МИНИСТЕРСТВО ЭНЕРГЕТИКИ, ПРОМЫШЛЕННОСТИ И СВЯЗИ СТАВРОПОЛЬСКОГО КРАЯ

**Государственное бюджетное**

**профессиональное образовательное учреждение**

**«Ставропольский колледж связи**

**имени Героя Советского Союза В.А. Петрова»**

Цикловая комиссия вычислительной техники

УТВЕРЖДАЮ

Заместитель директора по учебной работе

\_\_\_\_\_\_\_\_\_\_\_\_ И.О. Фамилия

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_201\_ г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

Шифр специальности Название специальности

Тема тема тема тема тема тема тема тема тема тема тема

|  |  |
| --- | --- |
|  | Выполнил: Бесленеев Станислав Борисович, группа П141 |
|  | Руководитель: Фамилия И.О. |
|  |  |
|  |  |
| Зав. отделением    Председатель ЦК | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* И.О. Фамилия  *(подпись)* И.О. Фамилия |
|  |  |
|  |  |
|  |  |

Ставрополь, 2017

**Содержание**

Введение 4

1 Теоретическое исследование темы 6

1.1 Актуальность проблемы исчерпания IP адресов 6

1.2 Технология NAT 6

1.2.1 Типы NAT 7

1.3 Разбор протокола STUN 10

1.3.1 Назначение протокола 10

1.3.2 Спецификация RF5389 11

1.3.2.1 Структура протокола 13

1.3.3 Принцип действия STUN сервера 14

1.4 Описание метода UDP Hole Punching 14

2 Реализация программной системы 18

2.1 Выбор средств реализации программного кода 18

2.2 Написание серверной части 20

2.2.1 Написание консольной версии сигнального сервера 20

2.2.2 Написание сигнального сервера с веб интерфейсом 27

2.2.2.1 Серверная часть 27

2.2.2.2 Веб интерфейс 29

2.3 Написание клиентской части 43

2.3.1 Модуль для работы с сигнальным сервером 43

2.3.2 Модуль для работы с UDP Hole Punching 46

Заключение 52

Список литературы 53

Приложение 1 54

**Введение**

С момента появления человека и до наших времен, люди всегда пытались установить связь между собой. Во времена первобытных людей, единственным видом связи был голос. Во времена древнего Египта, Рима, Греции связь осуществлялась с помощью гонцов. Спустя какое-то время люди поняли, что самый эффективный способ передачи информации это - свет. А спустя еще время – электричество.

Открытие электричества, людям позволило передавать информацию на большие расстояния и с большой скоростью. В 1753 году, стали появляться первые телеграфные машинки. Всего за полтора столетия, начиная с момента изобретения телеграфа и до наших дней, человечество освоило такие телекоммуникационные средства, которые позволили ему быть не только информированным, но и мобильным.

На сегодняшний момент эволюция связи человечества остановилась на Интернете. Интернет - всемирная система объединённых компьютерных сетей для хранения и передачи информации. Как и в любой системе передачи информации должен быть адресант (тот, кто посылает сообщение) и адресат (тот, кто получает сообщение). В сети Интернет адресация осуществляется с помощью протокола IP в настоящие время 4-я версия (IPv4) и менее распространённого IPv6.

IP адрес это уникальный идентификатор устройства, который подключен к сети Интернет. Распределением адресов IPv4 занимается американская некоммерческая организация IANA, а также пять региональных интернет-регистраторов, которые ответственны за назначение IP-адресов конечным пользователям на определённых территориях, и локальными интернет-регистраторами, такими как интернет-провайдеры. IPv4 позволяет использовать около 4,22 миллиарда адресов.

Проблема, с которой люди столкнулись 15 апреля 2011 года – это исчерпания адресов IPv4. Одна из пяти региональных интернет-регистраторов ARIN объявила о полном исчерпании IPv4 адресов. С каждым годом проблема начинает касается и других интернет-регистраторов.

Из-за нехватки IP адресов, интернет провайдеры стали брать дополнительную плату за выделенный IP. А для тех, кто не хочет платить выделяется 1 IP адрес на несколько пользователей. Эта технология называется NAT (Network address translation). Он является основным механизмом, уменьшающий скорость исчерпания адресов.

Из-за исчерпания IP адресов появляется трудность инициирования канала связи с тем, кто находится за NAT. Так как за одним выделенным IP адресом могут быть несколько пользователей. Для решения этой проблемы было создана программная система, которая сможет помочь установить канал связи между пользователями, которые находятся за NAT.

**1 Теоретическая часть**

**1.1 Актуальность проблемы**

Как говорилось ранее, проблема исчерпания IP адресов актуальна для тех кому нужен прямой доступ из Интернета. Такая возможность бывает не у всех, так как у интернет провайдеров ограниченное количество внешних IP адресов.

Например, небольшим провайдером выделено от 127 до 500 адресов. Естественно если каждому давать по адресу на всех не хватит. А за отдельную плату могут выделить отдельный «белый» IP адрес. Для решения этой проблемы провайдеры используют NAT.

Интернет поставщики подключают несколько своих клиентов к одному внешнему адресу. Из-за этого соединиться напрямую к компьютеру, который находится за NAT не получится.

**1.2 Технология NAT**

NAT (Network Address Translation) – механизм был придуман в 1994 году. Чаще всего он нужен для подключения локальной сети к Интернету. Дело в том, что теоретически существует 255\*255\*255\*255=4228250625(4 миллиарда адресов). Даже если бы у каждого жителя планеты был всего один компьютер, адресов бы уже не хватало. А на конец 2015 года в мире было 8,1 млрд. подключенных к Интернету устройств. В два раза больше чем возможное количество адресов. Специалистами было принято решение разделить пространство адресов на публичные (белые) и приватные (частные, серые). К последним относятся три диапазона:

* 10.0.0.0/8
* 172.16.0.0/12
* 192.168.0.0/16

Эти адреса могут использоваться в любой локальной сети или в любой частной сети. Следовательно, они могут совпадать. И для решения конфликта IP адресов приходит на помощь NAT. Простыми словами, он заменяет внутренний IP адрес на внешний.

**1.2.1 Типы NAT**

Существует 3 вида NAT

* Статический
* Динамический
* Many-To-One

Статический NAT – самый простой вид NAT. Он преобразует один внутренний адрес в один внешний. При этом все запросы из вне транслируются во внутреннею сеть. Как будто он напрямую подключен к Интернету.

Динамический NAT работает с пулом внешних адресов и динамически преобразует внутренний адрес в свободный из пула внешний адрес.

Many-To-One – у этого вида есть много названий. Например, PAT (Port Address Translation), NAT Overload. Является самым распространённым видом NAT. Его можно встретить на любом домашнем маршрутизаторе. Из названия понятно, что он транслирует несколько внутренних IP в один внешний. Именно этот вид позволяет решить проблему нехватки адресов и пускать в интернет всех желающих.

Рассмотрим работу NAT Overload на примере. Например, есть 2 хоста в одной локальной сети, имеющие следующие IP адреса

* 192.168.10.101
* 192.168.10.102

Оба хоста запрашивают страницу с сервера 74.125.232.227 (google.com). Подробно видно в таблице 1.

Таблица 1

Запрос к серверу

|  |  |  |  |
| --- | --- | --- | --- |
| Адрес отправителя | Порт отправителя | Адрес назначения | Порт назначения |
| 192.168.10.101 | 5648 | 74.125.232.227 | 443 |
| 192.168.10.102 | 3912 | 74.125.232.227 | 443 |

Когда пакеты доходят до локального маршрутизатора, маршрутизатор проводит декапсуляцию пакета. Он заменяет адрес отправителя на внешний адрес, а порт на свободный любой порт. Далее маршрутизатор создает таблицу маршрутизации, где показано соответствие внутреннего IP:PORT с внешним. В таблице 2 показан пример таблицы маршрутизации.

Таблица 2

Таблица маршрутизации

|  |  |  |  |
| --- | --- | --- | --- |
| Внутренний адрес | Внутренний порт | Внешний адрес | Внешний порт |
| 192.168.10.101 | 5648 | 91.225.32.19 | 11235 |
| 192.168.10.102 | 3912 | 91.225.32.19 | 43852 |

После создания таблицы пакет отправляется адресату. На веб-сервер пришло 2 разных запроса с одного IP адреса, но с разных портов. Сервер после получения запроса, отправляет ответ на внешний адрес и порт хоста как показано в таблице 3

Таблица 3

Ответ от сервера

|  |  |  |  |
| --- | --- | --- | --- |
| Адрес отправителя | Порт отправителя | Адрес назначения | Порт назначения |
| 74.125.232.227 | 443 | 91.225.32.19 | 11235 |
| 74.125.232.227 | 443 | 91.225.32.19 | 43852 |

Как только сообщение доходит до локального маршрутизатора он в соответствии созданной таблице маршрутизации (см. таблица 2) сверяет какому хосту послать дальше пакет.

Теперь немного о видах NAT Overload. Виды классифицируются по методу трансляции пары внутреннего и внешнего IP:PORT. Вот список существующий видов:

* Симметричный NAT (Symmetric NAT)
* Cone NAT, Full Cone NAT
* Address-Restricted cone NAT
* Restricted cone NAT

Симметричный NAT – осуществляет трансляцию, при котором каждое соединение транслируется в случайный уникальную случайную пару IP:PORT. Инициация подключения из публичной сети невозможна.

Full Cone NAT - взаимная трансляция между парами «внутренний адрес: внутренний порт» и «публичный адрес: публичный порт». Любой внешний хост может инициировать соединение с внутренним хостом.

