правому куті. Коли користувач створює кімнату, відкривається модальне вікно, в якому можна скопіювати посилання на цю кімнату, або дати

іншому користувачеві відсканувати QR-код, що також містить посилання на цю кімнату (рис. 3.7).

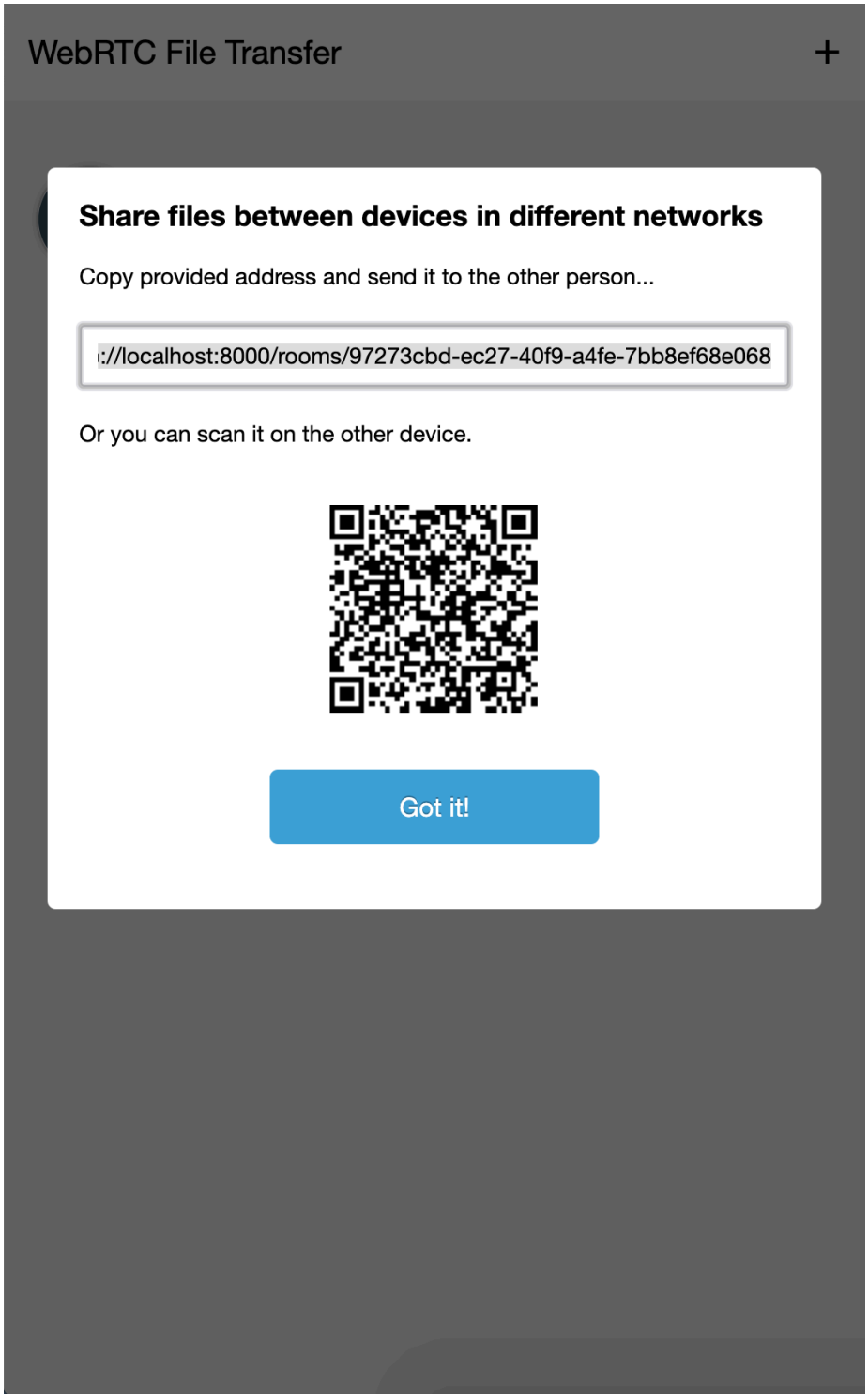

Рис. 3.7. Модальне вікно після створення кімнати

3.2. Архітектура додатку

Додаток має клієнт-серверну архітектуру. Тобто ділиться на дві незалежні частини: клієнтську та серверну. Клієнтська відповідає за графічний інтерфейс, з яким взаємодіє користувач. Також на цій стороні знаходиться WebRTC, де генеруються ICE Candidate. На серверній стороні знаходиться сокет-сервер, та база даних поточних користувачів, кімнат та повідомлень.

3.2.1. Серверна сторона

В якості серверного рішення було обрано Firebase від Google. Це безкоштовне рішення, що дозволяє запустити базу даних та сокет-сервер та взаємодіяти з ними в режимі реального часу. При цьому не потрібно вирішувати жодних інфраструктурних задач, на кшталт підключення хостингу або виділення фізичних ресурсів.

В безкоштовний план входять 100 одночасних підключень до бази даних, 1 ГБ даних та ліміт на 10ГБ даних на місяць на завантаження. Оскільки вся взаємодія між клієнтами відбувається peer-to-peer, то навантаження на базу буде незначним. Єдиним вузьким місцем є лімітована кількість одночасних підключень, але це можна виправити перейшовши на платний тариф.

Ще однією перевагою такого рішення є відсутність необхідності писати серверний код. Це дуже економить час розробки, та спрощує її в цілому. Вся взаємодія клієнта з сервером відбувається через веб-сокети. Якщо потрібно занести в базу даних будь-яке повідомлення, клієнт відправляє серверу повідомлення, той заносить необхідні дані та розсилає все, що необхідно іншим клієнтам (рис. 3.8).

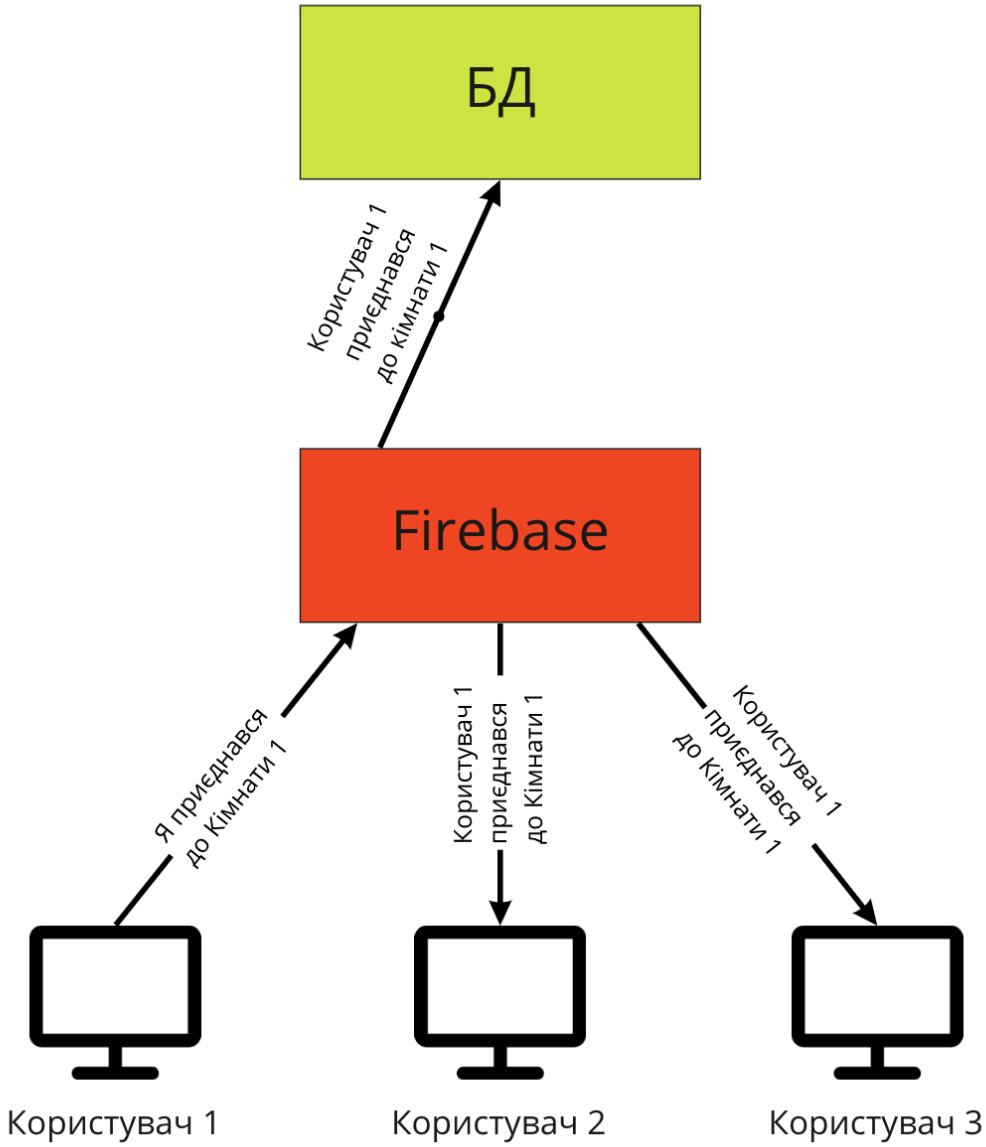

Рис. 3.8. Принцип роботи Firebase

База даних зберігає дані у форматі JSON. Цей формат було обрано, оскільки з ним найлегше працювати на стороні клієнта. Структура бази даних має наведена на рис. 3.9.

