**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ**

**Кафедра дифференциальных уравнений и системного анализа**

# РАЗРАБОТКА БИБЛИОТЕКИ ДЛЯ РАЗВЁРТЫВАНИЯ P2P СЕТИ

Курсовая работа

Боровского Ильи Евгеньевича
студента 3-го курса
специальности 1-31 03 09
«Компьютерная математика
и системный анализ»

Научный руководитель:
ст. преподаватель А. В. Кушнеров

Минск, 2023

# ОГЛАВЛЕНИЕ

# ВВЕДЕНИЕ

С ростом и развитием сети Интернет разработка одноранговых (peer-to-peer) сетей стала сложнее: с исчерпанием адресов IPv4 появилась технология Network Address Translator, объединяющая устройства в локальные сети и не дающая устройствам беспрепятственно взаимодействовать с устройствами из других локальных сетей. Теперь одноранговые сети обязаны иметь сервера в своей архитектуре, что роднит её с клиент-серверной архитектурой. Отличием являются лишь требования к характеристикам серверов.

Peer-to-peer сети имеют ряд недостатков:

- высокое потребление ресурсов на устройствах: и клиентская, и серверная часть объединены в одну программу, все данные хранятся у пользователя сети;
- ограничение один пользователь — одно устройство (в известных реализациях);
- ограниченность в возможностях;
- сложность в использовании;
- безопасность передачи информации.

В дипломной работе будет разработана библиотека yap2p для развёртывания peer-to-peer сети на языке программирования Rust. Библиотека будет предоставлять возможность разработки различных прикладных приложений, что будет достигаться новой топологией сети. YAP2P также будет предоставлять набор структур и протоколов, позволяющих пользователям сети иметь более одного устройства. Для более оптимальной защиты сообщений предложен новый протокол защиты информации на основе симметричного шифрования и технологии блокчейн.

В ходе выполнения дипломной работы были поставлены следующие задачи:

- Изучение теории компьютерных сетей.

- Изучение существующих подходов к построению peer-to-peer сетей.
- Проектирование собственной сети, решающей недостатки существующих.
- Разработка протокола связи в сети и подключения к ней.

# ГЛАВА 1

# Необходимые сведения о сетях

## 1.1 Эталонные модели

В теории компьютерных сетей выделяют две эталонные модели архитектуры: OSI и TCP/IP. Эти модели представляют собой деление сети на уровни, а также набор протоколов. Несмотря на то, что протоколы, связанные с эталонной моделью OSI, сейчас не используются, сама модель до сих пор весьма актуальна. В эталонной модели TCP/IP все наоборот: сама модель сейчас почти не используется, а ее протоколы являются едва ли не самыми распространенными. В современной сети модели OSI и TCP/IP служат двум разным целям. Модель TCP/IP — это модель реализации, поскольку она предоставляет руководство для тех, кто будет создавать сетевое оборудование или программное обеспечение, совместимое с TCP/IP. Модель OSI — это, скорее, абстрактная модель, которую можно использовать для понимания широкого спектра сетевых архитектур. [11, 13]

### 1.1.1 OSI

Модель OSI состоит из семи уровней:

1. Физический уровень занимается непосредственной передачей данных по каналу связи.

2. Канальный уровень обеспечивает взаимодействие физического и сетевого уровней. Обнаруживает и корректирует ошибки передачи данных на физическом уровне от сетевого уровня, а также обеспечивает согласование скоростей в сети.

3. Сетевой уровень занимается управлением операциями подсети, основной из которых является маршрутизация пакетов, то есть определение пути, по которому будет осуществляться пересылка пакетов. В случае высокой нагруз-

ки происходит перестройка маршрута, из-за чего последовательные пакеты могут идти по сети разными путями и приходить адресату в разнобой.

4. Транспортный уровень является прослойкой между сеансовым и сетевым уровнями. Обеспечивает правильную доставку данных, а также изолирует более высокие уровни от каких-либо изменений в аппаратной технологии с течением времени.

5. Сеансовый уровень позволяет пользователям различных устройств устанавливать сеансы связи друг с другом. При этом предоставляются различные типы сервисов, среди которых:

   - управление диалогом — отслеживание очередности передачи данных;
   - управление маркерами — предотвращение одновременного выполнения критичной операции несколькими системами;
   - синхронизация — установка служебных меток внутри длинных сообщений, позволяющих продолжить передачу с того места, на котором она оборвалась, даже после сбоя и восстановления.

6. Уровень представления занимается по большей части синтаксисом и семантикой передаваемой информации. Преобразует различные внутренние представления данных в стандартизированный вид, тем самым предоставляя возможность определения и изменения структур данных более высокого уровня. Для повышения эффективности обмена текстами и графическими изображениями уровень представления может оказывать услуги по сжатию/распаковке информации. К функциям уровня представления относится также кодирование графических изображений, аудио и видео в соответствии с различными стандартами, например JPEG, MPEG, TIFF. [12]

7. Прикладной уровень — уровень, на котором функционируют приложения. Также к этому уровню относится большая часть известных протоколов передачи данных, таких как HTTP, FTP, SMTP, DNS.

### 1.1.2 TCP/IP

В отличии от OSI, архитектура TCP/IP состоит из четырёх уровней:

1. Канальный уровень можно считать объединением физического и канального уровней модели OSI.

2. Межсетевой уровень, или уровень IP, соответствует сетевому уровню модели OSI. На этом уровне вводится протокол IP с дополнительным протоколом ICMP (Internet Control Message Protocol, интернет протокол управления сообщений), использующимся, например, в утилите traceroute.

3. Транспортный уровень объединяет в себе транспортный и сеансовый уровни модели OSI. Протоколы этого уровня будут рассмотрены в следующем параграфе. Далее в работе сеансовый уровень будет выделяться в рамках модели TCP/IP между транспортным и прикладным уровнями.

4. Прикладной уровень объединяет в себе уровень представления и прикладной уровень модели OSI.

В сети Интернет используются две версии протокола IP: IPv4 и IPv6. В начале восьмидесятых — на момент изобретения IPv4 — считалось, что Интернет будут использовать лишь университеты, правительство, крупные компании, поэтому создатели сети посчитали, что $2^{32}$ (более четырёх миллиардов, на деле в адресации участвует примерно на четверть меньше) адресов должно хватить. Однако уже через десять лет стало ясно, что четырёх миллиардов адресов мало, и появилась потребность расширять сеть. С разницей в два года появились два решения: Network Address Translator (NAT, май 1994-го) и IPv6 (декабрь 1995-го).

NATs создали новую проблему: предполагалось, что каждая машина, подключённая к сети, должна иметь возможность связаться с любой другой машиной сети напрямую; NATs же логически ввели концепцию локальных сетей в рамках сети Интернет и, как следствие, публичные и приватные адреса. Теперь можно было выдавать адрес не каждому устройству, а целой группе устройств. Более того, публичные адреса можно экономить и другим способом: при отключении локальной сети от глобальной, её публичный адрес передавался другой сети — динамическая адресация. Однако теперь пропала возможность прямой связи двух хостов, что затрудняет разработку peer-to-peer приложений. Более подробно эта проблема будет разобрана в 2.3.4.

