

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

Тема: Отдалечено управление на файлове чрез P2P връзка

Дипломант:

Христо Спасов

Научен ръководител:

Николай Спасов

СОФИЯ

2018



**ТЕХНОЛОГИЧНО УЧИЛИЩЕ
ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 06.11.2017 г.

Утвърждавам:.....

Дата на предаване: 06.02.2018 г.

/проф. д-р инж. Т. Василева/

ЗАДАНИЕ

за дипломна работа

на ученика Христо Бориславов Спасов 12 А клас

1.Тема: Отдалечено управление на файлове чрез P2P свързаност

2.Изисквания:

- Electron desktop-приложение, което сканира и предоставя на свързаните към него потребители съдържанието на избрана директория и нейните поддиректории.
- Уеб сайт, от който потребител се свързва към desktop-приложението с възможност за отваряне на директории, четене и промяна на файлове.
- Node.js сървър, който контролира комуникацията между двете страни и установява peer to peer връзка между тях чрез WebRTC.

3.Съдържание 3.1 Обзор

3.2 Същинска част

3.3 Приложение

Дипломант :.....

Ръководител:.....

/ Николай Спасов /

Директор:.....

/ доц. д-р инж. Ст. Стефанова /

УВОД

Тази дипломна работа представлява софтуерен продукт, състоящ се от три части: десктоп приложение, сървър и приложение от една страница (single-page application), което се отваря чрез уеб браузър и се предоставя от сървъра. Целта е изграждане на система, която позволява отдалечен достъп и управление на файлове. Крайният потребител трябва да може лесно от разстояние да използва файлове на избран от него компютър.

Чрез десктоп приложението трябва да бъде избрана папката, чието съдържание да бъде достъпвано, и компютърът, на който приложението работи, трябва да бъде оставен включен и свързан към Интернет. Десктоп приложението, което ще осигурява достъпа до файловете, ще бъде наричано “провайдър” (от англ. ез. “provider” - доставчик).

Уеб приложението, посредством което се осъществява достъпът до файловете, ще бъде наричано “клиент”. То трябва да предоставя възможност за преглед, изтегляне и редактиране на файловете, ако потребителят има права.

Трябва да има създадена система за управление на правата и е нужно да се предприемат мерки по отношение на сигурността. Комуникацията между клиент и провайдър трябва да бъде директна. Клиентът и провайдърът са “пиъри” (от англ. ез. “peer” - равни) в тази връзка. Необходима е система, чрез която пиърите да могат да се откриват и да се установява P2P връзка.

ПЪРВА ГЛАВА

1.1. Обзор на начините за осъществяване на отдалечен достъп до файлове

1.1.1. VPN

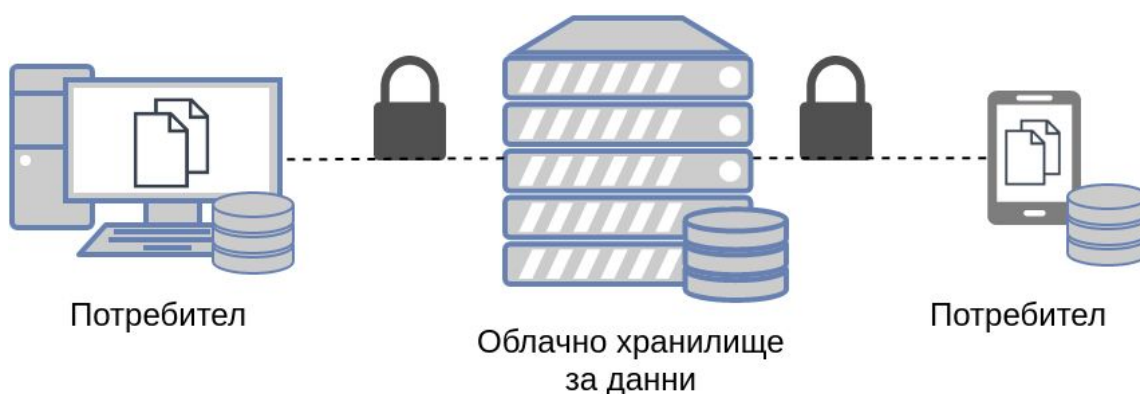
VPN^[3] е аббревиатура на “Virtual Private Network” - виртуална частна мрежа. VPN технологията позволява създаването на криптиран канал между клиенти посредством VPN сървър. Двете крайни точки могат да бъдат на произволни места в глобалната мрежа. Възможно е създаването на личен VPN сървър, който предоставя достъп до VPN мрежата. За да получи достъп до мрежата, потребителят трябва да може да се автентичира пред сървъра с предварително определени име и парола.

Този метод има следните предимства:

- информацията, които се предава, е добре защитена
- не се разчита на чужда услуга, която да съхранява данните

Това решение обаче не може да бъде използвано от всеки краен потребител. Нужни са познания по компютърни мрежи и мрежова сигурност, за да бъде настроен VPN сървър.

1.1.2. Облачна услуга



Фиг. 1.1: Диаграма на облачна услуга

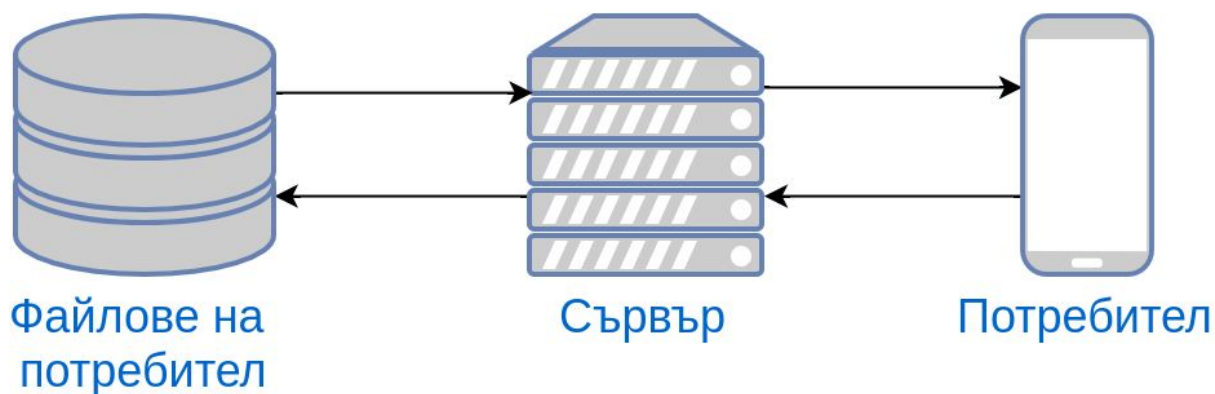
Друг начин за отдалечен достъп на файлове е чрез облачна услуга, на която файловете могат да се качат и след това да се достъпват. Примери за такива системи са Google Drive и Dropbox.

Предимството е, че конфигурациите, които потребителят трябва да направи, са минимални - на практика само автентикация. След нея потребителят е свободен да управлява файловете си. Той няма нужда да прави поддръжка на компютъра, на който се намират файловете - тази работа е оставена на поддръжката на услугата.

Последното освен предимство, също е и недостатък - потребителят трябва да плаща за услугата, а ако има безплатен вариант, то той е с ограничения. Също е нужна синхронизация между данните, които потребителят иска да достъпва, и данните, които се намират в облака. Всеки един файл, който може да бъде достъпван, трябва предварително да бъде качен.

Услугата не може да предостави неограничен по размер склад, който да побира произволно количество данни. Съществува и риск за сигурността - файловете на потребителя биват съхранявани при някого друг. Възможно е да бъдат откраднати и използвани за недоброжелателни цели, ако не са добре защитени.

1.1.3. Система със сървър-посредник между файлове и потребител



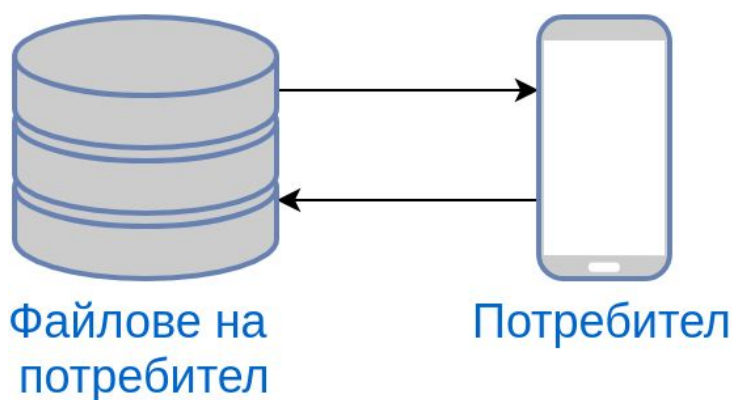
Фиг. 1.2: Система за отдалечен достъп до файлове чрез сървър-посредник

Възможно е да се изгради система, при която има приложение, което сканира файлове, и при заявка за достъп до тях от страна на потребителя приложението праща заявените файлове до сървъра, а той ги препраща към потребителя.

Предимството е, че информацията за файловете е винаги актуална, тъй като не се достъпва нейно копие, както в случая с облачната услуга, описан в т. 1.1.2. Цената за поддръжка на услугата намалява, тъй като няма нужда от склад за данните на всеки един потребител. Потребителят е свободен да избере достъп до която и да е папка. Конфигурациите, които потребителят трябва да направи, са минимални. Възможно е да се навигира сред огромно количество информация и да се изпрати само тази, която е необходима.

Но все още са нужни сървъри-посредници между потребителя и източника на файловете. Пращането на файл първо до сървър, а след това до потребител, е по-бавно от вариант, при който изпращането става директно до потребителя. Необходим е достатъчен брой сървъри, за да се осигури трафика на информация. На самия доставчик на файлове трябва да бъде осигурено постоянно захранване и Интернет връзка.

1.1.4. Отдалечено управление на файлове чрез P2P връзка



Фиг. 1.3: Достъп до файлове чрез P2P връзка

Системата за отдалечено управление на файлове чрез P2P връзка е подобна на описаната в т. 1.1.3. Разликата е, че отпада нуждата от сървъри-посредници - файловете могат да се изпращат директно към потребителя.

Но осъществяването на връзката се усложнява, тъй като двете крайни точки трябва по някакъв начин да се договорят да започнат връзката, да знаят

как да се откриват един друг, как да преодолеят защитните стени (firewall) и NAT (Network Address Translation).

1.1.4.1. TeamViewer

Teamviewer^[2] е софтуер за отдалечено управление, споделяне на екран и на файлове. Интерфейсът му е интуитивен за потребителите. Безплатен е за лична употреба. Работи на всички популярни операционни системи - Windows, Windows Phone, Mac OS X, iOS, GNU/Linux, Android, Chrome OS, BlackBerry. Възможен е отдалечен достъп дори през уеб браузър.

Във всяка връзка даден потребител може да бъде домакин, тоест да споделя екран, файлове или да дава възможност друг потребител да управлява компютъра, или гост, тоест да вижда екрана и/или управлява устройството на домакина и да приема споделените файлове. Домакинът трябва да предостави на гостите генерирани от Teamviewer идентификатор и парола и те трябва да ги въведат, за да се осъществи връзка.

Teamviewer не е съвсем удобен за управление на файлове. Функционалността за споделяне на файлове изисква да се качват файловете, които трябва да се достъпват. Тъй като тези файлове са налични единствено в рамките на една сесия този вариант за управление на файлове е по-ограничен откъм възможности от решението с облак, описано в т. 1.1.2.

От друга страна стартирането на сесия с отдалечено управление на компютър се доближава много до крайната цел на тази дипломна работа. Визуализира се екранът на домакина и може да се отвори файлов мениджър, с който да се визуализират файловете. Възможностите са много повече - на практика могат да се използват всички функционалности, които заредената на отдалечения компютър операционна система позволява. Недостатъкът е, че информацията за целия екран трябва да се изпраща от домакина към госта, а тя е ирелевантна и коства излишно прехвърляне на данни, ако единствената цел е да се достъпват и управляват файлове.

1.2. Обзор на начините за осъществяване на директна връзка

1.2.1. BitTorrent протокол

BitTorrent^[3] е протокол за споделяне на файлове чрез P2P връзка. За осъществяване на връзка е необходим BitTorrent клиент, например uTorrent, Transmission или друг.

Когато потребител желае да изтегли файл, трябва да се сдобие с .torrent файл, който съдържа метаданни и информация за тракери - сървъри, които намират пиъри, притежаващи въпросния файл.

Като алтернатива на .torrent файловете съществуват и магнитните връзки, които съдържат хешкод, чрез който могат да бъдат открити пиъри, притежаващи файла. Търсенето на пиъри чрез магнитна връзка става чрез DHT (Distributed Hash Table - в превод "разпределена хештаблица"), която всеки пиър притежава и която съдържа информация за налични други пиъри. При свързване на пиър към друг пиър те взаимно обменят информация за пиърите, записани в своите DHT, и по този начин хештаблицата се обогатява. Колкото повече пиъри притежават даден файл, толкова по-бързо файлът може да бъде изтеглен.

Този метод за осъществяване на P2P връзка работи отлично при разпространението на файлове, но има два основни проблема.

Първият недостатък е липсата на контрол върху споделянето на файлове. Всеки, разполагащ с .torrent файл или магнитна връзка може да изтегли съответстващия файл.

Вторият проблем е нуждата от създаване на магнитна връзка или .torrent файл за всяка информация, която трябва да бъде споделена. Който желае да получи информацията трябва първо по някакъв начин да се сдобие с .torrent файла или магнитна връзка.

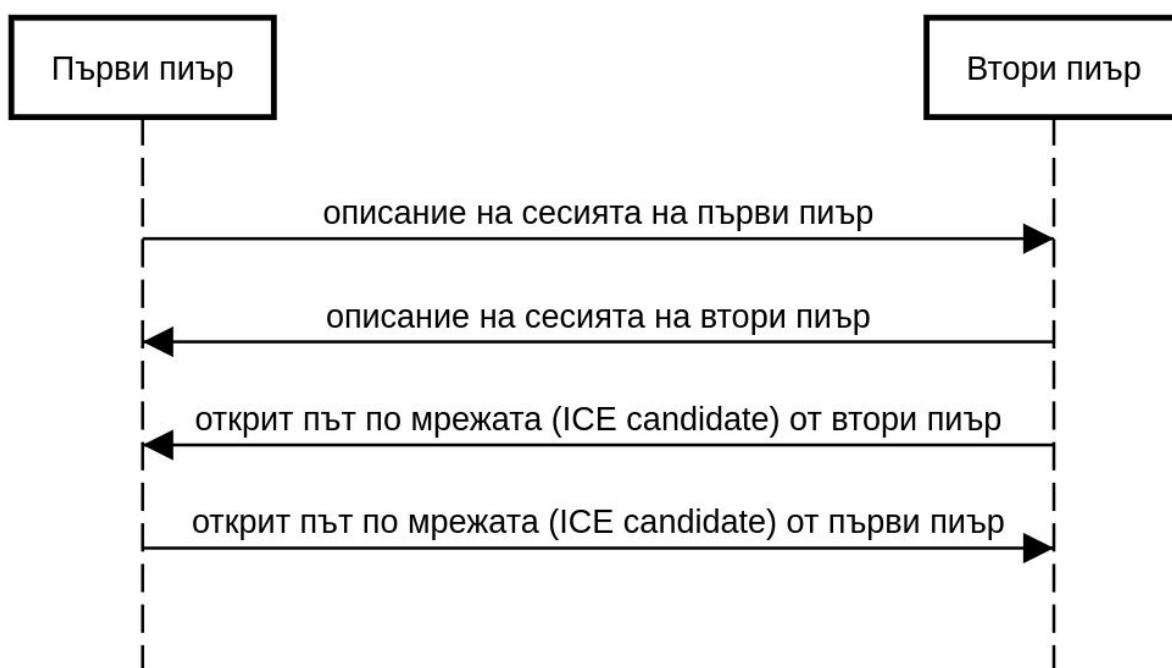
Изброените недостатъци правят BitTorrent протокола твърде неподходящ за целите на тази дипломна работа.

1.2.2. WebRTC

WebRTC^{[4][5]} е проект с отворен код, който предоставя средства за директна връзка между браузъри чрез обмен на аудио, видео или други данни

в реално време. Това е уеб стандарт и за неговата употреба не са нужни приставки или протоколи и технологии със затворен код. За да бъде възможна връзката между двама потребители, единственото софтуерно изискване е да разполагат с модерен уеб браузър.

Инициализацията на връзката се извършва чрез JSEP (JavaScript Session Establishment Protocol - в превод “JavaScript протокол за установяване на сесия”), чиято същност в опростен вид е показана на фигура 1.4.



Фиг. 1.4: Опростен преглед на JavaScript Session Establishment Protocol

Всеки пиър изпраща заявка до STUN (Session Traversal Utilities for NAT) сървър, в отговора на която получава своя публичен IP адрес.

STUN сървърите имат една единствена задача - в отговор на изпратена към тях заявка добавят IP адреса, от който са получили заявката. Това е необходимо, тъй като пиърите не знаят своите публични IP адреси, а трябва по някакъв начин да ги предоставят на отсрещния пиър, за да може той да знае до коя мрежова точка да се осъществи директна връзка.

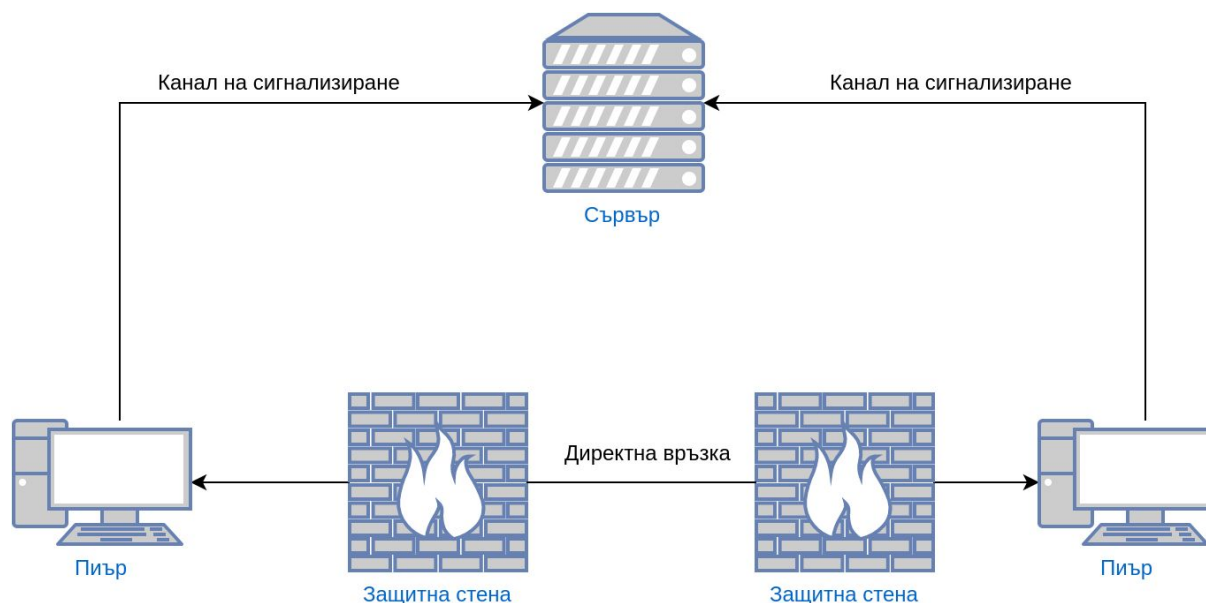
Единият пиър създава описание във вид на SDP (Session Description Protocol), в което се съдържат параметрите на текущата сесия, включително и публичният IP адрес. Генерираното описание се задава като локална конфигурация и се изпраща на другия пиър.

Начинът, по който става изпращането, не е дефиниран от стандарта, но самият процес бива означаван като сигнализиране (signalling).

Когато другият пиър получи описанието, го запазва като чуждо, след което създава ново, съдържащо параметри, съвместими с параметрите в чуждото описание. Резултатът бива запазен като локално описание и пратен на отсрещната страна по канала на сигнализиране. Първият пиър получава описанието и го запазва като чуждо.

След като пиърите обменят конфигурациите си започват да търсят най-добрия път по мрежата между тях чрез ICE (Interactive Connectivity Establishment) протокол^[6]. Системата, която избира най-добрия маршрут, се нарича “ICE Agent” и е вградена в браузърите, а всеки маршрут бива означен като “ICE candidate”.

Всеки открит маршрут от пиър бива изпратен по канала за сигнализиране до другия пиър, където бива предаден на ICE Agent. При намиране на подходящ маршрут P2P връзката се осъществява. За изпращане на файлове може да се създаде канал за данни, позволяващ предаването на произволен вид информация. Протоколът, който се използва е SCTP (Stream Control Transmission Protocol).



Фиг. 1.5: Комуникация между пиъри чрез WebRTC

Ако не може да се осъществи директна връзка между пиърите, би могло да се използва TURN сървър (Traversal Using Relays Around NAT) като резервен вариант. В такъв случай съобщенията между пиърите се предават през TURN сървъра. Налични са имплементации с отворен код на TURN сървър (например “coturn”^[7]).

1.2.3. SMB

SMB^[8] (Server Message Block) е протокол за споделяне на файлове в компютърна мрежа. Други функционалности, които се поддържат, са автентикация, забрана за промяна на файл от много потребители наведнъж, уведомления за промени по файлове, принтиране по мрежата и други. Комуникацията се извършва по модела “клиент-сървър”.

За предаване на информация чрез SMB между компютри с операционна система “Windows” не се изисква инсталиране на допълнителен софтуер. Поради тази причина протоколът е често използван.

1.2.3.1. Samba

Samba^[9] е софтуер с отворен код, който позволява споделяне на файлове и принтери в една мрежа чрез SMB протокол. Възможна е комуникация между различни операционни системи - Windows, Linux, macOS, BSD и други. Конфигурацията е твърде сложна за обикновения потребител - изисква работа с конзолен интерфейс и редактиране на конфигурационни файлове.

1.2.4. SSH

SSH^[10] (Secure Shell - в превод “сигурна обвивка”) е протокол за сигурен отдалечен достъп до сървър.

С цел отдалечено управление на файлове на компютър може да се пусне SSH сървър и чрез SSH клиент посредством конзолен интерфейс може да се изпълняват команди и програми, които позволяват създаване, променяне и изтриване на файлове и много други операции на сървъра.

Такава SSH сесия не позволява прехвърляне на файлове от и към сървъра, но това може да се осъществи чрез SCP (Secure Copy - в превод "сигурно копиране"), използвайки SSH. Примери за SSH клиенти са PuTTY, OpenSSH и Bitvise, за SSH server - OpenSSH, Dropbear и Copssh, а за SCP програми - scp и WinSCP.

Този подход има един сериозен недостатък - необходимо е множество конфигурации, най-вече за стартирането на сървър, до който да може да се осъществява достъп. От потребителя, чиято цел е да предостави достъп до файлове, се изисква да може да инсталира програми, да управлява фонові процеси, да конфигурира защитната стена, да променя файлове с конфигурации и да настройва рутера си. Повечето от тези операции се извършват най-често чрез команди в конзолата, а не посредством графичен интерфейс. Неопитните потребители биват отблъснати от този процес, а тези с познания не биха желали да го изпълняват често.

1.2.5. SFTP

SFTP^[11] (SSH File Transfer Protocol) е протокол за сигурно прехвърляне на файлове. Той изпълнява същата функционалност като FTP (File Transfer Protocol), но по сигурен начин, с имплементирани мерките за сигурност и автентикацията на SSH. Комуникацията се извършва по модел "клиент-сървър".

За да се свърже към сървър, клиентът въвежда потребителско име и парола. Клиентът може да бъде уеб браузър или друга програма, например FileZilla. SFTP поддържа следните операции: отваряне, затваряне, създаване, триене, преименуване на файлове и директории, четене и писане на файлове, разглеждане и промяна на атрибути на файл, прехвърлянето на файл може да се сложи на пауза и други.

Възможно е да се ползва SFTP клиент с графичен интерфейс, който да направи отдалеченото управление на файлове лесно за потребител. Но проблемът, който остава, е сложната за обикновения потребител конфигурация на SFTP сървър.

1.3. Обзор на desktop application development frameworks

1.3.1. Electron

Electron^[12] е библиотека, която предоставя възможност за създаване на мултиплатформени десктоп приложения. Базирана е на уеб браузъра Chromium. Възможни са операции върху файловата система директно от заредена HTML страница посредством средата Node.js. Потребителският интерфейс се изгражда с уеб технологии - CSS, HTML и JavaScript. Сред най-известните приложения, разработени чрез Electron, са Skype, Discord, Slack, Visual Studio Code, Atom, Twitch и други.

1.3.2. Qt

Qt^[13] е популярна технология за разработка на мултиплатформени десктоп приложения с програмния език C++. Известни приложения, използващи Qt са VLC media player, Teamviewer, VirtualBox, Autodesk Maya, Spotify for Linux и много други, както и десктоп средите KDE Plasma, LXQt, Lumina и други. Възможна е разработката на приложения със или без графичен интерфейс. Графичният интерфейс може да бъде консистентен спрямо десктоп средата, в която работи. Qt може да предостави WebRTC функционалност посредством Qt WebEngine.

1.3.3. JavaFX

JavaFX^[14] е платформа за създаване на мултиплатформени десктоп приложения. Използва се програмният език Java, заедно с множеството Java библиотеки. Налични са възможностите за извикване на системни функции и Интернет комуникация, чрез което може да се постигне прочитане на файлове или информация за тях и изпращането им до клиента, който ги е заявил.

JavaFX предоставя богат набор от възможности за създаване на потребителски интерфейс. Високопроизводителен медия енджин позволява вмъкването на аудио и видео съдържание в популярните формати. Възможно е вграждането на уеб съдържание чрез WebView компонент, използващ WebkitHTML.

1.4. Обзор на технологии за уеб разработка

1.4.1. JavaScript-базирани технологии за уеб разработка

JavaScript е слабо типизиран динамичен език за програмиране. Типът на всяка променлива не се декларира, а се определя според зададената ѝ стойност в хода на изпълнение на програмата. При смяна на стойността на променливата е възможно да се смени и нейният тип. Типовете в JavaScript са 7: string, number, boolean, null, undefined, symbol и object.

JavaScript е почти изцяло обектно-ориентиран и използва прототипно-базиран стил. Това означава, че създаването и унаследяването на обекти става от други обекти-прототипи, а не от класове. Към обектите може динамично да се добавят или премахват полета. Стойността на едно поле може да бъде от произволен тип. Поле, чиято стойност е функция, може да бъде наричано “метод на обекта”.

JavaScript поддържа функционален стил на програмиране. Функциите са обекти от първи клас и те могат да бъдат подавани като аргументи на други функции или да бъдат връщани като резултат от изпълнение на функция. Функциите могат да бъдат анонимни. Поддържат се обичайните за функционалния стил на програмиране операции “map”, “reduce” и “filter”.

JavaScript е език за програмиране, който традиционно е намирал приложение единствено в уеб браузърите. Това се променя след появата на Node.js през 2009г.

Node.js^[15] е среда, позволяваща изпълнението на JavaScript код извън рамките на браузъра, например на сървър. Node.js сървърите се отличават с това, че не се използва многонишковост за обработка на паралелно случващи се заявки, а асинхронен модел на опериране.

JavaScript не поддържа многонишковост. Всички операции се извършват в една единствена нишка. Но може да се симулира паралелно изпълнение на задачи чрез модел, базиран на събития. При асинхронно извикване на операция се запазват действия, които да се изпълнят след края на операцията. Не се изчаква операцията да завърши, а продължава изпълнението на следващите от текущия стек на изпълнение. При приключване на асинхронно изпълняващата се операция блокът от запазени за приключването ѝ действия

се нарежда на опашка от други подобни блокове и чака реда си да попадне върху стека на изпълнение.

1.4.1.1. Express.js

Express.js^[16] е популярна технология за разработка на уеб приложения с Node.js. Той е минималистичен и конфигурируем.

Към процеса на обработка на всяка заявка могат да бъдат добавени мидълуер функции, които биват извиквани с “request”, “response” и “next”. “request” е обект, който съдържа цялата информация за направената заявка, а “response” - обект, представляващ отговора, който сървърът връща. “next” е функция, която при извикване изпълнява следващия мидълуер.

Мидълуерите се изпълняват един след друг и могат да изпълняват различни роли - координиране на заявки, автентикация, превръщане на тялото на заявката в JavaScript обект и други.

1.4.1.2. Обзор на JavaScript технологии за изграждане на SPA

Поради естеството на P2P връзките, интерфейсът на потребителското приложение, което достъпва файловете, а също и на провайдъра, не може да бъде реализирано по друг начин освен чрез single page application, тъй като обновяването на информацията става независимо от сървър.

Single page application (SPA) представлява уеб приложение, което се доставя от сървъра като една единствена страница. При интеракция на потребителя с приложението невинаги се прави заявка към сървъра за нова страница - съдържанието на текущата динамично се пренаписва чрез JavaScript.

1.4.1.2.1. Angular

Angular или Angular 2+ е JavaScript фронт енд платформа^[17], поддържана от предимно от Google. Чрез Angular могат да се разработват както уеб страници, така и мобилни приложения.

AngularJS и Angular не са една и съща технология. Първата е създадена през 2010г., а втората - 2016г. Angular използва някои идеи от AngularJS, други

отхвърля и в резултат се получава технология, използваща различна архитектура и синтаксис. Двете технологии не са съвместими една с друга. Angular е по-оптимизирана, с повече функционалности от AngularJS и е по-добрият избор за нови проекти, но другата все още се поддържа и е широко използвана.

Целта на Angular е улесняване на разработката и тестването на приложенията. Angular е догматичен - обикновено има един единствен добър начин даден проблем да бъде решен. Създаването на потребителски интерфейс чрез Angular става по декларативен стил. Към HTML елементите се добавят директиви, които им задават поведение. Архитектурата на едно приложение представлява йерархия от компоненти.

Използва се TypeScript - език, проектиран и разработен от Microsoft, който може да се транспилира до JavaScript. TypeScript е JavaScript с добавено статично типизиране, интерфейси, enum и други функционалности, присъщи на програмните езици C# и Java.

1.4.1.2.2. React

ReactJS^[18] е JavaScript библиотека за изграждане на потребителски интерфейс. Поддържа се от Facebook и общността от уеб разработчици.

React се стреми към бързина и простота. Използва виртуален DOM за визуализация на промените с цел оптимизация.

Подобно на Angular, чрез React се прилага декларативно имплементиране на потребителския интерфейс. Интерфейсът представлява йерархия от компоненти. Тази архитектура дава възможност за лесно надграждане на функционалности в приложението.

Всеки компонент притежава метод "render", в който се задава визуализацията, поле "state", в което се запазва текущото състояние на компонента и поле "props", което представлява свойство или състояние, предадено от родителския компонент. Стойности на "props" и "state" могат да се използват при визуализация, като промените в стойностите им водят до повторно рендериране на компонента, в който се намират.