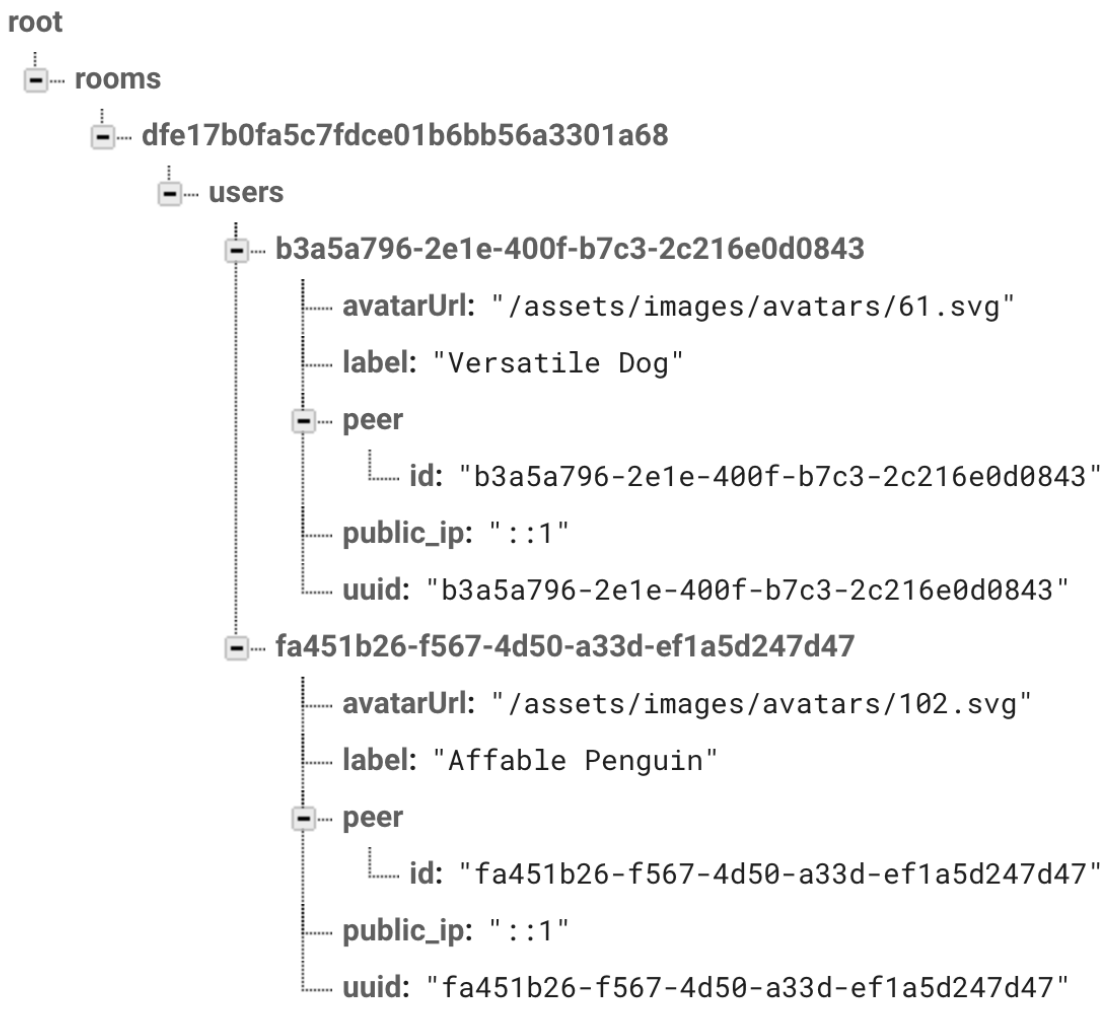

Рис. 3.9. Структура бази даних

База даних зберігає інформацію про відкриті кімнати та про активних користувачів цієї кімнати. Коли користувач залишає кімнату, його дані видаляються з бази даних. Коли останній користувач залишає кімнату, дані про кімнату видаляються з БД.

У Firebase є можливість прописувати правила для роботи з базою даних, щоб розмежувати доступ до неї користувачам. Правила описані у форматі JSON та мають певну структуру.

На рис. 3.10 наведені наступні правила:

1. Користувач може бачити інших користувачів в кімнаті тільки якщо він є в ній сам;
2. Користувач може модифікувати тільки свою інформацію;

3. Користувач обов’язково повинен мати атрибути uuid, public_ip та peer;
4. Користувач може відправляти повідомлення про передачу файлів будь-якому іншому користувачеві в кімнаті;
5. Користувач може читати тільки ті повідомлення, які було адресовано йому.

```
{
    "rules": {
        "rooms": {
            "$roomid": {
                // You can see people in the room only if you are in it as well
                ".read": "auth != null && data.child('users').hasChild(auth.id)",
                "users": {
                    "$userid": {
                        // You can modify only your own info
                        ".write": "auth != null && $userid == auth.id",

                        // Ensure that all required attributes are there
                        ".validate": "newData.hasChildren(['uuid', 'public_ip', 'peer']) && newData.child('peer').hasChildren(['id'])"

                    }
                },
                "messages": {
                    // You can send message to anybody in the room
                    ".write": "auth != null",
                    "$userid": {
                        // You can read only messages sent to you
                        ".read": "auth != null && $userid == auth.id"
                    }
                }
            }
        }
    }
}
```

Рис. 3.10. Правила доступу до БД

3.2.2 Клієнтська сторона

Клієнтська сторона базується на JavaScript-фреймворку Ember. Використання фреймворку значно спрощує контроль над навігацією та шаблонізацію елементів користувальницького інтерфейсу.

Скриптова мова JavaScript була обрана як найбільш універсальне та поширене рішення для багатоплатформових додатків. Готовий код може бути використано як у браузері, так і зібрано для Android, iOS, MacOS, Windows та Linux. Для десктопних додатків можна використати фреймворк Electron, який являє собою браузер Chrome, але розширений функціоналом для взаємодії з

операційною системою. Для смартфонів можна розробити нативну оболонку, всередині якої в елементі WebView буде запущено сам додаток.

Середовище виконання коду – програмна платформа NodeJS. Саме в даному середовищі відбувається збірка коду, та видача його клієнту.

Стилі для HTML-шаблонів написані за допомогою модуля SASS. Це розширений функціонально та синтаксично модуль, який збирається в CSS стилі. SASS значно спрощує написання стилів для компонентів, а також дозволяє використовувати змінні та міксини.

Посилання на Firebase та ключі зберігаються у файлі .env.

В якості STUN серверу використовується безкоштовний STUN сервер від Google. TURN сервер в даному додатку не використовується, оскільки немає необхідності транслювати контент.

3.3. Замір швидкості передачі даних

Щоб оцінити ефективність передачі файлів між пристроями потрібно заміряти швидкість їх передачі. Це комплексне завдання, шо поділене на декілька етапів.

На передачу даних значний вплив має мережа, в якій знаходяться пристрої. Передача даних може проходити при різних обставинах:

1. Коли обидва пристрої знаходяться в одній локальній мережі;
2. Коли обидва пристрої знаходяться в різних мережах та підключені до швидкісних мереж (100Мб/с);
3. Коли один з пристроїв підключено до мобільної мережі 4G.

Також важливе значення на передачу може мати розмір самого файлу. В даному експерименті буде досліджено три типи файлів з різними розмірами:

1. Медіа-файл – відео формату mp4 (1.5ГБ та 170МБ);
2. Медіа-файл – зображення формату jpeg 37МБ та 3.6МБ);
3. Текстовий документ формату docx 1.7МБ та 0.15КБ).

Всі файли будуть передаватися по 5 разів від пристрою-відправника до пристрою-приймача в кожній з зазначених типів мереж. При цьому фіксується час передачі файлу. Відлік часу розпочинається коли приймач натискає кнопку прийняти файл і рахується до моменту завершення передачі файлу. В таблицю заноситься середній час, що було отримано в результаті кожного з п’яти дослідів для кожного окремого типу файлів.

На основі отриманого часу та відомого розміру файлу розраховується середня швидкість передачі даних (в Мб/с) для кожного з типів файлів.

3.3.1. Швидкість в локальній мережі

Для тестування швидкості в локальній мережі використовувалось наступне обладнання:

- Пристрій-приймач: MacBook Pro A1398;
- Пристрій-відправник: HP 250 G6;
- Роутер: TP-Link TL-WR841N.

В даному тесті швидкість передачі даних в локальній мережі обмежена лише пропускною здатністю роутера та мережевих карт пристрою-приймача. Всі пристрої, що було використано для даного тесту підтримують стандарт передачі даних для бездротових мереж 802.11n. Цей стандарт здатен забезпечити швидкість передачі даних до 300Мб/с.

З результатів тесту (табл. 3.1) можна засвідчити, що тип файлу та його розмір не впливають на швидкість його передачі в локальній мережі. Середня швидкість складає 95 Мб/с. Швидкість передачі файлів менше 1 МБ відбувається майже миттєво, тож в замірі швидкості присутня значна похибка. Але дані з більшими файлами є більш точними та дозволяють робити висновки про загальну швидкість передачі.