IPv6 имеет уже $2^{128}$ адресов, что решает проблему нехватки адресов (и позволяет хорошо работать peer-to-peer приложениям), однако имеет свои недостатки[1].

## 1.2 Транспортные протоколы

В Интернете нашли своё применение два основных протокола транспортного уровня, один из которых требует установления соединения, другой — нет. Эти протоколы дополняют друг друга. Протоколом без установления соединения является UDP, не делающий практически ничего, кроме отправки пакетов между приложениями, позволяя последним надстраивать свои собственные протоколы. Протоколом с установлением соединения является TCP, обеспечивающий надежность сети, выполняя повторную передачу данных, а также осуществляет управление потоком данных и контроль перегрузки — и все это от лица приложений, которые его используют.

В следующих разделах мы рассмотрим TCP и UDP, а также протокол QUIC, формально не считающийся протоколом транспортного уровня, так как является надстройкой над UDP.

### 1.2.1 TCP

Протокол TCP (Transmission Control Protocol — протокол управления передачей) был специально разработан для обеспечения надежного сквозного байтового потока по ненадежной сети. Объединенная сеть отличается от отдельной сети тем, что её различные участки могут обладать сильно различающейся топологией, пропускной способностью, значениями времени задержки, размерами пакетов и другими параметрами. При разработке TCP основное внимание уделялось способности протокола адаптироваться к свойствам объединенной сети и отказоустойчивости при возникновении различных проблем. [11]

В рамках работы нас будет интересовать лишь две характеристики протокола: способность сохранять целостность данных, то есть доставлять все пакеты и восстанавливать их последовательность, и установка логического соединения

[1] habr.com: Что такое и зачем нужен IPv6? (`shorturl.at/iSW03`)

в небезопасной сети. Для решения задачи сохранения целостности данных используются порядковые номера пакетов, а также накопительное подтверждение получения пакетов. Эту возможность обеспечивает схема двойного рукопожатия (показаны синим на рисунках 1.1, 1.2), используемая для синхронизации состояния отправителя и получателя. [11]

Безопасность соединения обеспечивает протокол TLS (Transport Layer Security). TLS использует асимметричное шифрование для аутентификации, симметричное шифрование для конфиденциальности и коды аутентичности сообщений для сохранения целостности сообщений[2]. На данный момент используются две версии: TLS1.2 (использует два цикла связи, показано бежевым на рисунке 1.1), TLS1.3 (использует один цикл связи, показано бежевым на рисунке 1.2).

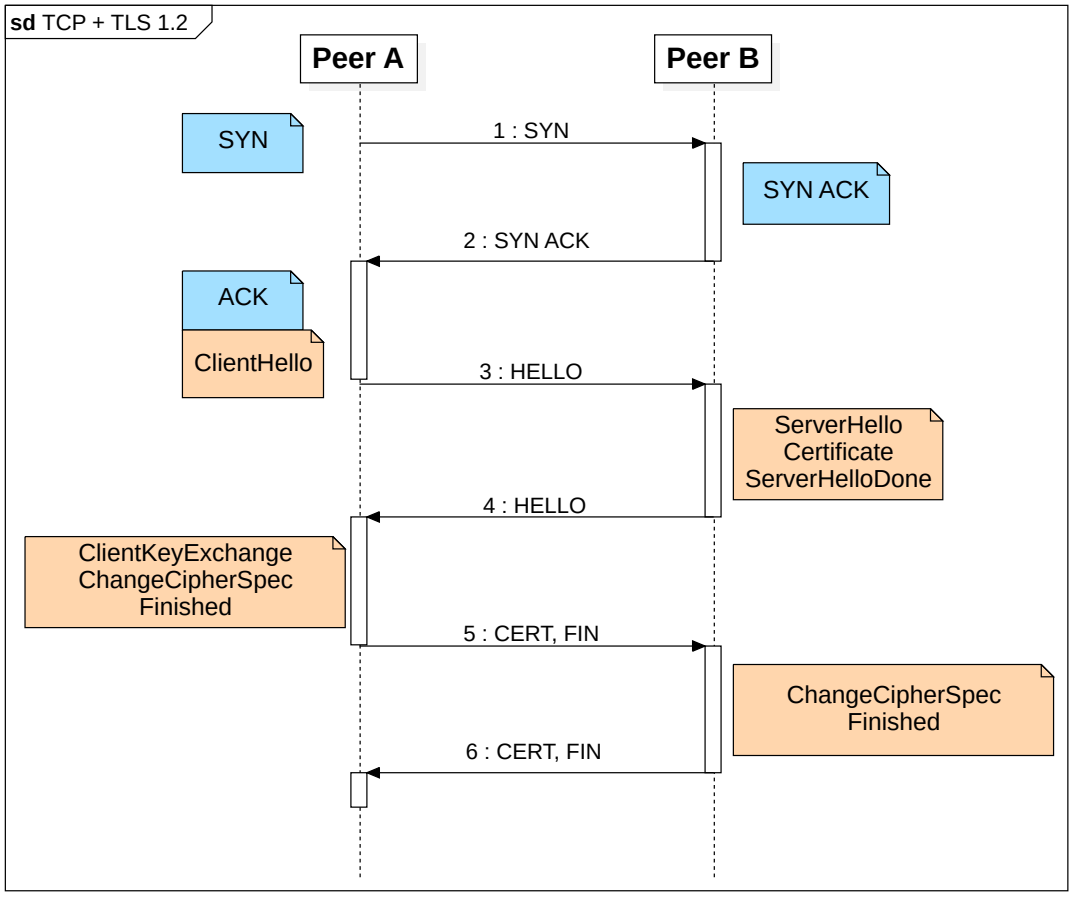



Рисунок 1.1 Схема установки соединения TCP с использованием TLS 1.2

Поверх протокола TCP построены протоколы прикладного уровня, требующие передачу данных без потерь, например FTP, SMTP и HTTP.