Address-Restricted cone NAT - Постоянная трансляция между парой «внутренний адрес: внутренний порт» и «публичный адрес: публичный порт». Любое соединение, инициированное с внутреннего адреса, позволяет в дальнейшем получать ему пакеты с любого порта того публичного хоста, к которому он отправлял пакет(ы) ранее.

Port-Restricted cone NAT — Трансляция между парой «внутренний адрес: внутренний порт» и «публичный адрес: публичный порт», при которой входящие пакеты проходят на внутренний хост только с одного порта публичного хоста — того, на который внутренний хост уже посылал пакет.

**1.3 Разбор протокола STUN**

**1.3.1 Назначение протокола**

Как говорилось ранее, NAT при инициировании подключения с локальной сети транслирует пару «внутренний адрес: внутренний порт» в пару «внешний адрес: внешний порт». Из-за этого проблематично узнать свой внешний адрес и порт, а это бывает нужно, когда необходимо принять подключение из публичной сети. На помощь приходит протокол STUN.

STUN расшифровывается как (Simple Traversal of User Datagram Protocol (UDP) through Network Address Translators) (упрощенное прохождение UDP пакетов через NAT). Его основной задачей является, помогать устройствам находящиеся за NAT узнать свой публичный адрес и порт. Его второстепенной задачей является обнаружение проброса портов, сделанным шлюзом. STUN не является решением обхода NAT. Скорее, это инструмент, помогающий это достичь.

**1.3.2 Спецификация RF5389**

Структура протокола STUN описана в спецификации RFC 5389. RFC (Request for Comments) —это документ из серии пронумерованных информационных документов Интернета, охватывающих технические спецификации и Стандарты, широко используемые во Всемирной сети. Этот документ описывает работу протокола как на сервере, так и на клиенте. Для продолжения описание протокола необходимо ознакомиться с некоторыми терминами.

STUN агент - это объект, который реализует STUN протокол. Объектом может быть либо STUN-клиент, либо STUN сервер.

STUN клиент - это объект, который отправляет запросы STUN серверу. Клиент STUN также может отправлять свои показания. В этой спецификации термины STUN-клиент и клиент синонимы.

Сервер STUN - это объект, который получает запросы от клиента и отправляет STUN ответы. Сервер STUN также может отправлять свои показания. В этом описании термины STUN-сервер и Сервер являются синонимами.

Адрес транспорта - комбинация IP-адреса и номера порта (Например, номер порта UDP или TCP).

Рефлексивный транспортный адрес – это транспортный адрес, полученный клиентом показывающий, как клиент виден в публичной сети.

Сопоставленный адрес – это тоже самое что рефлексивный адрес. Он сохранился только по историческим причинам и из-за наименования атрибутов MAPPED-ADDRESS и XOR-MAPPED-ADDRESS.

Long-Term Credential – это долгосрочный логин связанный с определенным паролем. Необходим для предоставление доступа клиенту к серверу.

Long-Term Password – это долгосрочный пароль

Short-Term Credential – это временный логин и пароль. Обычно выдается на время до 5 минут.

Short-Term Password – это временный пароль

STUN Indication (STUN Индикация) – Сообщения на которые не ожидается ответ.

RTO: Retransmission TimeOut, определяет начальный период времени между передачей запроса и первой повторной передачей этого запроса.

**1.3.2.1 Структура протокола**

Сообщения STUN кодируются в двоичном формате, используя формат, ориентированный на сеть. Все сообщения должны начинаться с 20-байтового заголовка, за которым следует атрибуты.

Заголовок STUN содержит тип сообщения STUN, Magic Cookie, идентификатор транзакции и длину сообщения. Подробней видно в таблице 4

Таблица 4

Структура заголовка сообщения STUN

|  |  |  |
| --- | --- | --- |
| 00 | Тип сообщения STUN | Длина сообщения |
| Magic Cookie | | |
| ID Транзакции (96 бит) | | |

Наиболее значимые первые 2 бита каждого сообщения STUN. Они должны быть нулями. Это можно использовать для дифференциации пакетов STUN из других протоколов, когда STUN мультиплексируется с другими протоколами на одном и том же порту.

Тип сообщения определяет класс сообщения (запрос, ответ успеха, ответ отказа или указание) и метод сообщения (основная функция). Хотя всего существует 4 типа класса сообщения.

В протоколе существует только два типа транзакций: транзакции запроса/ответа и операции индикации. Классы ответов разбиты на ошибки и успешные ответы.

Поле Magic Cookie должно содержать значение 0x2112A442. Magic Cookie позволяет различать пакеты STUN из пакетов других протоколов, когда STUN мультиплексируется с другими протоколами на одном и том же порту.

**1.3.3 Принцип действия STUN сервера**

STUN сервер – это сервер находящийся в публичной сети, который позволяет узнать свой публичный IP: PORT транслируемый локальным NAT.

Работу сервера осуществляется следующими шагами:

1. Сервер слушает порт 3478 на входящие сообщения
2. При получении сообщения, сервер в соответствии со спецификацией RF5389 расшифровывает входящее сообщение
3. Если сообщение было составлено корректно, сервер составляет ответ, в который он включает IP: PORT отправителя
4. Ответ посылается обратно.
5. В зависимости от вида NAT сообщение может быть получено или нет.

**1.4 Описание метода UDP Hole Punching**

User Datagram Protocol Hole Punching обычно используется в приложениях использующие NAT, для проброса UDP пакетов через NAT. Эта технология обычно нужна для клиент-клиент сетевых приложений.

UDP Hole Punching часто используется в торрентах, peer-2-peer соединениях, Direct Client-to-Client (DCC), VoIP (голос через интернет), Skype и много других приложения или протоколах где необходимо прямое соединение с клиентами.

UDP Hole Punching является методом установления двунаправленного UDP соединения между узлами сети в частных сетях, использующие трансляторы сетевых адресов. Метод не применим во всех случаях и со всеми типами NAT, так как эксплуатационные характеристики NAT не стандартизированы. Для инициации соединения требуется третья сторона- сервер, который виден обоим компьютерам. Обычно это STUN сервера

Для того что бы напрямую установить связь между 2-мя хостами необходимо сделать несколько шагов:

* Узнать внешний IP и порт удаленной машины. Для этого воспользуемся STUN – сетевым протоколом, который позволяет определить внешний IP-адрес.
* Передать эту информацию другому хосту
* Установить соединение и использовать его далее для обмена данными

Схематично представлено на рисунке 1

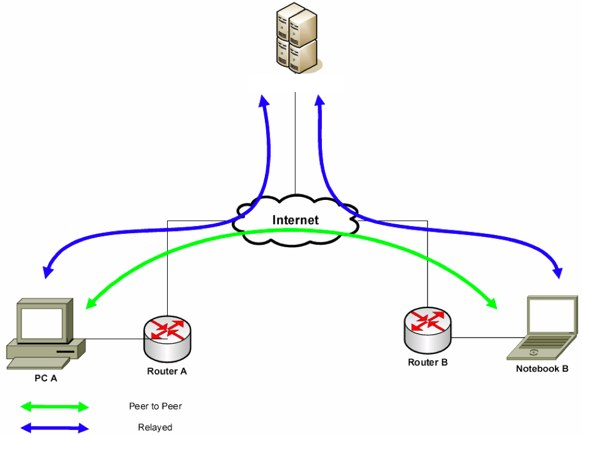


Рис. 1. Упрощённая схема UDP Hole Punching

Предположим A и B являются хостами, которые хотят установить двунаправленное соединение между собой и находящиеся в приватной сети. NA и NB являются NAT устройствами с глобальными адресами EIPA и EIPB. S является публичным сервером с адресом известным A и B.

* A и B начинают UDP соединение с сервером S. NAT устройства NA и NB транслируют UDP адрес и задает временный внешний порт EPA и EPB соответственно.
* Сервер S исследует UDP пакет и извлекает исходящий адрес A и B (внешний порт EPA и EPB)
* Сервер S посылает EIPA: EPA клиенту B и EIPB: EPB клиенту А. Теперь оба клиента знают внешние адреса для инициации подключения.
* Клиент А отправляет пакет на адрес EIPB: EPB
* NA исследует пакет создает соответствующую запись в таблице маршрутизации {Source-IP-A, EPA, EIPB, EPB}
* Клиент B отправляет пакет на адрес EIPA: EPA
* NB исследует пакет создает соответствующую запись в таблице маршрутизации {Source-IP-B, EPB, EIPA, EPA}
* Когда первый пакет B достигает NAT, в зависимости от состояния таблицы маршрутизации NA устройства, пакет может быть отброшен (если запись не была создана) или пропущен дальше в локальную сеть к клиенту А
* Когда первый пакет А достигает NAT, в зависимости от состояния таблицы маршрутизации NB устройства, пакет может быть отброшен (если запись не была создана) или пропущен дальше в локальную сеть к клиенту B
* Как только пакет А дошел до пакета B и как только пакет B дошел до А, можно сказать что связь установлена. Теперь А и B могут обмениваться сообщениями, несмотря на то, что они находятся за NAT устройствами.

Подробный процесс инициирования соединения между 2-ми хостами можно увидеть на рисунке 2

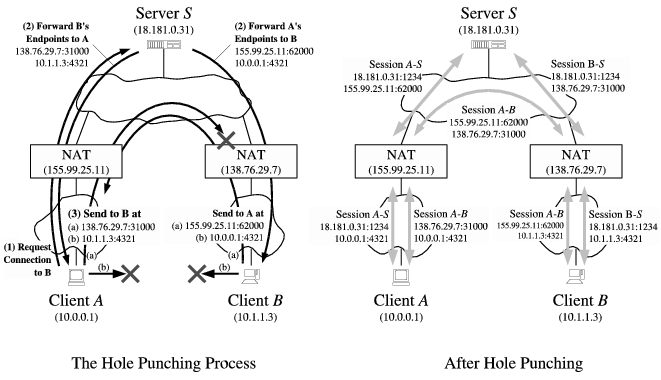


Рис. 2. Подробный процесс UPD Hole Punching

**2 Реализация программной системы**

**2.1 Выбор средств реализации программного кода**

На этапе реализации программного кода выполняется кодирование отдельных компонент программы в соответствии с разработанным техническим проектом.