Таблиця 3.1

Результати швидкості передачі файлів в локальній мережі

Тип файлу	Розмір, МБ	Час передачі (хв:сс:мс)	Швидкість передачі (Мб/с)
Медіа-файл (відео)	1570	02:06:17	94.8
Медіа-файл (відео)	117	00:09:90	95.1
Медіа-файл (зображення)	37	00:03:13	94.6
Медіа-файл (зображення)	3.6	00:00:30	96.0
Текстовий документ	1.7	00:00:14	97.1
Текстовий документ	0.15	00:00:01	120

3.3.2. Швидкість в глобальній мережі

В даному тесті обидва пристрої підключені до мережі різних провайдерів, що забезпечують швидкість до 100Мб/с. Перед передачею даних було зроблено замір швидкості за допомогою Speedtest для приймача та відправника. За результатами п’яти замірів середня швидкість передачі даних для відправника склала 49.7 Мб/с на завантаження та 42.27 Мб/с на віддачу, а для приймача 48.1 Мб/с на завантаження та 40.96 Мб/с на віддачу

Для тестування використовувалось наступне обладнання:

- Пристрій-приймач: MacBook Pro A1398;
- Пристрій-відправник: HP 250 G6;
- Роутер на стороні приймача: TP-Link TL-WR841N;
- Роутер на стороні відправника: TP-Link TL-WR841N.

Даний тест підтвердив відсутність впливу типу файлу та його розміру на швидкість передачі даних в глобальній мережі. Швидкість передачі даних була достатньою для того, щоб передати відео середнього розміру (117 МБ) менше ніж за хвилину. Усі результати тесту наведені в табл. 3.2.

Таблиця 3.2

Результати швидкості передачі файлів в глобальній мережі

Тип файлу	Розмір, МБ	Час передачі (хв:сс:мс)	Швидкість передачі (Мб/с)
Медіа-файл (відео)	1570	05:55:80	35.3
Медіа-файл (відео)	117	00:28:45	32.9
Медіа-файл (зображення)	37	00:08:48	34.7
Медіа-файл (зображення)	3.6	00:00:82	35.1
Текстовий документ	1.7	00:00:39	34.5
Текстовий документ	0.15	00:00:03	35.2

3.3.3. Швидкість в мобільній мережі

В даному тесті один з пристроїв було підключено до мобільної мережі, що передає дані за стандартом 4G. Перед передачею даних було зроблено замір швидкості за допомогою Speedtest. За результатами п’яти замірів середня швидкість мережі 4G склала 14 Мб/с на завантаження та 8.7 Мб/с на віддачу.

Для тестування використовувалось наступне обладнання:

- Пристрій-приймач: MacBook Pro A1398 (підключено до мережі 100Мб/с);
- Пристрій-відправник: iPhone X (підключено до мобільної мережі 4G).

Тест, коли один з пристроїв підключено до мобільної мережі також не виявив різниці в швидкості для різних типів та розмірів файлів. В цьому тесті не було відправлено великий медіа-файл, як в попередніх, оскільки швидкість самої мобільної мережі була доволі низька. Очевидно, що в реальних умовах вона не здатна забезпечити комфортну швидкість для відправки великих файлів, тому саме цей випадок було виключено з тесту. Усі результати тесту доступні в табл. 3.3.

Таблиця 3.3

Результати швидкості передачі файлів в мобільній мережі

Тип файлу	Розмір, МБ	Час передачі (хв:сс:мс)	Швидкість передачі (Мб/с)
Медіа-файл (відео)	1570	-	-
Медіа-файл (відео)	117	02:30:94	5.9
Медіа-файл (зображення)	37	00:47:73	6.2
Медіа-файл (зображення)	3.6	00:04:64	6.2
Текстовий документ	1.7	00:02:38	5.7
Текстовий документ	0.15	00:00:22	5.5

3.4. Висновки до третього розділу

Метою даного розділу було розробити додаток для передачі файлів між пристроями з різними операційними системами та протестувати його ефективність, швидкість передачі даних за допомогою технології WebRTC та загальну зручність користування додатком.

Для полегшення взаємодії користувача з додатком було розроблено декілька концепцій: користувач (кожен окремий пристрій є окремим користувачем), кімната (для розмежування користувачів одне від одного на основі локальної мережі чи за посиланням) та повідомлення (відправляються, коли користувачі мають намір передати файл). Також для додатку було розроблено графічний інтерфейс.

Замір швидкості передачі даних показав, що ефективність передачі даних через WebRTC є достатньо високою. В локальній мережі, максимальна швидкість якої становить 300Мб/с середня швидкість передачі даних була 95Мб/с. Коли один з пристроїв було підключено до мобільної мережі 4G з максимальною швидкістю 8.7Мб/с, передача даних відбувалась на 6Мб/с.

Ці дані були отримані шляхом проведення кількох тестів з різними файлами В результаті також було виявлено, що ні тип файлу, що передається, ні його розмір не впливають на швидкість його передачі між користувачами.

Ці результати доводять те, що через локальну мережу система може швидко та комфортно для користувача передавати навіть великі файли. Через мобільну мережу, відправка зображень та документів середнього розміру також відбудеться в межах 1 хвилини.

РОЗДІЛ 4
ЕКОНОМІКА

4.1. Визначення трудомісткості розробки програмного забезпечення

Одним з головних етапів при розробці програмного забезпечення є визначення трудомісткості та розрахунок витрат на створення програмного продукту. В даному розділі буде розраховано витрати на розробку додатку з для передачі файлів між пристроями.

Початкові дані:

- годинна заробітна плата програміста, грн / год — 200;
- вартість машино-години ЕОМ, грн / год — 40.

Нормування роботи в процесі створення ПЗ істотно ускладняється в силу творчого характеру роботи програміста, тому трудомісткість розробки ПЗ може бути розрахована на основі системних моделей з різною точністю оцінки.

Трудомісткість розраховується за формулою:

$$t = t_u + t_a + t_n + t_{\text{відл}} + t_{\text{д}},$$

де t_o – витрати праці на підготовку і опис поставленого завдання (50 год.);

t_u – витрати праці на дослідження алгоритму вирішення задач;

t_a – витрати праці на розробку блок-схем алгоритму;

t_n – витрати праці на програмування за готовим блок-схемою;

$t_{відл}$ – витрати праці на відладку програм на ПЕОМ;

$t_д$ -– витрати праці на підготовку документації.

Сукупні витрати праці визначаються виходячи з умовного числа операторів у розроблюваному ПЗ. Розрахунок очікуваних витрат праці:

$$Q = q * C * (1 + p), \text{ людино-годин},$$

де q – передбачуване число операторів;

C – коефіцієнт складності програми;

p – коефіцієнт корекції програми в ході її розробки.

В нашому випадку, *q* буде дорівнювати 1050, *C* – 1,3, *p* – 0,1.

$$Q = 1050 * 1.3 * (1 + 0.1) = 1500, \text{людино-годин}$$

Витрати праці на вивчення опису завдання визначаються з урахуванням уточнення опису і кваліфікації програміста за формулою:

$$t_u = \frac{QB}{(75 \dots 85)K}$$

де *B* – коефіцієнт збільшення витрат праці (внаслідок неповного опису завдання, *B* = 1,2…1,5);

K – Коефіцієнт кваліфікації програміста, який визначається в залежності від стажу роботи за фахом (якщо стаж роботи менший 2 років, то *K* = 1,2).

Витрати праці на дослідження алгоритму рішення задачі:

$$t_u = 1500 * 1.3 \,/\, (80 * 1.2) \; = 20.3, \text{людино-години}$$

Витрати на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20 \dots 25)K}$$

$$t_a = 1500 \,/\, (23 * 1.2) \; = 78.2, \text{людино-години}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20 \dots 25)K}$$

$$t_n = 1500 \,/\, (23 * 1.2) \; = 78.2, \text{людино-години}$$

Витрати праці на налагодження програми на ЕОМ:

$$t_{\text{відл}} = \frac{Q}{(4 \dots 5)K}$$

$$t_{\text{відл}} = 1500 \,/\, (4 * 1.2) \; = 450, \text{людино-години}$$

Витрати праці на підготовку документації:

$$t_{\text{д}} = t_{\text{др}} + t_{\text{до}}, \text{людино-годин}$$

де $t_{др}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{\text{др}} = \frac{Q}{(15 \dots 20)K}$$

$$t_{\text{др}} = 1500 \,/\, (17 * 1.2) \; = 105, \text{людино-години}$$

$t_{до}$ – трудомісткість редагування, печатки й оформлення документації

$$t_{\text{до}} = 0.75 * t_{\text{др}}, \text{людино-годин}$$
$$t_{\text{до}} = 0.75 * 105 = 79.4, \text{людино-годин}$$
$$t_{\text{д}} = 105 + 79.4 = 184.4, \text{людино-годин}$$

Отримуємо трудомісткість розробки програмного забезпечення:

$$t = 50 + 20.3 + 78.2 + 78.2 + 450 + 184.4 = 861.1, \text{людино-години}$$

4.2. Розрахунок витрат на створення програмного забезпечення

Витрати на створення програмного забезпечення $К_{пз}$ включають витрати на заробітну плату розробників програми ($З_{зп}$) і витрати машинного часу, необхідного для налагодження програми на ЕОМ ($З_{мв}$).