[2] https://ru.wikipedia.org/wiki/TLS

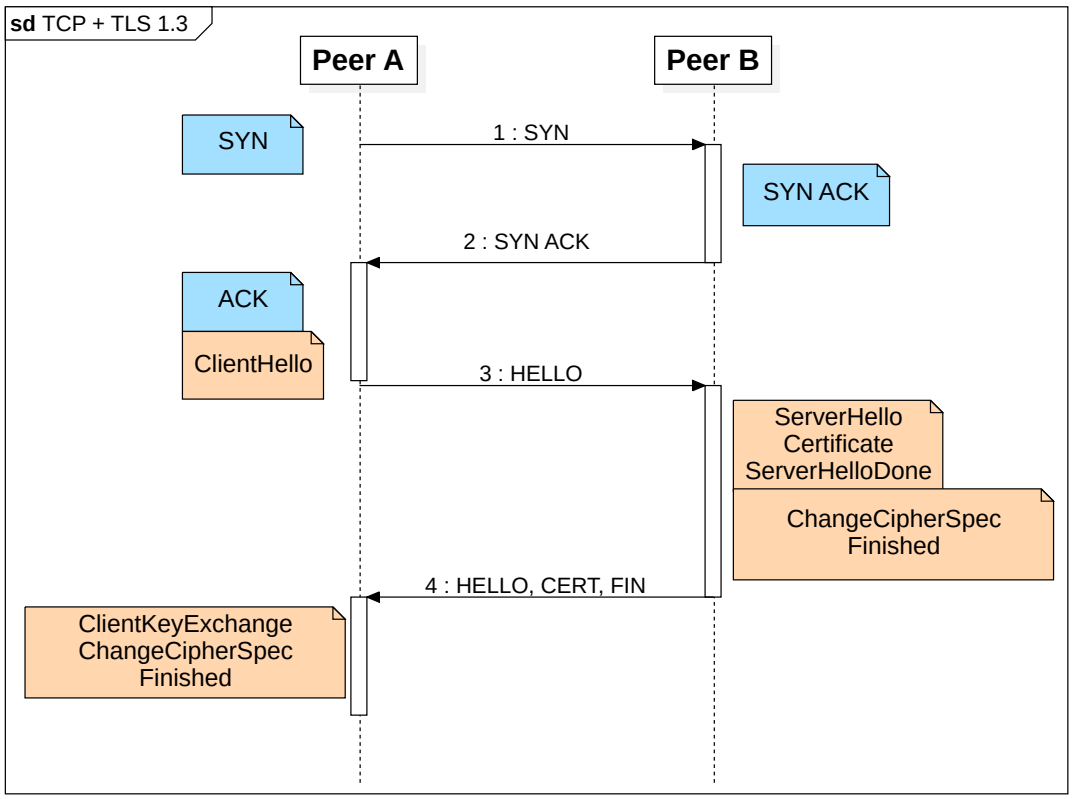



Рисунок 1.2 Схема установки соединения TCP с использованием TLS 1.3

### 1.2.2 UDP

В отличие от TCP, в UDP не предусмотрен контроль доставки пакетов. По сути происходит слепая пересылка данных без подтверждения от получателя. UDP используется в таких прикладных протоколах, как DNS (Domain Name System — система доменных имён[3]) и RPC (Remote Procedure Call – вызов удалённой процедуры[4]).

Ввиду отсутствия гарантий целостности и безопасности передачи данных, заголовок UDP содержит лишь порты отправителя и получателя, размер пакета и контрольную сумму. Протокол также не устанавливает предварительное соединение.

[3] `https://ru.wikipedia.org/wiki/DNS`

[4] `https://ru.wikipedia.org/wiki/Remote_procedure_call`

### 1.2.3 QUIC

Протокол QUIC (Quick UDP Internet Connections), как следует из названия, построен поверх протокола UDP. Протокол[5] был разработан компанией Google в 2013 году как альтернатива протоколу TCP с меньшей задержкой при установке соединения: синхронизация пакетов и настройка TLS происходит за один раунд (Рисунок 1.3). [8]

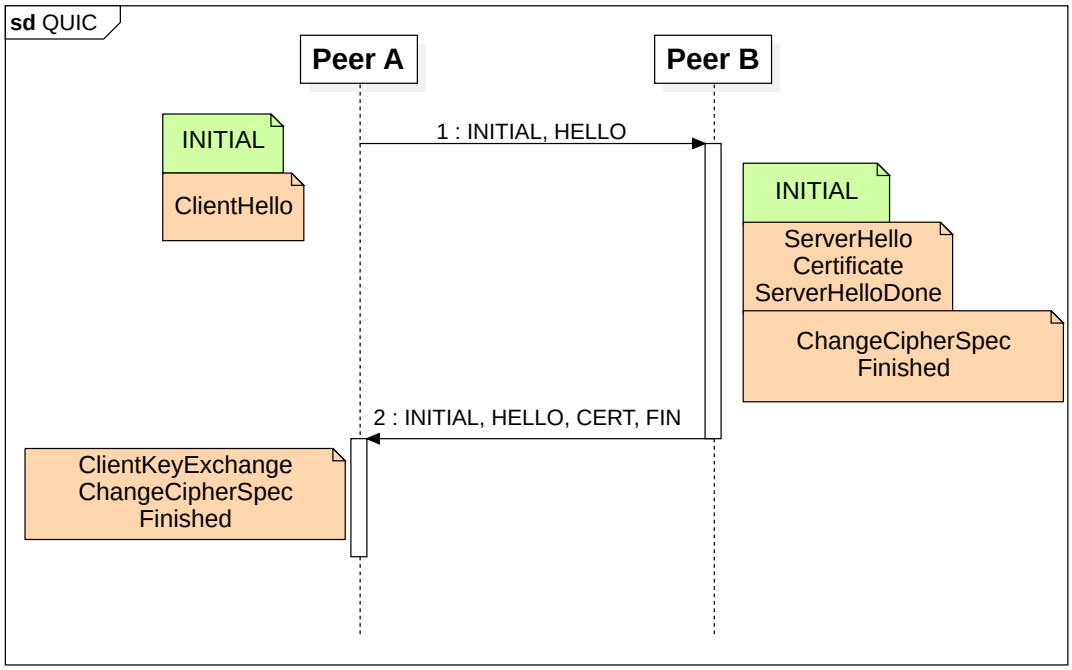



Рисунок 1.3 Схема установки соединения QUIC

UDP в связке с QUIC используется в HTTP/3 по сравнению с TCP+TLS1.3 в HTTP/2. [3]

## 1.3 Топологии сетей

Сетевая топология — это конфигурация графа, вершинам которого соответствуют конечные узлы сети (компьютеры и маршрутизаторы), а рёбрам — физические или информационные связи между вершинами.[6]

Далее будут представлены основные топологии сети. Обычно встречаются не отдельные топологии, а их комбинации.

[5] https://ru.wikipedia.org/wiki/QUIC

[6] https://en.wikipedia.org/wiki/Network_topology

### 1.3.1 Полносвязная

Топология, в которой каждое устройство непосредственно связано со всеми остальными устройствами в сети, то есть сообщения идут напрямую.

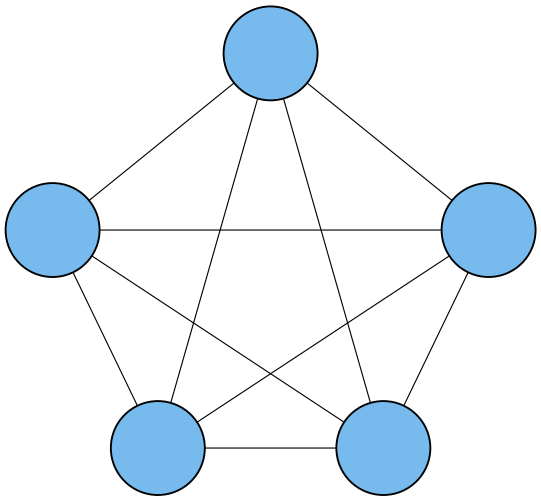


Рисунок 1.4

### 1.3.2 Шина

Топология, в которой каждое устройство подключено к общей магистрали. Сообщение пройдёт через каждое устройство, находящееся между отправителем и получателем.