Средства, которые могут быть применены, в значительной степени зависит от того, какие подходы были использованы во время проектирования и, кроме этого, от степени проработанности технического проекта.

Тем не менее, среди средств разработки программного кода необходимо выделить следующие основные виды средств (в скобках приведены примеры средств):

* методы и методики алгоритмирования.
* языки программирования (C++,Си, Java, C#, php и многие другие);
* средства создания пользовательского интерфейса (MFC, WPF, QT, GTK+ и т.д.)
* средства управления версиями программного кода (cvs, svn, VSS).
* средства получения исполняемого кода (MS Visual Studio, gcc и многие другие).
* средства управления базами данных (Оracle, MS SQL, FireBird, MySQL и многие другие).
* отладчики (MS Visual Studio, gdb и т.д.).

Мой выбор пал на языки программирования такие как Python и JavaScript, в качестве отладчика я выбрал встроенный отладчик в IDE PyCharm. Для контроля версии программной системы был использован git.

Разработанная программная система представляет собой клиент-серверное приложение. Клиентская часть написана на языке программирования Python и для связи с сервером была использована библиотека Socket IO и PySTUN.

Socket.IO — библиотека для веб-приложений и обмена данными в реальном времени. Состоит из двух частей: клиентской и серверной для node.js. Оба компонента имеют похожее API. Подобно node.js, Socket.IO событийно-ориентированная. Socket.IO главным образом использует протокол WebSocket, но если нужно, использует другие методы, например Adobe Flash сокеты, JSONP запросы или AJAX запросы, предоставляя тот же самый интерфейс. Помимо того, что Socket.IO может быть использована, как оболочка для WebSocket, она содержит много других функций, включая вещание на несколько сокетов, хранение данных, связанных с каждым клиентом, и асинхронный ввод/вывод.

Всем известный протокол HTTP является изначально синхронным, то есть построенным по модели «запрос — ответ». WebSocket — это серьезное расширение HTTP, которое позволит приложениям (в том числе и играм) поддерживать многопользовательское взаимодействие в режиме реального времени. Благодаря ему не станет клиента и сервера с фиксированными ролями, а появятся два равноправных участника обмена данными. Каждый при этом работает сам по себе, и когда надо отправляет данные другому. Вторая сторона отвечает по мере необходимости, и только если эта необходимость возникнет.

WebSocket — протокол полнодуплексной двунаправленной связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени. Программисты и ранее пытались решить проблему «равноправия» между сервером и клиентом. Они экспериментировали с двунаправленной связью, используя запросы XMLHttpRequest. Но это было очень медленно, так как каждый раз приходилось создавать новое TCP-соединение для каждого сообщения. В отличие от XMLHttpRequest WebSockets устанавливает одно TCP-соединение и подтверждает, что сервер может общаться с WebSocket, делая специальные проверки, после чего сервер и клиент могут отправлять текстовые сообщения через установленное соединение при необходимости, в результате чего связь становится быстрее. Соединение постоянно держится открытым, но не передаёт лишних HTTP заголовков. При этом в веб-сокетах нет ограничений на количество соединений.

В качестве серверной части был выбран NodeJS. Node.js (или просто Node) — это серверная платформа для работы с JavaScript через движок V8. С помощью Node можно писать полноценные приложения. Node умеет работать с внешними библиотеками, вызывать команды из кода на JavaScript и выполнять роль веб-сервера. Преимущество Node – это возможность легко масштабироваться. При одновременном подключении к серверу тысяч пользователей Node работает асинхронно, то есть ставит приоритеты и распределяет ресурсы грамотнее. Java же, например, выделяет на каждое подключение отдельный поток.

Так же как и на клиенте на сервере стоит модуль (библиотека) SocketIO, но теперь уже в качестве сервера, которая будет в реальном времени обрабатывать запросы пользователей

**2.2 Написание серверной части**

**2.2.1 Написание консольной версии сигнального сервера**

Сигнальный сервер — это координирующий центр, который обеспечивает коммуникацию между клиентами, инициализацию и закрытие соединения, отчеты об ошибках. Сигнальный сервер в этом случае это Node.js + socket.io + Express, он будет слушать порт 8001. Но в случае консольной версии Express становится не нужным, т.к. нет необходимости выдавать страницу пользователю.

Как можно видеть на рисунке 2, структура сигнального сервера состоит из самого скрипта – server.js, конфигурационный файл для npm и папки модулей node\_modules.



Рис. 2. Структура папок консольного сигнального сервера

Основным модулем является SocketIO. Он создает события и обработчики событий, что упрощает разработку. Основной задачей сервера является работа с пользователями, поэтому был написан модуль для работы с ним. Основной код сервера предоставлен в приложении 1.

Модуль называется user.js.В модуле находится основной класс User и вспомогательный UserStore. Эти модули продемонстрированы на рисунке 3 и рисунке 4. В классе User присутствует минимум для работы с пользователями. В нем присутствуют основные описывающую информацию о пользователе и функции для работы с ними.

* Id – уникальный идентификатор выдаваемый сервером SocketIO
* username – уникальное имя пользователя. По нему осуществляется поиск других хостов.
* localAddress – локальный адрес на который ожидается подключение
* localPort – локальный порт на который ожидается подключение
* remoteAddress – публичный адрес пользователя. Предварительно узнанный с помощью STUN сервера.
* remotePort – публичный порт
* setLocalAddress - функция устанавливающая локальный адрес пользователя.
* setUserName - функция устанавливающая имя пользователя
* setLocalPort - функция устанавливающая локальный порт
* setRemoteAddress - функция устанавливающая публичный адрес пользователя
* setRemotePort - функция устанавливающая публичный порт
* updateFields - функция устанавливающая те атрибуты которые были переданы функции. Это бывает удобно когда необходимо установить все поля сразу.

Поля и методы вспомогательного класса UserStore. Этот класс служит для упрощения работы с хранилищем пользователи. Упрощает добавление, поиск, удаление пользователей. Основные поля:

* addUser – функция добавления нового пользователя. На вход получает объект User.
* getUserList – функция возвращающая весь список пользователей
* findByName – функция поиска пользователя по имени. На вход получает имя. Возвращает массив объектов User.
* checkUsername - функция проверяющая существует ли пользователь или нет. Получает имя пользователя, а возвращает true или false.
* deleteById – функция удаляющая пользователя по ID. Входной параметр это ID пользователя.
* updateUser – обновление данных подключенного пользователя. Получает объект атрибутов для обновления и ID пользователя, которому необходимо обновить
* findAllByName – возвращает список совпадающих имен. Возвращает массив объектов User
* findById – функция осуществляющая поиск пользователя по ID. Возвращает объект User
* userList – массив для хранения всех пользователей подключенных к серверу

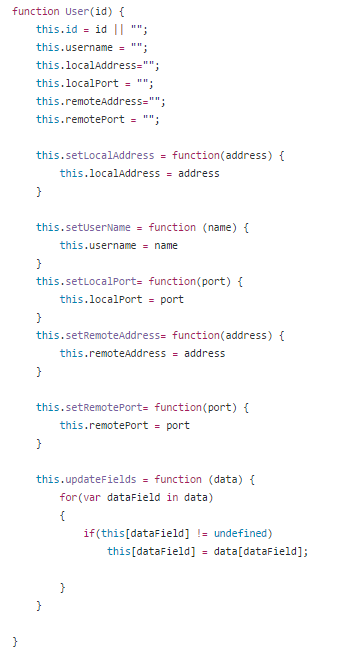


Рис. 3. Код класса User

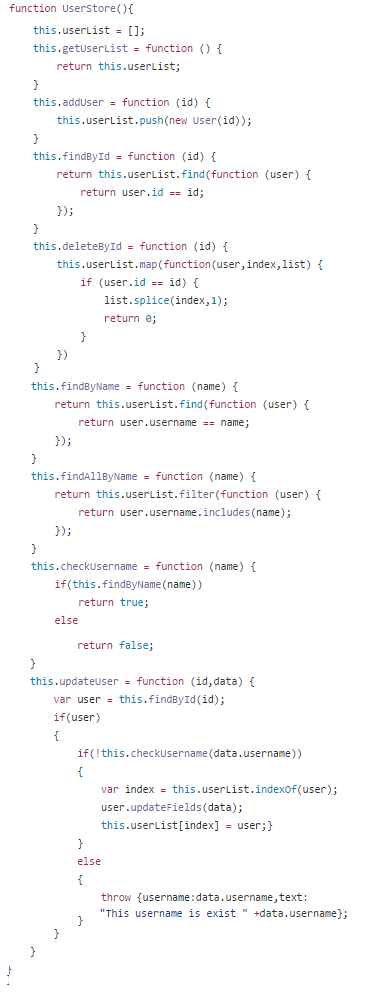


Рис. 4 Класс UserStore

Логика сервера следующая:

1. Сервер ожидает подключения пользователей
2. При подключении пользователя, сервер регистрирует его, выполняется событие «newConnection» которое продемонстрировано на рисунке 5. В нем выполняются следующие функции:
   1. updateConnectedClients – функция обновляющая список подключенных пользователей. Код функции продемонстрирован на рисунке 6.
   2. UserStore.addUser(client.id) – функция модуля UserStore, которая добавляет в хранилище нового пользователя, пока что без имени.
   3. Объекту client присваиваются события
      1. sendTo – события для обработки отправки сообщения другому клиенту
      2. update-info – события для обработки основной информации о клиенте. Сюда входят и имя пользователя, локальный адрес, порт, внешний адрес и порт.
      3. disconnecting – событие возникающее, когда пользователь отключился от сервера. Это нужно для того что бы обновить списки подключенных пользователей. Код обработчика этого события продемонстрирован на рисунке 6.
      4. get-user – событие возникающее, когда пользователь запрашивает информацию у сервера о другом клиенте.
3. После срабатывания newConnection, сервер ожидает команды от клиента и в зависимости от полученной команды сервер будет выполнять определенные функции, такие как:
   1. Регистрация нового пользователя
   2. Удаление пользователя
   3. Отправка информации о удаленном хосте
   4. Обновление информации о пользователе

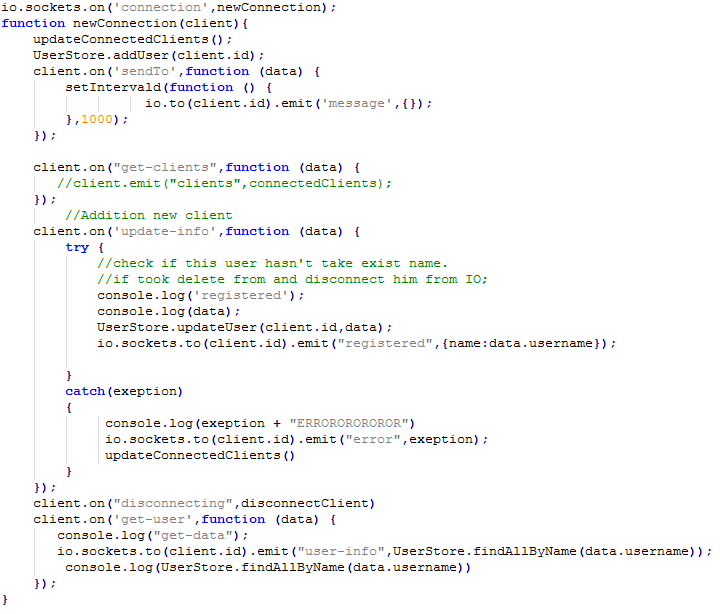


Рис. 5. Программный код события newConnection

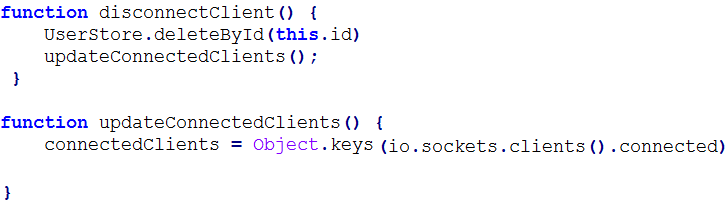


Рис. 6. Программный код обработчиков

Консольный сервер простой, но очень эффективный, т.к. он весит всего 5 мегабайт и выполняет все основные функции необходимые для UDP Hole Punching.

**2.2.2 Написание сигнального сервера с веб интерфейсом**

**2.2.2.1 Серверная часть**

Вторая версия серверной части базируется на консольной версии. Она работает с клиентом через веб-браузер, выдавая веб страницу. В этой версии так же в качестве сервера используется NodeJS, но уже в другой конфигурации. NodeJS + SocketIO + Express.

Express - это минималистичный и гибкий веб-фреймворк для приложений Node.js, предоставляющий обширный набор функций для мобильных и веб-приложений. Он спроектирован для создания веб-приложений и API. Де-факто является стандартным каркасом для Node.js. Автор фреймворка, описывает его как созданный на основе написанного на языке Ruby каркаса Sinatra, подразумевая, что он минималистичен и включает большое число подключаемых плагинов.

На этом сервере будет использован далеко не весь функционал, он служит только для выдачи страницы пользователям. Вторая версия сервера хоть и базируется на консольной версии, но все-таки отличается от нее. Поменялась структура папок. Была добавлена папка public. Она отвечает за веб клиентскую часть. В ней хранится все необходимые страницы для выдачи и необходимые библиотеки, css файлы, шрифты.

Главное отличие от консольного сервера были добавлены новые события и обработчики к ним. Их можно увидеть на рисунке 7.

* Событие update служит для обработки запроса на обновления данных от пользователя. В основном служит для обработки публичных и приватных IP: PORT
* Событие accept служит для оповещения пользователя о том, что пользователь принял его запрос на подключения
* Событие decline служит для оповещения пользователя о том, что пользователя отклонил его запрос на подключения

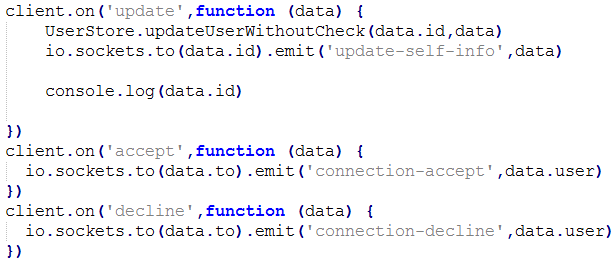


Рис. 7. Код добавленных событий

В этом и есть главное отличие консольного сигнального сервера от веб сигнального сервера. В нем так же используется чуть-чуть изменений модуль user.js, который включает в себя User и UserStore. В UserStore была добавлена новая функция updateUserWithoutCheck. Она обновляла информацию о пользователе без проверки существует пользователь или нет. В этом случае проверка не требуется, так как все проверки были до выполнения этой функции, а код этой функции продемонстрирован на рисунке 8.

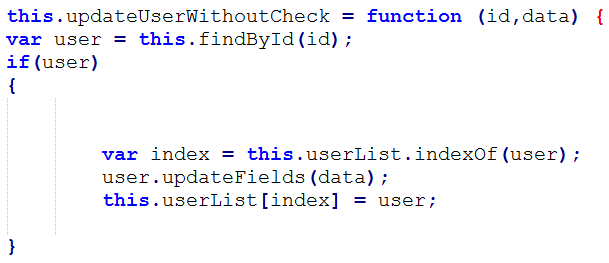


Рис. 8. Программный код функции updateUserWithoutCheck.

**2.2.2.2 Веб интерфейс**

Веб интерфейс представляет собой прослойку между клиентской частью и серверной. Он представляет собой веб сайт + SocketIO + Materializecss + JQuery. В этом случае SocketIO будет играть роль клиента, который будет общаться с сервером. Веб интерфейс лежит в папке public. Структура папок веб интерфейса показана на рисунке 9.

* Файл index.html является веб интерфейсом.
* Файл sketch.js является пользовательским скриптом для работы с сервером.
* Папка libraries содержит все необходимые библиотеки для работы веб интерфеса. По стандарту она разделена на папки
  + css – в этой папке лежит файлы стилей
  + js – в этой папке лежат дополнительные библиотеки для sketch.js
  + fonts – в этой папке лежат шрифты.

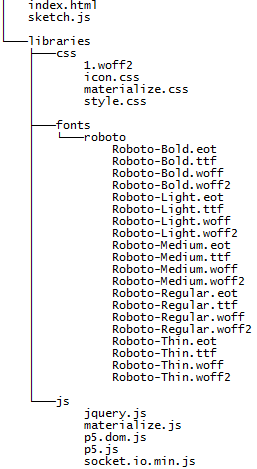


Рис. 9. Структура файлов веб интерфейса

Для веб интерфейса был использован материальный дизайн. Material Design (рус. Материальный дизайн) — дизайн программного обеспечения и приложений операционной системы Android от компании Google. Идея дизайна заключается в приложениях, которые открываются и сворачиваются как карточки, используя эффекты теней. По идее дизайнеров Google, у приложений не должно быть острых углов, карточки должны переключаться между собой плавно и практически незаметно.

Вдохновившись этой идей, энтузиасты начали делать свои библиотеки материального дизайна для веб приложений. Одна из них Materializecss. Она представляет собой CSS + JS+ JQuery библиотеку. Она очень проста в освоении, т.к. на сайте библиотеки есть хорошая документация.

Основной интерфейс разбит на несколько частей. Часть регистрации пользователя и часть для работы с приложением. Часть регистрации продемонстрирована на рисунке 10.

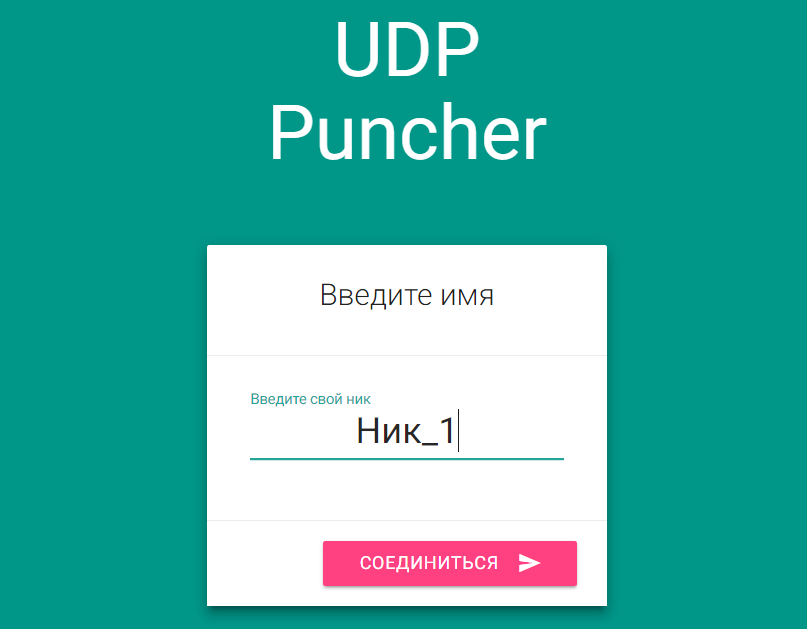


Рис. 10. Экран логина.