$$К_{\text{по}} = З_{\text{зп}} + З_{\text{мв}}$$
$$З_{\text{зп}} = t * С_{\text{пр}}$$

де t – загальна трудомісткість розробки ПЗ;

$C_{пр}$ – середня годинна заробітна плата програміста

$$З_{зп} = 861.1 * 200 = 172220 \text{ грн.}$$

Витрати машинного часу, необхідного для налагодження програми на ЕОМ ($З_{мв}$) визначаються за формулою:

$$З_{мв} = t_{відл} * С_{мч}$$

де $t_{відл}$ – трудомісткість налагодження програми на ЕОМ;

$C_{мч}$ – вартість машино-години ЕОМ.

$$З_{мв} = 450 * 40 = 18000 \text{ грн.}$$

Таким чином витрати на створення програмного забезпечення, складуть:

$$К_{по} = 172220 + 18000 = 190220 \text{ грн.}$$

Очікувана тривалість розробки ПО:

$$T = \frac{t}{B_k * F_p}$$

де $B_к$ – число розробників;

$F_р$– місячний фонд робочого часу (при 40-ка годинному робочому тижні $F_р$ = 176 годин).

$$t_{др} = 861.1 \ / \ 1 * 176 \ = 5 \text{ місяців}$$

4.3. Маркетингові дослідження

В процесі дослідження методів роботи передачі даних між користувачами, було створено додаток для особистого користування. Метою

додатку є покращення користувальницького досвіду при передачі файлів бездротовим способом між пристроями незалежно від їх операційної системи.

Принцип взаємодії користувача з додатком:

1. Користувач відкриває додаток, наприклад, на смартфоні та на іншому пристрої, на який він хоче передати файли;
2. На екрані графічного інтерфейсу додатку з’являється список з цими пристроями;
3. Користувач в додатку натискає на пристрій, на який хоче передати файли;
4. У діалоговому вікні користувач обирає необхідні для передачі файли та натискає кнопку «Відправити»;
5. На пристрої, що приймає ці файли, можна або прийняти або відхилити дані, що надсилаються.

Як результат, з процесу передачі даних повністю виключено будь-які USB-кабелі. Єдине що потребує додаток для передачі даних – підключення обох пристроїв до інтернету.

Додаток наразі не потребує затрат на своє обслуговування, оскільки серверна частина розгорнута на безкоштовному плані Google Firebase. Клієнтська частина являє собою веб-додаток, тож може бути розгорнута на Heroku, що також має достатній за характеристиками безкоштовний план.

Основним каналом розподілу додатку буде сама веб-версія додатку, що буде доступна за певним посиланням. В майбутньому планується розробити клієнти для мобільних пристроїв. Їх основними каналами розподілу стануть магазин додатків AppStore та Google Play. Щоб розмістити додаток в Google Play потрібно мати ліцензію розробника, що разово коштує $25. Ліцензія розробника для AppStore дорожча та коштує $100 на рік. Для ПК та ноутбуків доступні веб-версії додатку, а також планується розробити додаток, що можна буде завантажити на сайті.

Основною цільовою аудиторією додатку є люди молодого та середнього віку, що активно використовують декілька пристроїв протягом дня та мають

потребу в швидкому та простому способі передачі файлів з одного пристрою на інший. У таких людей, наприклад, може бути смартфон на базі ОС Android та ПК на базі Windows. Досі для того, щоб передати дані з одного пристрою на інший користувач мав би користуватися або USB-кабелем, або хмарними сервісами, що значно подовжує час передачі файлів, а також витрачає зайвий інтернет-трафік. А за допомогою розробленого додатку передачу файлів можна зробити в декілька кліків.

Додаток має зайняти нішу Apple AirDrop, тільки не лише для комп'ютерів компанії Apple, а для будь-яких інших пристроїв також, незалежно від того, яка на них встановлена операційна система – Windows, Linux, MacOS, Android чи iOS.

4.4. Економічна ефективність

Створена у ході дослідження система передачі файлів була зроблена як додаток для персонального користування, отже економічну ефективність цього продукту обчислити неможливо, але можна визначити його соціальний ефект.

Соціальним ефектом від розробленого продукту є покращення процесу передачі файлів між пристроями з різними операційними системами. Це було досягнуто за рахунок простого користувальницького інтерфейсу, швидкості передачі даних та сумісністю з усіма сучасними типами пристроїв. Таким чином, для передачі файлів, наприклад, зі смартфону на ПК тепер не потрібно використовувати жодних USB-кабелів, а лише зробити декілька кліків в додатку.

Окрім цього, напрацювання та технології, використані у цьому продукті, можуть бути повторно застосовані в інших системах передачі даних або продуктах, що потребують такої функції.

ВИСНОВКИ

В процесі дослідження були отримані наступні результати. При вивчені принципів побудови додатку для передачі файлів між пристроями з різними операційними системами, було вирішено розділити проектування даної системи на декілька етапів: вирішення проблем з підтримкою різних операційних систем, розробка архітектури додатку та розробка, власне, додатку.

Для вирішень проблем з підтримкою багатьох ОС було вирішено розробити веб-додаток з використанням багатоплатформової мови JavaScript. Такий підхід дозволяє додатку працювати в веб-браузері, а також бути вбудованим в додаток та скомпільованим для будь-якої операційної системи. Таким чином не потрібно розробляти окреме рішення для кожної окремої системи.

Оскільки це веб-додаток, то основний тип архітектури – клієнт-серверна. Цей тип використовується, коли дані про кожного користувача заносяться в базу даних, а також на сервері виконується розмежування їх доступу. Однак додаток передбачає передачу файлів, що можуть мати великий розмір та створити надлишкове навантаження на сервер. Щоб цього уникнути для передачі даних файлу було використано архітектуру peer-to-peer. Цей тип архітектури призначений для обміну даними між користувачами напряму без використання проміжного серверу.

Для передачі даних самого файлу була обрана технологія WebRTC. Вона була створена для передачі даних між двома вузлами за архітектурою peer-to-peer. Основне призначення WebRTC – передача потокових даних, наприклад аудіо та відео в реальному часі. Саме тому основне призначення цієї технології – онлайн відео та аудіо-конференції. Однак під час дослідження технології WebRTC було виявлено, що за її допомогою можна також відправляти і звичайні файли.

WebRTC була обрана для того, щоб спростити процес передачі між пристроями, оскільки вона вже має в собі всі потрібні механізми для передачі

та встановлення зв’язку. Таким чином значно спрощується розробка та підтримка додатку, а також підвищується стабільність його роботи.

WebRTC є багатоплатформовою технологією, оскільки підтримується багатьма веб-браузерами та операційними системами. З цією технологією можна взаємодіяти за допомогою скриптової мови JavaScript, що також є багатоплатформовим рішенням.

Для функціонування WebRTC потрібен STUN сервер. STUN – це сервер, що повідомляє вузлу його знаходження в мережі. Тобто він повертає IP адресу та порт вузла в глобальній мережі. Це необхідно для подолання NAT на маршрутизаторах та передачі файлів напряму між вузлами, що знаходяться навіть в різних мережах.

Ключовим моментом у встановленні зв’язку у WebRTC є генерація ICE Candidate. Це інформація про вузол, яку потрібно передати іншому вузлу для встановлення контакту. Це можна зробити будь-яким способом, але найбільш поширеним є використання веб-сокетів. Для цього потрібен простий сокет-сервер, що буде передавати невеликі повідомлення між вузлами та виконувати роль координатора їх роботи.

Після дослідження доступних технологій та розробки архітектури додатку, було створено сам додаток. В якості JavaScript-фреймворку було використано Ember. Цей фреймворк дозволяє легко працювати з навігацією та шаблонізувати додаток. Для серверної частини було використано Firebase. Цей сервіс має весь необхідний функціонал у вигляді сокет-серверу, бази даних та налаштувань їх роботи.

Після розробки додатку були проведені тести швидкості передачі файлів в різних мережах. Результати продемонстрували достатню швидкість, так, наприклад, в локальній мережі вона становила 95 Мб/с. Також було виміряно і середній час передачі файлів між пристроями. Час передачі зображення в 3.6 МБ навіть в повільній мобільній мережі становив всього 4 секунди, а в швидкій локальній мережі 0.3 секунди. На швидкість передачі тип

та розмір файлу не впливають, таким чином його можна використовувати не лише для передачі медіа-файлів, а і будь-яких інших файлів.

Зі зростаючим попитом на бездротові засоби передачі даних розроблене рішення може знайти застосування одразу в декількох сферах. По-перше, для персонального користування, щоб ділитися файлами між декількома власними пристроями. По-друге, для відправки файлу іншому користувачу, який може знаходитися взагалі в іншому місці та бути підключеним до іншої мережі. Маючи посилання на спільну кімнату с користувачем, передача даних також можлива. По-третє, як рішення для бізнесу для покращення комунікації між працівниками. Це дозволить не створювати зайвої мережевої інфраструктури, а ділитися файлами між працівниками напряму.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Albers V., Still B. (Eds.) Usability of Complex Information Systems: Evaluation of User Interaction. – CRC Press, 2011. – 392 p.