При разработката с React се препоръчва използването на по-новите стандарти за JavaScript (EcmaScript 6+) в комбинация с JSX. JSX е синтаксис, наподобяващ HTML, чрез който се създават React компоненти.

React е библиотека с неголямо API. Обикновено е необходимо да се прибавят допълнителни модули, например React Router за навигация, за да се постигне по-голяма функционалност.

1.4.1.2.3. Vue.js

Vue.js^[19] е JavaScript фреймуърк за изграждане на потребителски интерфейс в уеб приложения. Vue.js е вдъхновен от React и Angular. Отделните части от интерфейса се разделят на компоненти. Подобно на React, Vue.js използва виртуален DOM за визуализация на промените, чрез което се постига бързодействие. Поддържа се JSX, но не е задължителна употребата му - може да се работи с темплейтен синтаксис. Както при Angular, във Vue.js се използват директиви, които задават поведението на елементите, но Vue.js има по-прост дизайн и API от Angular и е по-конфигурируем.

1.4.2. Ruby on Rails

Ruby on Rails^[20] е технология за разработка на уеб приложения по архитектурата MVC - Model-View-Controller с програмния език Ruby. MVC е шаблон за дизайн, при който приложението е разделено на три взаимно свързани слоя - модел, изглед и контролер.

Ruby on Rails прилага принципа “конвенция над конфигурация”, с което определени решения за архитектурата на приложението са взети предварително. Спестява се време в разработката, тъй като често извършваните действия могат да се изпълняват автоматично. Това прави тази технология изключително подходяща за бързо създаване на прототипи. Разработката според конвенции води до писане на по-качествен код. Но скритото извършване на дейности невинаги е в полза на програмиста - например при търсене на грешки.

Друг принцип, който тази технология прилага, е DRY - “Don’t repeat yourself” (в превод - “Не се повтаряй”). Всяка информация, част от алгоритъма,

е дефинирана на едно единствено място, което значително улеснява поддръжката на приложението, защото промяна от едно естество се извършва само веднъж.

Ruby on Rails включва в себе си всичко, което е необходимо за разработката на уеб приложение - например поддръжка за уеб сокети, програмен модел за управление на бази данни (ORM) и изпращане на имейли.

1.4.3. Spring

Spring Framework^[21] е популярна технология с отворен код, която улеснява разработката на уеб приложения, работещи на Java платформа. Използва се езика Java, Groovy или Kotlin.

Spring Framework не е ограничаващ, дава се свобода на разработчика да вземе най-подходящите решения относно имплементацията на своето приложение. Друга част от философията му е всяка нова версия да бъде съвместима с предишните.

Spring Framework прилага аспектно-ориентирано програмиране. То решава проблема с оплитане на две или повече различни дейности, например необходимостта от автентикация преди всяко едно от множество действия. Целта е да се постигне модулност и да се избегнат проблемите, свързани с повторение на един и същи код.

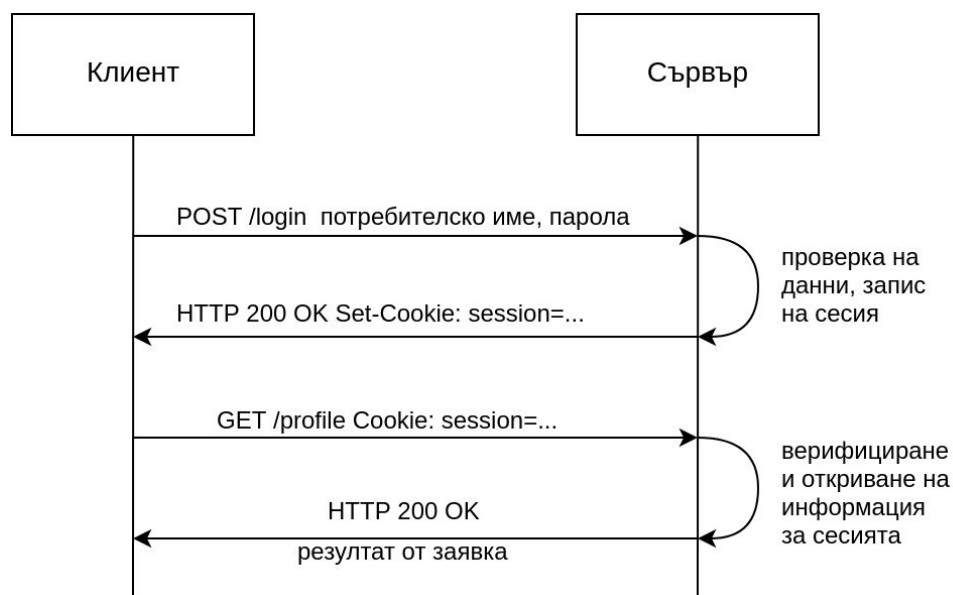
В Spring Framework се съдържа всичко необходимо за изграждането на едно уеб приложение.

1.5. Обзор на начини за автентикация

HTTP протоколът няма състояние. Това означава, че при всяка заявка за достъп до поверителна информация потребителят трябва да се автентичира. Но от него не бива да се изисква да въвежда своите данни при всяка заявка. Това занимание може не само да отегчи потребителя, но и да изложи на опасност от кражба данните за достъп до неговия профил. Съществуват различни решения на този проблем.

1.5.1. Бисквитки

Бисквитките (от англ. ез. “cookies”) представляват малко количество информация, чрез която клиент може със всяка заявка да се автентичира пред сървър.



Фиг. 1.6: Принцип на работа на автентикация с бисквитки

Принципът на работа на автентикация с бисквитки е показан на фигура 1.6. При извършване на определена операция, например вход в акаунт, потребител изпраща своите данни за автентикация. Сървърът запазва информация за сесията в база данни и в отговора на потребителската заявка предоставя “бисквитка”, съдържаща идентификатор на сесията. Потребителят запазва бисквитката и при следващи заявки до сървъра я използва, за да се автентичира.

Информацията, която може да се запази в бисквитка, е с много ограничен размер. Стандартът изисква максималният размер на бисквитките за всеки уеб браузър да бъде поне 4 килобайта. Това означава, че за да функционира правилно на всички уеб браузъри, приложението трябва да ограничи размера на създаваните бисквитки до 4 килобайта.

Подходът с бисквитките също изисква да се запазват сесийни данни на сървъра.

1.5.2. JSON Web Token

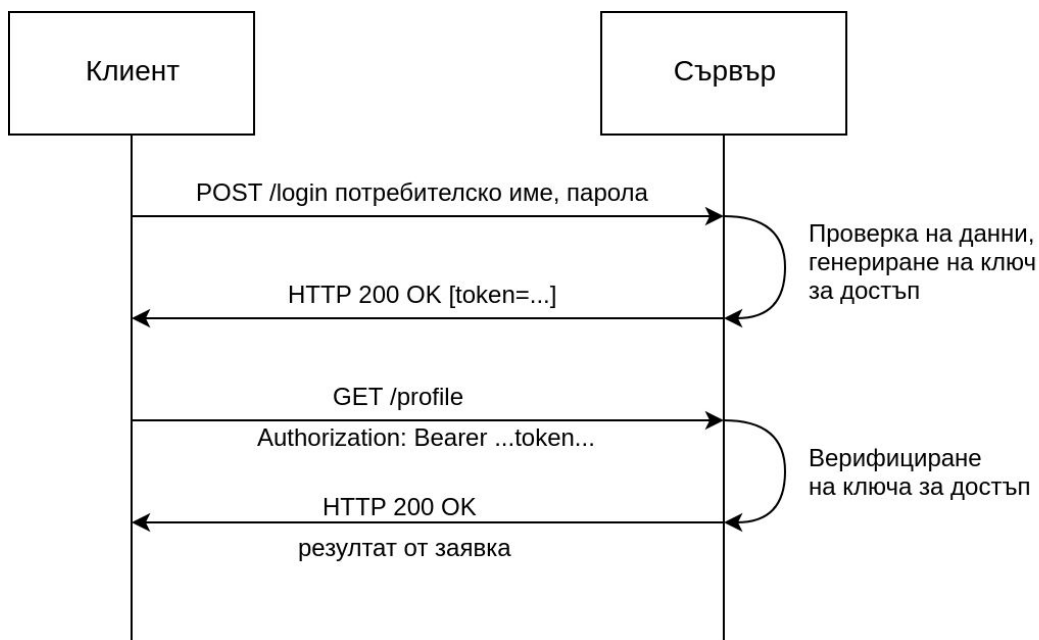
JSON Web Token (съкратено “JWT”) е стандарт за сигурно предаване на информация във вид на JSON обект, който се кодира като base64 символен низ. Информацията може да бъде подписана и верифицирана, използвайки таен ключ или двойка частен и публичен ключове.

За разлика от бисквитките, JWT няма ограничение за размер. Също така сървърът няма нужда да съхранява сесийни данни. Всички необходими данни за сесията могат да бъдат запазени в JWT ключа за достъп.

Потребителят изпраща своите данни за автентикация, сървърът връща JSON Web Token, който представлява ключ за достъп, и клиентът го запазва. При нужда от автентикация пред сървъра клиентът изпраща ключа за достъп. Сървърът го верифицира, декодира го и според запазената в него информация идентифицира клиента. (фиг. 1.7.)

JSON Web Token подходът има няколко недостатъка. Веднъж издаден, ключът за достъп остава валиден, докато не изтече периода на валидност, зададен при издаването му. Не е възможно преждевременно да се инвалидира.

Проблем е, ако ключът за достъп е с голям размер, тъй като се изпраща при всяка заявка до сървъра и това може да предизвика значителен по обем трафик на данни.



Фиг. 1.7: Начин на автентикация с JSON Web Token

ВТОРА ГЛАВА

2.1. Функционални изисквания

- Осъществяване на директна връзка между провайдер и клиент
- Съвързване на много клиенти към един провайдер
- Информация за свързаните към провайдер клиенти
- Система за права на достъп до провайдер
- Визуализация на файловете в текуща директория
- Отваряне на поддиректории от клиента
- Действия с файлове:
 - Изтегляне
 - Копиране
 - Преместване
 - Изтриване
 - Качване

2.2. Избор на технологии

2.2.1. WebRTC

WebRTC е набор от протоколи за комуникация, които позволяват осъществяване на директна връзка в реално време между браузъри. Той използва свободни протоколи и технологии и е уеб стандарт, който е имплементиран от повечето съвременни браузъри. Начинът, по който се осъществява P2P връзката, не усложнява потребителския интерфейс, а след нейното установяване данните се предават в криптиран вид.

2.2.2. JavaScript

JavaScript е един от основните три компонента, използвани за изграждането на уеб приложения. Той поддържа много парадигми на програмиране. Наличието на асинхронен модел на изпълнение, прототипно-базираното обектно-ориентирано програмиране, функциите като обекти от първи клас и слабото типизиране го правят изключително мощен и лесен за употреба.

2.2.3. Node.js

Node.js е технология, позволяваща създаването на сървърни и десктоп приложения с програмния език JavaScript, който традиционно е намирал приложение единствено за изграждане на динамични уеб страници. Чрез употребата на Node.js става възможно използването на един и същи език за програмиране на клиентската и сървърната част.

Тази технология използва асинхронен модел на опериране, при който времето за изчакване да завърши една операция бива ефективно използвано за изпълнение на друга операция. Това означава по-висока производителност при наличие на голямо количество заявки.

Чрез пакетния мениджър “npm” за Node.js лесно се достъпва регистър от готови решения на често срещани проблеми, което ускорява процеса на разработка.

2.2.4. PostgreSQL

PostgreSQL е определяна като богата на функционалности СУРБД с отворен код, която стриктно се придържа към SQL стандарта^[23]. Тя добре се интегрира с Node.js чрез npm модула “pg”, който има отлична поддръжка. MySQL е добра алтернатива, с която имам предишен опит. Причината да не я избира бе желанието ми да се запозная с друга СУБД, т.е. с PostgreSQL.

2.2.5. Redis

Redis^[24] е система за управление на данни във вид на ключ-стойност. Тези данни се съхраняват в оперативната памет, което прави Redis много бърза. Подходяща е за случаи, при които е изключително важна скоростта на работа на СУБД и/или случаи, при които данните не е нужно да се съхраняват вечно. Redis също поддържа функционалности за предаване на съобщения.

За целите на настоящата дипломна работа Redis се използва за съхранение на сесийни данни, които не е подходящо да бъдат записвани на дисково устройство. Redis също може да влезе в употреба при разширяване на мащабите на системата. Когато клиент и провайдер се свържат чрез уеб сокет до различни сървъри, използвайки Redis, може да се установи предаване на данни между сървърите и съответно между клиент и провайдер^[25].

2.2.6. Electron

Electron е фреймуърк за създаване на мултиплатформени десктоп приложения чрез уеб технологии - HTML, CSS и JavaScript. Използването на един и същи набор от средства за изграждане на уеб страницата и десктоп приложението улеснява разработката на цялостната система. Electron използва V8 енджинът на браузъра Chromium, което означава, че поддържа WebRTC API и може без допълнителни плъгини да се осъществява P2P връзка с клиента. Голямата популярност и това, че редовно използвани от мен десктоп приложения - Visual Studio Code, Slack и Skype, са създадени чрез технологията Electron, определиха избора ми.

2.2.7. React

React е популярна JavaScript библиотека за изграждане на динамичен потребителски интерфейс в уеб приложение. Създаването на уеб приложение става по елегантен декларативен начин. Един от принципите на React е създаване на компоненти, които могат да бъдат използвани многократно. Използва се виртуално DOM дърво, чрез което се оптимизира процеса на визуализация на промени по съдържанието. React не е тежка библиотека и нейният API е сравнително прост. Сред функционалностите, които липсват, могат да бъдат добавени като модули само тези, които са необходими.

2.2.8. JSON Web Token

JSON Web Token е стандарт за създаване на ключове за достъп. Той позволява голяма конфигурируемост на начина, по който се извършва автентикацията. Притежава степен на сигурност и е лесен за употреба. Подходящ е за уеб приложения, базирани на една страница (single-page application). Особеност в настоящата разработка е автентизирането на връзка на клиент към провайдер, осъществена посредством сървър. Това изисква ръчно дефиниране на подход, по който да се извърши автентикацията, и JWT позволява именно това.

2.3. Описание на алгоритъма

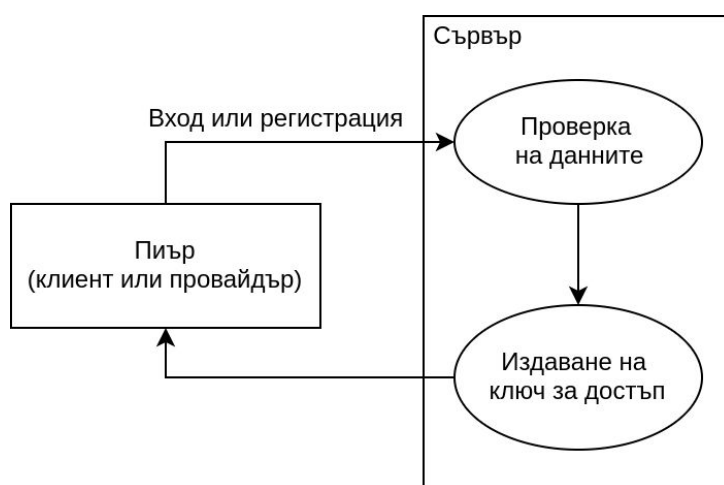
Приложението се състои условно от три части: провайдер, сървър и уеб интерфейс, изпратен от сървъра. Уеб интерфейсът ще бъде назоваван като “клиент”.

2.3.1. Сървър

Основната задача на сървъра е да установява директната връзка между клиент и провайдер.

За целта е нужна автентикация на всяка от двете страни. Сървърът запазва данни от регистрационни форми в база данни и при последващи опити за вход сравнява данните от форми за вход с тези в базата данни.

При вход на провайдър или клиент сървърът издава специален ключ за достъп, чрез който провайдърът или клиентът може да достъпва други функционалности от системата. Ключът е необходим например при промяна на данните за акаунта, но най-честият случай, при който се изисква, е установяване на директна връзка между клиент и провайдър.



Фиг. 2.1: Аутентикация на пиър

За установяване на връзката между провайдър и клиент е нужно предварително всеки от тях да установи връзка към сървъра чрез уеб сокет. При свързването се изпраща ключа за достъп и ако е валиден данните за сесията се запазват и се проверява дали има записани данни за другата страна - провайдър или клиент, тоест дали другият пиър е на линия. Ако това е така между провайдъра и клиента посредством сървъра започва обмяна на параметри по JSEP протокол, описан в т. 1.2.2.

След като се установи директна връзка между провайдър и клиент, уеб сокет връзките сървър-провайдър и сървър-клиент остават активни с цел възможност за рестартиране на P2P връзката, когато се наложи, и отчитане на момента, когато сесийните данни трябва да се премахнат - когато уеб сокет връзката се прекрати. Също така провайдърът трябва да бъде винаги готов да получи съобщение от сървъра през уеб сокет за наличието на клиент, който желае да се свърже към провайдъра.

2.3.2. Провайдер

Основната функция на провайдера е да сканира директории и да изпълнява инструкции от свързани към него клиенти.

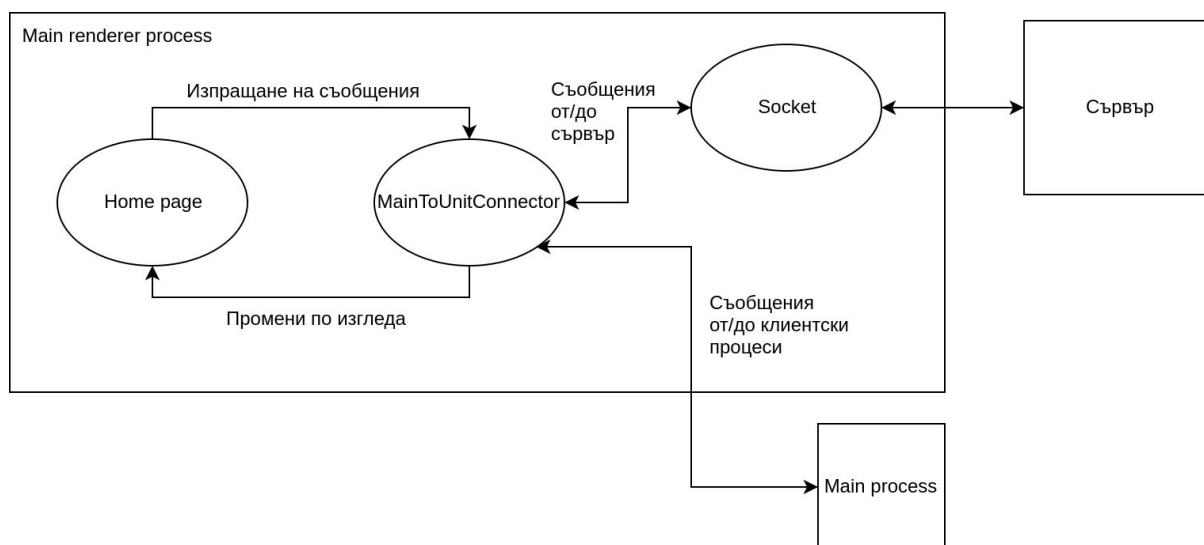
2.3.2.1. Поддръжка на много клиенти, свързани към един провайдер

За всеки свързан към провайдер клиент се създава нова нишка или нов процес. Това се извършва с цел ефективно използване на ресурсите на компютъра, на който работи провайдерът.

Текущото приложение използва JavaScript библиотеката Electron. Чрез този набор от технологии единствено могат да се създават процеси, но не и нишки, затова в следващите обяснения ще пропусна споменаването им.

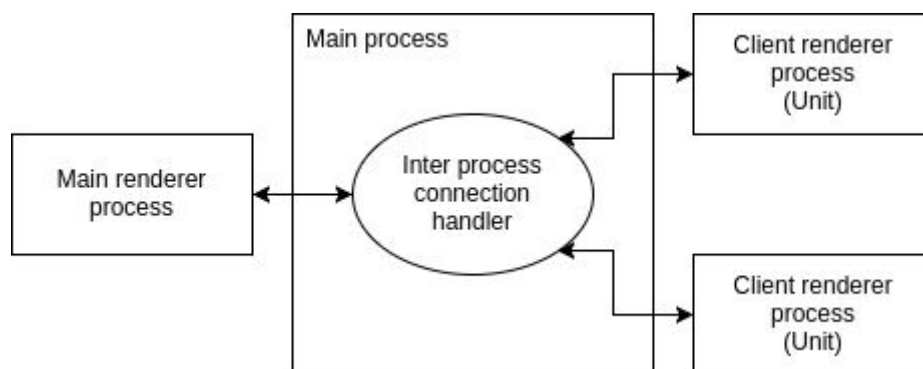
В един процес се извършва комуникацията със сървъра чрез уеб сокети, както и визуализацията на графичен интерфейс. Този процес ще бъде назоваван като “основен рендериращ процес”. Неговата структура е показана на фигура 2.2.

Определението “рендериращ” идва от двата вида процеси във всяко Electron приложение - “renderer” и “main”. Клиентските процеси също са рендериращи, но са скрити. Затова този рендериращ процес, в който е разположен потребителският интерфейс, се означава като “основен”.



Фиг. 2.2: Структура на основен рендериращ процес

Между основния рендериращ процес и клиентските процеси се осъществява междупроцесова комуникация (inter process communication), чрез която се обменят данни. Пример за нужда от обмяна е когато чрез уеб сокет в основния процес пристигне информация, отнасяща се за даден клиент, и тя трябва да се препрати до съответстващия на този клиент процес. Междупроцесовата комуникация се осъществява посредством главен процес, чиято структура е показана на фигура 2.3.



Фиг. 2.3: Структура на главен процес

2.3.2.2. Функционалности в главния рендериращ процес

Главният ренериращ процес съдържа форма за избор на директория, от която да започне сканирането, уеб сокет връзка към сървъра, информация за всички клиенти, начин за обмен на данни с техните процеси и начин за контрол на техните права.

2.3.2.3. Функционалности в клиентски процеси

Клиентските процеси се състоят от начини за комуникация с главния процес и комуникация с клиент, методи за сканиране на директории и обработка на команди, получени от клиент.

2.3.3. Клиент

Клиентът представлява уеб апликация, изпълнявана от браузър, посредством която може да се осъществи връзка към провайдър и да се изпращат команди за отдалечено управление на файловете.

2.3.4. Съобщения от клиент към провайдер

2.3.4.1. “openDirectory”

Изпратеното чрез WebRTC съобщение “openDirectory” от клиент към провайдер носи смисъла на команда да бъде отворена дадена директория.

Изпратеното съобщение към провайдера съдържа релативния път до директорията, чието съдържание трябва да бъде достъпно. Пътят до директорията е релативен спрямо пътя на избраната за начална точка на сканирането директория. При получаване на съобщението провайдерът запазва пътя до новата текуща директория, сканира съдържанието ѝ и изпраща към клиента метаданни за всеки един файл в нея. На клиентската страница се премахва съдържанието от предишната директория и се визуализира съдържанието на новата.

2.3.4.2. “downloadFile”

Съобщението “downloadFile” представлява заявка от страна на клиента да изтегли файл. Съобщението съдържа релативния път до избран за изтегляне файл. В резултат провайдерът го изпраща на парчета в отделни съобщения. Клиентът събира парчетата и добавя размера на всяко към един сбор. Когато сборът стане равен на размера на файла, това означава, че файлът е получен.

2.3.4.3. “uploadFile”

Когато клиентът избере да качи файл от своя компютър, на провайдера се изпраща съобщение “uploadFile” до провайдера, съдържащо метаданни за файла. Провайдерът запазва метаданните и се подготвя за предстоящото качване - създава поток за писане, след което уведомява клиента, че е готов, чрез “readyForFile” съобщение. Когато клиентът го получи, започва да изпраща файла на парчета. Провайдерът ги събира и когато сумата от размерите на всички получени парчета достигне големината на файла, се разбира, че файлът успешно е качен.

2.3.4.4. “copyFile”

Съобщението “copyFile” се изпраща от клиента, когато потребителят избере да копира файл. То съдържа пътя до файла. Когато провайдерът го получи, създава копие на избрания файл в текущо отворената от клиента директория.

2.3.4.5. “moveFile”

Съобщението “moveFile” се изпраща от клиента, когато потребителят избере да премести файл. В съобщението се съдържа пътят до файла. Провайдерът след като получи файла, съответстващ на пътя, го премества в текущо отворената от клиента директория.

2.3.4.6. “deleteFile”

Съобщението “deleteFile”, съдържащо път до файл, се изпраща на провайдера, когато потребителят избере да изтрие файл. Провайдерът изпраща в кошчето избрания файл.

2.3.5. Съобщения от провайдер към клиент

Освен съобщения от клиент към провайдер се изпращат и такива в противоположна посока, но те не са директен резултат от потребителско действие, а следствие от някакво събитие. Клиентът получава съобщение за успешно извършена операция, съдържащо данни, които могат да бъдат представени в интерфейса, или уведомление за наличие на грешка.

2.3.5.1. “add”, “addDir”, “change”, “unlink” и “unlinkDir”

По време на сканиране на съдържанието на дадена директория при открит файл се събират метаданните му и се изпраща съобщение “add”, а при открита папка - “addDir”. При клиента се запазва или визуализира новия файл или папка.

След приключване на сканирането клиентският процес на провайдера следи за промени по съдържанието в директорията.

При добавяне на нов файл се изпраща съобщение “add” с метаданните на файла, а при добавяне на папка - “addDir”.

При открита промяна в даден файл се изпраща съобщение “change” с новите метаданни. Клиентът премахва стария вариант на файла и добавя новия.

При изтриване на папка или файл се изпраща съобщение съответно “unlinkDir” или “unlink”, съдържащо пътя до вече несъществуващия файл. Клиентът го получава и чрез получения път открива кой файл да премахне.

2.3.5.2. “newScan”

Когато клиентът заяви да отвори нова директория или когато бъде променена началната точка на достъп на провайдъра, се извършва ново сканиране. Клиентът трябва да бъде уведомен за това. За тази цел провайдърът изпраща съобщение “newScan” до клиента, а той реагира като изчиства предишните данни за файлове.

2.3.6. Контрол на правата

Потребителският интерфейс на провайдъра позволява контрол на правата, с които свързващите се клиенти разполагат. Възможни са три режима на достъп, които се определят от два булеви параметъра - “readable” и “writable” (право за четене и право за писане).

Ако и двата са със стойност логическо отрицание, то следва, че е забранен всякакъв достъп до провайдъра.

Ако само “readable” е логическа истина, достъпът до провайдъра е разрешен, но са позволени само клиентски команди, свързани с четене на данни. Това са “openDirectory” и “downloadFile”.

Когато и двата параметъра са зададени като логическа истина е позволено четене и писане. Освен командите “openDirectory” и “downloadFile” са разрешени и “uploadFile”, “copyFile”, “moveFile” и “deleteFile”.

Всеки провайдър притежава една двойка параметри “readable” и “writable”. В неговия потребителски интерфейс чрез ключове могат да се задават стойностите на параметрите. При всяка промяна се изпраща заявка до

сървъра с новите права за достъп. Сървърът променя в базата данни записа за този провайдер с новите стойности на “readable” и “writable”.

Съществува възможност за задаване на индивидуални права на всеки клиент. Всички свързани клиенти и техните привилегии се визуализират в провайдъра. Промяната на правата се изпраща до сървъра и се запазва.

Режимът на достъп до провайдер се определя при свързването на клиент. Ако клиентът притежава индивидуално зададени права за достъп, те са определящи, в противен случай биват използвани стандартните права за достъп до провайдъра.

След осъществяване на връзката всякакви промени по правата за достъп не оказват влияние върху сесията на клиента. За да влязат в сила, той трябва да прекъсне връзката и да се свърже наново.

Ако е необходимо текущият режим на достъп за даден клиент да бъде прекратен незабавно, връзката му може да се прекъсне от потребителския интерфейс на провайдъра. При този случай до сървъра се изпраща заявка за инвалидация на ключа за достъп на клиента.

2.4. Структура на базата данни

Използват се две бази данни. Едната е за съхранение на постоянни данни, а другата - за сесийни.

2.4.1. Структура на базата данни за дълготрайна информация

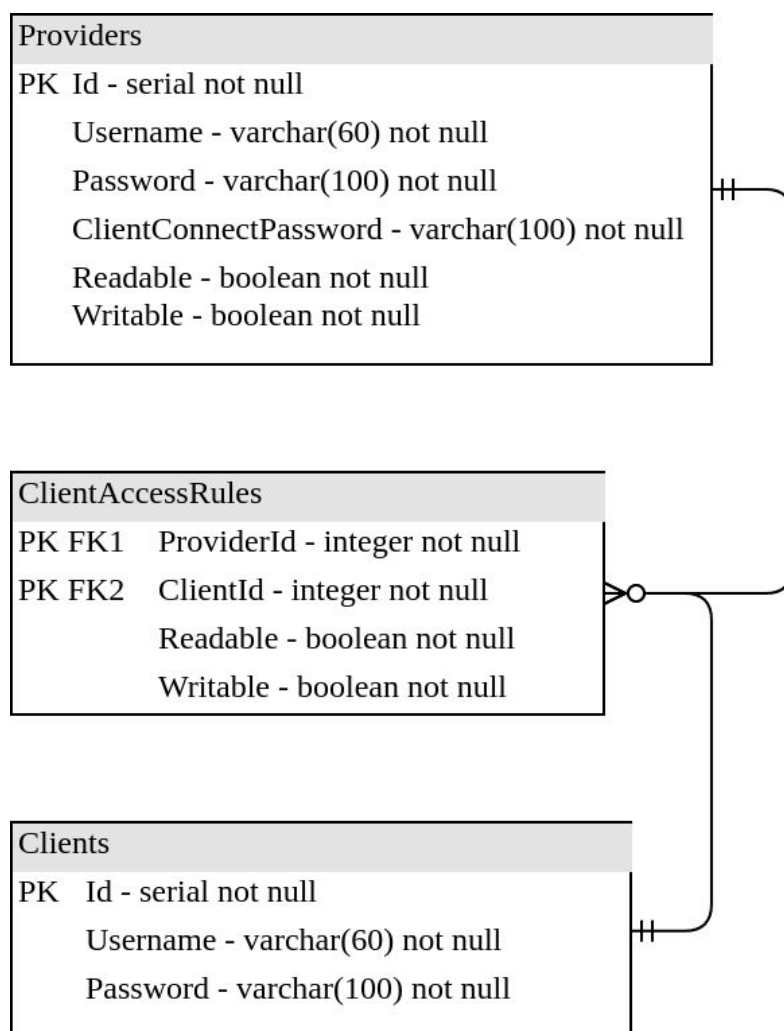
Информацията, която се запазва дълготрайно, се структурира в три таблици: “Providers”, “Clients” и “ClientAccessRules”.