При успешной регистрации пользователь получает зеленное уведомление как на рисунке 11.



Рис. 11. Успешная регистрация

А при неудачной регистрации. Например, когда такой пользователь уже подключён к серверу, пользователь получает оранжевое уведомление такое как на рисунке 12.



Рис. 12. Ошибка регистрации

После регистрации пользователь попадает в следующее меню, где он будет дальше работать. Это меню разделил на 3 блока, они так же продемонстрированы на рисунке 13:

* Блок, в котором показана собственная информация.
* Блок поиска других пользователей.
* Блок информации об удаленной пользователе.

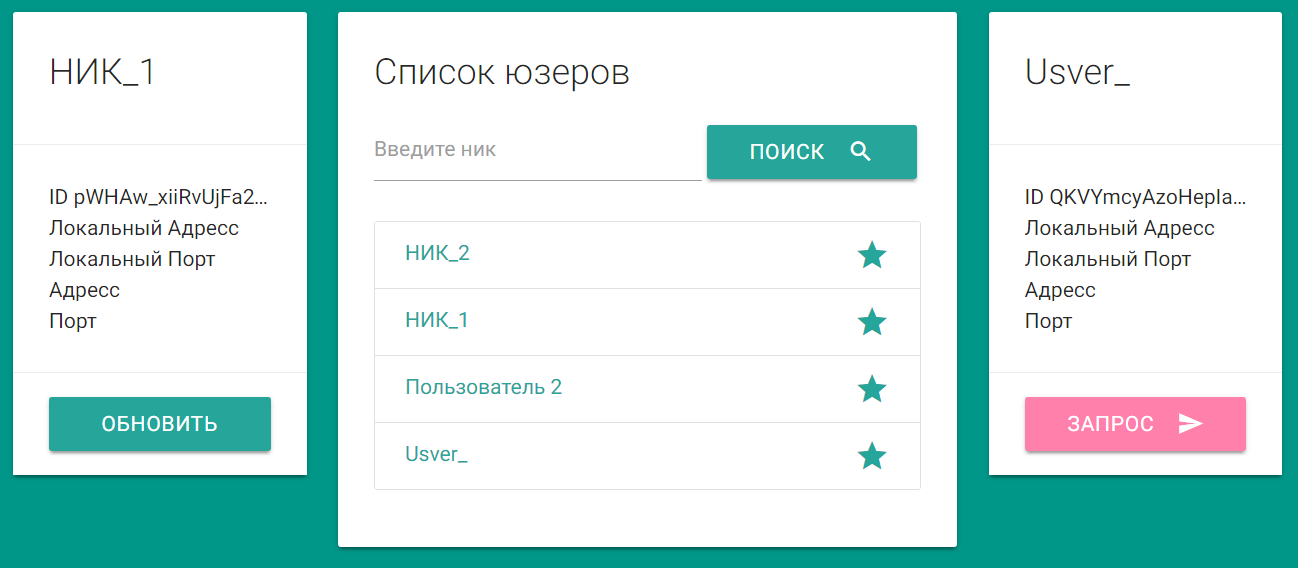


Рис. 13. Главное меню

Как можно видеть добавлены 3 кнопки. Обновить, поиск, запрос. С поиском все понятно, при нажатии на нее отправляется запрос на сервер на поиск подходящего пользователя. При нажатии кнопки обновить запускается основная клиентская программа, которая обновляет информацию о хосте (локальный порт, локальный адрес, публичный порт, публичный адрес). При нажатии кнопки запрос, на сервер отправляется запрос на обработку запроса, что в свое время пересылает его соответствующему пользователю. При получении у пользователя открывается соответствующее окно как на рисунке 14, в котором он может принять или отклонить запрос.

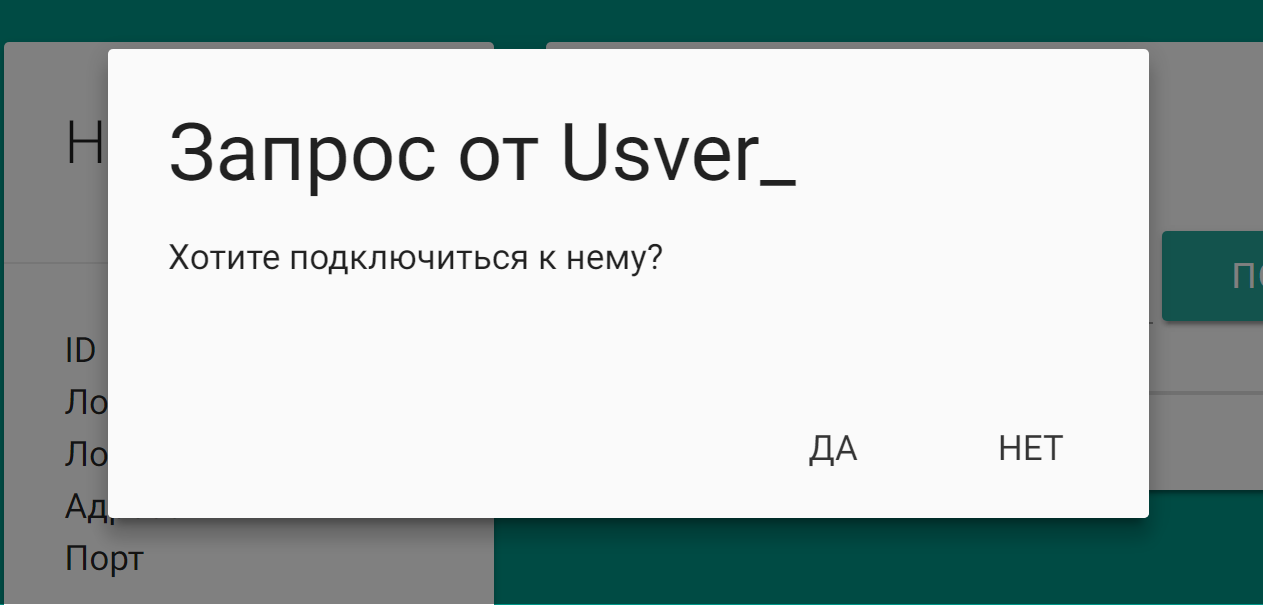


Рис. 14. Окно запроса на подключение

При отклонении запроса, адресанту посылается ответ и высвечивается сообщение об отказе. Пример сообщения продемонстрирован на рисунке 15. А при согласии, запускается скрипт, который начинает инициировать двустороннее соединение, а адресанту посылается уведомление, что пользователь принял запрос. Пример уведомления продемонстрирован на рисунке 16.



Рис. 15. Уведомление отказа на подключение



Рис. 16. Уведомления согласия на подключение

Все эти операции описаны в основном скрипте sketch.js. Клиентская часть работает следующим образом. Клиент при нажатии определенных кнопок, генерирует определенные события, в следствии сервер на них реагирует и выполняет те действия, которые описаны в обработчике события.

Первым делом, как только пользователь заходит на сайт, инициализируется скрипт, в котором до загрузки страницы объявляются некоторые глобальные переменные. Их можно увидеть на рисунке 16.

* socket – в этой переменной храниться объект SocketIO, который отвечает за связь в сервером. Любые запросы получает и генерирует эта переменная
* myID – собственный ID выданный сервером SocketIO
* servername – переменная которая нужна для определения имени сервера на, котором работает скрипт. Это нужно для того что бы клиентская часть знала куда отправлять данные клиента
* searchedUsers – переменная, которая хранит всех пользователей которые прислал сервер.
* currentUser – переменная хранящая объект пользователя, который пользователь выбрал в данный момент.
* stunUpdated – переменная для проверки, обновил ли пользователь свои данные.
* lastmodalID – переменная хранящая id последнего модального окна
* Объект selfInfo – объект, который хранит всю информацию о пользователе
* translatedInfo – объект созданный для упрощения локализации веб интерфейса. Нет необходимости менять текст по всему документы, необходимо просто ввести переведённый текст в этот объект.

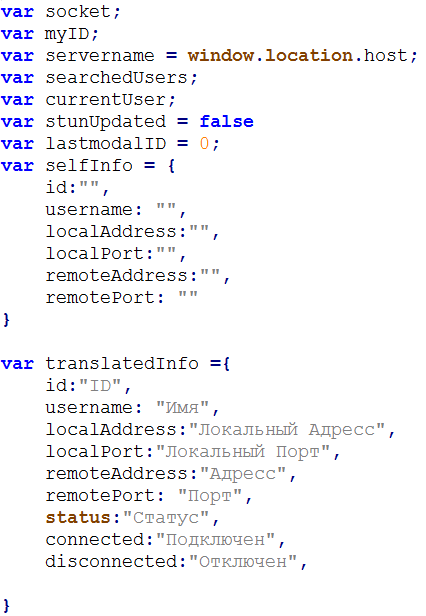


Рис. 16. Глобальные переменные

После загрузки документа, после того как DOM загрузился, скрипт подключается к SocketIO, который находится на сервере. После чего идет описание событий, которые сервер генерирует для клиента. Их можно увидеть на рисунке 17.