2. Ambler T. JavaScript Frameworks for Modern Web Dev / T. Ambler, N. Cloud – Apress, 2015. – 520 p.

3. Arlow Jim, Neustadt Ila. Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. – Addison–Wesley Professional, 2004. 528 p.

4. Aslaksen E.W. Designing Complex Systems: Foundations of Design in the Functional Domain. – Auerbach Publications; 1 edition (October 27, 2008). – 176 p.

5. Bryant R.E. Computer Systems: A Programmer's Perspective, 3 Edition / R.E. Bryant, A.R. O'Hallaron – Pearson India Education Services Pvt. Ltd., 2016. – 454p.

6. Brusilovsky, P. Adaptive and intelligent Web-based educational systems / P. Brusilovsky, C. Peylo // International Journal of Artificial Intelligence in Education. Special Issue on Adaptive and Intelligent Web-based Educational Systems – 2003. – № 13 (2-4). – P. 159-172.

7. Brusilovsky, P. Methods and techniques of adaptive hypermedia / P. Brusilovsky // User Modeling and User-Adapted Interaction. – 1996. – № 6 (2-3). – P. 87-129.

8. Buley L. The User Experience Team of One: A Research and Design Survival Guide / L. Buley – Rosenfeld Media, 2013. – 264p.

9. Burnett D. C. WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web / Burnett D. C. Burnett, Johnston A. B. Johnston – Digital Codex LLC, 2014. – 350p.

10. Casciaro M. Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques, 3rd Edition / M. Casciaro – Packt Publishing, 2020. – 660p.

11. Cravens J. Building Web Apps with Ember.js: Write Ambitious JavaScript / J. Cravens – O'Reilly Media, 2014. – 188p.

12. Domes S. Progressive Web Apps with React: Create lightning fast web apps with native power using React and Firebase / S. Domes – Packt Publishing, 2017. – 302p.

13. Dijkstra E. How do we tell truths that might hurt? / E. Dijkstra // Selected Writings on Computing: A Personal Perspective. – 1982. – № 1(23). – P. 89-131.

14. Elliot E. Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries / E. Elliot – O'Reilly Media, 2014. – 254p.

15. Flannagan D. JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language / D. Flannagan – O'Reilly Media, 2020. – 706p.

16. Frisbie M. Professional JavaScript for Web Developers / M. Frisbie – Wrox, 2019. – 1200p.

17. Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / E.Gamma, J. Vlissides, R. Johnson, R. Helm, 1994. – 395 p.

18. Gourley D. HTTP: The Definitive Guide: The Definitive Guide (Definitive Guides) / Gourley D. – O'Reilly Media, 2002. – 656p.

19. Hanchett E. Ember.js Cookbook / E. Hanchett – Packt Publishing, 2016. – 308p.

20. Godbolt M. Frontend Architecture for Design Systems: A Modern Blueprint for Scalable and Sustainable Websites / M. Godbolt – O'Reilly, 2016. – 198 p.

21. Gothelf J. Lean UX: Designing Great Products with Agile Teams / J. Gothelf, J. Seiden – O'Reilly Media, 2016. – 208p.

22. Grigorik I. High Performance Browser Networking: What every web developer should know about networking and web performance / I. Grigorik – O'Reilly Media, 2013. – 400p.

23. Haverbeke M. Eloquent JavaScript, 3rd Edition: A Modern Introduction to Programming / M. Haverbeke – Packt Publishing, 2015. – 472p.

24. Kereke F. Mastering JavaScript Functional Programming: Write clean, robust, and maintainable web and server code using functional JavaScript, 2nd Edition / F. Kereke – Packt Publishing, 2020. – 470p.

25. Kumar A. Mastering Firebase for Android Development: Build real-time, scalable, and cloud-enabled Android apps with Firebase / A. Kumar – Packt Publishing, 2018. – 394p.

26. Lim G. Beginning Node.js, Express & MongoDB Development / G. Lim – Greg Lim, 2019. – 153p.

27. Lombardi A. WebSocket: Lightweight Client-Server Communications / A. Lombardi – O'Reilly Media, 2015. – 144p.

28. Loretto S. Real-Time Communication with WebRTC: Peer-to-Peer in the Browser / Loretto S., Romano S.P. – O'Reilly Media, 2014. – 164p.

29. Marsh J. UX for Beginners: A Crash Course in 100 Short Lessons / J. Marsh – O'Reilly Media, 2016. – 258p.

30. Masse M. REST API Design Rulebook 1st Edition / M. Masse, 2011. – 114 p.

31. Moore D. Peer-to-Peer: Building Secure, Scalable, and Manageable Networks / D. Moore, J. Hebeler – McGraw-Hill Companies, 2001. – 512p.

32. Moroney L. The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform / L. Moroney – Apress, 2017. – 390p.

33. Myers M. A Smarter Way to Learn JavaScript. The new tech-assisted approach that requires half the effort / M. Myers – CreateSpace Independent Publishing Platform, 2014. – 254p.

34. Podmajersky T. Strategic Writing for UX: Drive Engagement, Conversion, and Retention with Every Word / T. Podmajersky – O'Reilly Media, 2019. – 194p.

35. Puri S. Ember.js Web Development with Ember CLI / S. Puri – Packt Publishing, 2015. – 176p.

36. Ristic D. Learning WebRTC / Ristic D. – Packt Publishing, 2015 – 186p.

37. Schneider T. Dynamics in Large-Scale Peer-to-Peer Networks: A new Quality of Computer Networking / T. Schneider – VDM Verlag Dr. Müller, 2008. – 84p.

38. Sergiienko A. WebRTC cookbook / Sergiienko A. – Packt Publishing, 2015. – 230p.

39. Silberstatz A. Operating System Concepts / A. Silberstatz, P.B. Galvin, G. Gagne – Wiley, 2012. – 976p.

40. Syed B. Beginning Node.js / B. Syed – Apress, 2020. – 326p.

41. Simpson K. You Don't Know JS Yet: Get Started / K. Simpson, S. St.Laurent, B. Holt – Independently published, 2020. – 143p.

42. Tanna M. Serverless Web Applications with React and Firebase: Develop real-time applications for web and mobile platforms / M. Tanna, H. Singh – Packt Publishing, 2018. – 143p.

43. Travis D. Think Like a UX Researcher: How to Observe Users, Influence Design, and Shape Business Strategy / D. Travis – CRC Press, 2019 – 306p

44. Ward John, Peppard Joe. Strategic Planning for Information Systems. – John Wiley & Sons, 2002. – 641 p.

45. Wexler J. Get Programming with Node.js / J. Wexler – Manning Publications, 2019. – 480p.

46. Yablonski J. Laws of UX: Using Psychology to Design Better Products & Services / J. Yablonski – O'Reilly Media, 2020. – 152p.

47. Yahiaoui H. Firebase Cookbook: Over 70 recipes to help you create real-time web and mobile applications with Firebase / H. Yahiaoui – Packt Publishing, 2017. – 288p.

48. Young A. R. Node.js in Practice / A. R. Young, M. Harter – Manning Publications, 2014. – 424p.

49. Zammetti F. Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker / F. Zametti – Apress, 2020. – 395p.

50. Гарсиа-Молина Г. Системы баз данных. Полный курс / Г. ГарсиаМолина Дж. Ульман , Дж. Уидом – М.: Вильямс, 2003. – 1088 с.

51. Грекул В.И., Денищенко Г.Н., Коровкина Н.Л. Проектирование информационных систем. – М.: Интернет–Университет Информационных Технологий «Интуит», 2016. – 570 с.

52. Документація Ember.js (Електрон. ресурс) / Режим доступу: https://emberjs.com/

53. Документація Firebase (Електрон. ресурс) / Режим доступу: https://developer.mozilla.org/ru/docs/Web/API/WebRTC_API

54. Документація Git (Електрон. ресурс) / Режим доступу: https://firebase.google.com/docs

55. Документація WebRTC (Електрон. ресурс) / Режим доступу: https://developer.mozilla.org/ru/docs/Web/API/WebRTC_API

56. Дрігалкін В. Веб-сайт на 100%. Як створити веб-сайт і зробити його видимим в Інтернеті – Діалектика-Вільямс, 2010 р. – 224с.

57. Кнут Д. Искусство программирования для ЭВМ. Сортировка и поиск / Д. Кнут. – М.: Вильямс, 2007. — 824 с.

58. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: Вильямс, 2013. – 1328 с.

59. Коцюба И.Ю., Чунаев А.В., Шиков А.Н. Основы проектирования информационных систем. Учебное пособие. – СПб.: Университет ИТМО, 2015. – 206 с.

60. Кузін А.В «Базы данных, 5-е издание» / Кузін А.В., Левонисова С.В. – К. : «Академия», 2012. – 317 с.

61. Куроуз Д. Компьютерные сети. Нисходящий подход, 6-е издание / Д. Куроуз, К. Росс – Москва : Издательство «Э», 2016 – 912 с.

62. Лаврищева Е. М. Методы программирования: теория, инженерия, практика / Е. М. Лаврищева. – К.: Наукова думка. – 2006. – 451 с.

63. Ніксон Р. «Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS и HTML5. 3-е издание» / Робін Ніксон – К. : «Питер», 2015. – 688 с.

64. Основы инженерии качества программных систем / Ф. И. Андон, Г. И. Коваль, Т. М. Коротун, В. Ю. Суслов – К: Академпериодика, 2002. – 502 с.