В “Providers” се съхраняват данните за провайдерите - потребителско име, парола на провайдер, парола за свързване на клиент към провайдер и права за достъп до провайдъра по подразбиране. Тези права по подразбиране се използват при свързване на клиент, когато за него не са дефинирани индивидуални права. Такива индивидуални права се запазват в таблицата “ClientAccessRules”.

“ClientAccessRules” съдържа задължителна двойка идентификатори на клиент и провайдер, както и правата, които този клиент притежава, когато се

свърже към провайдер. Връзката Providers-ClientAccessRules е one-to-many: относно един провайдер могат да се задават много клиентски права, а едно клиентско право задължително се отнася към един единствен провайдер.

Третата таблица е “Clients”, която съдържа потребителско име и парола на клиент. Между Clients и ClientAccessRules има връзка one-to-many.



Фиг. 2.4: Структура на база данни за дълготрайна информация

2.4.2. Описание на базата данни за съхранение на сесийна информация

Базата данни за съхранение на сесийна информация няма структура - всеки запис представлява двойка ключ-стойност. При такава архитектура и при нужда от съхранение на различни по смисъл данни се използва конвенция за

съставяне на ключа. Тази, която настоящото приложение използва, представлява идентификация, след това двуетоичие и накрая контекста.

Пример за ключ е “example:created”. В случая идентификацията е име на провайдер - “example”, следва двуетоичие и накрая “created” - контекст или смисъл на записа. “created” означава, че стойността е датата, в която провайдерът е създаден.

С цел по-лесно обяснение ще използвам синтаксиса на JavaScript за шаблонни литерали и вмъкване на символен низ в тях. Примерът “example:created” би се представил по следния начин: ``${providerName}:created``, където вмъкваме стойността на променливата “providerName” - името на провайдъра, в шаблонния литерал.

Когато клиент се свърже към провайдер се създава нова клиентска сесия. На ключ “clients” отговаря стойност, която е множество от сокет идентификации на всички свързани клиенти. Към множеството се добавя сокет идентификацията на новия клиент. На ключ ``${providerName}:clientSocketIds``, където “providerName” е името на провайдъра, към който клиентът иска да се свърже, отговаря стойност, която е множество от сокет идентификации на клиентите, предварително свързали се към този провайдер. Към двете множества се прибавя сокет идентификацията на новия клиент. Създават се две нови двойки ключ-стойност: ``${socketId}:providerName`` със стойност името на избрания провайдер и ``${socketId}:created`` със стойност датата на създаване на сесията (текущата дата), където “socketId” е сокет идентификацията на новия клиент. При прекратяване на сесията същите данни се изтриват.

На ключ “providers” съответства стойност, която е множество от имената на всички свързани провайдери. Когато се създава нова провайдер сесия към множеството се добавя името на новия провайдер. Създават се три нови двойки ключ-стойности. Първата е ключ ``${socketId}:providerName``, където socketId е сокет идентификацията на новия провайдер, със стойност името на новия провайдер. Втората и третата двойка са ``${providerName}:socketId`` и ``${providerName}:created``, чиито стойности представляват съответно сокет идентификацията на провайдъра и датата на създаване на сесията (текущата

дата), а “providerName” е името на провайдъра. Същите данни се изтриват при приключване на сесията.

Съществува още една двойка ключ-стойност: “invalidatedTokens” със стойност сортирано множество от ключове за достъп, които са отбелязани като невалидни. Всеки ключ автоматично става невалиден след определен при издаването му период от време, но при необходимост от предварителна инвалидация ключът за достъп се запазва в това множество - например когато клиент или провайдър излезе от профила си или когато клиентът избере да спре връзката си към провайдър. При всеки достъп до “invalidatedTokens” се премахват от него ключовете за достъп с изтекли периоди на активност.

ТРЕТА ГЛАВА

3.1. Структура на системата

3.1.1. Структура на сървърното приложение

- bin/www - началната точка на приложението
- config - папка, съдържаща конфигурации
- db
 - postgres - съдържа файлове със заявки към PostgreSQL базата данни
 - redis - съдържа файлове със заявки към Redis базата данни
- modules - папка, съдържаща модули
- public - директория, чието съдържание е публично за клиента
 - js - съдържа JavaScript файловете, директно използвани от клиента
 - stylesheets - съдържа CSS файловете, използвани от клиента
- routes - директория с файлове, в които се извършва обработката на заявки към сървъра
- scripts - папка със скриптове
- sockets - съдържа два файла, които представляват клиентска и сървърна обработка на информация, изпратена чрез уеб сокети
- store - съдържа логика, отнасяща се за създаване и промяна на състоянието на клиентското приложение

- views - директория, в която се съдържат изгледите на клиентското приложение
 - components - директория, съхраняваща React компоненти,,
 - containers - директория, в която е разположен главният компонент, контейнер на цялото приложение
 - pages - папка, в която са разположени React компоненти, които представляват отделни страници.
- app.js - файл, в който се конфигурира Express.

3.1.2. Структура на клиентското приложение

Клиентското приложение се състои от шест страници:

- IndexPage - начална страница, представяща приложението
- RegisterPage - страница с регистрационна форма
- LoginPage - страница с форма за вход в профил
- ConnectPage - страница с форма за свързване към провайдер
- HomePage - страницата, в която се осъществява самото свързване към провайдер с възможности за управление на файловете
- AccountSettingsPage - позволява промяна на парола и изтриване на акаунта

3.1.3. Структура на провайдер

- app - директорията, съдържаща изходния код на приложението
 - connections - поддиректория, в която се намира осъществяването на комуникация със сървъра чрез уеб сокети и комуникация на основния процес на провайдъра с клиентските процеси
 - store - съдържа създаването и промяната на състоянието на потребителския интерфейс
 - units - логика, свързана с клиентски процеси
 - connections - съдържа осъществяване на връзка с основния процес и осъществяване на директна връзка с клиента
 - modules - модули, свързани с работата на клиентския процес

- views - потребителски интерфейс на провайдъра
 - components - React компоненти
 - containers - директория, в която е разположен основния компонент, в който се съдържа цялото приложение
 - pages - директория със страниците, които приложението има
 - LoginPage - страница с форма за вход
 - RegisterPage - страница с форма за регистрация
 - HomePage - страница, в която може да се конфигурира провайдъра и да се следят свързаните клиенти
 - AccountSettingsPage - съдържа форма за промяна на парола или изтриване на акаунта
- public/js - папка, в която се намират използваните от приложението JavaScript файлове в транспилиран вариант
- index.html - страницата, която се отваря при стартиране на провайдъра
- ipc-handler.js - файл, чрез който се осъществява комуникацията между процесите на провайдъра
- main.js - началната точка, от която стартира приложението

3.2. Реализация на автентикация

При всички операции, свързани с автентикация и описани в т. 3.2., е налична една и съща група от действия, но имплементирани по различен начин - изпращане на форма, обработка на заявка до сървър, заявка до база данни и работа с ключ за достъп. В т. 3.2.1.1. е представена имплементацията на тези действия при регистрация на клиент. За останалите операции това е спестено - предоставя се препратка към файл от изходния код, в който се съдържа направената имплементация, и са представяни само най-особените моменти от нея.

3.2.1. Автентикация на клиент

3.2.1.1. Регистрация на клиент

Нови клиентски акаунти могат да се създават чрез форма за регистрация, която може да се отваря от потребител, който не е влязъл в профила си. При изпращане на данни чрез нея се изпълнява функция "handleSubmit" (фиг. 3.1).

```
handleSubmit(form) {  
  this.setState({ loading: true });  
  if (!(form.username && form.password)) {  
    this.setState({  
      loading: false,  
      hasFormErrors: true,  
      formErrors: ["empty"]  
    });  
    return;  
  }  
  
  if (form.password !== form.confirmPassword) {  
    this.setState({  
      loading: false,  
      hasFormErrors: true,  
      formErrors: ["match"]  
    });  
    return;  
  }  
  
  const formData = {  
    username: form.username,  
    password: form.password  
  };  
  
  fetch("/register", {  
    method: "POST",  
    headers: { "Content-Type": "application/x-www-form-urlencoded", },  
    body: formurlencoded(formData)  
  }).then(response => {  
    this.setState({ loading: false });  
    if (response.status === 201) {  
      return response.json();  
    } else {  
      throw response;  
    }  
  }).then(json => {  
    this.props.loginClient(json);  
    this.props.history.push("/connect");  
  });  
}
```

```

    }).catch(error => {
      this.setState({
        hasFormErrors: true,
        formErrors: [error.status]
      });
    });
  });
}

```

Фиг. 3.1: Изпращане на форма за регистрация на клиент

(изходен код: “datastreamer-server/src/views/pages/register-page.jsx”)

Правилният формат на данните се проверява както от клиентската страна, така и от сървърната. Те се изпращат на сървъра чрез POST заявка до крайната точка “/register”.

```

router.post("/register", [
  body("username").exists().trim().isLength({ min: 5, max: 60 }),
  body("password").exists().custom(passwordCheck)
], (req, res, next) => {
  if (!validationResult(req).isEmpty()) {
    res.status(400).end();
  } else {
    register(req.body.username.toLowerCase(),
    req.body.password).then(response => {
      if (response.success) {
        res.status(201).send({
          token: response.token,
          username: response.username
        });
      } else {
        res.status(412).end();
      }
    }).catch(error => {
      log.error(error);
      res.status(500).end();
    });
  }
});

```

Фиг. 3.2: Сървърна обработка на заявка за регистрация на клиент

(изходен код: “datastreamer-server/src/routes/client.js”)

Следва обработка на заявката (фиг. 3.2). Извиква се функцията “register”, отнасяща се за клиент и представена във фигура 3.3. Прави се заявка до

базата данни с извикване на създадената по начин, описан във фигура 3.4, функция “create_client”.

```
async function register(username, password) {
  try {
    const response = await db.query(`SELECT Username
    FROM create_client($1, $2)
    AS (Username VARCHAR);`, [username, password]);
    if (response.rows.length <= 0 || response.rows.length > 1) {
      throw `Invalid response from create_client: response.rows.length:
    ${response.rows.length}`;
    }
    if (!response.rows[0].username) {
      return { success: false };
    } else {
      log.info("New client successfully created...");
      log.verbose(`Username: ${response.rows[0].username}`);
      const token = await signClientToken(response.rows[0].username);
      return {
        success: true,
        token,
        username: response.rows[0].username
      };
    }
  } catch (error) {
    log.error("In register client:");
    throw error;
  }
}
```

Фиг. 3.3: Заявка до базата данни за регистрация на клиент

(изходен код: “datastreamer-server/src/db/postgres/client.js”)

```
CREATE OR REPLACE FUNCTION create_client(client_username varchar,
client_password varchar)
RETURNS RECORD AS $$
DECLARE
  client RECORD;
BEGIN
  SELECT INTO client * FROM Clients
  WHERE Username = client_username;
  IF FOUND THEN
    RETURN NULL;
  ELSE
    INSERT INTO Clients (Username, Password)
    VALUES (client_username, crypt(client_password, gen_salt('bf',
8)))
```



```

        RETURNING Clients.Username INTO client;
    RETURN client;
END IF;
END;$$ LANGUAGE plpgsql;

```

Фиг. 3.4. Създаване на нов клиент в базата данни^[27]

(изходен код: “datastreamer-server/src/scripts/db-init/init.sql”)

```

const issuer = "datastreamer-server";
const algorithm = "RS256";
const clientTokenSubject = "client";
const expiresIn = 60 * 60; // 1 hour

[...]

async function signClientToken(username) {
  try {
    const privateKey = await getPrivateKey();
    return jwt.signAsync({ username }, privateKey, {
      issuer,
      subject: clientTokenSubject,
      algorithm,
      expiresIn
    });
  } catch (error) {
    log.error("In sign client token:");
    throw error;
  }
}

```

Фиг. 3.5: Създаване на нов ключ за достъп за клиент^[28]

(изходен код: “datastreamer-server/src/modules/token-actions.js”)

Функцията “create_client” приема потребителското име и паролата и създава нов клиент, ако не съществува такъв със същото потребителско име. Паролите се хешират, използвайки PostgreSQL модула pgcrypto^[26]. Използва се алгоритъм “bf” с 8 итерации. След новия запис сървърът издава нов ключ за достъп със субект “client”, съдържащ потребителското име на клиента (фиг. 3.5). Ключът за достъп се предоставя на клиента чрез отговора на заявката до сървъра.

3.2.1.2. Вход на клиент

Чрез формата за вход на страницата LoginPage потребител може да влезе в своя акаунт, ако въведе правилните потребителско име и парола. Имплементацията на страницата се намира във файл *“datastreamer-server/src/views/pages/login-page.jsx”* от изходния код на приложението.

Форматът на въведените данни се валидира и от клиентската, и от сървърната страна. Те се изпращат на сървъра чрез POST заявка до крайната точка *“/login”*, което е имплементирано във файл *“datastreamer-server/src/routes/client.js”*.

С аргументи потребителското име и паролата се извиква функцията *“login”*, отнасяща се за клиент, в която се прави заявка до базата данни. (изходен код: *“datastreamer-server/src/db/postgres/client.js”*)

Проверява се дали съществува потребител с предоставените данни. Ако това е така се издава ключ за достъп със субект *“client”* и се предоставя на клиента чрез отговора на заявката, където се запазва. При наличие на грешка формата предоставя съобщение за грешка според статус кода, който сървърът е върнал.

3.2.2. Автентикация на провайдер

3.2.2.1. Регистрация на провайдер

Създаването на провайдер се извършва чрез формата за регистрация в компонента *“RegisterPage”*. Тя изисква въвеждане на име и парола на провайдер, както и парола за свързване на клиент към провайдер. Изисква се всяка парола да бъде въведена повторно за потвърждение. Форматът на данните се валидира. (изходен код: *“datastreamer-provider/src/app/views/pages/register-page.jsx”*)

При изпращане се прави POST заявка до сървъра на крайна точка *“/provider/register”*. (изходен код: *“datastreamer-server/src/routes/provider.js”*)

При обработката ѝ се извиква функцията *“register”* (изходен код: *“datastreamer-server/src/db/postgres/provider.js”*), отнасяща се за провайдер, в която се прави заявка до базата данни с извикване на дефинираната функция

“create_provider”. Начинът, по който предварително е създадена, е показан във фигура 3.6.

```
CREATE OR REPLACE FUNCTION create_provider(provider_username varchar,  
    provider_password varchar, client_connect_password varchar)  
    RETURNS RECORD AS $$  
    DECLARE  
        provider RECORD;  
    BEGIN  
        SELECT INTO provider * FROM Providers  
        WHERE Providers.Username = provider_username;  
        IF FOUND THEN  
            RETURN NULL;  
        ELSE  
            INSERT INTO Providers (  
                Username, Password, ClientConnectPassword, Readable,  
                Writable)  
            VALUES (  
                provider_username,  
                crypt(provider_password, gen_salt('bf', 8)),  
                crypt(client_connect_password, gen_salt('bf', 8)),  
                FALSE,  
                FALSE  
            ) RETURNING Username, Readable, Writable INTO provider;  
            RETURN provider;  
        END IF;  
    END;$$ LANGUAGE plpgsql;
```

Фиг. 3.6: SQL за създаване на функцията “create_provider”, която прави нов запис за провайдер в базата данни

(изходен код: “datastreamer-server/src/scripts/db-init/init.sql”)

Тази функция приема като аргументи име на провайдер, парола за вход в провайдер и парола, използвана при свързване на клиент. Проверява се дали съществува провайдер със същото име. Ако не съществува се добавя нов запис към провайдерите.

Паролите се криптират, а като стойности на параметрите “readable” и “writable”, определящи режима на достъп до провайдера, се задава логическа неистина. Така достъпът на клиенти до провайдера трябва изрично да се разреши.

След създаване на записа в базата данни се издава ключ за достъп със субект "provider", съдържащ името на провайдъра, и се предава на десктоп приложението чрез отговора на заявката, при което се запазва.

3.2.2.2. Вход на провайдър

Формата за вход (изходен код: "datastreamer-provider/src/app/views/pages/login-page.jsx") е първата част от потребителския интерфейс, която се визуализира при стартиране на приложението. При въвеждане на име на провайдър и парола за вход данните се изпращат на сървъра чрез POST заявка до крайна точка "/provider/login". (изходен код: datastreamer-server/src/routes/provider.js")

```
async function login(username, password) {
  try {
    const response = await db.query(`SELECT * FROM Providers
      WHERE Username = $1
      AND Password = crypt($2, Password);`, [username, password]);
    if (response.rows.length <= 0) {
      log.info("Unsuccessfull provider attempt to login.");
      log.verbose("Username provided:", username);
      return { success: false };
    }
    const result = response.rows[0];
    const clientAccessRules = await db.query(`SELECT
      Clients.Username AS Username,
      ClientAccessRules.Readable AS Readable,
      ClientAccessRules.Writable AS Writable
      FROM ClientAccessRules INNER JOIN Clients
      ON ClientAccessRules.ClientId = Clients.Id
      WHERE ClientAccessRules.ProviderId = $1;`, [result.id]);
    const token = await signProviderToken(result.username);
    return {
      success: true,
      token,
      username: result.username,
      readable: result.readable,
      writable: result.writable,
      clientAccessRules: clientAccessRules.rows
    };
  } catch (error) {
    log.error("In login a provider:");
    throw error;
  }
}
```

```
}
```

Фиг. 3.7: Вход на провайдер

(изходен код: `"datastreamer-server/src/db/postgres/provider.js"`)

Форматът на данните се валидира. Извиква се функцията `"login"`, отнасяща се за провайдер (фиг. 3.7), в която се прави заявка към базата данни с цел проверка дали съществува провайдер с предоставените име и парола. Ако това е така се вземат всички индивидуално зададени права за достъп до провайдера и се издава нов ключ за достъп със субект `"provider"`. Провайдерът получава ключа за достъп и клиентските права при отговора на заявката към сървъра и ги запазва.

3.2.3. Свързване на клиент към провайдер

Формата, разположена в компонента `"ConnectPage"` на клиентското приложение, позволява свързването към провайдер (изходен код: `"datastreamer-server/src/views/pages/connect-page.jsx"`). При нейното изпращане с въведени име на провайдер и парола за свързване на клиент към провайдер се прави POST заявка до крайна точка `"/connect"`. Изпраща се и клиентският ключ за достъп. (изходен код: `"datastreamer-server/src/routes/client.js"`)

При обработка на заявката сървърът извиква функцията `"connect"` (изходен код: `"datastreamer-server/src/db/postgres/client.js"`). В нея се верифицира ключът за достъп на клиента и се изважда потребителското му име.

След това се прави заявка към базата данни с извикване на функцията `"get_access_rules"`, чиято имплементация е представена във фигура 3.8.

```
CREATE OR REPLACE FUNCTION get_access_rules(provider_username varchar,  
      client_username varchar, client_connect_password varchar)  
  RETURNS RECORD AS $$  
  DECLARE  
    default_rules RECORD;  
    client_rules RECORD;  
  BEGIN  
    SELECT INTO default_rules Username, Readable, Writable  
    FROM Providers WHERE Username = provider_username AND
```

```

        ClientConnectPassword = crypt(client_connect_password,
ClientConnectPassword);
        IF FOUND THEN
            SELECT INTO client_rules Providers.Username,
ClientAccessRules.Readable, ClientAccessRules.Writable
            FROM ClientAccessRules INNER JOIN Providers
            ON ClientAccessRules.ProviderId = Providers.Id
            INNER JOIN Clients ON ClientAccessRules.ClientId = Clients.Id
            WHERE Providers.Username = provider_username AND
Clients.Username = client_username;
            IF FOUND THEN
                RETURN client_rules;
            ELSE
                RETURN default_rules;
            END IF;
        ELSE
            RETURN NULL;
        END IF;
    END;$$ LANGUAGE plpgsql;

```

Фиг. 3.8: SQL за създаване на функцията “get_access_rules”, изпълняваща се при опит за свързване на клиент към провайдер

(изходен код: “datastreamer-server/src/scripts/db-init/init.sql”)

Функцията “get_access_rules” приема името на провайдъра, потребителското име на клиента и паролата за свързване на клиент към провайдъра. Търси се запис за провайдер със съответните име и парола. Ако се открие, се търси запис за индивидуално зададени права за достъп на клиента до провайдъра. Ако се намерят такива, те се връщат като резултат, в противен случай се връщат правата, зададени в записа на провайдъра в базата данни.

След това се създава нов ключ за достъп със субект “clientConnection”, съдържащ потребителското име на клиента, името на провайдъра и правата за достъп до провайдъра. Правата за достъп, името на провайдъра и ключът за достъп се предават заедно със сървърният отговор на заявката. Клиентското приложение ги запазва.

3.3. Осъществяване на P2P връзка

За да се осъществи директна връзка между клиент и провайдер е необходимо те да се намерят един друг, което може да се реализира на сървъра.

3.3.1. Свързване на клиент към сървър чрез уеб сокети

При зареждане на компонента “HomePage” в клиента се осъществява свързване към сървъра чрез WebSocket протокол (фигури 3.9, 3.10 и 3.11). Използва се JavaScript библиотеката Socket.io^[28]

```
class Home extends React.Component {
  constructor(props) {
    super(props);
    [...]
    this.RTC = new RTC({
      connectionToken: this.props.provider.token,
      writeAccess: this.props.provider.writeAccess
    }, {
      handleMessage: this.messageHandler,
      handleChunk: this.chunkHandler,
      handleError: this.handleError,
      pageActionHandler: this.pageActionHandler
    });
  }
  [...]
}
```

Фиг. 3.9: Създаване на обект с прототип RTC

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

```
class RTC {
  constructor(connectData, handlers) {
    this.socket = new Socket(this, connectData.connectionToken,
    handlers.pageActionHandler).socket;
    [...]
  }
  [...]
}
```

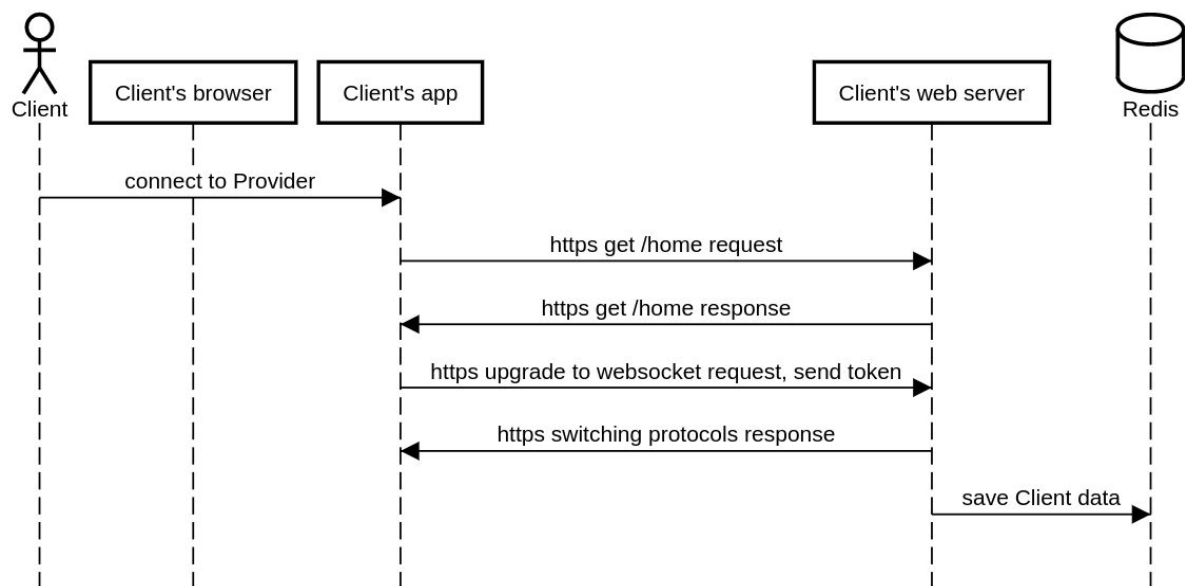
Фиг. 3.10: Създаване на обект с прототип Socket в конструктора на RTC

(изходен код: “datastreamer-server/src/modules/rtc.js”)

```
import io from "socket.io-client";
[...]
```

```
function Socket(RTC, token, pageActionHandler) {
  this.RTC = RTC;
  this.socket = io(`http://${window.location.host}`, {
    query: `token=${token}`
  });
[...]
```

Фиг. 3.11: Осъществяване на връзка на клиент със сървър чрез WebSockets
(изходен код: "datastreamer-server/src/sockets/client.js")



Фиг. 3.12: Диаграма на последователност при свързване на клиент към сървър чрез WebSockets

Свързвайки се, клиентът предава като параметър ключа си за достъп, с който е за установяване на връзка към провайдър. Сървърът автоматично генерира сокет идентификатор и на фигура 3.13. е показано извикването на функцията "createPeerSession". Нейната имплементация се съдържа във фигура 3.14.

```
io.on("connection", socket => {
  if (socket.handshake.query) {
    createPeerSession(
      socket.id,
      socket.handshake.query.token
    )
  }
})
```



```

).then(sessionInfo => {
  if (!sessionInfo) {
    io.to(socket.id).emit("connect_reject", "TokenExpiredError");
  } else {
    if (sessionInfo.type === "clientConnection") {
      if (sessionInfo.provider.isConnected) {
        io.to(socket.id).emit("provider_connect");
        io.to(sessionInfo.provider.socketId)
          .emit(
            "client_connect",
            socket.id,
            socket.handshake.query.token,
            sessionInfo.client.username, {
              readable: sessionInfo.readable,
              writable: sessionInfo.writable
            });
      } else {
        io.to(socket.id).emit("connect_reject",
          "ProviderNotConnectedError");
      }
    }
  }
}).catch(error => {
  log.error(error);
  io.to(socket.id).emit("connect_reject", error.name);
  socket.disconnect(true);
});
} else {
  socket.disconnect(true);
}
}

```

Фиг. 3.13: Операции на сървъра при свързване на клиент чрез WebSockets
(изходен код: "datastreamer-server/src/sockets/server.js")

```

async function createPeerSession(socketId, token) {
  try {
    const isInvalidated = await checkIfInvalidated(token);
    if (isInvalidated) {
      return null;
    }
    const decoded = await verifyToken(token);
    if (!decoded) {
      return null;
    }
    let sessionInfo;
    switch (decoded.sub) {
      case "provider":
        sessionInfo = await createProviderSession(socketId,

```

```

decoded.username);
    return sessionInfo;
  case "clientConnection":
    sessionInfo = await createClientSession(socketId,
decoded.provider);
    return {
      ...sessionInfo,
      client: { username: decoded.client },
      readable: decoded.readable,
      writable: decoded.writable
    };
  default:
    return null;
}
} catch (error) {
  log.error("While creating peer session:");
  throw error;
}
}

```

Фиг. 3.14: Верифициране на ключ за достъп и определяне на типа на новата сесия

(изходен код: "datastreamer-server/src/db/redis/peer-session.js")

Във функцията "createPeerSession" се прави проверка дали предоставеният ключ е инвалидиран, след което се верифицира. Ако не премине проверката, сокет връзката се прекратява. В противен случай се извиква функцията "createClientSession", показана във фигура 3.15. Чрез нея се запазва информацията за клиента в базата данни за съхранение на сесийна информация и се проверява дали в нея са записани данни за провайдъра, към който клиентът се опитва да се свърже, тоест дали провайдърът е на линия. Ако е на линия, сървърът му изпраща данните на клиента с наименование на съобщението "client_connect", показано във фигура 3.13., с което започва процесът на сигнализация (signalling), описан в т. 3.3.3.

```

async function createClientSession(socketId, providerName) {
  try {
    const response = await redisClient.multi()
      .sismember("providers", providerName)
      .get(`${providerName}:socketId`)
      .sadd("clients", socketId)
      .sadd(`${providerName}:clientSocketIds`, socketId)

```

```

        .set(`${socketId}:providerName`, providerName)
        .set(`${socketId}:created`, new Date().getTime())
        .execAsync();
    log.info(`New client session successfully created...`);
    log.verbose(`Redis response: ${response}`);
    return {
        type: "clientConnection",
        provider: {
            providerName,
            isConnected: response[0],
            socketId: response[1]
        }
    };
} catch (error) {
    log.error(`There was an error creating the client session for client "${socketId}": ${error}`);
    throw error;
}
}

```

Фиг. 3.15: Създаване на нова клиентска сесия

(изходен код: "datastreamer-server/src/db/redis/client-session.js")

3.3.2. Свързване на провайдър към сървър чрез уеб сокети

При зареждане на компонента "HomePage" провайдърът се свързва към сървъра чрез WebSocket протокол, предавайки своя ключ за достъп със субект "provider". Имплементацията на този процес е показан във фигури 3.16., 3.17. и 3.18.

```

class Home extends React.Component {
    constructor(props) {
        super(props);

        this.pageAccessor = this.pageAccessor.bind(this);
        this.connector = new MainToUnitConnector(this.props.provider.token,
        this.pageAccessor);
    }
    [...]
}

```

Фиг. 3.16: Създаване на обект с прототип MainToUnitConnector

(изходен код: "datastreamer-provider/src/app/views/pages/home-page.jsx")

```

class MainToUnitConnector {

```

```

    constructor(token, pageAccessor) {
        this.socket = new Socket(this, token, pageAccessor).socket;
    }
    [...]
}

```

Фиг. 3.17: Създаване на обект с прототип *Socket*

(изходен код:

“*datastreamer-provider/src/app/connections/main-to-unit-connector.js*”)

```

import io from "socket.io-client";
import config from "../../config.json";

function Socket(connector, token, pageAccessor) {
    this.connector = connector;
    this.socket = io(config.uri, {
        query: `token=${token}`,
        secure: true
    });
    [...]
}

```

Фиг. 3.18: Осъществяване на връзка на провайдер със сървър чрез *WebSockets*

(изходен код: “*datastreamer-provider/src/app/connections/socket.js*”)

Сървърът автоматично генерира сокет идентификатор и изпълнява “*createPeerSession*”, чиято дефиниция е показана във фигура 3.14. Изпълнението на тази функция протича подобно на описаното в т. 3.3.1., но вместо “*createClientSession*” в нея се извиква функцията “*createProviderSession*” (фиг. 3.19). Чрез нея се запазва сесийна информация за провайдера.

```

async function createProviderSession(socketId, providerName) {
    try {
        const response = await redisClient.multi()
            .smembers(`${providerName}:clientSocketIds`)
            .sadd("providers", providerName)
            .set(`${socketId}:providerName`, providerName)
            .set(`${providerName}:socketId`, socketId)
            .set(`${providerName}:created`, new Date().getTime())
            .execAsync();
        return {
            type: "provider",

```

```

        clientSocketIds: response[0],
        socketId,
        providerName
    };
} catch (error) {
    log.error("There was an error creating the provider session");
    throw error;
}
}

```

Фиг. 3.19: Създаване на сесия на провайдер

(изходен код: “datastreamer-server/src/db/redis/provider-session.js”)

Провайдерът изпраща до сървъра съобщение с наименование “client_tokens_request”, което представлява заявка да бъдат пратени на провайдъра данните за всички клиенти, желаещи да се свържат към него. Това е представено във фигури 3.20. и 3.21.

```

class Home extends React.Component {
    [...]
    componentDidMount() {
        if (this.props.provider.token) {
            window.addEventListener("beforeunload", this.connector.deleteAll);
        }
        this.connector.connectToClients();
    }
    [...]
}