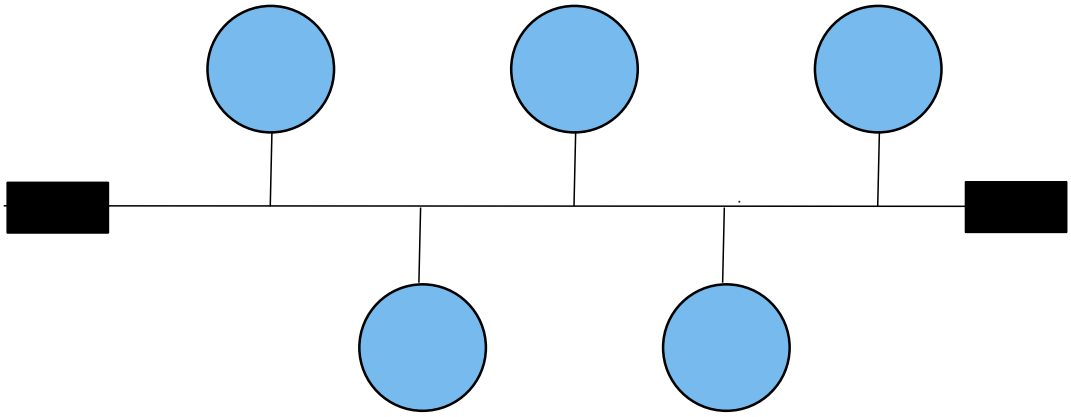


Рисунок 1.5

### 1.3.3 Звезда

Каждое устройство в сети подключено к одному хабу, через который и происходит коммуникация. Далее в работе на месте хаба будет другое устройство сети, который выполняет роль хаба-маршрутизатора.

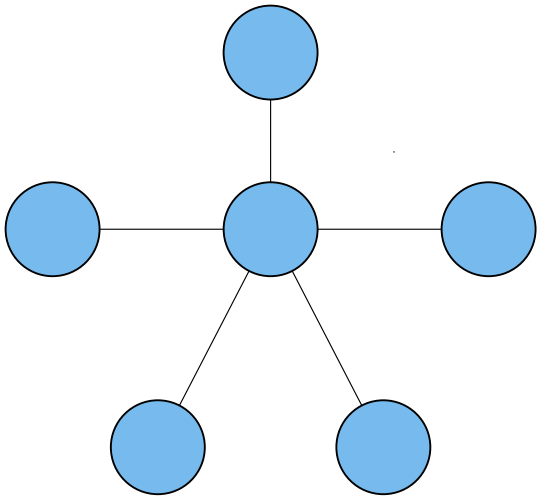


Рисунок 1.6

### 1.3.4 Кольцо

В данной топологии все устройства связаны в неразрывное кольцо. Эту топологию можно также представить как замкнутую "шину".

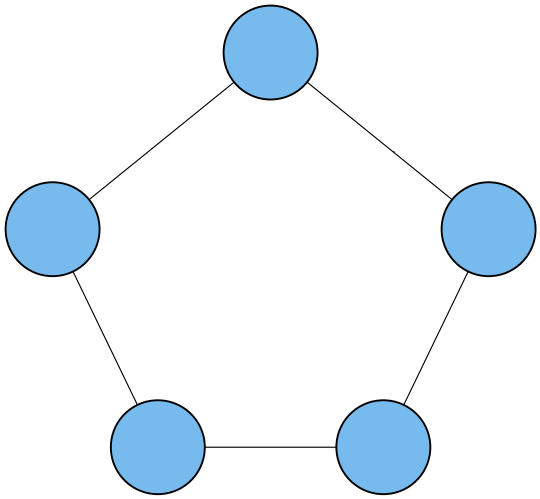


Рисунок 1.7

# ГЛАВА 2
# Yet Another P2P

В рамках этой работы разрабатывается библиотека для развёртывания peer-to-peer сети на языке программирования Rust [1, 2, 6, 9]. Принципы построения сети в библиотеке отличаются от привычных, являются новаторскими. Вводится новая концепция пользователей в сети, а именно их возможности и принцип подключения к сети (2.1). Разработан протокол передачи сообщений в сети, построенный поверх UDP (2.3.2). Также библиотека предоставляет инструментарий для обеспечения надёжного шифрования симметричным алгоритмом (2.5).

## 2.1 Пользователи сети

В современном мире у пользователей сети Интернет обычно имеется более одного устройства с доступом к Интернету. В библиотеке используются следующие концепции:

- ‘Peer‘ — уникальный пользователь сети;
- ‘Node‘ — устройство пользователя, то есть каждый пользователь может иметь более одного устройства.

Пользователь в сети идентифицируется парой ‘PeerId‘ и именем пользователя, где ‘PeerId‘ — 128-битный адрес, используемый также для обмена ключами шифрования (подробнее о построении и использовании в 2.5). Рассмотрим используемые структуры.

```
struct PeerId {
    inner: [u8; 16]
}
```

Первым байтом устанавливается число ‘111‘ (как идентивикатор сети YAP2P), следующие семь байтов используются для хранения публичного ключа (подробнее в 2.5), последние восемь байтов являются проверочной суммой хэша имени пользователя.

```rust
struct Peer {
    id: PeerId,
    name: String
}
```

‘Peer‘ конструируется при первом запуске, смена имени пользователя не предусмотрена.

Для введения структуры ‘Node‘ необходимо ввести структуру для хранения адреса устройства в сети Интернет. В структуре ‘Addr‘ предусмотрена возможность использования как IPv4, так и IPv6 (‘Option‘ — особая enum-структура, используемая в Rust для избежания значений ‘null‘[1]).

```rust
struct Addr {
    V4: Option<Ipv4Addr>,
    V6: Option<Ipv6Addr>
}
```

‘Addr‘ может одновременно хранить обе версии адреса. Это сделано с целью простоты поддержки в будущем.

```rust
struct Node {
    peer: Peer,
    device: u16,
    addrs: Mutex<Addr>
}
```

Обсудим подробнее поля структуры:

- ‘peer‘ — пользователь, которому принадлежит устройство;
- ‘device‘ — номер устройства, аналог порта в сетевом адресе;
- ‘addrs‘ — текущий сетевой адрес устройства; ‘Mutex‘ здесь используется для реализации паттерна Interior Mutability, что позволяет на лету безопасно изменять неизменяемые структуры данных.