65. Самойлов В. Д. Модельное конструирование компьютерных приложений / В. Д. Самойлов. – К.: Наукова думка, 2007. – 198 с.

66. Сміт Б. «Beginning JSON» / Б. Сміт – К. : «Apress», 2015. – 324 с

67. Чмир І.О. Моделювання систем у середовищі UML (Unified Modeling Language) : навч. посібник / І. О. Чмир. – Черкаськ. акад. менеджменту. - Черкаси : ЧАМ, 2004. – 100 с.

68. Чмирь И. А. Объектно-ориентированное моделирование / И. А. Чмирь. – Одесса: НАДУ, 2005. – 243 с.

69. Шуйтенов Г.Ж. Технологии передачи данных в системах видеоконференции /Г.Ж.Шуйтенов, Т.Ж.Айдынбай // Научный журнал Евразийского национального университета имени Л.Н.Гумилева. – 2015. – №4.- с.77-83.

70. Эрих Гамма. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – М.:Microsoft, 2016. – 366 с.

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

Конфігурація бази даних Firebase

```
{
    "rules": {
        "rooms": {
            "$roomid": {
                // You can see people in the room only if you are in it as well
                ".read": "auth != null && data.child('users').hasChild(auth.id)",
                "users": {
                    "$userid": {
                        // You can modify only your own info
                        ".write": "auth != null && $userid == auth.id",

                        // Ensure that all required attributes are there
                        ".validate": "newData.hasChildren(['uuid', 'public_ip', 'peer']) &&
newData.child('peer').hasChildren(['id'])"

                    }
                },
                "messages": {
                    // You can send message to anybody in the room
                    ".write": "auth != null",
                    "$userid": {
                        // You can read only messages sent to you
                        ".read": "auth != null && $userid == auth.id"
                    }
                }
            }
        }
    }
}
```

package.json

```
{
  "name": "webrtc-file-sharing",
  "version": "1.0.0",
  "private": true,
  "description": "WebRTC file sharing",
  "license": "MIT",
  "author": "Max Kurylo",
  "directories": {
    "doc": "doc",
    "test": "tests"
  },
  "scripts": {
    "build": "ember build --environment=production",
    "lint": "npm-run-all --aggregate-output --continue-on-error --parallel lint:*",
    "develop": "yarn install --frozen-lockfile && nf --procfile=Procfile.dev start",
    "lint:hbs": "ember-template-lint .",
    "lint:js": "eslint .",
    "start": "ember serve",
    "test": "npm-run-all lint:* test:*",
    "test:ember": "ember test"
  },
  "devDependencies": {
    "@ember/jquery": "^1.1.0",
    "@ember/optional-features": "^2.0.0",
    "@glimmer/component": "^1.0.2",
    "@glimmer/tracking": "^1.0.2",
    "babel-eslint": "^10.1.0",
```

```
    "broccoli-asset-rev": "^3.0.0",
    "ember-auto-import": "^1.6.0",
    "ember-cli": "~3.21.2",
    "ember-cli-app-version": "^3.2.0",
    "ember-cli-babel": "^7.22.1",
    "ember-cli-dependency-checker": "^3.2.0",
    "ember-cli-dotenv": "^3.1.0",
    "ember-cli-htmlbars": "^5.3.1",
    "ember-cli-inject-live-reload": "^2.0.2",
    "ember-cli-sass": "^10.0.0",
    "ember-cli-sri": "^2.1.1",
    "ember-cli-terser": "^4.0.0",
    "ember-export-application-global": "^2.0.1",
    "ember-fetch": "^8.0.2",
    "ember-load-initializers": "^2.1.1",
    "ember-maybe-import-regenerator": "^0.1.6",
    "ember-qrcode-shim": "^0.4.0",
    "ember-qunit": "^4.6.0",
    "ember-resolver": "^8.0.2",
    "ember-source": "~3.21.3",
    "ember-template-lint": "^2.13.0",
    "eslint": "^7.10.0",
    "eslint-config-airbnb-base": "^14.1.0",
    "eslint-config-prettier": "^6.12.0",
    "eslint-plugin-ember": "^9.2.0",
    "eslint-plugin-import": "^2.22.1",
    "eslint-plugin-node": "^11.1.0",
    "eslint-plugin-prettier": "^3.0.1",
    "foreman": "3.0.1",
    "husky": "^4.3.0",
    "lint-staged": "^10.4.0",
    "loader.js": "^4.7.0",
    "npm-run-all": "^4.1.5",
    "prettier": "^2.1.2",
    "qunit-dom": "^1.5.0",
    "sass": "^1.26.11"
  },
  "dependencies": {
    "@sentry/browser": "5.24.2",
    "@sentry/integrations": "5.24.2",
    "body-parser": "^1.10.0",
    "compression": "^1.2.2",
    "cookie-parser": "^1.3.3",
    "cookie-session": "^1.1.0",
    "express": "^4.10.6",
    "firebase-token-generator": "~2.0.0",
    "jszip": "^3.5.0",
    "lodash": "^4.17.20",
    "morgan": "^1.5.0",
    "newrelic": "^6.13.1",
    "stream": "^0.0.2",
    "uuid": "^8.3.1"
  },
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "*.js": [
      "yarn run prettier --write",
      "yarn run lint:hbs",
      "yarn run lint:js"
    ]
  },
  "engines": {
    "node": ">=12.*"
  },
  "ember": {
    "edition": "octane"
  },
  "ember-addon": {
    "paths": [
      "lib/google-analytics"
    ]
  }
}
```

server.js

```
/* eslint-env node */

if (process.env.NODE_ENV === 'production') {
  // eslint-disable-next-line global-require
  require('newrelic');
}

// Room server
const http = require('http');
const path = require('path');
const express = require('express');
const logger = require('morgan');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const cookieSession = require('cookie-session');
const compression = require('compression');
const { v4: uuidv4 } = require('uuid');
const crypto = require('crypto');
const FirebaseTokenGenerator = require('firebase-token-generator');

const firebaseTokenGenerator = new FirebaseTokenGenerator(
  process.env.FIREBASE_SECRET,
);
const app = express();
const secret = process.env.SECRET;
const base = ['dist'];

app.enable('trust proxy');

app.use(logger('combined'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(cookieParser());
app.use(
  cookieSession({
    cookie: {
      // secure: true,
      httpOnly: true,
      maxAge: 30 * 24 * 60 * 60 * 1000, // 30 days
    },
    secret,
    proxy: true,
  }),
);
app.use(compression());

//
// Web server
//
base.forEach((dir) => {
  const subdirs = ['assets'];

  subdirs.forEach((subdir) => {
    app.use(
      `/${subdir}`,
      express.static(`${dir}/${subdir}`, {
        maxAge: 31104000000, // ~1 year
      }),
    );
  });
});

//
// API server
//
app.get('/', (req, res) => {
  const root = path.join(__dirname, base[0]);
  console.log({ root });
  res.sendFile(`${root}/index.html`);
});

app.get('/rooms/:id', (req, res) => {
  const root = path.join(__dirname, base[0]);
  res.sendFile(`${root}/index.html`);
});
```

```
app.get('/room', (req, res) => {
  const ip = req.headers['cf-connecting-ip'] || req.ip;
  const name = crypto.createHmac('md5', secret).update(ip).digest('hex');

  res.json({ name });
});

app.get('/auth', (req, res) => {
  const ip = req.headers['cf-connecting-ip'] || req.ip;
  const uid = uuidv4();
  const token = firebaseTokenGenerator.createToken(
    { uid, id: uid }, // will be available in Firebase security rules as 'auth'
    { expires: 32503680000 }, // 01.01.3000 00:00
  );

  res.json({ id: uid, token, public_ip: ip });
});

http
  .createServer(app)
  .listen(process.env.PORT)
  .on('listening', () => {
    console.log(
      `Started ShareDrop web server at http://localhost:${process.env.PORT}...`,
    );
  });
```