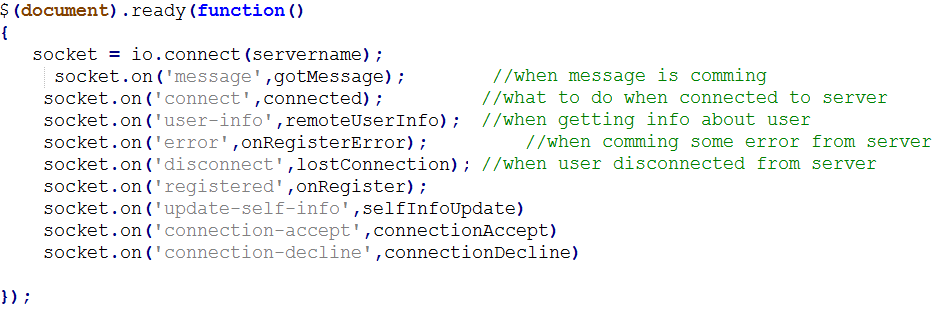


Рис. 17. Описание событий

* Событие message – возникает при получении сообщении от сервера или другого клиента
* Событие connect – возникает, когда скрипт подключается к SocketIO
* Событие user-info – возникает, когда сервер присылает информацию о пользователях
* Событие error – возникает при ошибке регистрации. Когда имя пользователя уже занято
* Событие disconnect – возникает при отключении от сервера
* Событие registered – возникает при успешной регистрации
* Событие update-self-info – возникает, когда клиентская часть обновляет свои данные
* Событие connection-accept – возникает при согласии на подключении
* Событие connection-decline – возникает при отклонении запроса на подключения

Как только пользователь нажимает кнопку обновить (см. рис. 13), скрипт обрабатывает функция updateStun, которая показан на рисунке 18.

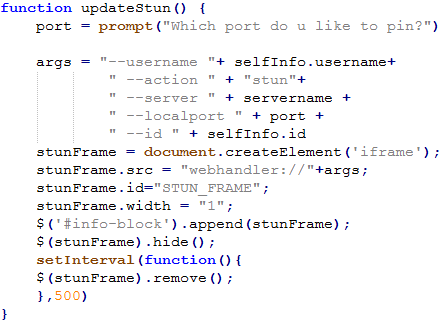


Рис. 18. Функция updateStun

Она создает фрейм с адресом webhandler:// + некие параметры для скрипта, который находится на локальном компьютере, который в свою очередь делает STUN запрос и отправляет ответ серверу, который сохраняет результат в UserStore и посылает параллельно пользователю, который нажал кнопку обновить. По умолчанию при обращении на адрес webhandler:// ничего не произойдет, т.к. операционная система не знает, как обрабатывать такой запрос. Для решения этой проблемы были созданы скрипты, написанные на скриптовых языках Batch и Bash для Windows и для Linux с графической оболочкой от GNOME. Они создают правило в ОС для обработки пользовательских протоколов таких как webhandler:// и tunnel://. Эти скрипты можно увидеть на рисунке 19 и на рисунке 20



Рис. 19. Скрипт install.bat для Windows

Выше указанный скрипт создает 2 файла, tunnel.reg и webhandler.reg. Это файлы регистра Windows. После создания файлов, пользователь должен запустить их, что бы Windows могла обработать webhandler:// и tunnel://.

Скрипт сам по себе очень простой, он командой where узнает где лежит бинарный файл python и записывает его в временный файл tmp. Далее в переменную pythonpath загружается содержимое файла tmp и все / заменяются на //.

Это необходимо для того что бы реестр Windows понимал, что это директория. А после настройки переменных сервер просто записывает создает построчно 2 реестровых файла. С подстановками переменных. После чего удаляется временный файл tmp.

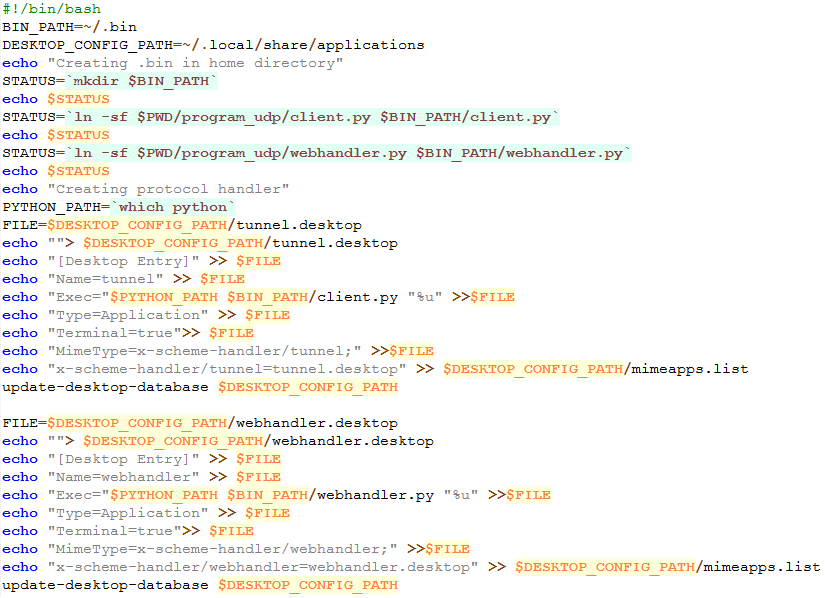


Рис. 20. Скрипт install.sh для Linux

Скрипт для Linux работает почти точно так же, кроме вместо создания реестровых файла, он создает 2 .desktop файла. Это конфигурационные файлы для графического окружения GNOME.

После настройки переменных и создания файлов, скрипт заносит в файл mimeapps.list данные о том что следует обрабатывать протоколы webhadler:// и tunnel:// файлами webhandler.desktop и tunnel.desktop. После чего Linux знает как обрабатывать эти протоколы.

После того как были созданы пользовательские протоколы, можно вернуться к скрипту, который работает на веб интерфейсе.

После того как пользователь нажал кнопку обновить его попросят ввести номер локального порта, на который будет ожидаться соединение. После чего откроется консоль и данные в собственном поле обновятся. За отображение и обработку данных от сервера отвечают функции selfInfoUpdate и renderSelfInfo. Они изображены на рисунке 21 и рисунке 22.

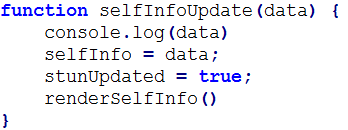


Рис. 21. Функция selfInfoUpdate

Функция selfInfoUpdate получает от сервера объект selfInfo, который заменяет на существующий. После чего отрисовывает эти данные на экран функцией renderSelfInfo, код которой показан ниже.

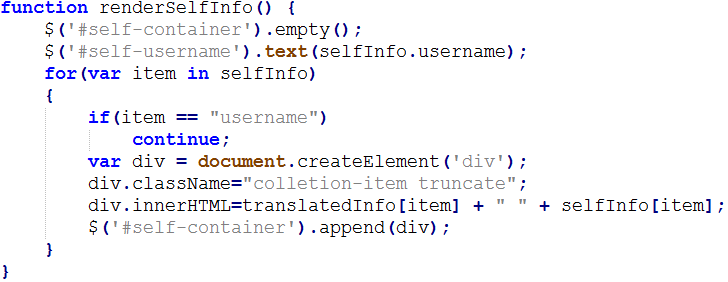


Рис. 21. Функция renderSelfInfo

Если пользователь нажимает по кнопке поиск, скрипт генерирует событие get-user и сервер отправляет ответ список всех пользователей, у которых имя совпадает с тем что ввел пользователь. Ответ от сервера генерирует событие на стороне клиента под названием user-info, обработчиком события которого является функция remoteUserInfo. Код этой функции предоставлен на рисунке 22.

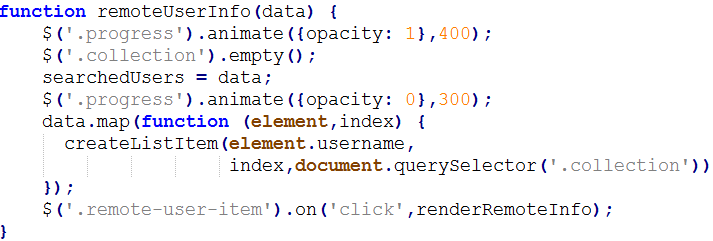


Рис. 22. Функция remoteUserInfo

Функция получает массив объектов User и через функцию Array.map создает и отрисовывает список на экран пользователя с помощью функции createListItem. Функция получает имя пользователя, индекс в массиве и элемент, к которому необходимо привязать элемент списка в роли дочернего элемента. Код функции продемонстрирован на рисунке 23.

Функция создает 3 элемента и задает им классы css, id и текст списка, после чего функцией appendChild привязывает полученный список к переданному элементу и потом отрисовывается на экране.

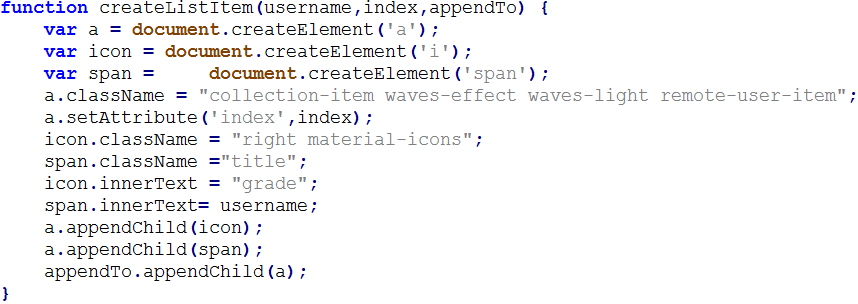


Рис. 23. Функция createListItem

Последняя кнопка в веб интерфейсе - это кнопка запроса. По нажатию на нее скрипт создает событие, на сервере которое сигнализирует о том, что пользователь хочет инициировать соединение. Делает он это с помощью функции socket.emit(‘sendTo’,data). Вместо data отслаивается объект User с которым необходимо инициировать подключение.

При получении такого запроса на подключения срабатывает событие message. Обрабатывается оно функцией gotMessage. Которая сигнализирует о том что пользователь пытается соединиться. И дается 2 кнопки. Да или нет. В зависимости от того что нажмет пользователь, скрипт сгенерирует определенное событие на сервере.