[1] https://doc.rust-lang.org/std/option/enum.Option.html

Для более удобного и эффективного хранения всех устройств пользователя будем использовать структуру ‘Contact‘:

```
struct Contact {
    peer: Peer,
    addrs: Mutex<HashMap<u16, (Addr, u16)>>
}
```

Здесь помимо информации о пользователе ‘Peer‘ хранится коллекция, ставящая в соответствие номеру устройства его адрес ‘Addr‘ и используемый порт. Ввиду указания порта, эта структура применима лишь на конкретном уровне сети (2.2.2, 2.3.2).

## 2.2 Топология сети

Большинство недостатков peer-to-peer сетей, приведённых во введении, предлагается решать введением новой топологии. Поэтому топология отличается от наиболее часто используемых в одноранговых сетях. Так, наиболее распространённым подходом является использование Distributed Hash Tables для поиска контактов и иногда маршрутизации сообщений[2]. Альтернативным подходом является использование меш-сетей, как, например, в сети Yggdrasil[3].

В YAP2P предлагается разделить сеть на уровни. Основные уровни сети (уровни, предлагаемые в работе, однако возможно создание дополнительных уровней):

- "ContactNet" — основной уровень сети, на котором происходит общение пользователей;
- "SelfNet" — уровень, позволяющий одному пользователю иметь сразу несколько устройств;
- "MeshNet" — уровень, используемый с целью избавиться от вспомогательных серверов, а также позволяющий реализовывать различные приложения.

[2] https://jenkov.com/tutorials/p2p/index.html

[3] https://yggdrasil-network.github.io/, https://habr.com/en/articles/547250/

### 2.2.1 ContactNet

Как было описано в 2.1, каждый пользователь сети — ‘Peer‘ — может иметь несколько устройств ‘Node‘. "ContactNet" является единственным из предлагаемых уровней сети, который работает с пользователями, а не с отдельными устройствами.

На этом уровне происходит обмен сообщениями. Так как сообщение пересылается от одного устройства одного пользователя всем устройствам другого пользователя, то необходим протокол передачи данных, способный к широковещанию — пересылке одних и тех же данных сразу нескольким устройствам. Для этого был разработан протокол сеансового уровня SDP (2.3.2), построенный поверх протокола UDP.

Для обеспечения защиты пересылаемой информации предлагается использовать представленное в 2.5 решение.

"ContactNet" имеет топологию "звезда"(Рисунок 1.6), вершинами которой являются пользователи. Этот граф для каждого устройства свой, к тому же данные с этого уровня не должны напрямую попадать ни на другие уровни сети, ни к другим пользователям.

### 2.2.2 SelfNet

"SelfNet" позволяет синхронизировать состояния (список контактов, список устройств, истории сообщений) устройств одного пользователя. Для передачи сообщений на этом уровне используется модифицированный протокол SDP — SSDP (2.3.2).

"SelfNet" имеет полносвязную топологию (Рисунок 1.4) — каждое устройство хранит список остальных устройств пользователя (наложение нескольких "звёзд" даёт полносвязный граф).

### 2.2.3 MeshNet

"MeshNet" связывает все устройства сети, при этом оставляя невозможными поиск и навигацию (2.3.3).

"MeshNet" является меш-сетью, разбитой на графы "звезда"для каждого устройства. На такой архитектуре возможно создание различных приложений

с модификациями из-за особенности топологии.

Примером такого приложения может быть распределённая база данных в какой-либо компании:

- у каждого сотрудника может быть несколько устройств;
- выделенные устройства хранят часть базы данных;
- сотрудник может сделать запрос к распределённой базе;
- запрос расходится по сети, и, если у какого либо устройства находится ответ на запрос, этот ответ возвращается по сети к устройству, с которого делался запрос, либо к любому устройству сотрудника, делавшего запрос;
- данные должны дублироваться на нескольких устройствах, то есть никакая запись не должна храниться на единственном устройстве в сети.

Таким образом, "MeshNet" позволяет расширять функционал сети реализациями нужных алгоритмов.

## 2.3 Протоколы сети

В этом параграфе будут разобраны некоторые протоколы, предлагаемые библиотекой как на идеологическом уровне, так и на техническом. К идеологическим отнесём предлагаемые решения по подключению новых пользователей к сети, к техническим — протоколы сеансового уровня, работающие на различных уровнях сети. Некоторые из представленных далее описаний протоколов являются техническими заданиями к будущим модулям библиотекам и продуктам, построенным с помощью этой библиотеки.

### 2.3.1 Подключение к сети

Yet Another P2P претендует на звание friend-to-friend[4] сети, то есть одноранговой сети, в которой прямые соединения устанавливаются только с заранее выбранными пользователями. Хотя уровень "MeshNet" и устанавливает связь

[4] `https://ru.wikipedia.org/wiki/Friend-to-friend`

с незнакомыми лично пользователями, предполагается, что алгоритмы на этом уровне будут изолированы от других и будут использоваться только для служебных целей.

Первое подключение нового пользователя (PeerB, Bob) к сети осуществляется посредством другого пользователя сети (PeerA, Alice). При этом Alice предоставляет Bob конфигурационную информацию для подключения к сети, в которой также содержится информация о текущем устройстве Alice — 'Node'. Bob отправляет сообщение с запросом о подключении, в котором уже содержится идентификатор чата (2.4). Alice и Bob обмениваются информацией об активных устройствах друг друга и инициализируют чат. Также Alice передаёт Bob информацию о своих соседях на уровне "MeshNet" для первого подключения Bob к ней.

Добавление нового пользователя в контакты от добавления нового пользователя в сеть главным образом отличается отсутствием синхронизации на уровне "MeshNet": статические IP адреса являются редкостью и представляют большую ценность для сети, так как через них может устанавливаться соединение между пользователями и подключение старых пользователей при утрате соединения. То есть если не указано иное, на уровень "MeshNet" передаются данные о любых новых известных статических IP адресах.

Добавление новых пользователей в беседы и каналы (2.4) отличается от добавления в контакты следующим: новая история сообщений не инициализируется, вместо этого новый пользователь получает конфигурацию уже существующей вместе с текущим приватным ключом шифрования. Если Alice и Bob ещё не были в контактах друг у друга, то помимо конфигурации беседы/канала, происходит и создание нового диалога. Далее все пользователи чата, к которому происходило подключение, проверяют, есть ли Bob у них в контактах, и если есть, то Bob теперь может получать обновления не только от Alice, но и от других знакомых из этого чата.

### 2.3.2 Связь в сети

Для осуществления связи в сети необходим транспортный протокол, не устанавливающий соединения (подробнее в 2.3.4). Также этот протокол должен предоставлять возможность широковещания. Таким образом, не подходит про-

токол TCP. QUIC в свою очередь не получил достаточно широкого распространения и так же не поддерживает широковещание. Поэтому, для связи в сети были разработаны протоколы сеансового уровня, построенные поверх UDP. Протоколы, работающие на уровнях "ContactNet" и "SelfNet", рассмотрены далее в этом разделе.