services/web-rtc.js

```
// TODO:
// - provide TURN server config once it's possible to create rooms with custom names
// - use Ember.Object.extend()

import $ from 'jquery';
import File from './file';

const WebRTC = function (id, options) {
  const defaults = {
    config: { iceServers: [{ urls: 'stun:stun.l.google.com:19302' }] },
    debug: 3,
  };

  this.conn = new window.Peer(id, $.extend(defaults, options));

  this.files = {
    outgoing: {},
    incoming: {},
  };

  // Listen for incoming connections
  this.conn.on('connection', (connection) => {
    $.publish('incoming_peer_connection.p2p', { connection });

    connection.on('open', () => {
      console.log('Peer:\t Data channel connection opened: ', connection);
      $.publish('incoming_dc_connection.p2p', { connection });
    });

    connection.on('error', (error) => {
      console.log('Peer:\t Data channel connection error', error);
      $.publish('incoming_dc_connection_error.p2p', {
        connection,
        error,
      });
    });

    this._onConnection(connection);
  });

  this.conn.on('close', () => {
    console.log('Peer:\t Connection to server closed.');
  });

  this.conn.on('error', (error) => {
```

```
    console.log('Peer:\t Error while connecting to server: ', error);
  });
};

WebRTC.CHUNK_MTU = 16000;
WebRTC.CHUNKS_PER_ACK = 64;

WebRTC.prototype.connect = function (id) {
  const connection = this.conn.connect(id, {
    label: 'file',
    reliable: true,
    serialization: 'none', // we handle serialization ourselves
  });

  connection.on('open', () => {
    console.log('Peer:\t Data channel connection opened: ', connection);
    $.publish('outgoing_dc_connection.p2p', { connection });
  });

  connection.on('error', (error) => {
    console.log('Peer:\t Data channel connection error', error);
    $.publish('outgoing_dc_connection_error.p2p', {
      connection,
      error,
    });
  });

  $.publish('outgoing_peer_connection.p2p', { connection });
  this._onConnection(connection);
};

WebRTC.prototype._onConnection = function (connection) {
  const self = this;

  console.log('Peer:\t Opening data channel connection...', connection);

  connection.on('data', (data) => {
    // Lame type check
    if (data.byteLength !== undefined) {
      // ArrayBuffer
      self._onBinaryData(data, connection);
    } else {
      // JSON string
      self._onJSONData(JSON.parse(data), connection);
    }
  });

  connection.on('close', () => {
    $.publish('disconnected.p2p', { connection });
    console.log('Peer:\t P2P connection closed: ', connection);
  });
};

WebRTC.prototype._onBinaryData = function (data, connection) {
  const self = this;
  const incoming = this.files.incoming[connection.peer];
  const { info, file, block, receivedChunkNum } = incoming;
  const chunksPerAck = WebRTC.CHUNKS_PER_ACK;

  // TODO move it after requesting a new block to speed things up
  connection.emit(
    'receiving_progress',
    (receivedChunkNum + 1) / info.chunksTotal,
  );
  // console.log('Got chunk no ' + (receivedChunkNum + 1) + ' out of ' + info.chunksTotal);

  block.push(data);

  incoming.receivedChunkNum = receivedChunkNum + 1;
  const nextChunkNum = incoming.receivedChunkNum;
  const lastChunkInFile = receivedChunkNum === info.chunksTotal - 1;
  const lastChunkInBlock =
    receivedChunkNum > 0 && (receivedChunkNum + 1) % chunksPerAck === 0;

  if (lastChunkInFile || lastChunkInBlock) {
    file.append(block).then(() => {
      if (lastChunkInFile) {
        file.save();

        $.publish('file_received.p2p', {
```

```
          blob: file,
          info,
          connection,
        });
      } else {
        // console.log('Requesting block starting at: ' + (nextChunkNum));
        incoming.block = [];
        self._requestFileBlock(connection, nextChunkNum);
      }
    });
  }
};

WebRTC.prototype._onJSONData = function (data, connection) {
  switch (data.type) {
    case 'info': {
      const info = data.payload;

      $.publish('info.p2p', {
        connection,
        info,
      });

      // Store incoming file info for later
      this.files.incoming[connection.peer] = {
        info,
        file: null,
        block: [],
        receivedChunkNum: 0,
      };

      console.log('Peer:\t File info: ', data);
      break;
    }
    case 'cancel': {
      $.publish('file_canceled.p2p', {
        connection,
      });

      console.log('Peer:\t Sender canceled file transfer');
      break;
    }
    case 'response': {
      const response = data.payload;

      // If recipient rejected the file, delete stored file
      if (!response) {
        delete this.files.outgoing[connection.peer];
      }

      $.publish('response.p2p', {
        connection,
        response,
      });

      console.log('Peer:\t File response: ', data);
      break;
    }
    case 'block_request': {
      const { file } = this.files.outgoing[connection.peer];

      // console.log('Peer:\t Block request: ', data.payload);

      this._sendBlock(connection, file, data.payload);
      break;
    }
    default:
      console.log('Peer:\t Unknown message: ', data);
  }
};

WebRTC.prototype.getFileInfo = function (file) {
  return {
    lastModifiedDate: file.lastModifiedDate,
    name: file.name,
    size: file.size,
    type: file.type,
    chunksTotal: Math.ceil(file.size / WebRTC.CHUNK_MTU),
  };
};
```

```
WebRTC.prototype.sendFileInfo = function (connection, info) {
  const message = {
    type: 'info',
    payload: info,
  };

  connection.send(JSON.stringify(message));
};

WebRTC.prototype.sendCancelRequest = function (connection) {
  const message = {
    type: 'cancel',
  };

  connection.send(JSON.stringify(message));
};

WebRTC.prototype.sendFileResponse = function (connection, response) {
  const message = {
    type: 'response',
    payload: response,
  };

  if (response) {
    // If recipient accepted the file, request required space to store the file on HTML5 filesystem
    const incoming = this.files.incoming[connection.peer];
    const { info } = incoming;

    new File({ name: info.name, size: info.size, type: info.type }).then(
      (file) => {
        incoming.file = file;
        connection.send(JSON.stringify(message));
      },
    );
  } else {
    // Otherwise, delete stored file info
    delete this.files.incoming[connection.peer];
    connection.send(JSON.stringify(message));
  }
};

WebRTC.prototype.sendFile = function (connection, file) {
  // Save the file for later
  this.files.outgoing[connection.peer] = {
    file,
    info: this.getFileInfo(file),
  };

  // Send the first block. Next ones will be requested by recipient.
  this._sendBlock(connection, file, 0);
};

WebRTC.prototype._requestFileBlock = function (connection, chunkNum) {
  const message = {
    type: 'block_request',
    payload: chunkNum,
  };
  connection.send(JSON.stringify(message));
};

WebRTC.prototype._sendBlock = function (connection, file, beginChunkNum) {
  const { info } = this.files.outgoing[connection.peer];
  const chunkSize = WebRTC.CHUNK_MTU;
  const chunksPerAck = WebRTC.CHUNKS_PER_ACK;
  const remainingChunks = info.chunksTotal - beginChunkNum;
  const chunksToSend = Math.min(remainingChunks, chunksPerAck);
  const endChunkNum = beginChunkNum + chunksToSend - 1;
  const blockBegin = beginChunkNum * chunkSize;
  const blockEnd = endChunkNum * chunkSize + chunkSize;
  const reader = new FileReader();
  let chunkNum;

  // Read the whole block from file
  const blockBlob = file.slice(blockBegin, blockEnd);

  // console.log('Sending block: start chunk: ' + beginChunkNum + ' end chunk: ' + endChunkNum);
  // console.log('Sending block: start byte : ' + begin + ' end byte : ' + end);

  reader.onload = function (event) {
```

```
    if (reader.readyState === FileReader.DONE) {
      const blockBuffer = event.target.result;

      for (
        chunkNum = beginChunkNum;
        chunkNum < endChunkNum + 1;
        chunkNum += 1
      ) {
        // Send each chunk (begin index is inclusive, end index is exclusive)
        const bufferBegin = (chunkNum % chunksPerAck) * chunkSize;
        const bufferEnd = Math.min(
          bufferBegin + chunkSize,
          blockBuffer.byteLength,
        );
        const chunkBuffer = blockBuffer.slice(bufferBegin, bufferEnd);

        connection.send(chunkBuffer);

        // console.log('Sent chunk: start byte: ' + begin + ' end byte: ' + end + ' length: ' +
chunkBuffer.byteLength);
        // console.log('Sent chunk no ' + (chunkNum + 1) + ' out of ' + info.chunksTotal);

        connection.emit('sending_progress', (chunkNum + 1) / info.chunksTotal);
      }

      if (endChunkNum === info.chunksTotal - 1) {
        $.publish('file_sent.p2p', { connection });
      }
    }
  };

  reader.readAsArrayBuffer(blockBlob);
};

export default WebRTC;
```

models/user.js

```
import Peer from './peer';

const User = Peer.extend({
  serialize() {
    const data = {
      uuid: this.uuid,
      public_ip: this.public_ip,
      label: this.label,
      avatarUrl: this.avatarUrl,
      peer: {
        id: this.get('peer.id'),
      },
    };

    return data;
  },
});

export default User;
```

models/peer.js

```
import EmberObject, { observer } from '@ember/object';
import Evented, { on } from '@ember/object/evented';

export default EmberObject.extend(Evented, {
  uuid: null,
  label: null,
  avatarUrl: null,
  public_ip: null,
  peer: null,
  transfer: null,
```

```
  init(...args) {
    this._super(args);

    const initialPeerState = EmberObject.create({
      id: null,
      connection: null,
      // State of data channel connection. Possible states:
      // - disconnected
      // - connecting
      // - connected
      state: 'disconnected',
    });
    const initialTransferState = EmberObject.create({
      file: null,
      info: null,
      sendingProgress: 0,
      receivingProgress: 0,
    });

    this.set('peer', initialPeerState);
    this.set('transfer', initialTransferState);
  },

  // Used to display popovers. Possible states:
  // - idle
  // - has_selected_file
  // - establishing_connection
  // - awaiting_response
  // - received_file_info
  // - declined_file_transfer
  // - receiving_file_data
  // - sending_file_data
  // - error
  state: 'idle',

  // Used to display error messages in popovers. Possible codes:
  // - multiple_files
  errorCode: null,

  stateChanged: on(
    'init',
    observer('state', function () {
      console.log('Peer:\t State has changed: ', this.state);

      // Automatically clear error code if transitioning to a non-error state
      if (this.state !== 'error') {
        this.set('errorCode', null);
      }
    }),
  ),
});
```