```

Фиг. 3.20: Извикване на функцията “connectToClients”

(изходен код: “datastreamer-provider/src/app/views/pages/home-page.jsx”)

```

class MainToUnitConnector {
    [...]
    connectToClients() {
        this.socket.emit("client_tokens_request");
    }
    [...]
}

```

Фиг. 3.21: Изпращане на заявка за ключовете за достъп на клиентите, правещи опит да се свържат към провайдер

(изходен код:

“datastreamer-provider/src/app/connections/main-to-unit-connector.js”)

Както е показано във фигури 3.22., 3.23., 3.24. и 3.25., сървърът намира клиентите, желаещи да се свържат към провайдъра, и на всеки един от тях изпраща съобщение с наименование "token_request", което е заявка клиентът да предостави данните си. Данните, които са необходими, са тези, които не са запазени като сесийна информация - ключът за достъп за свързване на клиент към провайдър и записаните в него права за достъп до провайдъра.

```
socket.on("client_tokens_request", () => {
  findClientSessionsByProviderSocketId(socket.id).then(clientSessions => {
    clientSessions.forEach(client => {
      io.to(client).emit("token_request");
    });
  }).catch(error => {
    log.error(error);
    socket.disconnect(true);
  });
});
```

Фиг. 3.22: Изпращане на заявка "token_request" към всички клиенти, правещи опит да се свържат към провайдъра

(изходен код: "datastreamer-server/src/sockets/server.js")

```
async function findClientSessionsByProviderSocketId(socketId) {
  try {
    const providerName = await findProviderNameBySocketId(socketId);
    return findClientSessionsByProviderName(providerName);
  } catch (error) {
    log.error("In find client sessions by provider socketId");
    throw error;
  }
}
```

Фиг. 3.23: Намиране на клиенти, опитващи да се свържат към провайдъра, чрез уеб сокет идентификатора на провайдъра

(изходен код: "datastreamer-server/src/db/redis/provider-session.js")

```
async function findProviderNameBySocketId(socketId) {
  try {
    return redisClient.getAsync(`${socketId}:providerName`);
  } catch (error) {
    log.error("In find provider name by socketId:");
    throw error;
  }
}
```

```
}
```

Фиг. 3.24: Функция, връщаща името на провайдер при подаден уеб сокет идентификатор

(изходен код: "datastreamer-server/src/db/redis/provider-session.js")

```
async function findClientSessionsByProviderName(providerName) {
  try {
    return redisClient.smembersAsync(`${providerName}:clientSocketIds`);
  } catch (error) {
    log.error("In find client session by provider name:");
    throw error;
  }
}
```

Фиг. 3.25: Функция, получаваща като аргумент името на провайдер и връщаща множество от сокет идентификатори на клиенти, опитващи да се свържат към провайдъра

(изходен код: "datastreamer-server/src/db/redis/provider-session.js")

Всеки клиент отговаря със съобщение "token_response", съдържащо ключа за достъп, което е показано във фигура 3.26.

```
function Socket(RTC, token, pageActionHandler) {
  [...]
  this.socket.on("token_request", () => {
    this.socket.emit("token_response", token);
  });
}
```

Фиг. 3.26: Изпращане до сървъра ключ за достъп на клиент

(изходен код: "datastreamer-server/src/sockets/client.js")

Фигура 3.27. показва как сървърът верифицира и декодира всеки ключ и при успех изпраща до провайдъра съобщение с наименование "client_connect", съдържащо данните на клиента, с което започва процесът на сигнализация (signalling), описан в т. 3.3.3.

```
socket.on("token_response", token => {
  let providerSocketId;
  findProviderSocketIdByClientSocketId(socket.id).then(socketId => {
    providerSocketId = socketId;
    return verifyToken(token);
  }).then(decoded => {
```

```

    if (!decoded) {
      io.to(socket.id).emit("connect_reject", "TokenExpiredError");
    } else {
      io.to(providerSocketId).emit("client_connect", socket.id, token,
        decoded.client, {
          readable: decoded.readable,
          writable: decoded.writable
        });
      io.to(socket.id).emit("provider_connect");
    }
  }).catch(error => {
    log.error(error);
    socket.disconnect(true);
  });
});

```

Фиг. 3.27: Изпращане до провайдъра ключа за достъп за свързване на клиент (изходен код: "datastreamer-server/src/sockets/server.js")

```

async function findProviderSocketIdByClientSocketId(clientSocketId) {
  try {
    const providerName = await
      redisClient.getAsync(`${clientSocketId}:providerName`);
    return findProviderSocketIdByProviderName(providerName);
  } catch (error) {
    log.error("In find provider socketId by client socketId:");
    throw error;
  }
}

```

Фиг. 3.28: Функция, получаваща като аргумент уеб сокет идентификатор на клиент и връщаща уеб сокет идентификатор на провайдъра, към който клиентът опитва да се свърже

(изходен код: "datastreamer-server/src/db/redis/provider-session.js")

```

async function findProviderSocketIdByProviderName(providerName) {
  try {
    return redisClient.getAsync(`${providerName}:socketId`);
  } catch (error) {
    log.error("In find provider socketId by provider name:");
    throw error;
  }
}

```

Фиг. 3.29: Функция, приемаща име на провайдър и връщаща неговия сокет идентификатор

(изходен код: "datastreamer-server/src/db/redis/provider-session.js")

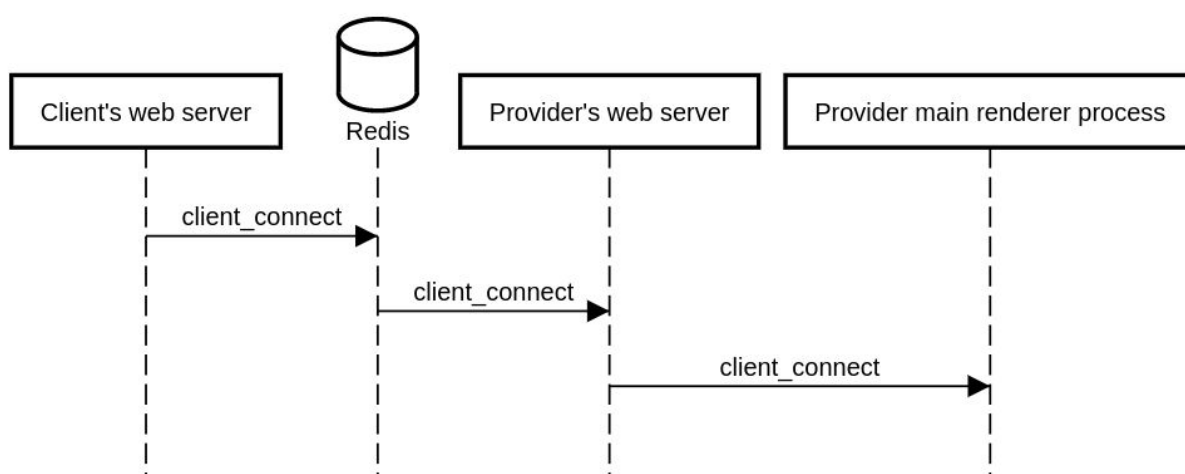
3.3.3. Процесът “сигнализиране”

Важно условие за установяване на P2P връзка е предварителният обмен на данни, които служат за конфигуриране на параметрите на връзката и определяне на мрежовия път, през който ще се осъществява комуникацията. Този процес се нарича “сигнализиране” (signalling). В настоящата разработка сигнализирането се извършва посредством сървър чрез уеб сокети.

Процесът стартира след като единият пиър се свърже към сървъра чрез уеб сокети и открие, че другият също е свързан. Сигнализирането започва с изпращането на съобщение “client_connect” до провайдъра. Както е показано на фигури 3.31., 3.33., 3.34., 3.35. и 3.36., създава се нов клиентски процес, в който се създава нов обект, чийто прототип е “RTCPeerConnection”.

На фигури 3.38. и 3.47. на конструктора на “RTCPeerConnection” не са подадени аргументи. С такава конфигурация осъществяването на P2P връзка е възможно само, ако пиърите се намират в една и съща мрежа. На конструктора може да се подаде обект “RTCConfiguration” с поле “iceServers”, в което са описани STUN/TURN сървъри, към които пиърът може да се свърже. Това е задължително да се направи при пускане на системата в експлоатация. Смисълът на STUN/TURN сървърите е описан в т. 1.2.2.

Фигури 3.33., 3.34. и 3.35. показват междупроцесова комуникация. Нейният смисъл и начин на работа са обяснени в точка 3.4.



Фиг. 3.30: Последователност при изпращане на съобщението “client_connect” от сървъра на клиента до основния рендериращ процес на провайдъра

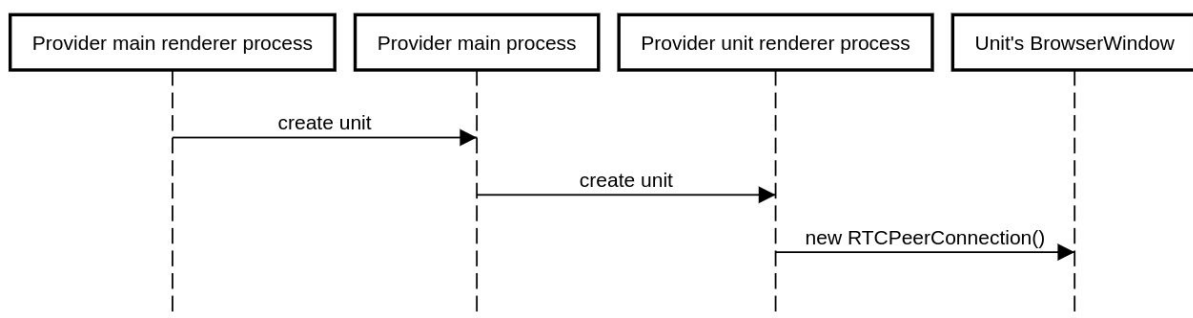
```

this.socket.on("client_connect", (clientSocketId, token, username, accessRules)
=> {
  this.connector.createUnit({
    clientSocketId, token, username, accessRules
  }, this.connector.selectedMainDirectory);
  pageAccessor(function () {
    this.props.addClient({
      id: clientSocketId,
      token,
      username,
      readable: accessRules.readable,
      writable: accessRules.writable,
      directory: this.connector.selectedMainDirectory
    });
  });
});
});

```

Фиг. 3.31: Съобщение до MainToUnitConnector за създаване на нов клиентски процес в провайдъра

(изходен код: "datastreamer-provider/src/app/connections/socket.js")



Фиг. 3.32: Диаграма на последователности за създаване на нов клиентски процес в провайдъра

```

createUnit(unitData, selectedMainDirectory) {
  ipcRenderer.send("create unit", unitData, selectedMainDirectory);
}

```

Фиг. 3.33: Изпращане на съобщение от основния рендериращ процес до главния процес за създаване на нов клиентски рендериращ процес

(изходен код:

"datastreamer-provider/src/app/connections/main-to-unit-connector.js")

```

ipcMain.on("create unit", (event, unitData, selectedMainDirectory) => {
  const browserWindow = new BrowserWindow({
    parent: mainWindow,

```

```

        show: false
    });
    browserWindow.once("ready-to-show", () => {
        browserWindow.show();
        browserWindow.webContents.send("initialize", unitData,
selectedMainDirectory);
    });
    socketIdUnitMap.set(unitData.clientSocketId, {
        browserWindow,
        token: unitData.token,
        accessRules: unitData.accessRules
    });
    let unit = socketIdUnitMap.get(unitData.clientSocketId);
    unit.browserWindow.webContents.openDevTools();
    unit.browserWindow.loadURL(url.format({
        pathname: path.join(__dirname, "app/units/provider-unit.html"),
        protocol: "file:",
        slashes: true
    }));
    }));
});

```

Фиг. 3.34: Създаване на нов клиентски процес

(изходен код: "datastreamer-provider/src/ipc-handler.js")

```

ipcRenderer.on("initialize", (event, unitData, selectedMainDirectory) => {
    const client = new Client(unitData, selectedMainDirectory);
});

```

Фиг. 3.35: Създаване на нов обект с прототип "Client"

(изходен код: "datastreamer-provider/src/app/units/init.js")

```

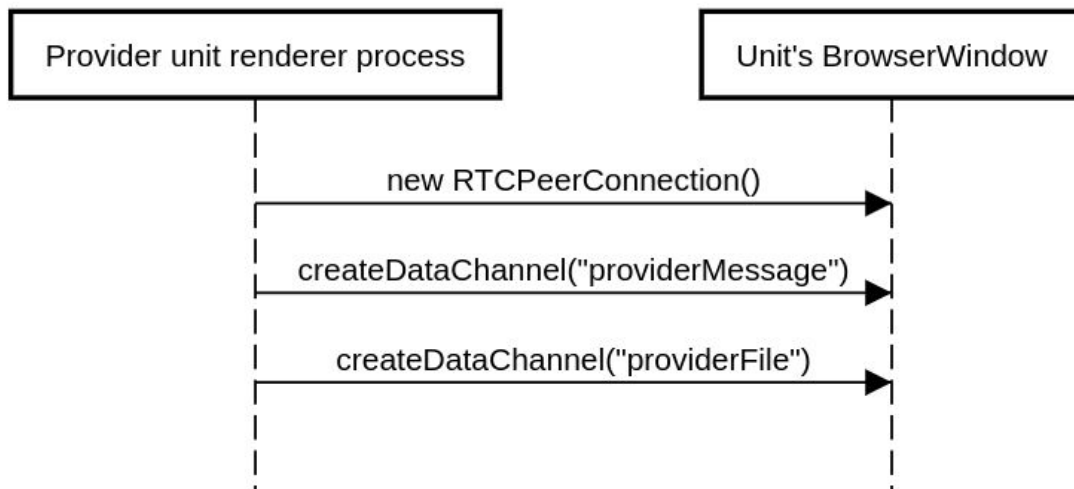
class Client {
    constructor(unitData, selectedMainDirectory, currentDirectory = ".",
watcherOptions = {
        ignored: /[\\\/\.\.]\./,
        alwaysStat: true,
        usePolling: true,
        depth: 0
    }) {
        [...]
        this.prepareConnectionInitialization(unitData.accessRules);
    }
}

```

Фиг. 3.36: Извикване на функция за създаване на *RTCPeerConnection*

(изходен код: "datastreamer-provider/src/app/units/client.js")

Създават се канали за комуникация с клиент, задават се поведение при наличие на възможен път за комуникация между пиърите (ICE candidate) и поведение при открит канал за директна комуникация, създаден от другия пиър (фиг. 3.38.).



Фиг. 3.37: Създаване на `RTCPeerConnection` и канали за изпращане на съобщения и файлове от провайдер към клиент

```
function prepareConnectionInitialization(accessRules) {
  this.peerConnection = new RTCPeerConnection();
  this.peerConnection.onicecandidate = event => {
    if (event.candidate) {
      this.connector.sendICECandidate(event.candidate);
    }
  };

  this.peerConnection.ondatachannel = event => {
    switch (event.channel.label) {
      case "clientMessage":
        if (accessRules.readable) {
          this.receiveMessageChannel = event.channel;
          this.receiveMessageChannel.onmessage = event => {
            this.handleMessage(JSON.parse(event.data));
          };
        }
        break;
      case "clientMessageWritable":
        if (accessRules.readable && accessRules.writable) {
          this.receiveMessageWritableChannel = event.channel;
          this.receiveMessageWritableChannel.onmessage = event => {
            this.handleMessageWritable(JSON.parse(event.data));
          };
        }
        break;
    }
  };
}
```

```

        };
    }
    break;
    case "clientFile":
        if (accessRules.readable && accessRules.writable) {
            this.receiveFileChannel = event.channel;
            this.receiveFileChannel.binaryType = "arraybuffer";
            this.receiveFileChannel.onmessage = event => {
                this.handleChunk(event.data);
            };
        }
        break;
    }
}

this.sendMessageChannel =
this.peerConnection.createDataChannel("providerMessage", this.dataConstraint);
this.sendFileChannel = this.peerConnection.createDataChannel("providerFile",
this.dataConstraint);
this.sendFileChannel.binaryType = "arraybuffer";
this.sendFileChannel.bufferedAmountLowThreshold = 1024 * 1024; // 1 MB

this.sendMessageChannel.onopen = () => {
    if (this.selectedMainDirectory) {
        this.scanDirectory();
    }
}
this.connector.requestP2PConnection();
}

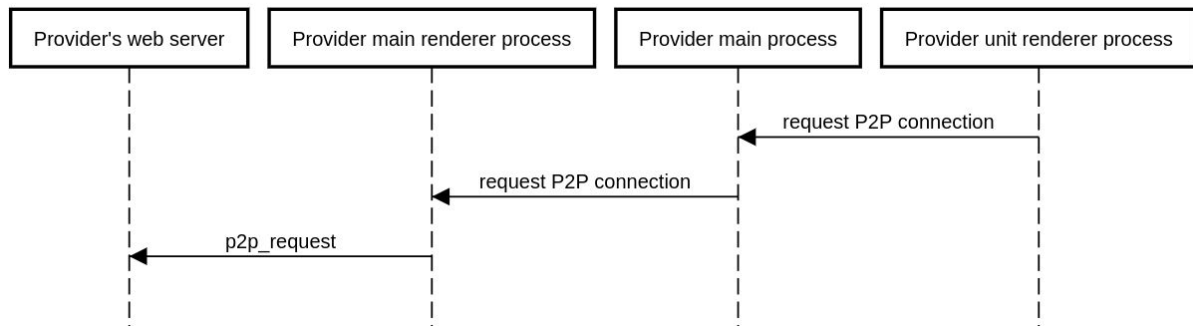
```

Фиг. 3.38: Инициализация на директна комуникация при провайдър
(изходен код: "datastreamer-provider/src/app/units/connections/rtc-initialization.js")

Провайдърът създава два канала с обозначения "providerMessage" и "providerFile". Първият канал служи за изпращане на съобщения до клиента, а вторият - за изпращане на файлове. Задава се поведение когато каналът за изпращане на съобщение стане отворен да започне сканиране на файлове в избрана директория.

След тези конфигурации се изпраща към сървъра съобщение "p2p_request" (фиг. 3.42). Преди това е нужна междупроцесова комуникация, описана чрез фигури 3.39., 3.40. И 3.41. Фигура 3.44. представя препращането на съобщението от сървъра към клиента, а фигура 3.45. - получаването на съобщението. Целта е клиентът да бъде уведомен, че инициализацията,

извършена на провайдъра, е завършила, и че може да започне своята инициализация.



Фиг. 3.39: Предаване на съобщението за завършила инициализация на провайдъра от клиентски процес до сървъра на провайдъра

```

class UnitToMainConnector {
[...]
  requestP2PConnection() {
    ipcRenderer.send("request P2P connection", this.client.id);
  }
[...]}

```

Фиг. 3.40: Изпращане на съобщението “request P2P connection” от клиентски процес до главния процес

(изходен код:

“datastreamer-provider/src/app/units/connections/unit-to-main-connector.js”)

```

function ipcHandler(mainWindow) {
[...]
  ipcMain.on("request P2P connection", (event, clientSocketId, arg) => {
    mainWindow.webContents.send("request P2P connection", clientSocketId);
  });
[...]}

```

Фиг. 3.41: Препращане на съобщението “request P2P connection” от главния процес до основния рендериращ процес

(изходен код: “datastreamer-provider/src/ipc-handler.js”)

```

class MainToUnitConnector {
  constructor(token, pageAccessor) {

```

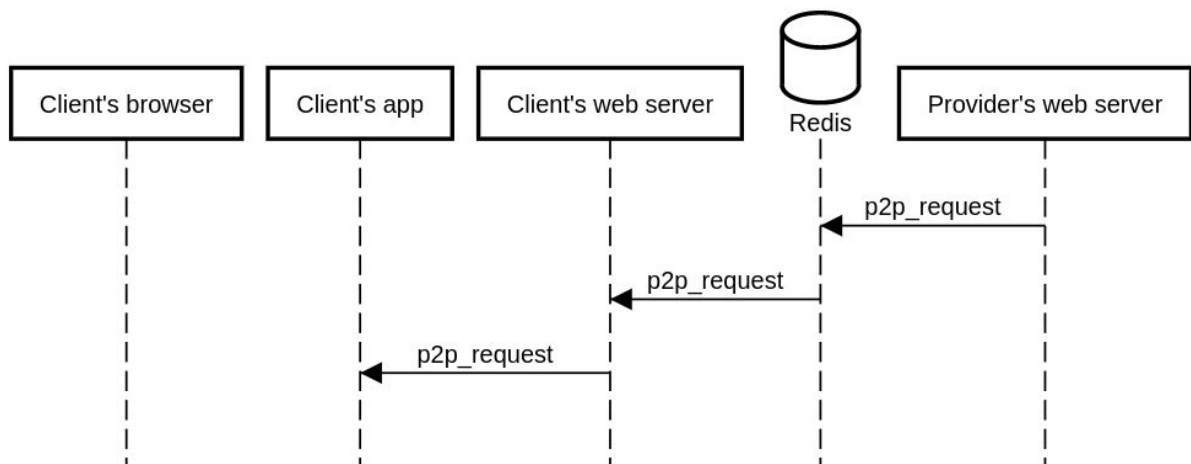
```

    this.socket = new Socket(this, token, pageAccessor).socket;
[...]
    ipcRenderer.on("request P2P connection", (event, clientSocketId) => {
        this.socket.emit("p2p_request", clientSocketId);
    });
[...]
}
[...]
```

Фиг. 3.42: Изпращане на съобщение "p2p_request" до сървъра

(изходен код:

"datastreamer-provider/src/app/connections/main-to-unit-connector.js")



Фиг. 3.43: Диаграма на последователности за предаване на съобщението "p2p_request" от сървър до клиент

```

socket.on("p2p_request", clientId => {
    io.to(clientId).emit("p2p_request");
});

```

Фиг. 3.44: Препращане на съобщението "p2p_request" от сървър до клиент

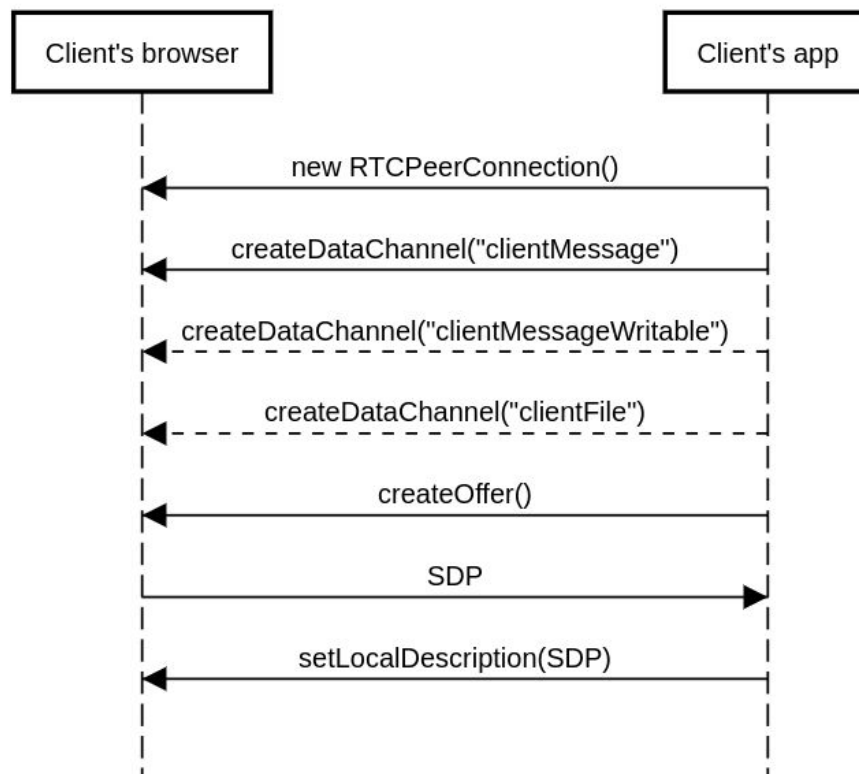
(изходен код: "datastreamer-server/src/sockets/server.js")

```

function Socket(RTC, token, pageActionHandler) {
[...]
    this.socket.on("p2p_request", () => {
        pageActionHandler(function () {
            this.props.setLoaderMessage("Initializing P2P connection...");
        });
        this.RTC.initializeP2PConnection();
    });
[...]
```

```
}
```

Фиг. 3.45: Получаване на съобщението "p2p_request" от клиент
(изходен код: "datastreamer-server/src/sockets/client.js")



Фиг. 3.46: Последователност на инициализация на RTC при клиента

На фигура 3.46. е представена диаграма на последователности, описваща инициализация при клиента, нужна за осъществяване на директна връзка, на фигура 3.47. - самата имплементация. В клиента се създава нов обект с прототип "RTCPeerConnection". Създава се канал с обозначение "clientMessage", който служи за изпращане на съобщения до провайдъра, свързани единствено с четене на информация.

```
class RTC {
[...]
  initializeP2PConnection() {
    this.peerConnection = new RTCPeerConnection();
    this.sendMessageChannel =
this.peerConnection.createDataChannel("clientMessage", this.dataConstraint);
    if (this.writeAccess) {
      this.sendMessageWritableChannel =
```



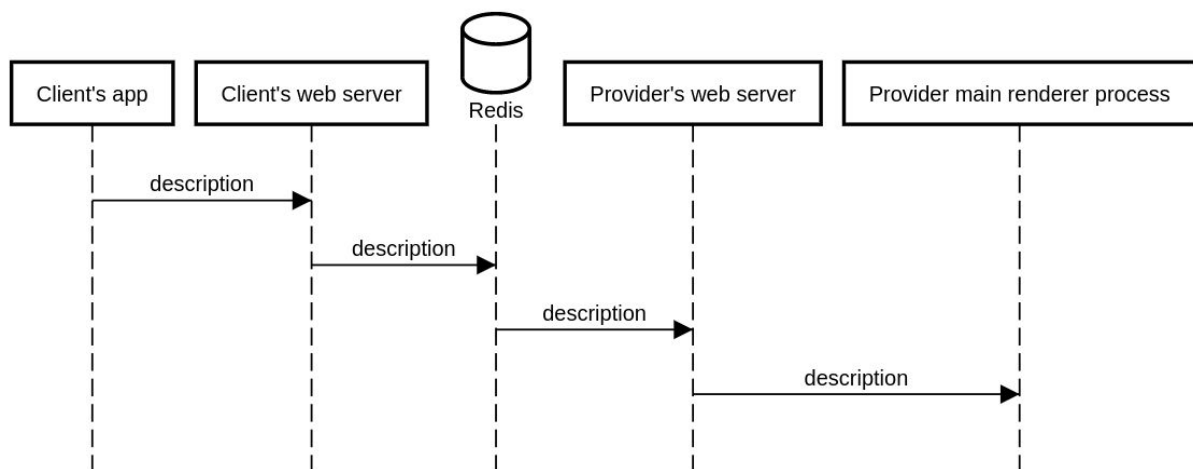
```

this.peerConnection.createDataChannel("clientMessageWritable",
this.dataConstraint);
    this.sendFileChannel =
this.peerConnection.createDataChannel("clientFile", this.dataConstraint);
    this.sendFileChannel.binaryType = "arraybuffer";
    this.sendFileChannel.bufferedAmountLowThreshold = 1024 * 1024; // 1
MB
}
this.peerConnection.onicecandidate = event => {
    if (event.candidate) {
        this.socket.emit("ice_candidate",
JSON.stringify(event.candidate));
    }
};
this.peerConnection.ondatachannel = event => {
    switch (event.channel.label) {
        case "providerMessage":
            this.receiveMessageChannel = event.channel;
            this.receiveMessageChannel.onmessage = event => {
                this.handleMessage(JSON.parse(event.data));
            };
            break;
        case "providerFile":
            this.receiveFileChannel = event.channel;
            this.receiveFileChannel.binaryType = "arraybuffer";
            this.receiveFileChannel.onmessage = event => {
                this.handleChunk(event.data);
            };
            break;
    }
}
this.peerConnection.createOffer().then(description => {
    return this.peerConnection.setLocalDescription(description);
}).then(() => {
    this.socket.emit("description",
JSON.stringify(this.peerConnection.localDescription));
}).catch(error => {
    this.handleError({
        type: "generic",
        message: error
    });
    this.deleteP2PConnection(error);
});
}
[...]
```

Фиг. 3.47: Инициализация на директна комуникация при клиент
(изходен код: "datastreamer-server/src/modules/rtc.js")

Ако клиентът има права за писане се създават още два канала - "clientMessageWritable" и "clientFile". Смисълът на първия е изпращане на команди, свързани с промяна на информация, а на втория - изпращане на файл. Задават се поведение при открит път за комуникация между пиърите (ICE candidate) и поведение при наличие на канал за директна комуникация, създаден от провайдъра.

След това чрез "createOffer" се създава предложение във вид на SDP (session description protocol) - протокол с описание на параметрите на сесията. Запазва се като локално описание и се изпраща със съобщение "description" до сървъра, където се препраща до провайдъра. Фигури 3.48. и 3.51. представят чрез диаграма на последователности действията, необходими за предаване на описанието от клиента до провайдъра, а фигури 3.49., 3.50., 3.52., 3.53. и 3.54. - самата имплементация.