При согласии пользователя на подключения у обоих пользователей открывается адрес tunnel:// + параметры необходимые для подключения и начинается двусторонняя попытка подключения к друг другу.

**2.3 Написание клиентской части**

**2.3.1 Модуль для работы с сигнальным сервером**

Клиентская часть состоит из двух частей. Обработчик протокола webhandler:// он же модуль для работы с сигнальным сервером и tunnel:// модуль для работы с UDP Hole Punching.

Для написания модулей был выбран язык python и ряд разных библиотек. Основные SocetIO и PySTUN. PySTUN это библиотека для работы с STUN сервером, она делает запросы к STUN серверу и обрабатывает запросы от него.

Задача модуля работающий с сигнальным сервером состоит в том, чтобы сделать запрос к STUN серверу и отослать эти данные на сервер, сообщив ему от кого пришли и передать данные полученные от STUN сервера. Дальше сигнальный сервер автоматически обновит данные у себя и оповестит пользователя о том, что данные обновлены.

Структуру папок клиентской части можно увидеть на рисунке 24. Большинство папок — это необходимые модули для работы SocketIO и PySTUN, кроме модуля sub\_tools. В этой папке хранятся вспомогательные модули, которые были написаны на python.

Основными исполняемыми файлами являются webhadler.py и client.py. Файл webhandler.py отвечает за обработку адреса webhandler://.

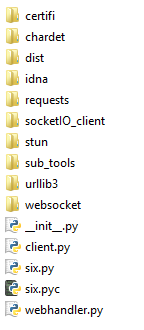


Рис. 24. Структура папок клиентской части

Основной код webhandler.py продемонстрирован на рисунке 25. Первым делом при запуске программы происходит импортирование необходимых библиотек таких как SocketIO, stun, sys, time. После чего фильтрует входные данные которые приходят с браузера. Она приходят в виде строки, эту строку необходимо отфильтровать строку от префикса протокола, после разбить строку на элементы массива.

После фильтрации строки, массив параметров обрабатывается, в результате чего программа узнает какие параметры были переданы, а какие нет. Если какие-нибудь параметры не были переданы, то программа завершается с ошибкой.

После проверки переданных параметров, скрипт совершает запрос к STUN серверу и создает на основе результата от сервера объект selfInfo в скрипте он называется DATA\_TO\_SEND и отсылает на сервер данные о пользователе, который совершил запрос на обновления данных.

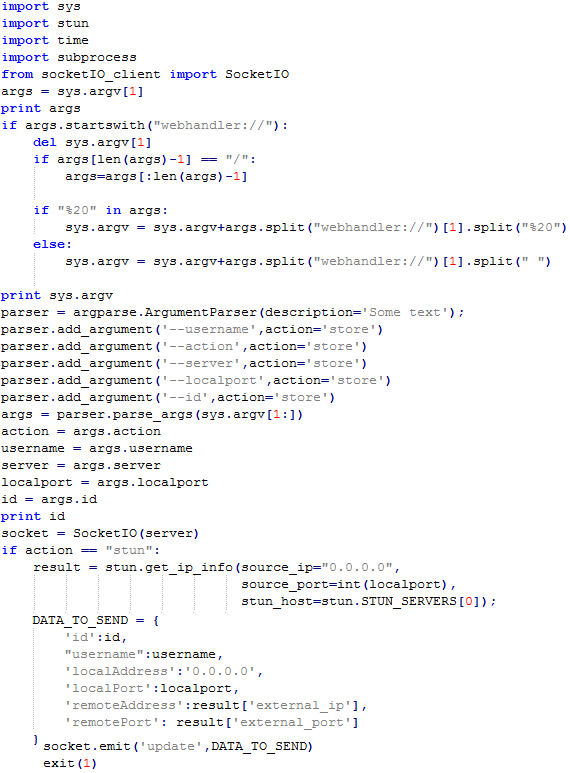


Рис. 25. Файл webhandler.py

**2.3.1 Модуль для работы с UDP Hole Punching**

В отличии от webhandler.py скрипт client.py работает независимо от типа сервера, он работает, как и на консольном так и на сервере с веб интерфейсом. Изначально он был создан для работы с консольной версией сервера.

Как и sketch.js он генерирует события на сервер и так же их обрабатывает. Логика выполнения программы делится на 2 части. Если в качестве параметра задать –d, то произойдет попытка прямого подключение без использования сервера. Если нет этого параметра, то программа будет работать с консольной версией сервера. Он будет пытаться найти определенного клиента и попробует подключиться к нему.

Первым делом скрипт импортирует необходимые библиотеки, кроме SocketIO, stun и т.д. он импортирует argcontroller, externalvars, tunnelmaker. Эти модули были написаны для того что бы упростить логику программы.

* argcontreller – модуль для работы с входными параметрами. Код модуля продемонстрирован на рисунке 26.
* externalvars – модуль в котором содержатся вспомогательные функции
* tunnelmaker – модуль в котором содержатся функции для работы с UDP Hole Punching

Как видно на рисунке 26 модуль содержит несколько функций. Функция directConnection служит для прямого подключения 2х хостов. При передаче параметра –d происходит выполнения этой функции. Функция breakthroughTunnel использует метод UDP Hole Punching для соединения 2х хостов находящиеся за NAT.

В данном модуле присутствует еще несколько функций, но они более опциональны чем обязательны.

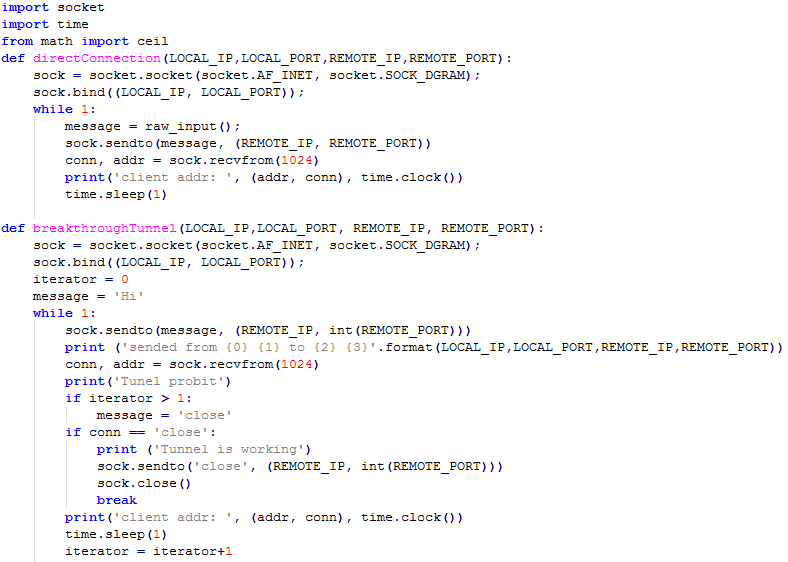


Рис. 26. Код модуля tunnelmaker

Модуль externalvars содержит вспомогательные функции и переменные. Вот список основных переменных и функций находящиеся в этом модуле:

* Переменная args. Она получает список переданных параметров через функцию getargs() которая находится в модуле argcontroller
* DIRECTCONNECTION тоже самое что и args.directconnection. Служит для сигнализации о том что программа запущенна в режиме прямого подключения.
* SOURCE\_PORT тоже самое что и args.sourceport. Содержит локальный IP адрес
* SOURCE\_HOST тоже самое что и args.sourcehost. Содержит номер локального порта
* REMOTE\_HOST тоже самое что и args.remotehost. Содержит IP адрес удаленного хоста.
* REMOTE\_PORT тоже самое что и args.remoteport. Содержит удаленный порт для подключения
* IS\_SERVER тоже самое что и args.server. Указывает включать ли расшаркивания локального порта
* OWN\_EXTERNAL\_IP переменная которая содержит собственный публичный адрес
* OWN\_EXTERNAL\_PORT переменная которая содержит собственный публичный порт
* SIGNALSERVERHOST тоже само что и args.signalserver. Содержит в себе адрес сигнального сервера.
* SOCKETIO\_CLIENT содержит в себе объект клиента SocketIO
* doStunRequest – функция которая обращается к STUN серверу. Код функции предоставлен на рисунке 27.