initializers/prerequisites.js

```
/* jshint -W030 */
import $ from 'jquery';
import { Promise } from 'rsvp';
import config from 'sharedrop/config/environment';

import FileSystem from '../services/file';
import Analytics from '../services/analytics';

export function initialize(application) {
  function checkWebRTCSupport() {
    return new Promise((resolve, reject) => {
      // window.util is a part of PeerJS library
      if (window.util.supports.sctp) {
        resolve();
      } else {
        // eslint-disable-next-line prefer-promise-reject-errors
        reject('browser-unsupported');
      }
    });
```

```
  }

  function clearFileSystem() {
    return new Promise((resolve, reject) => {
      // TODO: change File into a service and require it here
      FileSystem.removeAll()
        .then(() => {
          resolve();
        })
        .catch(() => {
          // eslint-disable-next-line prefer-promise-reject-errors
          reject('filesystem-unavailable');
        });
    });
  }

  function authenticateToFirebase() {
    return new Promise((resolve, reject) => {
      const xhr = $.getJSON('/auth');
      xhr.then((data) => {
        const ref = new window.Firebase(config.FIREBASE_URL);
        // eslint-disable-next-line no-param-reassign
        application.ref = ref;
        // eslint-disable-next-line no-param-reassign
        application.userId = data.id;
        // eslint-disable-next-line no-param-reassign
        application.publicIp = data.public_ip;

        ref.authWithCustomToken(data.token, (error) => {
          if (error) {
            reject(error);
          } else {
            resolve();
          }
        });
      });
    });
  }

  // TODO: move it to a separate initializer
  function trackSizeOfReceivedFiles() {
    $.subscribe('file_received.p2p', (event, data) => {
      Analytics.trackEvent('received', {
        event_category: 'file',
        event_label: 'size',
        value: Math.round(data.info.size / 1000),
      });
    });
  }

  application.deferReadiness();

  checkWebRTCSupport()
    .then(clearFileSystem)
    .catch((error) => {
      // eslint-disable-next-line no-param-reassign
      application.error = error;
    })
    .then(authenticateToFirebase)
    .then(trackSizeOfReceivedFiles)
    .then(() => {
      application.advanceReadiness();
    });
}

export default {
  name: 'prerequisites',
  initialize,
};
```

controllers/index.js

```
import Controller, { inject as controller } from '@ember/controller';
import { alias } from '@ember/object/computed';
```

```
import $ from 'jquery';

import WebRTC from '../services/web-rtc';
import Peer from '../models/peer';

export default Controller.extend({
  application: controller('application'),
  you: alias('application.you'),
  room: null,
  webrtc: null,

  _onRoomConnected(event, data) {
    const { you } = this;
    const { room } = this;

    you.get('peer').setProperties(data.peer);
    // eslint-disable-next-line no-param-reassign
    delete data.peer;
    you.setProperties(data);

    // Initialize WebRTC
    this.set(
      'webrtc',
      new WebRTC(you.get('uuid'), {
        room: room.name,
        firebaseRef: window.Sharedrop.ref,
      }),
    );
  },

  _onRoomDisconnected() {
    this.model.clear();
    this.set('webrtc', null);
  },

  _onRoomUserAdded(event, data) {
    const { you } = this;

    if (you.get('uuid') !== data.uuid) {
      this._addPeer(data);
    }
  },

  _addPeer(attrs) {
    const peerAttrs = attrs.peer;

    // eslint-disable-next-line no-param-reassign
    delete attrs.peer;
    const peer = Peer.create(attrs);
    peer.get('peer').setProperties(peerAttrs);

    this.model.pushObject(peer);
  },

  _onRoomUserChanged(event, data) {
    const peers = this.model;
    const peer = peers.findBy('uuid', data.uuid);
    const peerAttrs = data.peer;
    const defaults = {
      uuid: null,
      public_ip: null,
    };

    if (peer) {
      // eslint-disable-next-line no-param-reassign
      delete data.peer;
      // Firebase doesn't return keys with null values,
      // so we have to add them back.
      peer.setProperties($.extend({}, defaults, data));
      peer.get('peer').setProperties(peerAttrs);
    }
  },

  _onRoomUserRemoved(event, data) {
    const peers = this.model;
    const peer = peers.findBy('uuid', data.uuid);

    peers.removeObject(peer);
  },
```

```
_onPeerP2PIncomingConnection(event, data) {
  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  // Don't switch to 'connecting' state on incoming connection,
  // as p2p connection may still fail.
  peer.set('peer.connection', connection);
},

_onPeerDCIncomingConnection(event, data) {
  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  peer.set('peer.state', 'connected');
},

_onPeerDCIncomingConnectionError(event, data) {
  const { connection, error } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  switch (error.type) {
    case 'failed':
      peer.setProperties({
        'peer.connection': null,
        'peer.state': 'disconnected',
        state: 'error',
        errorCode: 'connection-failed',
      });
      break;
    case 'disconnected':
      // TODO: notify both sides
      break;
    default:
      break;
  }
},

_onPeerP2POutgoingConnection(event, data) {
  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  peer.setProperties({
    'peer.connection': connection,
    'peer.state': 'connecting',
  });
},

_onPeerDCOutgoingConnection(event, data) {
  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);
  const file = peer.get('transfer.file');
  const { webrtc } = this;
  const info = webrtc.getFileInfo(file);

  peer.set('peer.state', 'connected');
  peer.set('state', 'awaiting_response');

  webrtc.sendFileInfo(connection, info);
  console.log('Sending a file info...', info);
},

_onPeerDCOutgoingConnectionError(event, data) {
  const { connection, error } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  switch (error.type) {
    case 'failed':
      peer.setProperties({
        'peer.connection': null,
        'peer.state': 'disconnected',
        state: 'error',
        errorCode: 'connection-failed',
      });
      break;
```

```
    default:
      break;
  }
},

_onPeerP2PDisconnected(event, data) {
  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  if (peer) {
    peer.set('peer.connection', null);
    peer.set('peer.state', 'disconnected');
  }
},

_onPeerP2PFileInfo(event, data) {
  console.log('Peer:\t Received file info', data);

  const { connection, info } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  peer.set('transfer.info', info);
  peer.set('state', 'received_file_info');
},

_onPeerP2PFileResponse(event, data) {
  console.log('Peer:\t Received file response', data);

  const { connection, response } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);
  const { webrtc } = this;

  if (response) {
    const file = peer.get('transfer.file');

    connection.on('sending_progress', (progress) => {
      peer.set('transfer.sendingProgress', progress);
    });
    webrtc.sendFile(connection, file);
    peer.set('state', 'receiving_file_data');
  } else {
    peer.set('state', 'declined_file_transfer');
  }
},

_onPeerP2PFileCanceled(event, data) {
  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  connection.close();
  peer.set('transfer.receivingProgress', 0);
  peer.set('transfer.info', null);
  peer.set('state', 'idle');
},

_onPeerP2PFileReceived(event, data) {
  console.log('Peer:\t Received file', data);

  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);

  connection.close();
  peer.set('transfer.receivingProgress', 0);
  peer.set('transfer.info', null);
  peer.set('state', 'idle');
  peer.trigger('didReceiveFile');
},

_onPeerP2PFileSent(event, data) {
  console.log('Peer:\t Sent file', data);

  const { connection } = data;
  const peers = this.model;
  const peer = peers.findBy('peer.id', connection.peer);
```

```
    peer.set('transfer.sendingProgress', 0);
    peer.set('transfer.file', null);
    peer.set('state', 'idle');
    peer.trigger('didSendFile');
  },
});
```

controllers/application.js

```
import Controller from '@ember/controller';
import { inject as service } from '@ember/service';
import { v4 as uuidv4 } from 'uuid';

import User from '../models/user';

export default Controller.extend({
  avatarService: service('avatar'),

  init(...args) {
    this._super(args);

    const id = window.Sharedrop.userId;
    const ip = window.Sharedrop.publicIp;
    const avatar = this.avatarService.get();
    const you = User.create({
      uuid: id,
      public_ip: ip,
      avatarUrl: avatar.url,
      label: avatar.label,
    });

    you.set('peer.id', id);
    this.set('you', you);
  },

  actions: {
    redirect() {
      const uuid = uuidv4();
      const key = `show-instructions-for-room-${uuid}`;

      sessionStorage.setItem(key, 'yup');
      this.transitionToRoute('room', uuid);
    },
  },
});
```

router.js

```
import EmberRouter from '@ember/routing/router';
import config from 'sharedrop/config/environment';

export default class Router extends EmberRouter {
  location = config.locationType;

  rootURL = config.rootURL;
}

// eslint-disable-next-line array-callback-return
Router.map(function () {
  this.route('room', {
    path: '/rooms/:room_id',
  });
});
```

ДОДАТОК Б

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу магістра

на тему:

«Інформаційна технологія удосконалення процесу передачі файлів бездротовим способом між пристроями з різними операційними системами на базі технології WebRTC»

студента групи 121м-19-1 Курило Максима Васильовича

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

ДОДАТОК В

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	Опис
Пояснювальні документи	
Kurylo_122m_19_1_diploma.docx	Пояснювальна записка до магістерської роботи. Документ Word.
Kurylo_122m_19_1_diploma.pdf	Пояснювальна записка до магістерської роботи. Документ PDF.
Програма	
app.rar	Архів. Містить код програми.
Презентація	
Kurylo_122m_19_1_presentation.pptx	Презентація до магістерської роботи