Фиг. 3.48: Последователност при предаване на протокол с описание на параметрите на сесията на клиента от клиентското приложение до основния рендериращ процес на провайдъра.

```

socket.on("description", (description, receiver) => {
  if (receiver) {
    io.to(receiver).emit("description", description);
  } else {
    findProviderSocketIdByClientSocketId(socket.id).then(socketId => {
      if (!socketId) {
        io.to(socket.id).emit("connect_reject",
        "ProviderNotConnectedError");
      } else {
        io.to(socketId).emit("description", socket.id, description);
      }
    });
  }
});
  
```

```

    }
  }).catch(error => {
    log.error(error);
    socket.disconnect(true);
  });
}
});

```

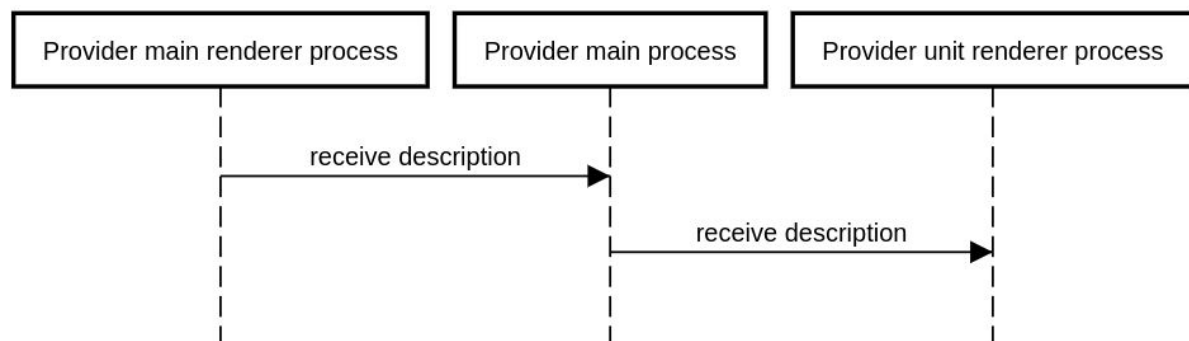
Фиг. 3.49: Предаване на описание от сървър към пиър
(изходен код: "datastreamer-server/src/sockets/server.js")

```

function Socket(connector, token, pageAccessor) {
  [...]
  this.socket.on("description", (clientSocketId, description) => {
    this.connector.receiveDescription(clientSocketId, description);
  });
  [...]
}

```

Фиг. 3.50: Получаване на клиентско описание в основния рендериращ процес на провайдъра и предаването му на MainToUnitConnector
(изходен код: "datastreamer-provider/src/app/connections/socket.js")



Фиг. 3.51: Последователност на предаване на описание от основния рендериращ процес на провайдъра до клиентски процес

```

class MainToUnitConnector {
  [...]
  receiveDescription(clientSocketId, description) {
    ipcRenderer.send("receive description", clientSocketId, description);
  }
  [...]
}

```

Фиг. 3.52: Предаване на клиентско описание от основния рендериращ процес до главния процес

(изходен код:

“datastreamer-provider/src/app/connections/main-to-unit-connector.js”)

```
function ipcHandler(mainWindow) {  
  [...]  
  ipcMain.on("receive description", (event, clientSocketId, description) => {  
    let unit = socketIdUnitMap.get(clientSocketId);  
    unit.browserWindow.webContents.send("receive description", description);  
  });  
  [...]  
}
```

Фиг. 3.53: Предаване на клиентско описание от главния до клиентския процес

(изходен код: “datastreamer-provider/src/ipc-handler.js”)

```
class UnitToMainConnector {  
  constructor(client) {  
    [...]  
    ipcRenderer.on("receive description", (event, remoteDescription) => {  
      this.client.respondToOffer(JSON.parse(remoteDescription));  
    });  
  }  
  [...]  
}
```

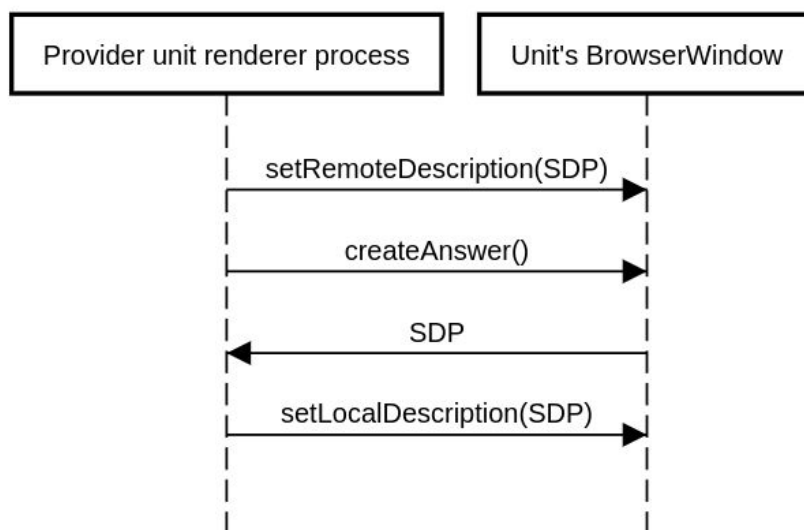
Фиг. 3.54: Получаване на клиентско описание в клиентския процес

(изходен код:

“datastreamer-provider/src/app/units/connections/unit-to-main-connector.js”)

Провайдърът получава описанието и го задава като чуждо. Чрез “createAnswer” генерира ново, чиито параметри са съвместими с тези на чуждото, и го запазва като локално. Тази последователност е представена на фигура 3.55., а фигура 3.56. показва самата имплементация.

Новото описание се праща до сървъра със съобщение “description” и информация за кой клиент се отнася. Сървърът го препраща до съответния клиент, където се запазва като чуждо описание (фиг. 3.58., 3.59., 3.60. и 3.62.). На фигури 3.57. и 3.61. е представена последователността на тези операции.



Фиг. 3.55: Последователност на инициализация на RTC в клиентски процес на провайдер

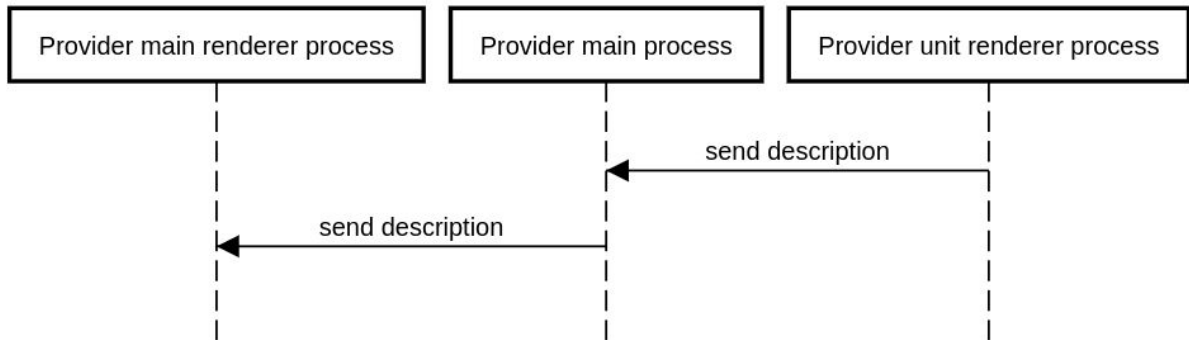
```

function respondToOffer(remoteDescription) {
  try {
    this.peerConnection.setRemoteDescription(remoteDescription);
    this.peerConnection.createAnswer().then(localDescription => {
      return this.peerConnection.setLocalDescription(localDescription);
    }).then(() => {

this.connector.sendDescription(this.peerConnection.localDescription);
    }).catch(error => {
      this.connector.deleteClient(error);
    });
  } catch (e) {
    [...]
  }
}

```

Фиг. 3.56: Инициализация на RTC в клиентски процес на провайдер
(изходен код: "datastreamer-provider/src/app/units/connections/rtc-initialization.js")



Фиг. 3.57: Диаграма на последователности с предаване на описание на провайдер от клиентски процес до основния рендериращ процес

```

class UnitToMainConnector {
  [...]
  sendDescription(localDescription) {
    ipcRenderer.send("send description", this.client.id,
    JSON.stringify(localDescription));
  }
  [...]
}
  
```

Фиг. 3.58: Предаване на описание на провайдер от клиентски процес до главния процес

(изходен код:

"datastreamer-provider/src/app/units/connections/unit-to-main-connector.js")

```

function ipcHandler(mainWindow) {
  [...]
  ipcMain.on("send description", (event, clientSocketId, description) => {
    mainWindow.webContents.send("send description", clientSocketId,
    description);
  });
  [...]
}
  
```

Фиг. 3.59: Предаване на описание на провайдер от главния процес до основния рендериращ процес

(изходен код: "datastreamer-provider/src/ipc-handler.js")

```

class MainToUnitConnector {
  constructor(token, pageAccessor) {
  [...]
    ipcRenderer.on("send description", (event, clientSocketId, description)
    => {
  
```

```

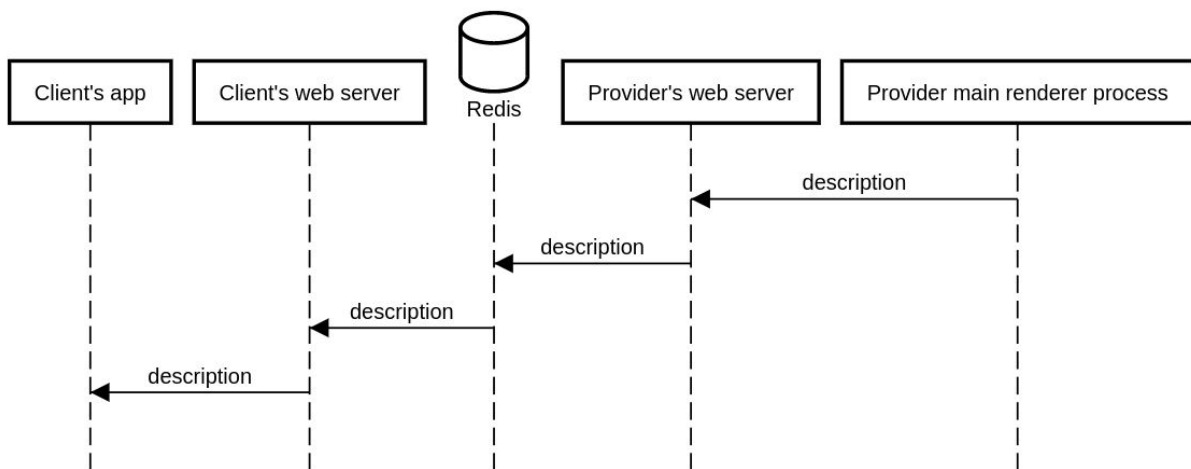
        this.socket.emit("description", description, clientSocketId);
    });
    [...]
  }
  [...]
}

```

Фиг. 3.60: Получаване на описание от главния процес в основния рендериращ процес и изпращане на описание до сървъра

(изходен код:

“datastreamer-provider/src/app/connections/main-to-unit-connector.js”)



Фиг. 3.61: Последователност на изпращане на описанието на провайдъра от основния му рендериращ процес до клиентското приложение

```

function Socket(RTC, token, pageActionHandler) {
  [...]
  this.socket.on("description", description => {
    try {

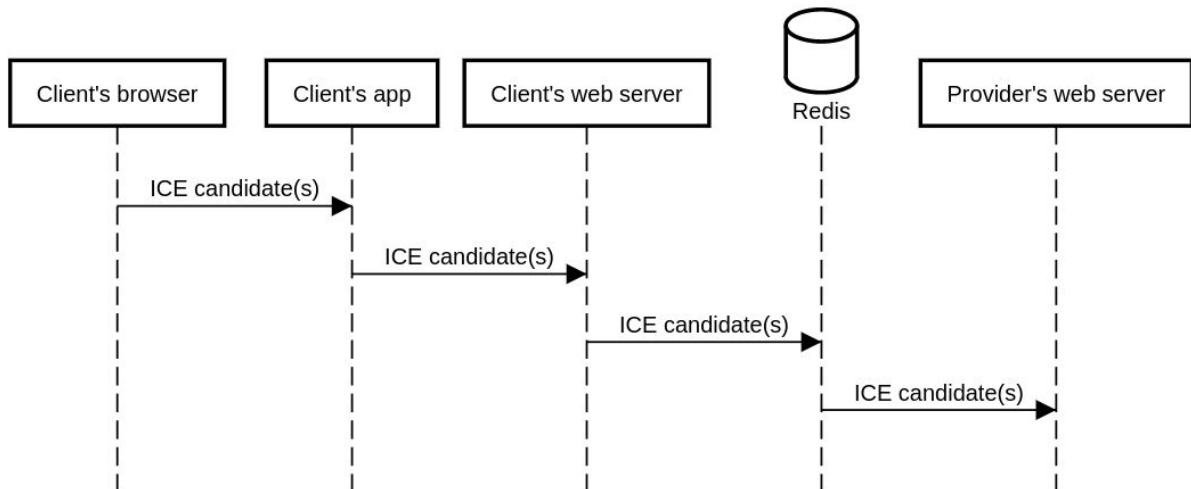
      this.RTC.peerConnection.setRemoteDescription(JSON.parse(description));
      pageActionHandler(function () {
        this.props.setLoaderMessage("Peer negotiation...");
      });
    } catch (error) {
      [...]
    }
  });
  [...]
}

```

Фиг. 3.62: Приемане на описанието на провайдъра от клиента

(изходен код: “datastreamer-server/src/sockets/client.js”)

След това провайдърът и клиентът започват да търсят най-добрия мрежови път, по който да се осъществи директната връзка. Когато клиент или провайдър открие възможен път (ICE candidate) го изпраща до сървъра със съобщение "ice_candidate", след което бива препращано до другия пиър, където се запазва при чуждото описание (фиг. 3.64. - 3.72.).



Фиг. 3.63: Последователност при предаване на ICE candidate от браузъра на клиента до сървър

```

this.peerConnection.onicecandidate = event => {
  if (event.candidate) {
    this.socket.emit("ice_candidate", JSON.stringify(event.candidate));
  }
};

```

Фиг. 3.64: Предаване на генериран ICE candidate към сървъра

(изходен код: "datastreamer-server/src/modules/rtc.js")

```

socket.on("ice_candidate", (candidate, receiver) => {
  if (receiver) {
    io.to(receiver).emit("ice_candidate", candidate);
  } else {
    findProviderSocketIdByClientSocketId(socket.id).then(socketId => {
      if (!socketId) {
        io.to(socket.id).emit("connect_reject",
          "ProviderNotConnectedError");
      } else {
        io.to(socketId).emit("ice_candidate", socket.id, candidate);
      }
    }).catch(error => {
      log.error(error);
    });
  }
});

```



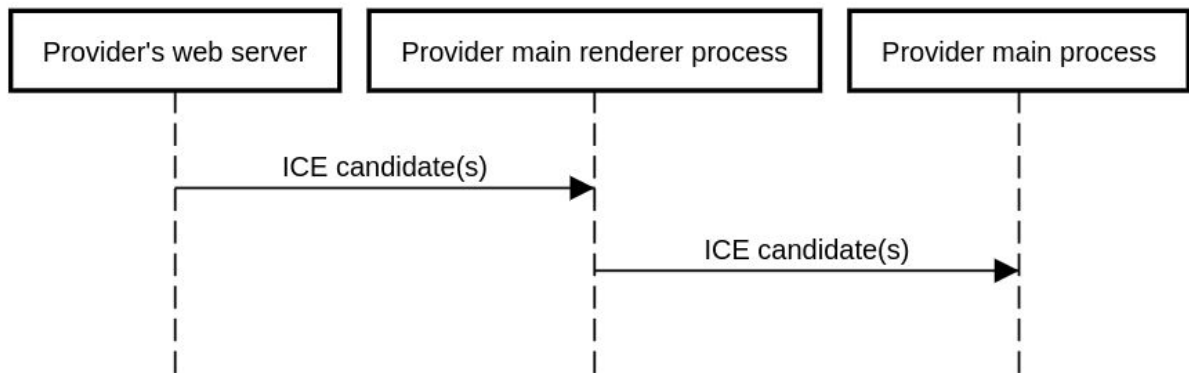
```

        socket.disconnect(true);
    });
}
});

```

Фиг. 3.65: Препращане на ICE candidate от сървър към пиър

(изходен код: “datastreamer-server/src/sockets/server.js”)



Фиг. 3.66: Последователност при предаване на ICE candidate от сървър до главния процес на провайдер

```

function Socket(connector, token, pageAccessor) {
    [...]
    this.socket.on("ice_candidate", (clientSocketId, candidate) => {
        this.connector.receiveICECandidate(clientSocketId, candidate);
    });
    [...]
}

```

Фиг. 3.67: Получаване на ICE candidate в основния рендериращ процес и изпращане до главния процес чрез MainToUnitConnector

(изходен код: “datastreamer-provider/src/app/connections/socket.js”)

```

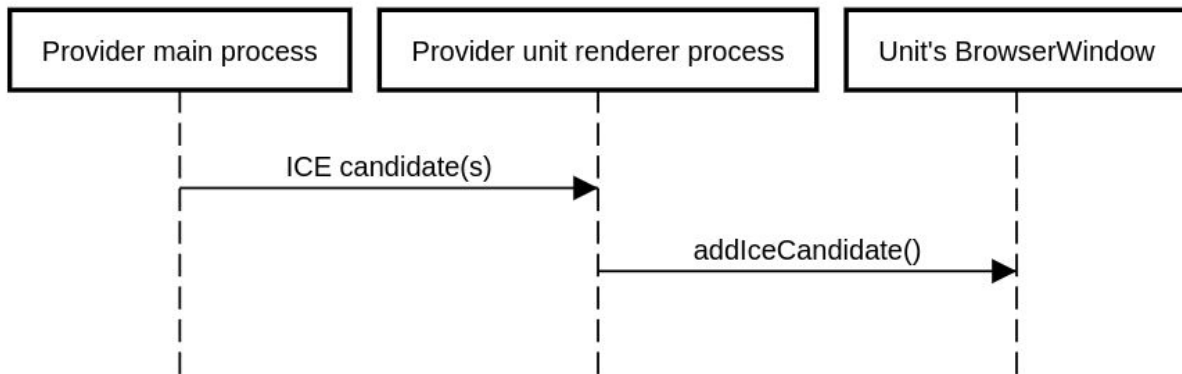
class MainToUnitConnector {
    [...]
    receiveICECandidate(clientSocketId, candidate) {
        ipcRenderer.send("receive ICE candidate", clientSocketId, candidate);
    }
    [...]
}

```

Фиг. 3.68: Изпращане на ICE candidate до главния процес

(изходен код:

“datastreamer-provider/src/app/connections/main-to-unit-connector.js”)



Фиг. 3.69: Последователност на изпращане на ICE candidate от главния процес до клиентския процес, където се запазва

```

function ipcHandler(mainWindow) {
  [...]
  ipcMain.on("receive ICE candidate", (event, clientId, candidate) => {
    let unit = socketIdUnitMap.get(clientId);
    unit.browserWindow.webContents.send("receive ICE candidate", candidate);
  });
  [...]
}

```

Фиг. 3.70: Препращане на ICE candidate от главния процес до клиентски процес

(изходен код: "datastreamer-provider/src/ipc-handler.js")

```

class UnitToMainConnector {
  constructor(client) {
    [...]
    ipcRenderer.on("receive ICE candidate", (event, candidate) => {
      this.client.receiveICECandidate(JSON.parse(candidate));
    });
    [...]
  }
  [...]
}

```

Фиг. 3.71: Получаване на ICE candidate в клиентски процес

(изходен код:

"datastreamer-provider/src/app/units/connections/unit-to-main-connector.js")

```

function receiveICECandidate(candidate) {
  try {

```

```

        this.peerConnection.addIceCandidate(candidate).catch(error => {
            this.connector.deleteClient(error);
        });
    } catch (e) {
        [...]
    }
}

```

Фиг. 3.72: Запазване на ICE candidate в клиентски процес

(изходен код: "datastreamer-provider/src/app/units/connections/rtc-initialization.js")

На фигури 3.64. - 3.72. е показан процесът на предаване на открит мрежов път в посока от клиент към провайдер. Процесът на предаване на открит мрежов път в обратна посока е подобен. Когато се намери най-добрият мрежови път всеки пиър получава и запазва създадените от другия пиър канали. Задава се поведение при получаване на съобщение по всеки канал (фиг. 3.73. и 3.74.).

```

this.peerConnection.ondatachannel = event => {
    switch (event.channel.label) {
        case "providerMessage":
            this.receiveMessageChannel = event.channel;
            this.receiveMessageChannel.onmessage = event => {
                this.handleMessage(JSON.parse(event.data));
            };
            break;
        case "providerFile":
            this.receiveFileChannel = event.channel;
            this.receiveFileChannel.binaryType = "arraybuffer";
            this.receiveFileChannel.onmessage = event => {
                this.handleChunk(event.data);
            };
            break;
    }
}

```

Фиг. 3.73: Дефиниране на поведение на клиент при получване на съобщение по канали, създадени от провайдер

(изходен код: "/datastreamer-server/src/modules/rtc.js")

```

this.peerConnection.ondatachannel = event => {
  switch (event.channel.label) {
    case "clientMessage":
      if (accessRules.readable) {
        this.receiveMessageChannel = event.channel;
        this.receiveMessageChannel.onmessage = event => {
          this.handleMessage(JSON.parse(event.data));
        };
      }
      break;
    case "clientMessageWritable":
      if (accessRules.readable && accessRules.writable) {
        this.receiveMessageWritableChannel = event.channel;
        this.receiveMessageWritableChannel.onmessage = event => {
          this.handleMessageWritable(JSON.parse(event.data));
        };
      }
      break;
    case "clientFile":
      if (accessRules.readable && accessRules.writable) {
        this.receiveFileChannel = event.channel;
        this.receiveFileChannel.binaryType = "arraybuffer";
        this.receiveFileChannel.onmessage = event => {
          this.handleChunk(event.data);
        };
      }
      break;
  }
}

```

Фиг. 3.74: Дефиниране на поведение на провайдър при получаване на съобщение по канали, създадени от клиент

(изходен код: `"/datastreamer-provider/src/app/units/connections/rtc-initialization.js"`)

Когато клиентът получи съобщение по канала с означение "providerMessage", то бива обработено от функцията "handleMessage" (фиг. 3.75.).

```

messageHandler (message) {
  switch (message.type) {
    case "scanFinished":
      this.props.removeLoaderMessage();
      break;
    case "readyForFile":
      this.sendFiles();
      break;
  }
}

```

```

    case "newScan":
        this.props.clearFiles();
        this.files = [];
        break;
    case "add":
    case "addDir":
        this.handleAddFile(message.payload);
        break;
    case "change":
        this.props.changeFile(message.payload);
        break;
    case "unlink":
    case "unlinkDir":
        this.props.unlink(message.payload);
        break;
  }
}

```

Фиг. 3.75: Обработка на съобщения от провайдер

(изходен код: "datastreamer-server/src/views/pages/home-page.jsx")

Съобщенията по канал "providerFile" се обработват от функцията "chunkHandler" (фиг. 3.87.).

Когато провайдерът получи съобщение по канал "clientMessage" изрично се проверява дали клиентът има права за четене и ако ги притежава, функцията "handleMessage" обработва съобщението (фиг. 3.77.). При открит канал "clientMessageWritable" или "clientFile" клиентът трябва да разполага с права за четене и писане. В първия случай съобщението се обработва от "handleMessageWritable" (фиг. 3.76.), а във втория - от "handleChunk" (фиг. 3.95.).

След обмяната на канали за комуникация директната връзка между провайдер и клиент е установена.

```

handleMessageWritable(message) {
  switch (message.type) {
    case "copyFile":
        this.connector.lockFile(message.payload, () => {
            this.copyFile(message.payload);
        });
        break;
    case "moveFile":
        this.connector.lockFile(message.payload, () => {

```

```

        this.moveFile(message.payload);
    });
    break;
case "deleteFile":
    this.connector.lockFile(message.payload, () => {
        this.deleteFile(message.payload);
    });
    break;
case "uploadFile":
    this.prepareUpload(message.payload);
    break;
default:
    this.handleMessage(message);
}
}

```

Фиг. 3.76: Обработка на съобщение, получено в провайдер по канал "clientMessageWritable"

(изходен код: "datastreamer-provider/src/app/units/client.js")

```

handleMessage(message) {
    switch (message.type) {
        case "openDirectory":
            this.changeDirectory(message.payload);
            this.scanDirectory();
            break;
        case "downloadFile":
            this.connector.lockFile(message.payload, () => {
                this.sendFile(message.payload);
            });
            break;
    }
}
}

```

Фиг. 3.77: Обработка на съобщение, получено в провайдер по канал "clientMessage"

(изходен код: "datastreamer-provider/src/app/units/client.js")

3.4. Многопроцесова архитектура на провайдер

Тъй като провайдерът е изграден с библиотеката Electron, той работи чрез два вида процеси - главен процес (main) и рендериращ процес (renderer). Главният е този, който се изпълнява при стартиране на приложението, и чрез който се пускат всички останали процеси. Нови рендериращи процеси се получават чрез създаване на нови браузър прозорци.

Главният процес след стартирането си създава нов браузър прозорец, който съдържа потребителския интерфейс за конфигуриране на провайдъра, както и уеб сокет връзката към сървъра (фиг. 3.78.).

```
let mainWindow;

const createWindow = () => {
  mainWindow = new BrowserWindow({ width: 800, height: 600 });
  mainWindow.loadURL(url.format({
    pathname: path.join(__dirname, "index.html"),
    protocol: "file:"
  }));

  mainWindow.on("closed", () => {
    mainWindow = null;
  });
  ipcHandler(mainWindow);
}
app.on("ready", createWindow);
```

Фиг. 3.78: Създаване на основния рендериращ процес
(изходен код: “datastreamer-provider/src/main.js”)

Клиентските процеси представляват скрити браузър прозорци. В главния процес се запазва асоциативен масив, като всеки елемент съдържа информация за клиент, включително инстанцията на неговия браузър прозорец. Елементите се достъпват чрез сокет идентификатор на клиент.

Необходима е комуникация между клиентските процеси и процеса с потребителския интерфейс, но всички те са рендериращи процеси, а Electron библиотеката позволява единствено комуникация между рендериращ процес и главен. Проблемът се решава чрез координация на съобщенията в главния процес. Всички съобщения се предават на главния процес, а той ги препраща до отбелязания краен получател.

3.5. Сканиране на файлове от провайдър

Сканирането на файлове се извършва във всеки отделен клиентски процес. Node модулът “chokidar”^[29] представлява абстракция над способите на Node.js за следене на промени във файловата система.

Фигура 3.79. представя имплементация на сканиране на файлове. Най-напред се задава директорията, чието съдържание да бъде сканирано, като се използва нейният абсолютен път. Той се получава чрез комбиниране на зададената главна директория с избраните от клиента поддиректории. След това се дефинира поведение при добавяне, промяна или премахване на файл или директория.

При добавяне или промяна на файл или папка се взема MIME (Multipurpose Internet Mail Extensions) типа чрез проверка на магическото число (сигнатурата) на файла. След това се вземат други метаданни за всеки файл - име, път, тип, права за достъп и размер.

В началото на сканирането всички открити файлове или директории се третират като току що добавени.

```
function scanDirectory() {
    let isCurrentDirectory = true;
    const path = pathModule.join(this.selectedMainDirectory,
    this.currentDirectory);
    if (this.watcher) {
        this.watcher.close();
    }
    this.watcher = chokidar.watch(path, this.watcherOptions);

    this.sendMessage("newScan");
    this.watcher
        .on("add", (path, stats) => {
            magic.detectFile(path).then(mime => {
                return this.getFileMetadata(path, stats, mime);
            }).then(fileMetadata => {
                this.sendMessage("add", fileMetadata);
            }).catch(error => {
                this.sendMessage("error", error);
            });
        })
        .on("addDir", (path, stats) => {
            if (isCurrentDirectory) {
                isCurrentDirectory = false;
            } else {
                this.getFileMetadata(
                    path, stats, "inode/directory"
                ).then(fileMetadata => {
                    this.sendMessage("addDir", fileMetadata);
                }).catch(error => {
                    this.sendMessage("error", error);
                });
            }
        })
    }
```



```

        });
    }
})
.on("change", (path, stats) => {
    magic.detectFile(path).then(mime => {
        return this.getFileMetadata(path, stats, mime);
    }).then(fileMetadata => {
        this.sendMessage("change", fileMetadata);
    }).catch(error => {
        this.sendMessage("error", error);
    });
})
.on("unlink", path => {
    this.sendMessage("unlink", this.getRelativePath(path));
})
.on("unlinkDir", path => {
    this.sendMessage("unlinkDir", this.getRelativePath(path));
})
.on("error", error => {
    this.sendMessage("error", error);
})
.on("ready", () => {
    this.sendMessage("scanFinished");
});

```

Фиг. 3.79: Сканиране на съдържанието на избрана директория

(изходен код: "datastreamer-provider/src/app/units/modules/scan-directory.js")

3.6. Представяне на съдържанието на директория в клиента

В процеса на сканиране при отчитане на промяна във файловата система се изпраща съобщение до клиента.

При добавяне на файл или папка провайдърът изпраща на клиента съобщение съответно "add" или "addDir", заедно с метаданните на файла. В резултат данните се запазват и се визуализират на клиента.

При промяна на файл провайдърът изпраща съобщение "change", заедно с новите метаданни на файла. Редактирането на името на файл или негово преместване не се отчита като промяна на един файл, а като изтриване на един и добавяне на нов. Затова когато клиентът получи съобщението "change", от запазените при клиента файлове се изтрива този със същия път и се добавя новата информация.

При изтриване на файл или папка провайдърът изпраща на клиента съобщение съответно “unlink” или “unlinkDir”, заедно с пътя на премахнатия файл или директория. Клиентът получава съобщението и премахва запазения файл или провайдър с предоставения път. (фиг. 3.75.)

3.7. Отваряне на поддиректория

При отваряне на поддиректория от страна на клиента се изпраща съобщение “openDirectory” към провайдъра, съдържащо релативния път до отворената папка (фиг. 3.80.). В клиентския процес се променя текущата директория и се изпълнява ново сканиране (фиг. 3.81., 3.82.).

```
navigate(directoryPath) {  
  this.props.clearFiles();  
  this.RTC.sendMessage("openDirectory", directoryPath);  
}
```

Фиг. 3.80: Заявка от клиент за отваряне на директория

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

```
handleMessage(message) {  
  switch (message.type) {  
    case "openDirectory":  
      this.changeDirectory(message.payload);  
      this.scanDirectory();  
      break;  
    [...]  
  }  
}
```

Фиг. 3.81: Обработка на получено съобщение “openDirectory” в провайдър

(изходен код: “datastreamer-provider/src/app/units/client.js”)

```
changeDirectory(selectedDirectory) {  
  const path = this.resolvePath(selectedDirectory);  
  if (!selectedDirectory) {  
    throw `Invalid directory ${selectedDirectory}`;  
  }  
  if (this.watcher) {  
    this.watcher.close();  
  }  
  this.watcher = null;
```

```

    this.currentDirectory = path.substring(this.selectedMainDirectory.length);
    this.connector.changeDirectory(path);
}

```

Фиг. 3.82: Промяна на текущата директория

(изходен код: “datastreamer-provider/src/app/units/client.js”)

3.8. Изтегляне на файл

При заявка от потребител да изтегли файл метаданните му се запазват в масив, като контекстът е “file” (фиг. 3.83.).

```

addToDownloads(file, context) {
    this.RTC.downloads.push({
        ...file,
        context,
        chunkArray: [],
        received: 0
    });
    if (this.RTC.downloads.length === 1) {
        this.requestDownload();
    }
}

```

Фиг. 3.83: Добавяне на файл за изтегляне

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

Контекстът указва как да бъде обработен файла след изтеглянето му. В случая означава, че този файл трябва да бъде записан на дисковото пространство на клиента. В друг контекст файлът би се визуализирал в брауъра като изображение или текст.