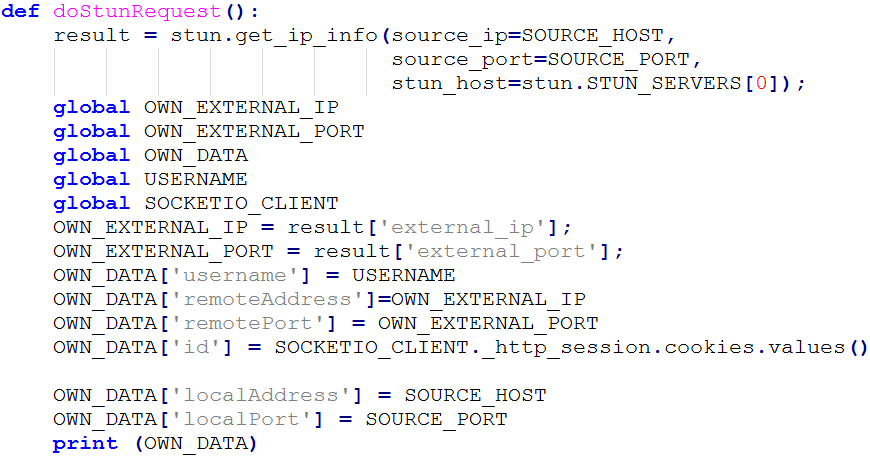


Рис. 27. Код функции doStunRequest

* OWN\_DATA – переменная которая содержит информацию о себе:
  + id
  + username
  + localAddress
  + localPort
  + remoteAddress
  + remotePort
* REPEAT\_TIME\_USERSEARCHER – переменная, которая содержит время повторного запроса к серверу, если искомый клиент еще не зарегистрировался на сервере
* Функции regUser, updUser, getUser используются для общения с сервером. Они генерируют определенные события на сервере. Код этих функций представлен на рисунке 27

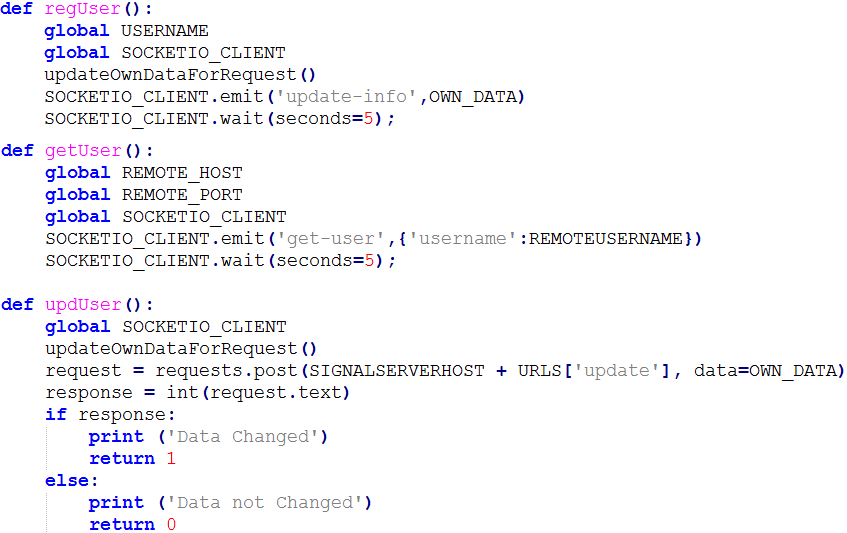


Рис. 27. Функции для работы с сервером

После инициализации параметров программа client.py создает поток который будет держать открытым внешний порт. Он не позволит внешним роутерам поменять порт. Далее в зависимости от параметра –d программа либо делает попытку прямого подключения или подключается к сигнальному серверу попутно создавая события и обработчиков к ним. Это все можно увидеть на рисунке 28. Как видно на рисунке поток autostun включается сразу при прямом подключении.

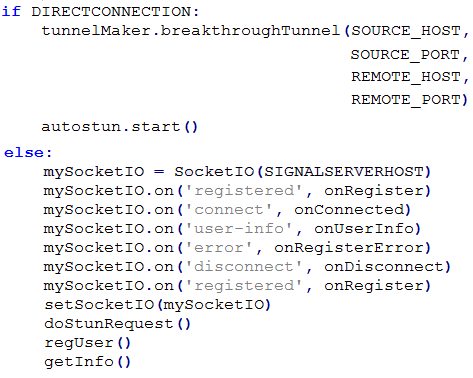


Рис. 28. Код логической развязки client.py

После успешной регистрации сервер генерирует событие registered и client.py запускает функцию onRegister и запрашивает информацию у сервера о удаленном клиенте выполняя функцию getInfo. При ответе, сервер генерирует событие user-info которое обрабатывается функцией onUserInfo. Как только клиент получил информацию он начинает подключение к удаленному хосту. Для наглядного. Код функции onRegister предоставлен на рисунке 29, а код функции onUserInfo на рисунке 30.

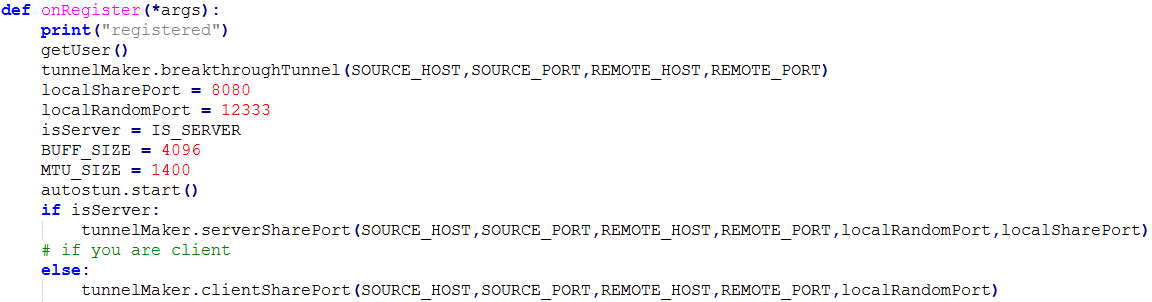


Рис. 29. Код функции onRegister

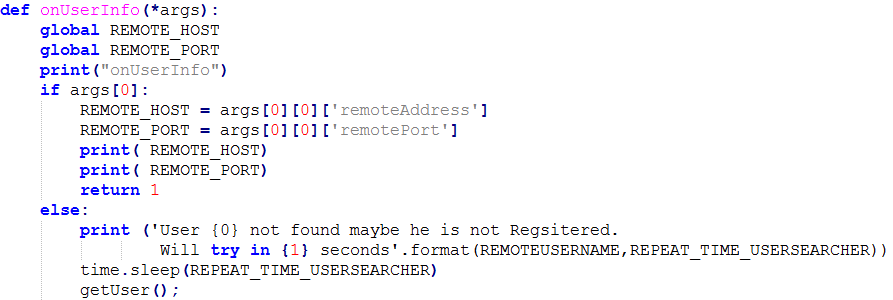


Рис. 30. Код функции onUserInfo

**Заключение**

В ходе выполнения дипломной работы была создана программная система, позволяющая людям находящиеся за NAT подключаться к друг другу на прямую. Данная программная система использует методику UDP Hole Punching для обеспечения связи между хостами.

В ходе работы были созданы 2 сигнальных сервера, веб интерфейс для работы упрощения работы с сигнальным сервером. Была создана клиентская часть, которая обеспечивала связь между хостами и сервером. Благодаря библиотеке SocketIO работающая с WebSocket, которая упростила код серверной и клиентской части. Без этой библиотеки весь код программной систему увеличился бы в несколько раз, что позволило бы ухудшится понимание, масштабирования кода в дальнейшем.

Созданная программная система доступна в git репозитории по адресу https://github.com/ruLegen/UDP-TCP-Tunnel и доступная для скачивания. Весь программный код можно найти в этом репозитории.

**Список литературы**

1. Документ из серии пронумерованных информационных документов Интернета «RFC 5389» от октября 2008 года
2. Bryan Ford, Pyda Srisuresh, Dan Kegel “Peer-to-Peer Communication Across Network Address Translators”
3. Сети для самых маленьких. Часть 5 ACL и NAT [Электронный ресурс]. URL: http://linkmeup.ru/blog/16.html. (Дата обращения: 20.05.2017)
4. Peer-to-Peer Communication Across Network Address Translators [Электронный ресурс]. URL: http://www.brynosaurus.com/pub/net/p2pnat/. (Дата обращения: 21.05.2017)
5. Wikipedia Свободная энциклопедия. NAT [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/NAT. (Дата обращения: 20.05.2017)
6. How Stuff Works Network Address Translation [Электронный ресурс]. URL: http://computer.howstuffworks.com/nat.htm. (Дата обращения: 20.05.2017)
7. 3CX. Что такое STUN сервер [Электронный ресурс]. URL: https://www.3cx.ru/voip-sip/stun-server/. (Дата обращения: 22.05.2017)
8. Wikipedia Свободная энциклопедия. STUN [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/STUN (Дата обращения: 22.05.2017)
9. Wikipedia UDP Hole Punching [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/UDP\_hole\_punching . (Дата обращения: 23.05.2017)
10. Хабрахабр UDP hole punching для Symmetric NAT [Электронный ресурс]. URL: https://habrahabr.ru/post/155803/ (Дата обращения: 23.05.2017)
11. Node JS Docs [Электронный ресурс]. URL: https://nodejs.org/en/docs/. (Дата обращения: 24.05.2017)
12. Microsoft Registering an Application to a URI Scheme [Электронный ресурс]. URL: https://msdn.microsoft.com/en-us/library/aa767914(v=vs.85).aspx (Дата обращения: 1.06.2017)

Приложение 1

**Программный код консольной версии сигнального сервера**

var express = require('express');

var socket = require('socket.io');

var user = require('user');

var app = express();

var server = app.listen(8002);

app.use(express.static('public'));

var io = socket(server);

var UserStore = new user.UserStore();

var connectedClients = [];

io.sockets.on('connection',newConnection);

function newConnection(client){

updateConnectedClients();

UserStore.addUser(client.id);

client.on('sendTo',function (data) {

setIntervald(function () {

io.to(client.id).emit('message',{});

},1000);

});

client.on("get-clients",function (data) {

});

client.on('update-info',function (data) { //Addition new client

try {

console.log('registered');

console.log(data);

UserStore.updateUser(client.id,data);

io.sockets.to(client.id).emit("registered",{name:data.username});//check if this user hasn't take exist name. if took delete from and disconnect him from IO;

}

catch(exeption)

{

console.log(exeption + "ERROROROROROR")

io.sockets.to(client.id).emit("error",exeption);

updateConnectedClients()

}

});

client.on("disconnecting",disconnectClient)

client.on('get-user',function (data) {

console.log("get-data");

io.sockets.to(client.id).emit("user-info",UserStore.findAllByName(data.username));

console.log(UserStore.findAllByName(data.username))

});

}

function disconnectClient() {

UserStore.deleteById(this.id)

updateConnectedClients();

}

function updateConnectedClients() {

connectedClients = Object.keys(io.sockets.clients().connected)

}