Рассмотрим некоторую общую для всех протоколов логику, а именно заголовок пакетов. Его структура выглядит следующим образом:

```
struct Header {
    protocol_type: ProtocolType,
    packet_type: PacketType,
    length: u16,
    src_id: PeerId,
    rec_id: PeerId
}
```

- ‘protocol_type‘ — тип используемого протокола; ‘ProtocolType‘ является восьмибитовым флагом (построенным при помощи библиотеки bitflags[5]), в работе рассмотрены ‘ProtocolType::{SDP, SSDP}‘.
- ‘packet_type‘ — тип пакета, содержащий в себе флаги для определения логики обработки пакета.
- ‘length‘ — длина всего пакета в байтах, максимальное значение предполагается в 1220.
- ‘src_id‘ — идентификатор отправителя.
- ‘rec_id‘ — идентификатор получателя.

Для протоколов обмена сообщениями, разобранными далее в этом разделе, так же существует общая логика. Рассмотрим некоторые из общих структур.

[5] https://crates.io/crates/bitflags

```rust
struct Packet {
    header: Header,
    chat_sync: ChatSynchronizer,
    packet_sync: PacketSynchronizer,
    payload: Vec<u8>
}
```

Пакет является минимальной единицей отправки. Помимо заголовка ‘header‘ он содержит синхронизатор чата ‘chat_sync‘, указывающий, к какому чату относится пакет, синхронизатор пакетов ‘packet_sync‘, используемый главным образом для восстановления порядка пакетов при получении, и саму нагрузку ‘payload‘, содержащую пересылаемую информацию. Размер пакета ограничивается 1220 байтами для того, чтобы удовлетворять Maximum Transition Unit[6]; QUIC из тех же соображений ограничивает размер пакетов 1200 байтами [8]. Заголовок, синхронизатор чата и синхронизатор пакетов занимают соответственно 36, 40 и 24 байта. Таким образом, размер нагрузки ограничивается 1120 байтами. Такое ограничение используется, так как 1120 является максимальным допустимым размером, разбивающимся на блоки по 16 байтов — размер блока шифрования AES (2.5).

```rust
struct ChatSynchronizer {
    chat_id: [u8; 32],
    timestamp: u64
}
```

- ‘chat_id‘ — идентификатор чата;
- ‘timestamp‘ — время последнего сообщения из чата.

```rust
struct PacketSynchronizer {
    timestamp: u64,
    n_packets: u64,
    packet_id: u64
}
```

[6] https://en.wikipedia.org/wiki/Maximum_transmission_unit

- ‘timestamp‘ — время создания отправляемого сообщения;
- ‘n_packets‘ — количество пакетов в транзакции, возможна ситуация, когда пакет в транзакции один;
- ‘packet_id‘ — идентификатор пакета в транзакции.

Данные для отправки не всегда могут поместиться в один пакет. С целью разрешения этой проблемы вводится понятие транзакции, и вместе с ним структура ‘Transaction‘.

```
enum Transaction {
    First {
        first_packet: Packet,
        rest_of_payload: VecDeque<Vec<u8>>
    },
    Rest {
        first_packet_id: u64,
        payload: Mutex<VecDeque<Packet>>
    }
}
```

‘Transaction‘ является перечислением с двумя возможными вариантами: ‘First‘ и ‘Rest‘ для первого и остальных пакетов соответственно.

Также используются две вспомогательные структуры ‘MessageHander‘ и ‘MessageW ‘MessageWrapper‘ используется для передачи сообщений на отправку и обработки принятых сообщений. ‘MessageHander‘ является вспомогательной структурой, служащей для обработки ещё до конца не принятых транзакций.

### Symmetric Datagram Protocol

Symmetric Datagram Protocol, далее SDP, является основным протоколом связи в сети. Этот протокол построен поверх UDP. Так как протокол UDP не устанавливает соединения с удалённым устройством, SDP требует раздельной логики для отправки и получения пакетов.

‘SdpConnection‘ служит для отправки сообщений пользователю ‘Contact‘ (2.2.1). Соединение привязано к сокету ‘socket‘, который в структуре представляется указателем[7] на UDP-сокет. ‘state‘ указывает на состояние соединения ‘ConnectionState::{Receiving, Sending, Pending}‘. Рассмотрим виды пакетов (‘PacketType‘), которые отправляются структурой.

```
enum ConnectionState {
    Receiving,
    Pending,
    Sending
}

struct ConnectionWaker {
    state: ConnectionState,
    waker: Option<Waker>
}

struct SdpConnection {
    socket: Arc<UdpSocket>,
    contact: Contact,
    send_queue: Mutex<HashMap<u16, Transaction>>,
    state: Mutex<ConnectionWaker>
}
```

‘PacketType::INIT‘ — пакет инициализации, отправляемый при создании новой истории сообщений. В придачу к этому флагу устанавливается флаг, указывающий на тип чата:

- ‘Chat::OneToOne‘ ⇒ ‘PacketType::CHAT‘ — диалог;
- ‘Chat::Group‘ ⇒ ‘PacketType::CONV‘ — беседа/группа;
- ‘Chat::Channel‘ ⇒ ‘PacketType::CHAN‘ — канал.

[7] https://doc.rust-lang.org/std/sync/struct.Arc.html

‘PacketType::HI‘ похож по семантике на ‘PacketType::INIT‘, однако он используется, когда чат уже имеется, и мы хотим синхронизировать историю. Ещё одним отличием является то, что одним пакетом можно запросить синхронизацию сразу группы чатов.

Разобранные выше пакеты являются одиночными и могут повторяться без изменений. Подтверждения этих пакетов выглядят точно так же, однако дополнительно включен флаг ‘PacketType::ACK‘.

‘PacketType::ECHO‘ — служебный пакет, не требующий подтверждения. Может использоваться с целью поддержания соединения.

‘PacketType::SYN‘ и ‘PacketType::ACK‘ — пакеты отправки и подтверждения соответственно, к которым также добавляются флаги типа чата. Установки соединения требуют только эти типы пакетов. Рассмотрим процедуру отправки сообщений подробнее.

1. В метод отправки сообщений ‘send‘ передаются тип чата ‘chat_t‘, сообщение ‘message‘ (2.5) и синхронизатор чата ‘chat_sync‘.

2. Проверяется состояние соединения: отправлять новое сообщение можно только тогда, когда состояние имеет значение ‘ConnectionState::Pending‘. Если соединение находится в другом состоянии, отправка невозможна.

3. Состояние соединения устанавливается в значение ‘ConnectionState::Sending‘.

4. Исходя из типа чата, выбирается тип пакета.

5. Сообщение разбивается на нагрузки пакетов длиной по 1120 байтов.

6. Генерируется синхронизатор первого пакета: ‘timestamp‘ берётся из сообщения, ‘n_packets‘ — количество пакетов, ‘packet_id‘ — случайное натуральное 64-битное число, такое, что ‘packet_id‘ + ‘n_packets‘ меньше, чем максимальное натуральное число, помещающееся в этот тип.

7. Для каждого устройства получателя создаётся транзакция ‘Transaction::First‘ с первым пакетом и всеми нагрузками, кроме первой.