Ако няма процес на изтегляне до провайдъра се изпраща съобщение “downloadFile” заедно с пътя на файла, чиято информация е записана в първия елемент от масива (фиг. 3.84.).

```

requestDownload() {
    const download = this.RTC.downloads[0];
    this.RTC.sendMessage("downloadFile", download.path);
}

```

Фиг. 3.84: Команда до провайдър файл да бъде изтеглен

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

В провайдъра се създава поток за четене и файлът започва да се изпраща до клиента на парчета по канала с наименование “providerFile” (фиг. 3.85., 3.86.).

```
case "downloadFile":
  this.connector.lockFile(message.payload, () => {
    this.sendFile(message.payload);
  });
  break;
```

Фиг. 3.85: Обработка на съобщението “downloadFile” изтеглянето на избран файл да започне

(изходен код: “datastreamer-provider/src/app/units/client.js”)

```
sendFile(filePath) {
  try {
    const path = this.getAbsolutePath(filePath);
    this.readStream = fs.createReadStream(path);
    this.sendFileChannel.onbufferedamountlow = () => {
      if (this.readStream) {
        this.readStream.resume();
      }
    }
    this.readStream.on("data", chunk => {
      if (this.sendFileChannel.bufferedAmount >
this.bufferedAmountHighThreshold) {
        this.readStream.pause();
      }
      this.sendFileChannel.send(chunk);
    });
    this.readStream.on("end", () => {
      this.readStream = null;
      this.connector.unlockFile(filePath);
    });
  } catch (e) {
    [...]
  }
}
```

Фиг. 3.86: Изпращане на файл от провайдър до клиент на парчета

(изходен код: “datastreamer-provider/src/app/units/client.js”)

Обикновено четенето на файл от диска е по-бързо от скоростта на изпращането му по мрежата и при липса на специални мерки буферът на канала се препълва, когато всяко прочетено парче се пусне за изпращане.

Решението на този проблем е чрез редуване на прекъсване и възстановяване на четенето на файла. При достигане на определена граница потокът се спира временно. Зададена е стойност на свойството “bufferedAmountLowThreshold” на канала. Когато количеството буферирани данни в канала стане по-малко от тази стойност потокът за четене стартира и прочетените парчета данни се пускат в канала. Спирането и пускането на потока се повтаря многократно до успешното изпращане на файла.

```
chunkHandler(chunk) {  
  try {  
    this.RTC.downloads[0].chunkArray.push(chunk);  
    this.RTC.downloads[0].received += chunk.byteLength;  
    if (this.RTC.downloads[0].received >= this.RTC.downloads[0].size) {  
      this.finishDownload();  
    }  
  } catch (error) {  
    this.RTC.downloads.shift();  
    if (this.RTC.downloads.length > 0) {  
      this.requestDownload();  
    }  
  }  
}
```

Фиг. 3.87: Обработка на получено от клиента парче от файл

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

Всички парчета се запазват в браузъра на клиента. Когато сумата от размерите на всички парчета достигне размера на целия файл се проверява контекста на изтеглянето (фиг. 3.88.).

```
finishDownload() {  
  const downloaded = this.RTC.downloads.shift();  
  try {  
    this.setState({  
      downloadPercent: 0
```

```

    });
    const received = new Blob(downloaded.chunkArray, { type: downloaded.mime
});
    switch (downloaded.context) {
        case "file":
            FileSaver.saveAs(received, path.basename(downloaded.path));
            break;
        case "image":
            this.props.setImage(URL.createObjectURL(received));
            break;
        case "text":
            this.props.setText(chunkArrayToText(downloaded.chunkArray));
            break;
    }
    if (this.RTC.downloads.length > 0) {
        this.requestDownload();
    }
} catch (error) {
    if (this.RTC.downloads.length > 0) {
        this.requestDownload();
    }
}
}
}

```

Фиг. 3.88: Обработка на приключило изтегляне на файл

(изходен код: "datastreamer-server/src/views/pages/home-page.jsx")

В случая контекстът е "file", при което се създава блоб, който може да се запише в избрано място от дисковото пространство на потребителя. Елементите в масива с файлове за изтегляне се измества. Ако има информация за файл, разположена на първия индекс от масива, започва процедура по сваляне на нов файл.

3.8.1. Визуализация на изображения

За да се визуализира едно изображение е необходимо файлът да се изтегли. Процесът е същият като описания в т. 3.8., но контекстът е "image". При приключване на изтеглянето се конструира блоб, за който се създава URL (Universal Resource Locator - универсален указател на ресурс) и чрез него се визуализира изображението.

3.8.2. Визуализация на текст

Възможно е да се прочете съдържанието на един текстов файл без да е необходимо да се сваля на дисковото пространство. Той се изтегля чрез процедурата, описана в т. 3.8., но контекстът е “text”. След получаването на всички парчета от файла те се преобразуват до текст, който се визуализира.

3.9. Качване на файл

При избор на файл за качване неговите данни се запазват в клиента, а до провайдъра се изпраща съобщение “uploadFile” с името и размера на файла (фиг. 3.89.).

```
handleUploadFiles(files) {  
  let file = files[0];  
  this.setState({  
    uploads: files  
  });  
  this.RTC.sendMessageWritable("uploadFile", {  
    name: file.name,  
    size: file.size  
  });  
}
```

Фиг. 3.89: Заявка за изпращане на файл от клиент

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

Провайдърът запазва тези данни, създава поток за писане и изпраща съобщение “readyForFile” до клиента (фиг. 3.90., 3.91.).

```
case "uploadFile":  
  this.prepareUpload(message.payload);  
  break;
```

Фиг. 3.90: Обработка на получено съобщение “uploadFile” от клиент до провайдър

(изходен код: “datastreamer-provider/src/app/units/client.js”)

```
prepareUpload(fileData) {  
  this.uploadedFileData = fileData;  
  const sanitizedFileName = sanitize(pathModule.basename(fileData.name));
```

```

    if (!sanitizedFileName) {
      return;
    }
    const filePath = pathModule.join(
      this.selectedMainDirectory,
      this.currentDirectory,
      sanitizedFileName
    );
    this.writeStream = fs.createWriteStream(filePath);
    this.writeStream.on("finish", () => {
      this.writeStream = null;
      this.uploadedFileData = null;
    });
    this.writeStream.on("error", error => {
      this.writeStream.end();
      this.sendMessage("error", error);
    });
    this.receivedBytes = 0;
    this.sendMessage("readyForFile");
  }

```

Фиг. 3.91: Подготовка на провайдър за получаване на файл от клиент
(изходен код: "datastreamer-provider/src/app/units/client.js")

Клиентът създава генератор на парчета от файла (фиг. 3.93., 3.94.). Генерирането стартира и всяко парче се праща по канала с обозначение "clientFile".

```

case "readyForFile":
  this.sendFiles();
  break;

```

Фиг. 3.92: Обработка на получено от провайдър съобщение "readyForFiles" в клиента

(изходен код: "datastreamer-server/src/views/pages/home-page.jsx")

```

sendFiles() {
  const reader = new FileReader();
  const chunkGenerator = fileChunkGenerator(this.state.uploads[0], reader,
  this.RTC.chunkSize);
  let received = 0;
  this.RTC.sendFileChannel.onbufferedamountlow = () => {
    if (received.value < this.state.uploads[0].size) {
      received = chunkGenerator.next();
    }
  };
  reader.onload = load => {

```



```

    this.RTC.sendFileChannel.send(load.target.result);
    if (received.value < this.state.uploads[0].size) {
        if (this.RTC.sendFileChannel.bufferedAmount < this.RTC.bufferLimit)
        {
            received = chunkGenerator.next();
        }
        else {
            chunkGenerator.return();
        }
    };
    received = chunkGenerator.next();
}

```

Фиг. 3.93: Изпращане на файл на парчета от клиент до провайдер
(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

```

function* fileChunkGenerator(file, reader, chunkSize) {
    let received = 0;
    while (file.size > received) {
        const slice = file.slice(received, received + chunkSize);
        reader.readAsArrayBuffer(slice);
        received += chunkSize;
        yield received;
    }
}

```

Фиг. 3.94: Генератор на парчета от файл
(изходен код: “datastreamer-server/src/modules/file-chunk-generator.js”)

Съществува проблем, подобен на описания в т. 3.8. Буферът на канала може да се препълни и в такъв случай генерирането на парчета временно се прекъсва. При достигане на долна граница, в която е прието, че каналът вече не е препълнен, генерирането и изпращането на парчета се възстановява. Този процес се повтаря многократно до успешното изпращане на файла.

Провайдерът следи за количеството получени данни и при достигане на размера на файла потокът за писане се затваря (фиг. 3.95.).

```

handleChunk(chunk) {
    try {
        this.receivedBytes += chunk.byteLength;
        if (this.receivedBytes >= this.uploadedFileData.size) {

```

```

        this.writeStream.end(Buffer.from(chunk));
    } else {
        this.writeStream.write(Buffer.from(chunk));
    }
} catch (error) {
    this.sendMessage("error", error);
}
}

```

Фиг. 3.95: Обработка на получено от провайдер парче файл

(изходен код: “datastreamer-provider/src/app/units/client.js”)

В процеса на следене на директорията за промени, описан в т. 3.5., се отчита добавянето на нов файл и данните за него се изпращат до клиента по начин, описан в т. 3.6.

Клиентът вижда качения файл в отворената от него директория и това е признак за успешно качване.

3.10. Копиране на файл

Когато клиентът избере да копира даден файл се изпраща до провайдъра съобщение “copyFile” с пътя на файла (фиг. 3.96.).

```

copyFiles() {
    this.props.selection.selected.forEach(file => {
        this.RTC.sendMessageWritable("copyFile", file.path);
    });
    this.props.clearSelection();
}

```

Фиг. 3.96: Заявка от клиента за копиране на файл

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

Провайдърът копира съответстващия файл в текущо отворената от клиента директория (фиг. 3.97. и 3.98.).

```
case "copyFile":
  this.connector.lockFile(message.payload, () => {
    this.copyFile(message.payload);
  });
  break;
```

Фиг. 3.97: Обработка на съобщението “copyFile”, изпратено от клиент до провайдер

(изходен код: “datastreamer-provider/src/app/units/client.js”)

```
copyFile(filePath) {
  const source = this.getAbsolutePath(filePath);
  const basename = sanitize(pathModule.basename(filePath));
  if (!basename) {
    return;
  }
  const destination = pathModule.join(this.selectedMainDirectory,
  this.currentDirectory, basename);
  fs.copy(source, destination, {
    overwrite: false,
    errorOnExist: true
  }).then(() => {
    this.connector.unlockFile(filePath);
  }).catch(error => {
    this.sendMessage("error", error);
  });
}
```

Фиг. 3.98: Осъществяване на копиране на файл

(изходен код: “datastreamer-provider/src/app/units/client.js”)

3.11. Преместване на файл

Когато клиентът избере да премести даден файл се изпраща до провайдъра съобщение “moveFile” с пътя на файла (фиг. 3.99.).

```
moveFiles() {
  this.props.selection.selected.forEach(file => {
    this.RTC.sendMessageWritable("moveFile", file.path);
  });
  this.props.clearSelection();
}
```

Фиг. 3.99: Заявка от клиента за преместване на файл

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

Провайдърът премества избрания файл в текущо отворената от клиента директория (фиг. 3.100. и 3.101.).

```
case "moveFile":
  this.connector.lockFile(message.payload, () => {
    this.moveFile(message.payload);
  });
  break;
```

Фиг. 3.100: Обработка на съобщението “moveFile”, изпратено от клиент до провайдър

(изходен код: “datastreamer-provider/src/app/units/client.js”)

```
moveFile(filePath) {
  const source = this.getAbsolutePath(filePath);
  const basename = sanitize(pathModule.basename(filePath));
  if (!basename) {
    return;
  }
  const destination = pathModule.join(this.selectedMainDirectory,
  this.currentDirectory, basename);
  fs.move(source, destination).then(() => {
    this.connector.unlockFile(filePath);
  }).catch(error => {
    this.sendMessage("error", error);
  });
}
```

Фиг. 3.101: Имплементация на преместване на файл

(изходен код: “datastreamer-provider/src/app/units/client.js”)

3.12. Изтриване на файл

Когато клиентът избере да изтрие файл съобщението “deleteFile” се изпраща до провайдъра, заедно с пътя до файла (фиг. 3.102.).

```
deleteFiles() {
  this.props.selection.selected.forEach(file => {
    this.RTC.sendMessageWritable("deleteFile", file.path);
  });
  this.props.clearSelection();
}
```

Фиг. 3.102: Заявка от клиента за изтриване на файл

(изходен код: “datastreamer-server/src/views/pages/home-page.jsx”)

Съобщението се обработва (фиг. 3.103.) и избраният файл бива преместен в кошчето от провайдъра (фиг. 3.104.).

```
case "deleteFile":
  this.connector.lockFile(message.payload, () => {
    this.deleteFile(message.payload);
  });
  break;
```

Фиг. 3.103: Обработка на съобщението “deleteFile”, изпратено от клиент до провайдър

(изходен код: “datastreamer-provider/src/app/units/client.js”)

```
deleteFile(filePath) {
  const source = this.getAbsolutePath(filePath);
  trash([source], { glob: false }).then(() => {
    this.connector.unlockFile(filePath);
  }).catch(error => {
    this.sendMessage("error", error);
  });
}
```

Фиг. 3.104: Имплементация на преместване на файл в кошчето

(изходен код: “datastreamer-provider/src/app/units/client.js”)

3.13. Управление на права за достъп

Потребителският интерфейс на провайдъра предоставя възможност за управление на правата за достъп до него.

Налични са два ключа, с които се задават правата за достъп до провайдър по подразбиране.

Тези права се използват, когато към провайдър се свързва клиент, за който няма индивидуално зададени права. При промяна в състоянието на ключ се извиква една и съща функция - “handleToggleDefaultAccessRule”, но с различна стойност на параметъра “accessRule”, който приема - “readable” или “writable”, тоест промяна в правата съответно за четене или писане. (изходен код: “datastreamer-provider/src/app/views/pages/home-page.jsx”) Генерира се новият вариант на правата за достъп и заедно с ключа за достъп на провайдъра се изпраща до сървъра чрез POST заявка до крайна точка

“access/default”.

(изходен

код:

“datastreamer-server/src/routes/client-access-rules.js”)

Извиква се функцията “setProviderDefaultRule”. Сървърът валидира форматът на изпратената информация, верифицира изпратения ключ за достъп, след което запазва в базата данни новите стойности на параметрите. (изходен код: “datastreamer-server/src/db/postgres/client-access-rules.js”)

Когато клиент се свърже към провайдер се дава възможност от потребителския интерфейс на провайдера да се задават индивидуални права на достъп за клиента.

При промяна на параметър се изпълнява функция “handleToggleAccessRule” с аргументи сокет идентификацията на клиента, чийто индивидуални права се променят, и параметъра, който трябва да се промени.

На сървъра се изпраща POST заявка на крайна точка “access/client” с ключа за достъп на провайдера и избраните нови права за достъп. (изходен код: “datastreamer-server/src/routes/client-access-rules.js”) Форматът на данните се валидира, предоставеният ключ за достъп се верифицира и новите права се запазват в базата данни. (изходен код: “datastreamer-server/src/db/postgres/client-access-rules.js”)

При наличие на индивидуални права за достъп за даден клиент те винаги ще бъдат с приоритет пред стандартните, принадлежащи на провайдера, при промяна на които не се въздейства върху индивидуалните права.

3.14. Настройки на акаунт

Провайдърът и клиентът разполагат със страница за настройки на профила. Тя позволява промяна на парола или изтриване на акаунта. При изпращане на промените се изпраща и ключ за достъп. Във файла “datastreamer-server/src/views/components/change-password-component.jsx” от изходния код е представена формата за промяна на парола на клиент а във файла “datastreamer-server/src/views/components/delete-account-component.jsx” - формата за изтриване на акаунта на клиент.

На сървъра ключът за достъп се инвалидира. (изходен код: *“datastreamer-server/src/db/postgres/client.js”*) В случаите с промяна на парола се издава нов ключ, който се връща с отговора.

Във файловете *“datastreamer-provider/src/app/views/components/change-account-password-component.jsx”*, *“datastreamer-provider/src/app/views/components/change-client-connect-password-component.jsx”* и *“datastreamer-provider/src/app/views/components/delete-account-component.jsx”* са представени формите съответно за промяна на парола на акаунт на провайдер, за промяна на парола за свързване на клиент към провайдер и за изтриване на акаунта на провайдер. Тези операции се извършват на сървъра във файла *“datastreamer-server/src/db/postgres/provider.js”* от изходния код.

ЧЕТВЪРТА ГЛАВА

4.1. Изисквания към компютърна конфигурация и инсталация

4.1.1. Конфигуриране и стартиране на сървър

Сървърната част от системата за отдалечено управление на файлове чрез P2P връзка може лесно да се конфигурира и стартира в Docker контейнери. Изисква се единствено да са инсталирани Docker CE и Docker Compose.

Необходимо е чрез конзола да се отвори директорията от изходния код на сървъра, в която се намират Dockerfile и docker-compose.yml:

```
$ cd datastreamer-server/src
```

Чрез следващата команда се извикват скриптове за създаване на docker контейнери, инсталиране на Node пакети, създаване на база данни и конфигуриране. Накрая сървърът е готов да обработва заявки:

```
$ docker-compose up --build
```

Веднъж след като е изграден, docker контейнерът може да се стартира без флага “build”:

```
$ docker-compose up
```

Горните команди изискват администраторски права, за да бъдат изпълнени.

Сървърното приложение е тествано на операционна система Debian GNU/Linux 9 (Stretch) 64-bit, пусната на виртуална машина с хипервайзор VirtualBox. За инсталация на Docker CE са следвани инструкциите на тази страница:

“<https://docs.docker.com/install/linux/docker-ce/debian/#install-using-the-repository>”.

За инсталация на Docker Compose са следвани инструкциите за Linux на тази страница: “<https://docs.docker.com/compose/install/#install-compose>”.

4.1.2. Стартиране на клиентско приложение

За да може да бъде отворено клиентското приложение, е необходимо да има работещ локален сървър, следвайки стъпките в т. 4.1.1. Приложението се отваря чрез браузър на адрес “localhost”. Поддържаните браузъри са Chrome версия 57.0 или по-нова и Firefox версия 52 или по-нова.

4.1.3. Конфигуриране и стартиране на провайдър

За изпълняване на десктоп приложението (провайдър) е необходимо да бъде инсталиран Node.js с версия 8.9.4. или по-нова. Следват инструкции за инсталация за операционна система Debian 9:

1. Необходимо е чрез конзола да се отвори изходния код на приложението:

```
$ cd datastreamer-provider/src
```

2. За компилиране на модулите е необходимо инсталирането на определени пакети:

```
$ sudo apt install libssl-dev libgconf2-4 zlib1g-dev
```


3. Инсталиране и компилиране на модули:

```
$ npm install
```

4. Конфигуриране на модулите да работят с текущата версия на Electron:

```
$ ./node_modules/.bin/electron-rebuild
```

5. Пакетиране на модулите:

```
$ npm run build
```

6. Стартиране на приложението:

```
$ npm start
```

Провайдърът е тестван на операционна система Debian GNU/Linux 9 (Stretch) 64-bit с десктоп среда Gnome 3.22. ОС е пусната на виртуална машина с хипервайзор VirtualBox. Следвани са инструкциите за инсталация на Node.js 8.x, предоставени на тази страница: ["https://nodejs.org/en/download/package-manager/#debian-and-ubuntu-based-linux-distributions"](https://nodejs.org/en/download/package-manager/#debian-and-ubuntu-based-linux-distributions).

4.2. Ръководство на потребителя

4.2.1. Използване на провайдър за предоставяне на файловете

4.2.1.1. Вход

При стартиране на десктоп приложението се визуализира форма за вход, чрез която провайдърът може да се идентифицира (фиг. 4.1.). Тя се състои от име на провайдър и парола.

The screenshot shows a web interface for 'Datastreamer'. In the top left corner, the word 'Datastreamer' is displayed. In the top right corner, there is a blue plus icon followed by the text 'Create provider'. The central part of the page features the heading 'Log into your account'. Below this heading is a login form. The form consists of two input fields: the first is labeled 'Username' with a person icon, and the second is labeled 'Password' with a lock icon. Below these fields is a dark button with the text 'Log in'.

Фиг. 4.1: Форма за вход в провайдер

4.2.1.2. Регистрация

Потребителят може да се регистрира, ако не разполага с акаунт. Регистрационната форма (фиг. 4.2.) се отваря чрез хипервръзката “Create provider”, разположен в горната дясна част на приложението при отворена форма за вход.

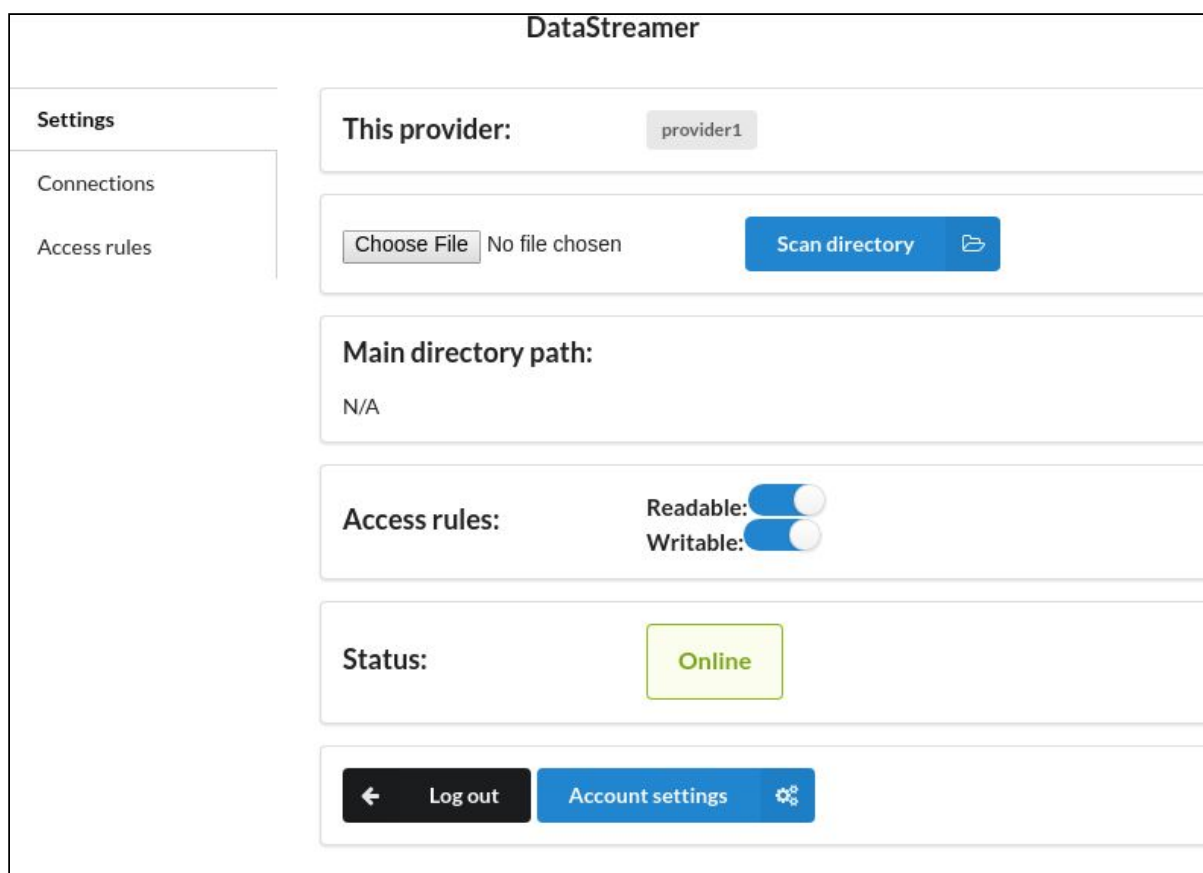
The screenshot shows a web interface for 'Datastreamer'. At the top left is the title 'Datastreamer' and at the top right is a 'Go back' link with a blue arrow. The main heading is 'Create new provider'. Below it is a registration form with five input fields, each with a lock icon on the left: 'Username', 'Password', 'Confirm password', 'Client connect password', and 'Confirm client connect password'. At the bottom of the form is a black 'Register' button.

Фиг. 4.2: Форма за регистрация на провайдер

За да се регистрира успешно, потребителят трябва да въведе в регистрационната форма име на провайдер, парола, потвърждение на парола, парола за клиентски достъп и потвърждение на парола за клиентски достъп, след което да натисне бутона “Register”. Полетата не трябва да бъдат празни. Името на провайдъра трябва да бъде с дължина между 5 и 60 символа. Паролите се изисква да бъдат с дължина между 8 и 100 символа, да съдържат поне една цифра, поне една главна буква и поне една малка буква.

4.2.1.3. Контролен панел на провайдер

След успешен вход или регистрация се визуализира контролният панел на провайдъра (фиг. 4.3.). В част от прозореца е разположено навигационно меню с три подпрозореца - “Settings”, “Connections” и “Access rules”.



Фиг. 4.3: Контролен панел на провайдер

4.2.1.3.1. Конфигурация на провайдер

По подразбиране е отворен подпрозорецът “Settings”, показан на фигура 4.3., който предоставя възможност за конфигурация. В него най-отгоре се визуализира името на провайдера.

Под името на провайдера е разположена форма за избор на директория, която да бъде начална точка за сканиране. Потребителите, които успешно се свържат към провайдера, ще могат да достъпват всяка една поддиректория на избраната начална точка, но не и нейните родителски директории.

Под формата се изписва абсолютният път на текущо избраната директория.

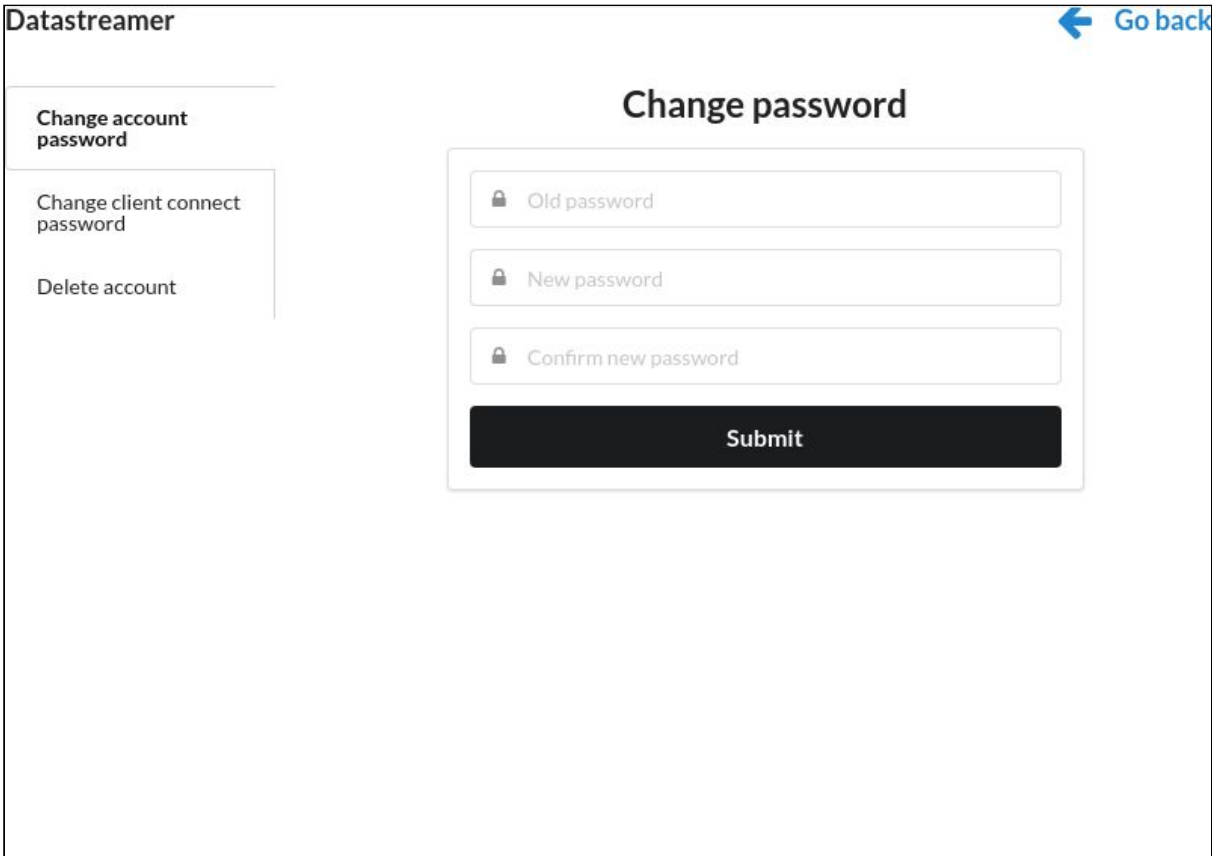
Следват настройки на достъп до провайдера, които представляват два ключа: “Readable” и “Writable”, означаващи наличие или липса съответно на право за четене и право за писане. Ключът “Writable” е скрит, когато не е

налично право за писане. Промените по правата за достъп до провайдъра се отразяват на клиента при следващото му свързване.

Следва информация за статуса на провайдъра. Той показва “Online”, ако има уеб сокет връзка със сървъра, или “Offline” в противния случай. При наличие на грешки се оцветява в червено и се изписва информация за грешката. През останалото време е оцветен в зелено.

4.2.1.3.2. Настройки на профил

Най-отдолу е разположена връзка към страница с настройки на акаунта. Отварянето ѝ прекъсва всички връзки към провайдъра.

The screenshot shows a web interface for 'Datastreamer'. At the top right, there is a blue arrow pointing left and the text 'Go back'. On the left side, there is a vertical menu with three items: 'Change account password', 'Change client connect password', and 'Delete account'. The main content area is titled 'Change password' and contains three input fields, each with a lock icon on the left: 'Old password', 'New password', and 'Confirm new password'. Below these fields is a large black button with the word 'Submit' in white text.

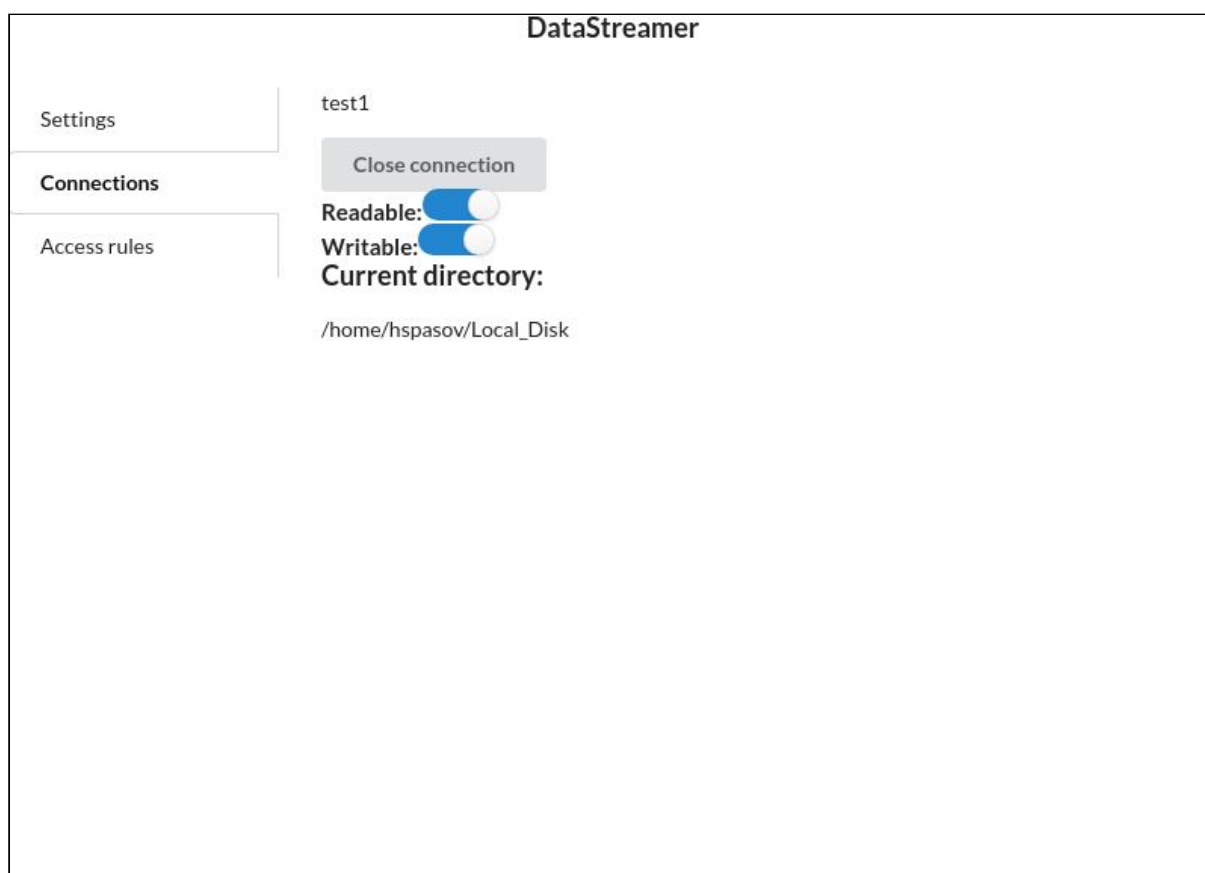
Фиг. 4.4: Форма за промяна на парола на акаунт

Тя се състои от три подпрозореца (фиг. 4.4.). В първия е разположена форма за промяна на паролата на провайдъра а във втория - форма за промяна на паролата за клиентски достъп до провайдъра. И в двата случая е необходимо въвеждане на настоящата парола на провайдъра, въвеждане на

новата парола от съответния тип, както и потвърждение на новата парола чрез повторно въвеждане. В третия подпрозорец е разположена форма за изтриване на акаунта. Тя изисква въвеждане на паролата на провайдъра. След изпращане на формата автоматично се излиза от профила и се визуализира формата за вход в акаунт.

4.2.1.3.3. Преглед и контрол на свързани клиенти

В контролния панел вторият подпрозорец е “Connections”, в който се визуализират всички свързали се към провайдъра клиенти (фиг. 4.5.).



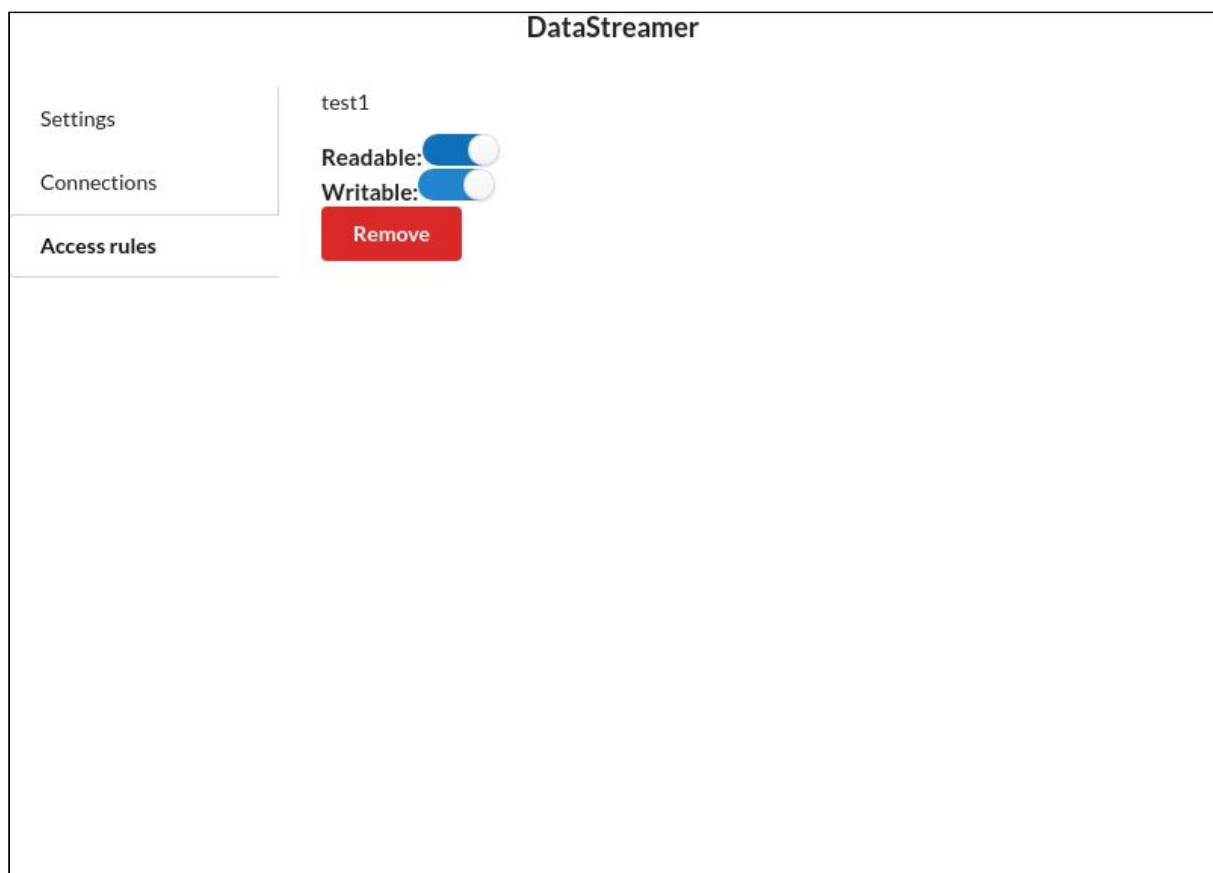
Фиг. 4.5: Визуализация на свързаните към провайдър клиенти

Предоставя се информация за потребителското име на клиента и директорията, която разглежда. Възможно е задаване на индивидуални права за достъп до провайдъра на всеки клиент. Това става чрез промяна на състоянието на ключовете “Readable” и “Writable”, подобно на настройките за

достъп до провайдер по подразбиране. Връзката може да бъде прекратена чрез натискане на бутон “Close connection” за даден клиент.

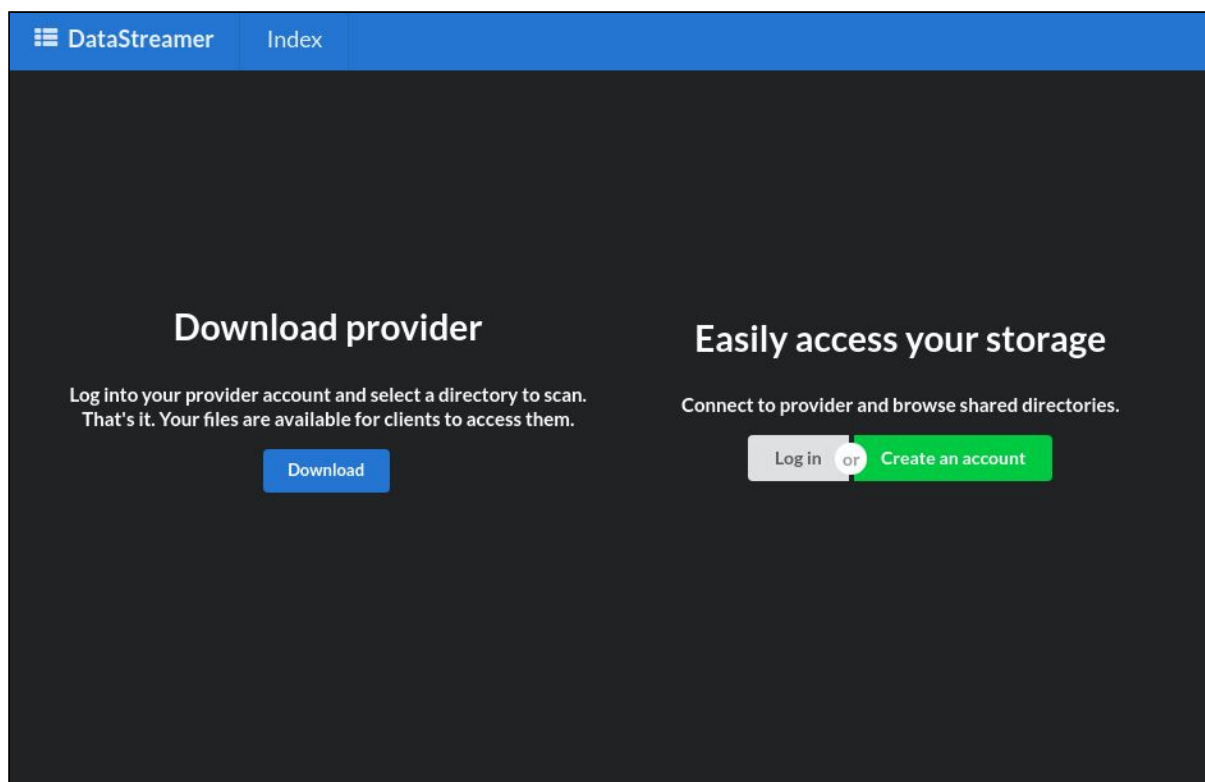
4.2.1.3.4. Преглед и контрол на индивидуално зададени клиентски права за достъп

Третият подпрозорец е “Access rules”, в който се визуализират всички клиенти, за които има зададени индивидуални права за достъп (фиг. 4.6.). Опцията за промяна на техните права, независимо дали са се свързали или не, са налични.



Фиг. 4.6: Визуализация на клиентите със зададени индивидуални права за достъп

4.2.2. Използване на клиент за достъпване на файлове



Фиг. 4.7.: Начална страница на клиентското приложение

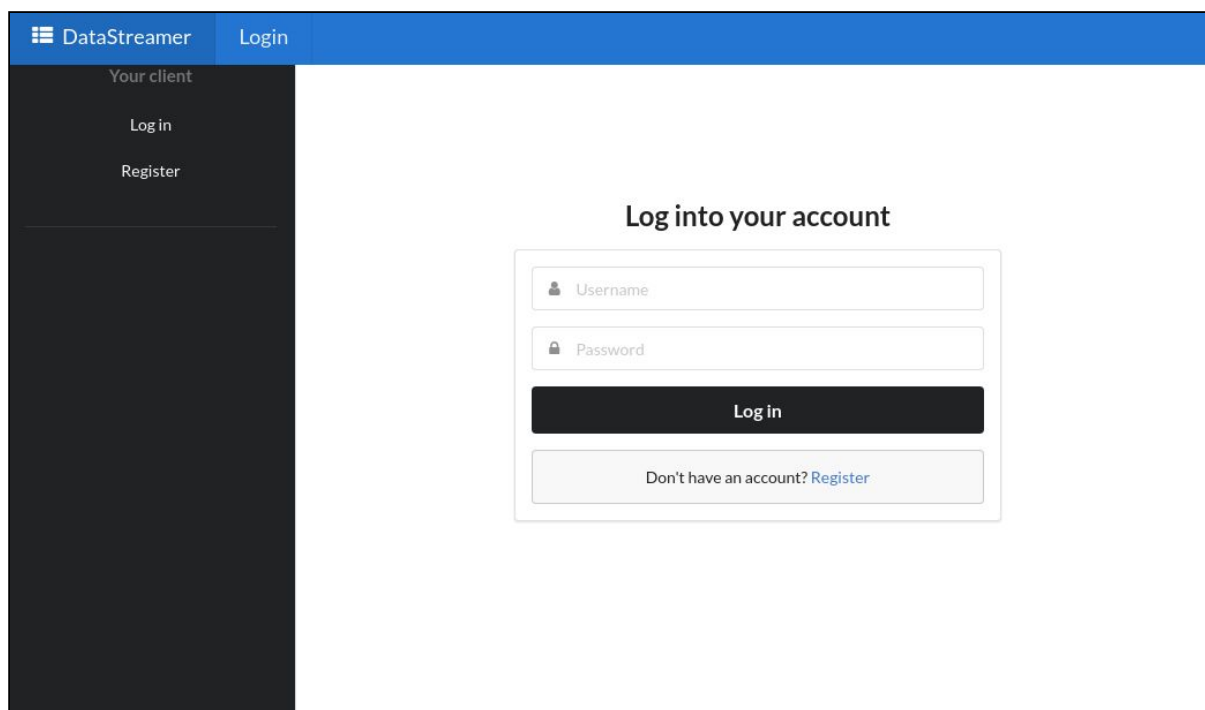
Уеб страницата притежава горно и странично меню.

Горното меню се състои от надпис “DataStreamer” и до него е разположено името на страницата, която е отворена. (фиг. 4.7.) При натискане на надписа “DataStreamer” страничното меню отляво може да се показва или скрива.

Страничното меню показва връзки за навигация, информация за текущия клиент и провайдъра, към който е свързан. Ако няма влязъл в профила си потребител в менюто са разположени две хипервръзки - “Log in” и “Register”.

4.2.2.1. Вход

При натискане на “Log in” се отваря форма за вход в профил (фиг. 4.8.).



Фиг. 4.8: Вход на клиент

Потребителят трябва да въведе своето потребителско име и парола, след което да изпрати формата чрез натискане на бутона “Log in”, за да влезе в профила си.

4.2.2.2. Регистрация

Ако потребителят не разполага с акаунт може да натисне връзката “Register”, разположена в страницата за вход или в страничното меню.

The screenshot displays a web application interface for 'DataStreamer'. At the top, there is a blue navigation bar with a hamburger menu icon, the text 'DataStreamer', and a 'Register' tab. On the left side, there is a dark sidebar with the text 'Your client' and two links: 'Log in' and 'Register'. The main content area is white and features a 'Create new account' form. The form includes three input fields: 'Username' (with a person icon), 'Password' (with a lock icon), and 'Confirm password' (with a lock icon). Below these fields is a dark 'Register' button. At the bottom of the form is a light gray button with the text 'Already have an account? [Log in](#)'.

Фиг. 4.9: Регистрация на клиент

Отваря се регистрационна форма (фиг. 4.9.), която изисква въвеждане на потребителско име, парола и повторение на парола. Потребителското име трябва да бъде с дължина между 5 и 60 символа. Паролата трябва да съдържа поне една малка буква, поне една голяма буква, поне една цифра и да бъде с дължина между 8 и 100 символа.

4.2.2.3. Форма за свързване към провайдер

След успешен вход или регистрация се отваря страница с форма за свързване към провайдер (фиг. 4.10.).

The screenshot displays the 'DataStreamer' application interface. The top navigation bar is blue with 'DataStreamer' and 'Connect' tabs. The left sidebar is dark grey and contains the following elements: a user profile 'test1', links for 'Account settings' and 'Log out', a 'Connection' section header, a red status message 'No provider', and a '+ Connect to provider' button. The main content area is white and features a 'Connect to provider' form. This form includes two input fields: 'Provider name' (with a person icon) and 'Client connect password' (with a lock icon), and a dark grey 'Connect' button at the bottom.

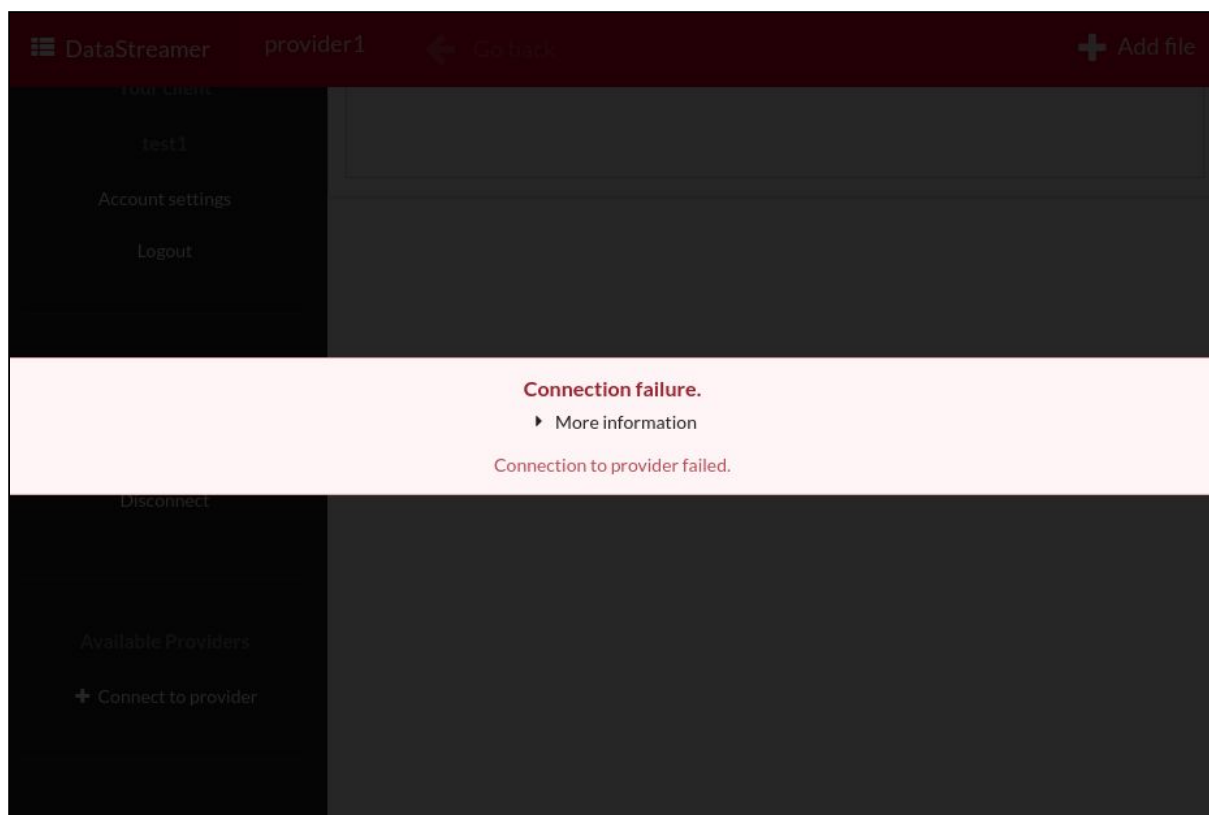
Фиг. 4.10: Форма за свързване към провайдер

В страничното меню се показва потребителското име на текущия клиент, връзка към страница с настройки на профила (“Account Settings”) и връзка за изход от профила (“Log out”). Показва се информация, че клиентът не е свързан към провайдер и има хипервръзка за отваряне на страницата с форма за свързване (“Connect to provider”).

Във формата за свързване потребителят трябва да въведе име на провайдер и парола за свързване, след което да натисне бутона “Connect”, за да се свърже към избран от него провайдер.

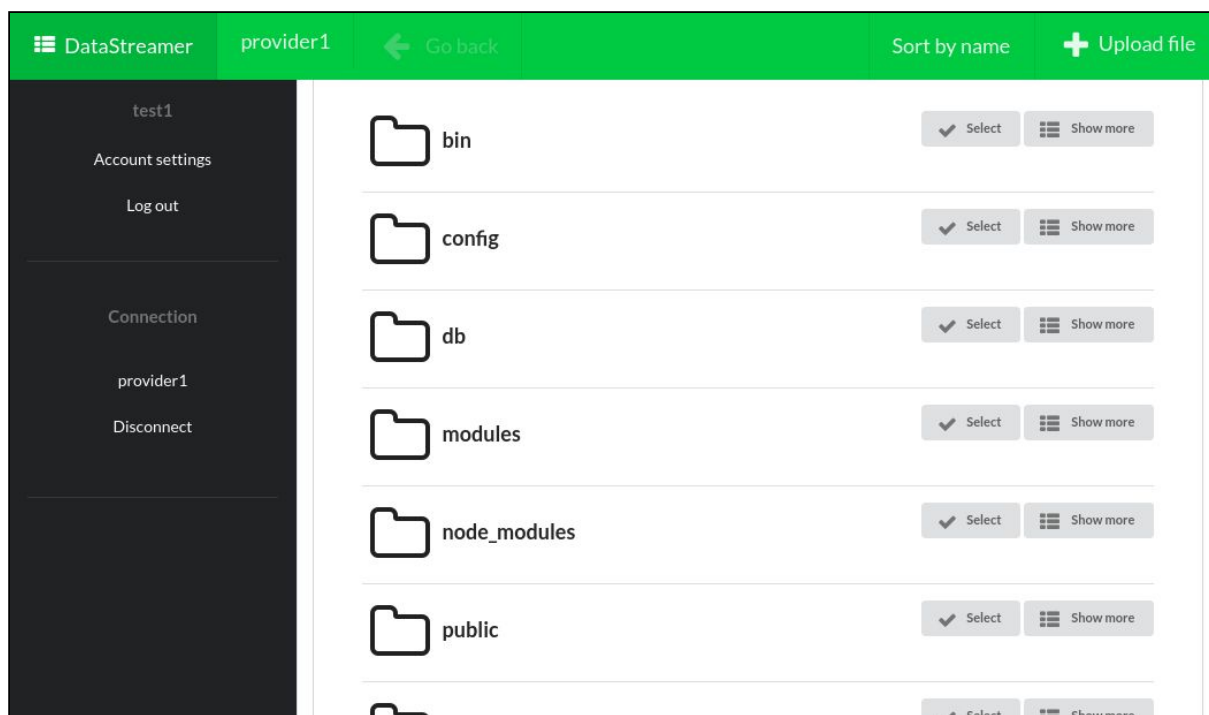
4.2.2.4. Осъществяване на връзка към провайдер

При правилно въведени данни се отваря нова страница, в която се прави опит за свързване. Ако провайдерът не е включен се изписва съобщение за грешка и връзката не може да се осъществи (фиг. 4.11.).



Фиг. 4.11: Изглед при опит за връзка към изключен провайдер

Ако е на линия се осъществява сесия с права за четене или сесия с права за писане и четене в зависимост от зададените в провайдъра настройки. Потребителят вижда съдържанието на избраната за начална точка на достъп директория (фиг. 4.12.).



Фиг. 4.12: Визуализация на файлове

4.2.2.5. Настройки на профил

Страничното меню съдържа връзка към страница с настройки на акаунта - "Account settings". При отваряне на тази страница се визуализира поле за промяна на паролата (фиг. 4.13.). Потребителят трябва да въведе настоящата си парола и два пъти новата парола, след което да натисне бутона "Submit".

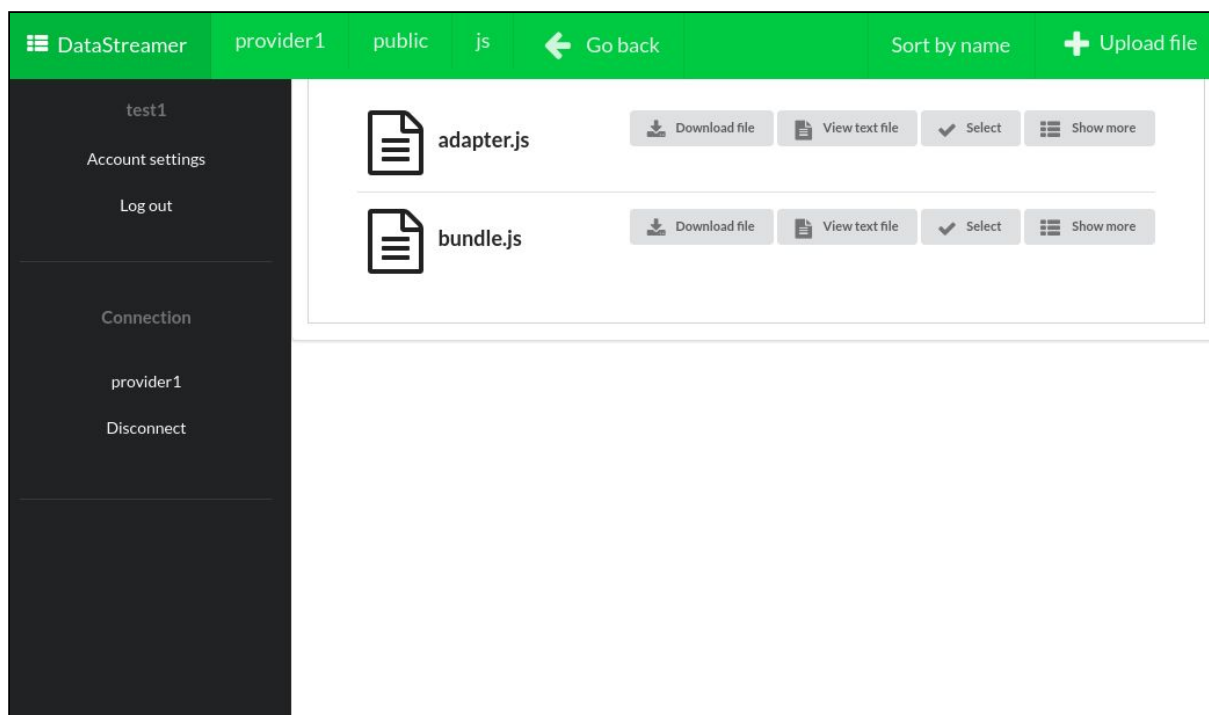
The screenshot shows a web application interface. At the top, there is a blue header bar with the text 'DataStreamer' and 'Settings'. Below the header, on the left, is a dark sidebar with the following items: 'test1', 'Account settings', 'Log out', 'Connection', 'provider1', and 'Disconnect'. The main content area is white and contains a 'Change password' form. The form has three input fields: 'Old password', 'New password', and 'Confirm new password', each with a lock icon on the left. Below these fields is a black 'Submit' button. To the right of the form, there is a red button labeled 'Delete account'.

Фиг. 4.13: Форма за промяна на паролата на клиент

В страницата с настройки на акаунта е налична опцията за изтриване на профил “Delete account”. При натискане на този бутон се отваря форма с едно поле. За да изтрие своя акаунт, потребителят трябва да въведе своята парола и да натисне бутона “Delete”.

4.2.2.6. Навигиране из директории

Потребителят вижда имената на файловете в отворената директория, както и икони според типа на всеки файл. Имената на папките се визуализират като връзки, при натискането на които съдържанието на папка може да бъде отворено (фиг. 4.14.). Имената на отворените поддиректории се наслагват в горната лента. При натискане на името на някоя поддиректория се навигира обратно до нея. При натискане на бутона “Go back”, намиращ се в горната лента, се отваря родителската директория на текущо отворената.



Фиг. 4.14: Преглед на файлове в поддиректория

4.2.2.7. Преглед на информация за файл

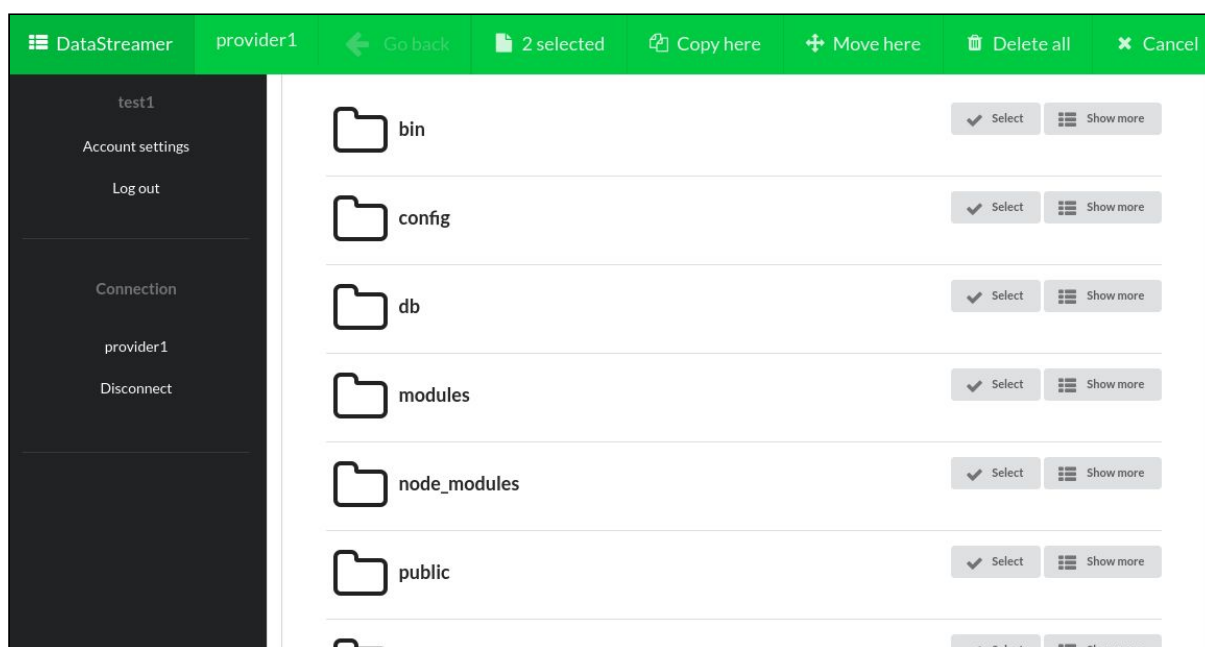
Срещу всеки файл има бутон “Show more”, при натискането на който се визуализира допълнителна информация за файла - тип, размер, път и права за достъп (фиг. 4.15.).