8. Из ‘SdpDriver‘ вызывается функция ‘poll_send‘. Если эта функция вызывается впервые, то устанавливается ‘state.waker‘, служащий[8] для автоматического вызова функции ‘poll_send‘. [1, 6]

9. Для каждого устройства пользователя происходит отправка очередного пакета из соответствующей транзакции. Если этот пакет первый в транзакции, то после отправки изменяется его идентификатор — номер первого сообщения в транзакции. Если первый пакет подтверждён, то отправляется первый пакет из очереди, при этом пакет помечается как отправленный, после чего кладётся в конец очереди. Пакеты будут отправляться снова и снова, пока не будут подтверждены.

10. Если все транзакции завершены, то состояние соединения устанавливается ‘ConnectionState::Pending‘, а ‘state.waker‘ устанавливается в значение ‘None‘.

‘SdpDriver‘ служит для отправки и получения пакетов, их обработки и сборки сообщений пользователей из датаграмм.

```
struct SdpDriver {
    socket: Arc<UdpSocket>,
    peer_id: PeerId,
    connections: HashMap<Peer, SdpConnection>,

    sending: mpsc::Receiver<MessageWrapper>,
    sending_deque: VecDeque<MessageWrapper>,

    receiving: mpsc::Sender<MessageWrapper>,
    handling: HashMap<[u8; 32], MessageHandler>
}
```

Структура содержит в себе коллекцию соединений на каждый контакт "ContactNe
Передача сообщений драйверу для отправки и передача полученных драйвером сообщений программе осуществляется по mpsc каналам[9]. Отправляемые и получаемые сообщения "спрятаны" в специальную обёртку ‘MessageWrapper‘ —

[8] https://rust-lang.github.io/async-book/02_execution/03_wakeups.html
[9] https://docs.rs/futures/latest/futures/channel/mpsc/index.html

перечисление структур, описывающих необходимые аргументы для отправки пакетов соответствующих типов. Дополнительно, для обработки отправляемых и получаемых сообщений используется очередь для отправки ‘sending_deque‘ и коллекция получаемых сообщений ‘handling‘.

‘SdpDriver‘ необходим, так как на UDP сокет приходят датаграммы с внешних адресов без разделения по соединениям. Это разделение драйвер и осуществляет. Он обрабатывает приходящие пакеты и, если нужно, собирает из них сообщения. Так как протокол SDP предполагает подтверждение пакетов, за их обработку также отвечает драйвер.

‘SdpDriver‘ реализует трэйт ‘Future‘[10] — структура, реализующая этот трэйт может быть возвращена из асинхронной функции. При вызове функции, которая возвращает этот тип, вызывается функция ‘poll‘, реализованная на этой структуре. Данная функция возвращает состояние готовности вызываемой функции: ‘Poll::Pending‘, если результат функции ещё не готов, и ‘Poll::Ready‘, если готов. Структуру, реализующую ‘Future‘, можно "вызывать"и без функции, тогда просто будет вызвана функция ‘poll‘. Драйвер должен работать на протяжении всего выполнения программы, что достигается бесконечным циклом внутри ‘poll‘: ‘Poll::Ready‘ возвращается только в случае фатальной ошибки. Опишем действия, выполняемые на каждой итерации этого цикла.

1. Проверяем все элементы коллекции принимаемых сообщений. Если все пакеты получены, то подтверждаем остающиеся неподтверждёнными пакеты и вызываем функцию ‘handle_message‘. Если не все пакеты получены, но неподтверждённых пакетов набирается на окно подтверждения[11], отправляется пакет подтверждения набора пакетов — функция ‘ack_window‘ соответствующего соединения ‘SdpConnection‘.

2. Если на сокет пришли данные, то от них вызывается функция ‘handle_datagram‘, которая обрабатывает приходящие пакеты.

3. Проверяем, есть ли в очереди ‘sending_deque‘ сообщения, которые можно отправить. Напомним, что отправить сообщения можно только по соедине-

[10] `https://doc.rust-lang.org/std/future/trait.Future.html`

[11] Пакеты подтверждаются не по одному, а окнами. При этом окном считается набор уникальных пакетов, а не подмножество пакетов на определённом промежутке. Это возможно, так как при отправке пакеты высылаются по кругу.

ниям, которые ни отправляют, ни получают пакетов в данный момент. Если какое-то сообщение можно отправить, то оно передаётся в функцию ‘send‘ соответствующего соединения. Если транзакции удалось сформировать, то вызывается функция ‘poll_send‘ на том же соединении, которая устанавливает waker состояния соединения и отсылает первый пакет из транзакции.

4. Считываем, если это возможно, следующее сообщение из канала. Это сообщение также пробуем отправить, как в предыдущем пункте. Если сообщение отправить не удаётся, оно добавляется в очередь ‘sending_deque‘.

5. Делаем прогресс по отправке сообщений на каждом соединении, где это возможно.

Рассмотрим теперь работу функции ‘handle_message‘. Эта функция предназначена для объединения ‘SYN‘ пакетов в сообщение. Предполагается, что все пакеты получены в единственном экземпляре. Таким образом, пакеты сортируются по их номеру, а их нагрузка соединяется в один массив. Конструируется обёртка ‘MessageWrapper::Receiving‘, после чего она отправляется по каналу ‘receiving‘ в основную программу.

Работа функции ‘handle_datagram‘ несколько сложнее. Пакет содержит ‘PeerId‘ отправителя и адрес сокета, с которого пришёл этот пакет. ‘handle_datagram‘ определяет соединение, соответствующее отправителю, и проверяет наличие адреса отправки пакета среди известных устройств отправителя. Если не удалось найти подключение к отправителю или адрес среди адресов отправителя, то обработка пакета невозможна. В случае, если пришедший пакет имеет один из типов ‘PacketType::{INIT, HI, ACK_INIT, ACK_HI}‘, то функция конструирует соответствующие обёртки и отправляет их по каналу ‘receiving‘ в основную часть программы, так как нагрузка этих пакетов не должна превосходить размера одного пакета. Разберём поведение функции при обработке других видов пакетов.

- ‘PacketType::SYN‘ — обычное сообщение пользователя. Определяется соответствующее адресу отправителя подключение. По идентификатору чата проверяется, начат ли приём пакетов для транзакции на этом чате. Если

приём пакетов начат, то нагрузка полученного пакета просто добавляется в список вместе со своим номером, добавляемым также в список пакетов, получение которых ещё не было подтверждено (см. 1-ый пункт работы функции ‘poll‘). Если данный пакет является первым в транзакции, то создаётся новая обёртка сообщения, которая добавляется в коллекцию ‘handling‘, а состояние подключения к отправителю теперь имеет статус ‘ConnectionState::Receiving‘, что блокирует отправку.

- ‘PacketType::ACK_SYN‘ — пакет подтверждения первого пакета транзакции. Пакет из ‘Transaction::First‘ не требует подтверждения, вместо этого вызывается функция ‘construct_rest‘, которая для определённого устройства отправителя пакета подтверждения и номера первого пакета (меняется при каждой отправке). ‘construct_rest‘ генерирует оставшиеся пакеты транзакции ‘Transaction::Rest‘, исходя из номера подтверждённого первого пакета.