Фиг. 4.15: Преглед на допълнителна информация за файл

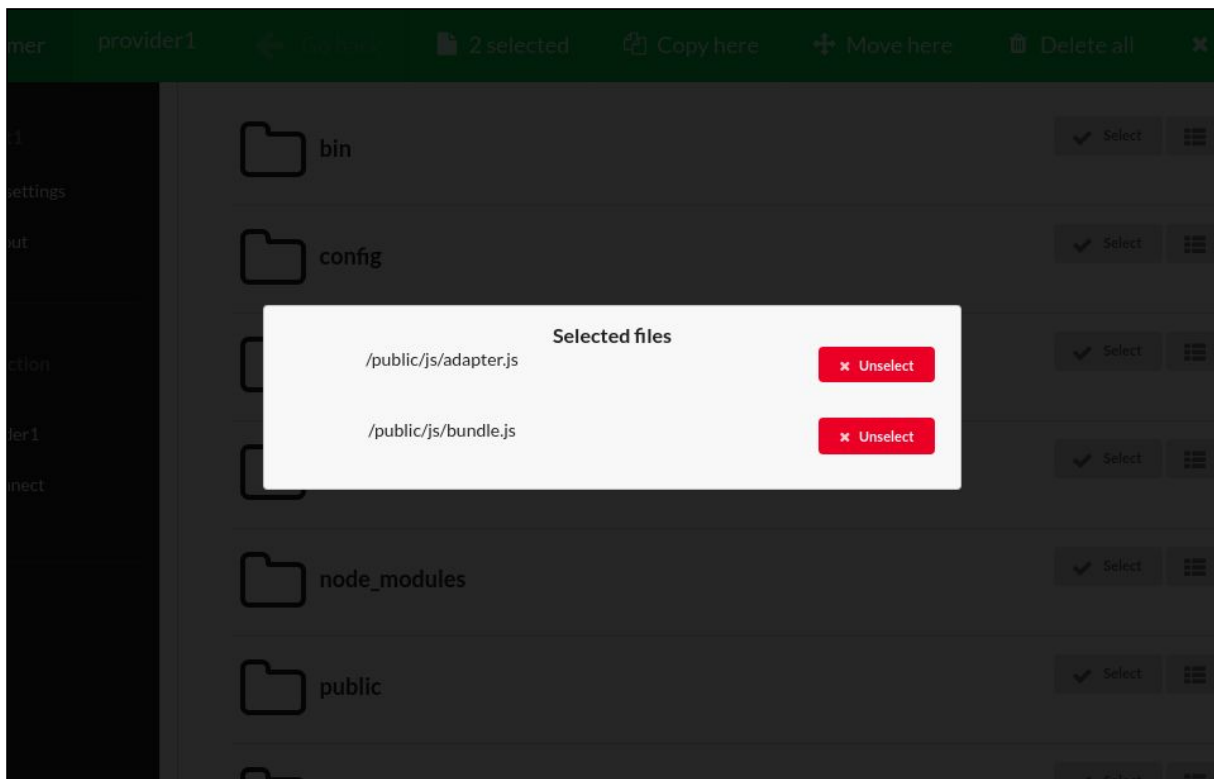
4.2.2.8. Селектиране на файлове

Всеки файл може да бъде селектиран чрез натискане на бутона “Select”. Тази опция позволява групово изтегляне на файлове, а при наличие на права за писане - изтриване, копиране или преместване на файлове. Възможните опции се визуализират в горната лента (фиг. 4.16.).



Фиг. 4.16: Селектиране на файлове

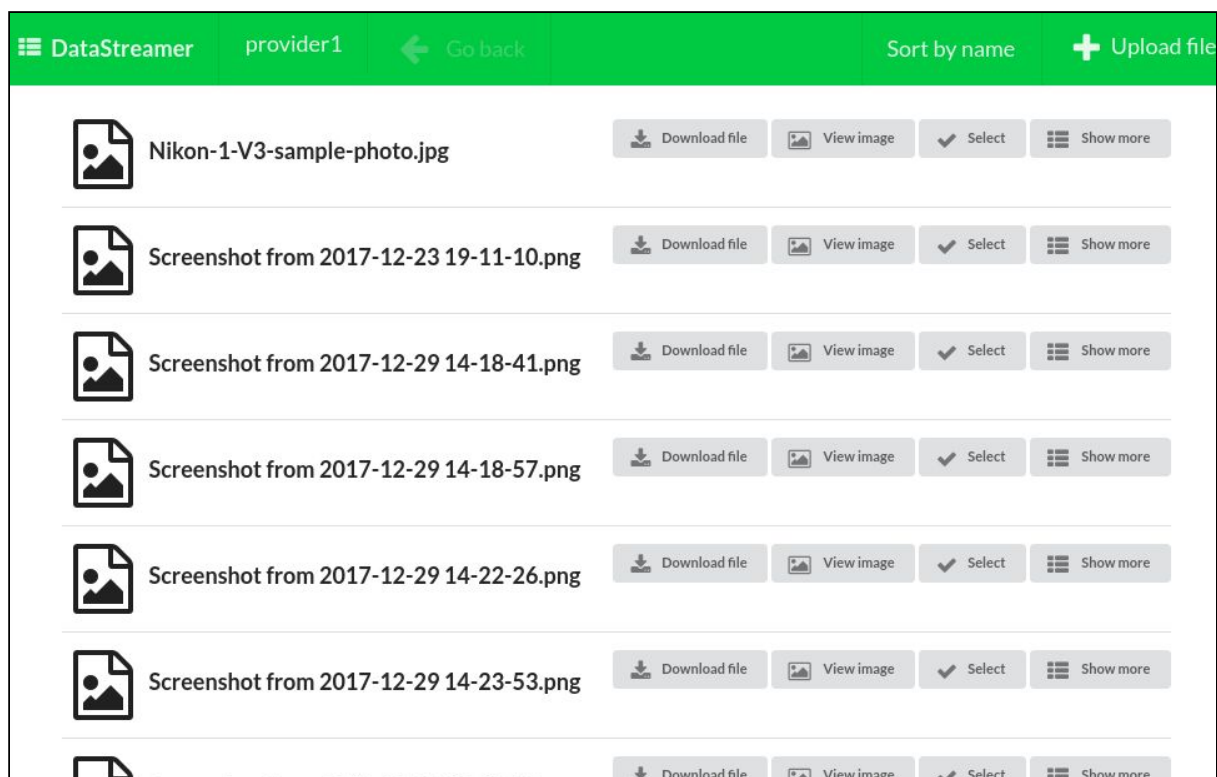
Показва се броят на селектираните файлове и при натискане на този бутон се визуализират пътищата с имената на тези файлове (фиг. 4.17.). При избор на “Unselect” даден файл може да бъде премахнат от избраните, а при клик върху затъмнената област от страницата показаният прозорец може да се премахне. При натискане на бутон “Cancel” в горната лента се премахват селекциите.



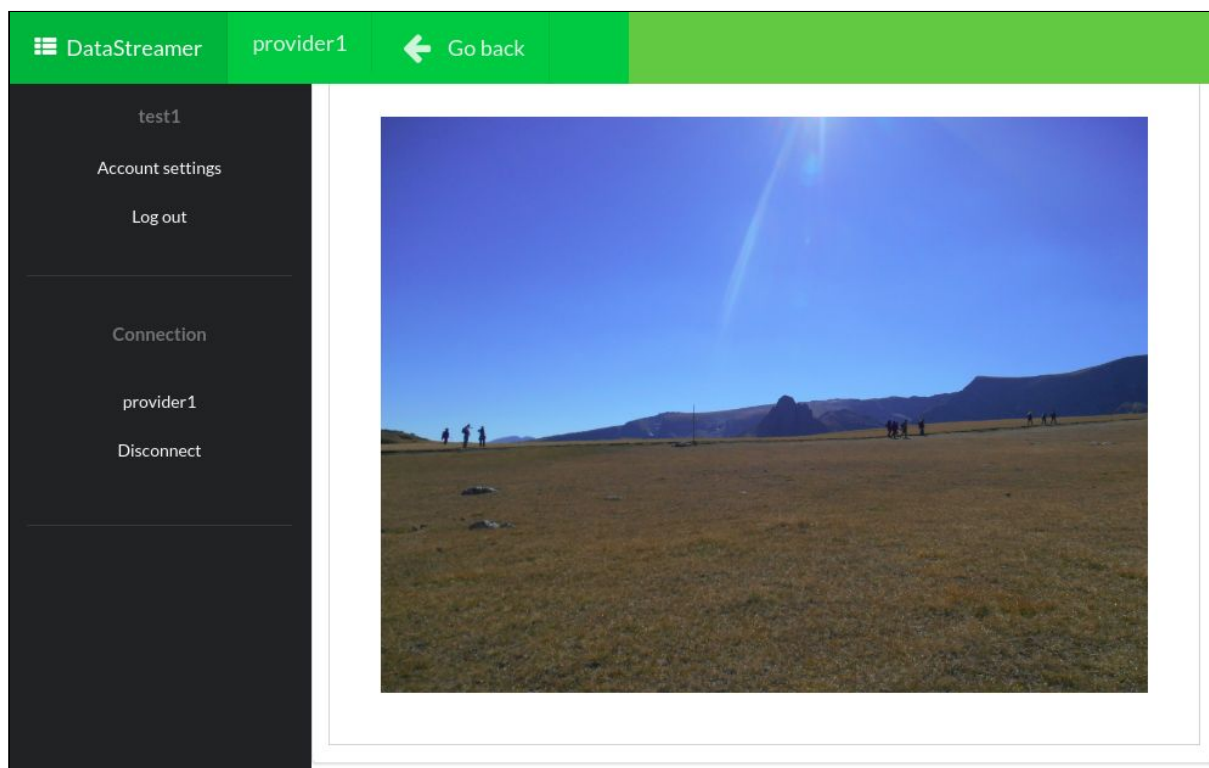
Фиг. 4.17: Преглед на селектирани файлове

4.2.2.9. Преглед на изображения

Ако файл е от тип JPEG или PNG, за него съществува опцията “View image” (фиг. 4.18.). При натискането му се отваря файла като изображение (фиг. 4.19.). Отворено изображение се затваря чрез бутона “Go back” в горното меню.



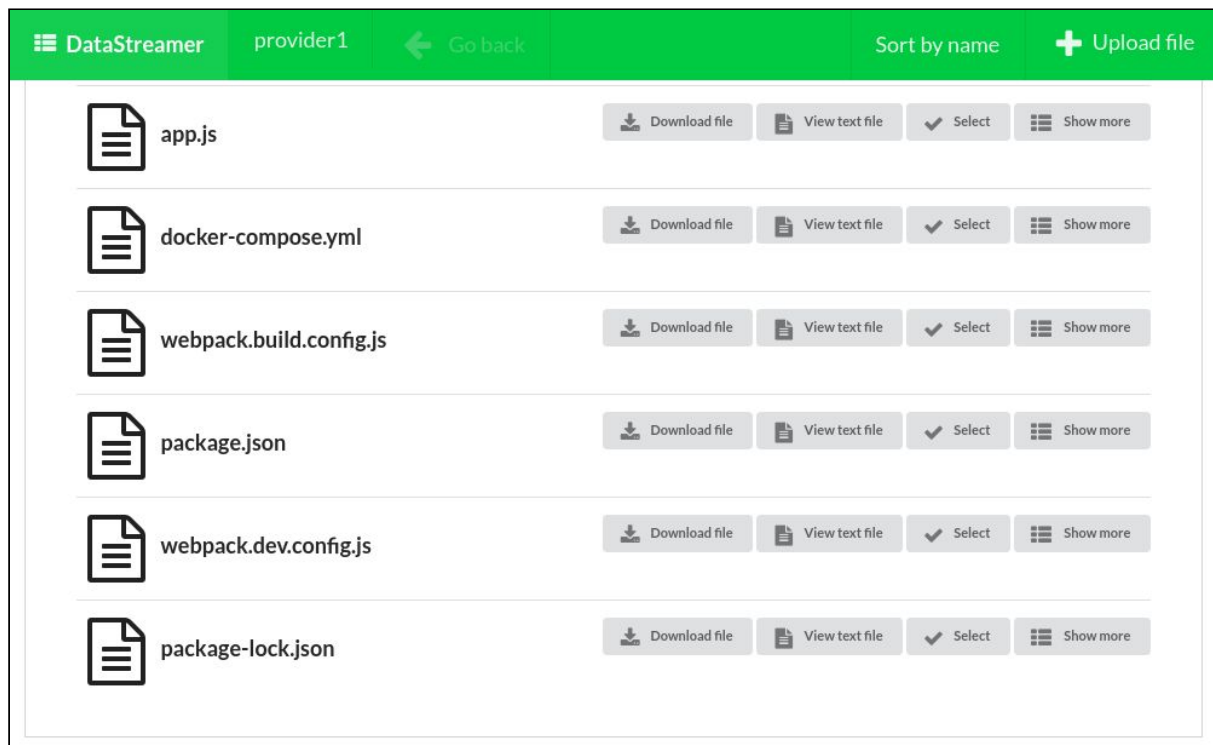
Фиг. 4.18: Опции за работа с изображения



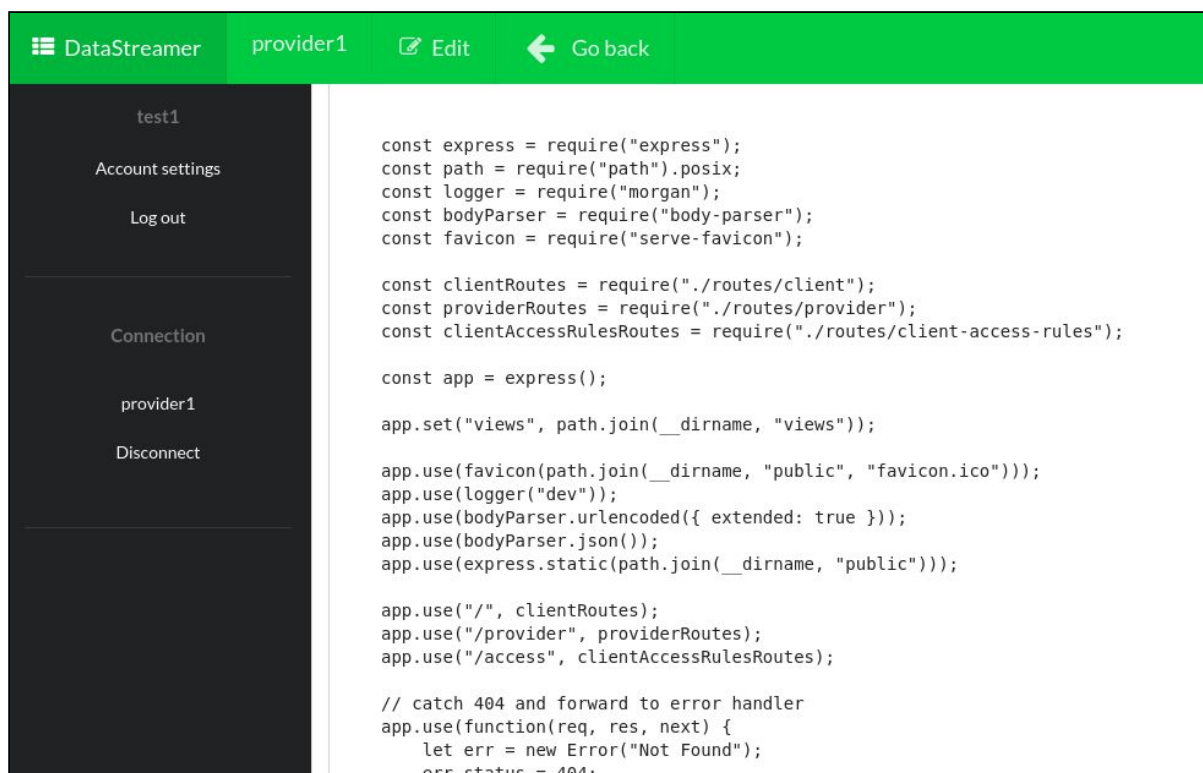
Фиг. 4.19: Визуализация на изображение

4.2.2.10. Преглед на текстови файлове

Ако файл е от текстов тип, за него съществува опцията “View text file” (фиг. 4.20.). При натискането му се визуализира текстовото съдържание на файла (фиг. 4.21.). При натискане на бутона “Go back” в горната лента файлът се затваря.



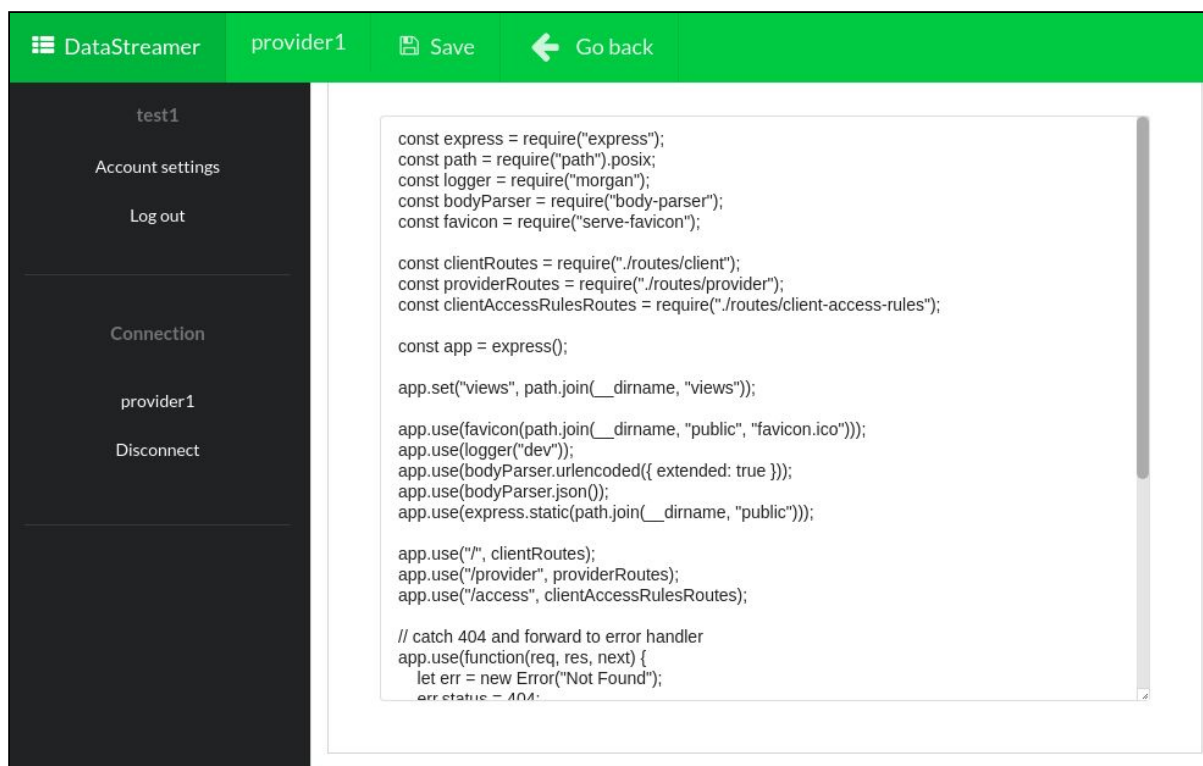
Фиг. 4.20: Опции за работа с текстови файлове



Фиг. 4.21: Визуализация на текстов файл

4.2.2.11. Редактиране на текстови файлове

При наличие на права за писане и при отворен текстов файл в горното меню се показва опцията “Edit” (фиг. 4.21.). При натискането ѝ се предоставя възможност за редакция на текста (фиг. 4.22.). Бутонът “Save” в горната лента запазва промените по файла, а “Go back” без да запазва промените премахва полето за редакция.



Фиг. 4.22: Редакция на текстов файл

4.2.2.12. Добавяне на файлове

При наличие на права за писане в горното меню се показва опцията "Add file". При избирането ѝ се предоставя възможност на потребителя да качи един или повече файлове в текущо отворената директория в провайдъра.

ЗАКЛЮЧЕНИЕ

В настоящата дипломна работа бяха разгледани протоколи за осъществяване на P2P връзка и управление на файлове. Изградена бе система от десктоп приложение, клиентско приложение, работещо посредством уеб браузър, и сървър.

Сървърът позволява десктоп приложението и клиентът да се автентичират, да се откриват взаимно и да обменят параметри, чрез което да се осъществи директна връзка между тях. Постигнатата цел е възможност за отдалечено управление на файлове чрез P2P връзка.

Клиентското приложение притежава потребителски интерфейс, чрез който могат да се визуализират данни за файловете и да се задават команди за управление: отваряне на поддиректории, качване, изтегляне, преместване, копиране и триене на файлове, визуализация на изображение и показване и редактиране на съдържанието на текстов файл.

Десктоп приложението (провайдър) предоставя информация за съдържанието на избрана директория и нейните поддиректории и изпълнява командите, получени от клиента. Налична е възможността за свързване на много клиенти към един провайдър. Имплементирана е система за управление на режима на достъп до файловете.

Съществуват много начини, по които разработката може да се подобри.

P2P връзките, осъществени чрез WebRTC, са сигурни по подразбиране^[30], но са необходими достатъчни мерки за защита на данните, предавани по време на процеса на сигнализиране, който е най-уязвим. Установяването на P2P връзка може да се направи по-сигурно чрез използване на SaltyRTC протокол за сигнализиране с криптиране от край до край^[31].

В клиентското приложение могат да се прибавят допълнителни функционалности за интеракция с файлове - търсене, отваряне на видео или аудио файл, преглеждане на съдържанието на pdf файлове, документи, таблици, презентации и други.

При работа с файлове съществува огромен брой случаи на употреба, които в бъдеще могат да се имплементират.

Потребителският интерфейс на клиентското приложение може да се пригоди за устройства със всякаква резолюция и работата с него да се направи по-удобна за потребителите. Необходимо е инсталацията на провайдер да се улесни.

Възможно е да се направи мобилно приложение, което да може да бъде в ролята както на провайдер, така и на клиент. То би позволило и изтегляне на много големи файлове от провайдер, тъй като това не е възможно чрез браузър, защото не се предоставя достъп до дисковото пространство и целият файл трябва първо да се запази в отворения браузър, в RAM паметта, преди да се предложи на потребителя опцията да го запази в дисковото пространство.

Поради същата причина може да се добави възможността десктоп приложението да работи като клиент. Също би могло да се направи провайдер с конзолен интерфейс.

ИЗПОЛЗВАНА ЛИТЕРАТУРА

- [1] VPN Technical Reference,
[https://technet.microsoft.com/en-us/library/cc780737\(v=ws.10\).aspx](https://technet.microsoft.com/en-us/library/cc780737(v=ws.10).aspx), март 2003г.
- [2] Уеб сайт на TeamViewer, <https://www.teamviewer.com>, февруари 2018г.
- [3] **David Harrison**, *Index of BitTorrent Enhancement Proposals*,
http://www.bittorrent.org/beps/bep_0000.html, юни 2017г.
- [4] **Sam Dutton**, *Getting Started with WebRTC*,
<https://www.html5rocks.com/en/tutorials/webrtc/basics/>, февруари 2014г.
- [5] **Sam Dutton**, *WebRTC in the real world: STUN, TURN and signaling*,
<https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/>, ноември 2013г.
- [6] **Jonathan Rosenberg**, *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*,
<https://tools.ietf.org/html/rfc5245>, април 2010г.
- [7] *coturn README.md*, <https://github.com/coturn/coturn>, февруари 2018г.
- [8] *Microsoft SMB Protocol and CIFS Protocol Overview*,
[https://msdn.microsoft.com/en-us/library/windows/desktop/aa365233\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365233(v=vs.85).aspx),
февруари 2018г.
- [9] Уеб сайт на Samba, <https://www.samba.org/>, февруари 2018г.
- [10] **Tatu Ylonen**, *SSH (SECURE SHELL)*, <https://www.ssh.com/ssh/>, август 2017г.
- [11] SFTP – SSH SECURE FILE TRANSFER PROTOCOL,
<https://www.ssh.com/ssh/sftp/>, октомври 2017г.
- [12] *Electron Documentation*, <https://electronjs.org/docs>, февруари 2018г.
- [13] Уеб сайт на Qt, <https://www.qt.io/>, февруари 2018г.
- [14] **Monica Pawlan**, *JavaFX Overview*,
<https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>, април 2013г.
- [15] *About Node.js®*, <https://nodejs.org/en/about/>, февруари 2018г.
- [16] Уеб сайт на ExpressJS, <https://expressjs.com/>, февруари 2018г.
- [17] *Angular - Architecture Overview*, <https://angular.io/guide/architecture>,
февруари 2018г.
- [18] Уеб сайт на ReactJS, <https://reactjs.org/>, февруари 2018г.

- [19] *Vue.js introduction*, <https://vuejs.org/v2/guide/>, февруари 2018г.
- [20] **David Heinemeier Hansson**, *The Rails Doctrine*, <http://rubyonrails.org/doctrine/>, януари 2016г.
- [21] *Spring Framework Overview*, <https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/overview.html>, февруари 2018г.
- [22] **Ado Kukic**, *Cookies vs Tokens: The Definitive Guide*, <https://auth0.com/blog/cookies-vs-tokens-definitive-guide/>, май 2016г.
- [23] *PostgreSQL FAQ*, <https://www.postgresql.org/about/press/faq/>, февруари 2018г.
- [24] *Уеб сайт на Redis*, <https://redis.io/>, февруари 2018г.
- [25] *socket.io-redis README.md*, <https://github.com/socketio/socket.io-redis>, февруари 2018г.
- [26] **Nick Gauthier**, *Hashed Passwords with PostgreSQL's pgcrypto*, <https://www.meetspaceapp.com/2016/04/12/passwords-postgresql-pgcrypto.html>, април 2016г.
- [27] *PostgreSQL 9.6 Documentation*, <https://www.postgresql.org/docs/9.6/static/index.html>, февруари 2018г.
- [28] *Socket.IO - Docs*, <https://socket.io/docs/>, февруари 2018г.
- [29] *chokidar README.md*, <https://github.com/paulmillr/chokidar>, февруари 2018г.
- [30] *A Study of WebRTC Security*, <https://webrtc-security.github.io/>, февруари 2018г.
- [31] *Уеб сайт на SaltyRTC*, <https://saltyrtc.org/>, февруари 2018г.
- <https://jwt.io/>
- Node.js v8.9.4 Documentation*, <https://nodejs.org/dist/latest-v8.x/docs/api/>, февруари 2018г.
- MDN web docs*, <https://developer.mozilla.org/en-US/docs/Web/JavaScript>, февруари 2018г.

СЪДЪРЖАНИЕ

УВОД	4
ПЪРВА ГЛАВА	5
1.1. Обзор на начините за осъществяване на отдалечен достъп до файлове	5
1.1.1. VPN	5
1.1.2. Облачна услуга	5
1.1.3. Система със сървър-посредник между файлове и потребител	6
1.1.4. Отдалечено управление на файлове чрез P2P връзка	7
1.1.4.1. TeamViewer	8
1.2. Обзор на начините за осъществяване на директна връзка	9
1.2.1. BitTorrent протокол	9
1.2.2. WebRTC	9
1.2.3. SMB	12
1.2.3.1. Samba	12
1.2.4. SSH	12
1.2.5. SFTP	13
1.3. Обзор на desktop application development frameworks	14
1.3.1. Electron	14
1.3.2. Qt	14
1.3.3. JavaFX	14
1.4. Обзор на технологии за уеб разработка	15
1.4.1. JavaScript-базирани технологии за уеб разработка	15
1.4.1.1. Express.js	16
1.4.1.2. Обзор на JavaScript технологии за изграждане на SPA	16
1.4.1.2.1. Angular	16
1.4.1.2.2. React	17
1.4.1.2.3. Vue.js	18
1.4.2. Ruby on Rails	18
1.4.3. Spring	19
1.5. Обзор на начини за автентикация	19
1.5.1. Бисквитки	20
1.5.2. JSON Web Token	21
ВТОРА ГЛАВА	22
2.1. Функционални изисквания	22
2.2. Избор на технологии	23
2.2.1. WebRTC	23
2.2.2. JavaScript	23
2.2.3. Node.js	23
2.2.4. PostgreSQL	24
2.2.5. Redis	24
2.2.6. Electron	24

2.2.7. React	25
2.2.8. JSON Web Token	25
2.3. Описание на алгоритъма	25
2.3.1. Сървър	25
2.3.2. Провайдер	27
2.3.2.1. Поддръжка на много клиенти, свързани към един провайдер	27
2.3.2.2. Функционалности в главния рендериращ процес	28
2.3.2.3. Функционалности в клиентски процеси	28
2.3.3. Клиент	28
2.3.4. Съобщения от клиент към провайдер	29
2.3.4.1. "openDirectory"	29
2.3.4.2. "downloadFile"	29
2.3.4.3. "uploadFile"	29
2.3.4.4. "copyFile"	30
2.3.4.5. "moveFile"	30
2.3.4.6. "deleteFile"	30
2.3.5. Съобщения от провайдер към клиент	30
2.3.5.1. "add", "addDir", "change", "unlink" и "unlinkDir"	30
2.3.5.2. "newScan"	31
2.3.6. Контрол на правата	31
2.4. Структура на базата данни	32
2.4.1. Структура на базата данни за дълготрайна информация	32
2.4.2. Описание на базата данни за съхранение на сесийна информация	33
ТРЕТА ГЛАВА	35
3.1. Структура на системата	35
3.1.1. Структура на сървърното приложение	35
3.1.2. Структура на клиентското приложение	36
3.1.3. Структура на провайдер	36
3.2. Реализация на автентикация	37
3.2.1. Автентикация на клиент	38
3.2.1.1. Регистрация на клиент	38
3.2.1.2. Вход на клиент	42
3.2.2. Автентикация на провайдер	42
3.2.2.1. Регистрация на провайдер	42
3.2.2.2. Вход на провайдер	44
3.2.3. Свързване на клиент към провайдер	45
3.3. Осъществяване на P2P връзка	47
3.3.1. Свързване на клиент към сървър чрез уеб сокети	47
3.3.2. Свързване на провайдер към сървър чрез уеб сокети	51
3.3.3. Процесът "сигнализиране"	57
3.4. Многопроцесова архитектура на провайдер	79

3.5. Сканиране на файлове от провайдер	81
3.6. Представяне на съдържанието на директория в клиента	82
3.7. Отваряне на поддиректория	83
3.8. Изтегляне на файл	84
3.8.1. Визуализация на изображения	87
3.8.2. Визуализация на текст	88
3.9. Качване на файл	88
3.10. Копиране на файл	91
3.11. Преместване на файл	92
3.12. Изтриване на файл	93
3.13. Управление на права за достъп	94
3.14. Настройки на акаунт	95
ЧЕТВЪРТА ГЛАВА	96
4.1. Изисквания към компютърна конфигурация и инсталация	96
4.1.1. Конфигуриране и стартиране на сървър	96
4.1.2. Стартиране на клиентско приложение	97
4.1.3. Конфигуриране и стартиране на провайдер	97
4.2. Ръководство на потребителя	98
4.2.1. Използване на провайдер за предоставяне на файловете	98
4.2.1.1. Вход	98
4.2.1.2. Регистрация	99
4.2.1.3. Контролен панел на провайдер	100
4.2.1.3.1. Конфигурация на провайдер	101
4.2.1.3.2. Настройки на профил	102
4.2.1.3.3. Преглед и контрол на свързани клиенти	103
4.2.1.3.4. Преглед и контрол на индивидуално зададени клиентски права за достъп	104
4.2.2. Използване на клиент за достъпване на файлове	104
4.2.2.1. Вход	105
4.2.2.2. Регистрация	106
4.2.2.3. Форма за свързване към провайдер	107
4.2.2.4. Осъществяване на връзка към провайдер	108
4.2.2.5. Настройки на профил	110
4.2.2.6. Навигиране из директории	111
4.2.2.7. Преглед на информация за файл	112
4.2.2.8. Селектиране на файлове	113
4.2.2.9. Преглед на изображения	114
4.2.2.10. Преглед на текстови файлове	116
4.2.2.11. Редактиране на текстови файлове	117
4.2.2.12. Добавяне на файлове	118
ЗАКЛЮЧЕНИЕ	119
	124