- ‘PacketType::ACK‘ — пакет подтверждения окна пакетов транзакции. Подтверждённые пакеты удаляются из транзакции и больше не будут отправлены в функции ‘poll_send‘.

В случае получения пакетов других типов, возвращается ошибка, так как пакеты других типов не предусмотрены протоколом.

Таким образом, Symmetric Datagram Protocol обеспечивает отправку сообщений сразу всем известным устройствам получателя, при этом все пакеты подтверждаются.

### Self Symmetric Datagram Protocol

Self Symmetric Datagram Protocol (SSDP) является адаптацией протокола SDP для уровня "SelfNet". SSDP предназначен для синхронизации состояний устройств пользователя, для чего вводится структура ‘SelfSynchronizer‘.

```
struct SelfSynchronizer {
    other_nodes: HashMap<u16, (Addr, u16)>,
    contacts: Vec<Contact>,
    chat_syncs: ChatSynchronizers
}
```

‘SdpSelfConnection‘ вместо поля контакта типа ‘Contact‘ содержит адреса других устройств текущего пользователя.

```
struct SdpSelfConnection {
    socket: Arc<UdpSocket>,
    peer_id: PeerId,
    other_nodes: Mutex<HashMap<u16, (Addr, u16)>>,
    send_queue: Mutex<HashMap<u16, Transaction>>,
    state: Mutex<ConnectionWaker>
}
```

‘SdpSelfDriver‘ вместо коллекции подключений содержит лишь одно. Теперь не нужно проверять наличия подключения, а только адрес устройства.

```
struct SdpSelfDriver {
    socket: Arc<UdpSocket>,
    peer_id: PeerId,
    connection: SdpSelfConnection,

    sending: mpsc::Receiver<MessageWrapper>,
    sending_deque: VecDeque<MessageWrapper>,

    receiving: mpsc::Sender<MessageWrapper>,
    handling: HashMap<[u8; 32], MessageHandler>,
}
```

Обсудим некоторые особенности протокола:

- ‘src_id‘ заголовка является отправитель синхронизируемого сообщения, то есть это не всегда ‘PeerId‘ текущего пользователя.

- Для SSDP не предусмотрены пакеты типа ‘PacketType::INIT‘.

- При синхронизации происходит следующее. Строится синхронизатор ‘Self-Synchronizer‘, содержащий информацию об адресах других устройств пользователя ‘other_nodes‘, список контактов ‘contacts‘ и синхронизаторы чатов ‘chat_syncs‘. Этот синхронизатор отправляется остальным устройствам пользователя в пакете ‘PacketType::HI_INIT‘, на что приходят ответы типа ‘PacketType::ACK_HI_INIT‘, содержащие синхронизатор состояния, отличающийся от отправленного. Далее каждому устройству отправляется индивидуальный синхронизатор ‘PacketType::HI‘, после чего начинается синхронизация историй сообщений (адреса устройств и контакты были синхронизированы пакетом ‘PacketType::ACK_HI_INIT‘).

### 2.3.3 Псевдонавигация в сети

Уровень "MeshNet" представляет собой ненаправленный граф, в который входят все устройства (‘Node‘) сети. На таком графе возможна навигация, то есть поиск пути от одной вершины к другой. Однако, архитектура уровня сети делает это невозможным по следующим причинам.

- Имеет смысл искать не определённое устройство пользователя, а любые его устройства.

- Граф децентрализован, то есть не хранится на одном устройстве. Децентрализация достигается путём разделения на накладывающиеся графы — на каждом устройстве хранятся только его соседи.

- Поиск на графе осложняется ещё и тем, что граф может перестроиться быстрее, чем алгоритм завершит работу.

Такая архитектура обеспечивает анонимность в сети. Однако для сохранения звания friend-to-friend сети протоколы, реализуемые на этом уровне, должны сохранять изоляцию от других уровней, если пользователем не разрешено иное.

### 2.3.4 NAT Traversal

NAT (Network Address Translator) — технология, позволяющая устройствам, находящимся в локальной сети, использовать один[12] публичный адрес в сети Интернет. Основной функцией NAT является обеспечение исходящих подключений. При прохождении пакета через NAT в заголовке IP приватный IP заменяется на публичный IP, а в заголовке транспортного протокола порт с локальной машины заменяется на публичный порт. Эта замена не позволяет напрямую подключаться из сети Интернет к машине из локальной сети.

Существуют следующие виды NAT [4, 7].

- Full Cone — паре (локальный адрес, порт) ставится в соответствие единственная пара (публичный адрес, порт), при этом входящие пакеты могут приниматься от любой машины из внешней сети.
- Restricted Cone — паре (локальный адрес, порт) ставится в соответствие пара (публичный адрес, порт), при этом входящие пакеты могут приниматься только от адреса, которому отправлялись пакеты.
- Port Restricted Cone — паре (локальный адрес, порт) ставится в соответствие пара (публичный адрес, порт), при этом входящие пакеты могут приниматься только от адреса и порта, которому отправлялись пакеты.
- Symmetric — паре (локальный адрес, порт) ставится в соответствие уникальная случайная пара (публичный адрес, порт), при этом входящие пакеты могут приниматься только от адреса и порта, которому отправлялись пакеты, и инициализация соединения извне невозможна.

#### UDP Hole Punching

UDP Hole Punching, или пробрасывание UDP портов, — способ преодоления NAT при помощи служебного сервера. В YAP2P такими служебными серверами являются устройства пользователей со статическими IP адресами, владельцы которого разрешают их использовать. Разберём подробнее алгоритм UDP Hole Punching [4, 5, 10].

[12]Так как публичный адрес используется не постоянно, при переподключении к сети публичный адрес может изменяться.

Смысл алгоритма состоит в обмане NAT. Пусть у нас есть два устройства Node1 и Node2 со своими приватными и публичными адресами и портами. У Node1 и Node2 есть одно или более знакомое обоим устройство Node0 с публичным IP адресом, используемое как сервер. Упрощённый алгоритм для YAP2P состоит в следующем.

0. Node0 знает приватные и публичные адреса и порты (в случае YAP2P по несколько портов для разных уровней, при этом порт для "SelfNet"передаётся только устройствам того же пользователя) устройств Node1 и Node2.

1. Node1 обращается к Node0 с целью связаться с Node2 на определённом уровне сети.

2. Node0 сообщает Node1 публичные и приватные адреса и порты Node2, а Node2 — Node1.

3. Node1 начинает отправку сообщений на оба[13] адреса Node2, Node2 — Node1 (‘PacketType::HI‘).

4. Если установится соединение по приватным адресам — Node1 и Node2 находятся в одной локальной сети, то есть под одним NAT. Если по публичным адресам, то Node1 и Node2 находятся в разных локальных сетях (за разными NAT), либо одно или оба являются устройствами со статическими адресами, которые не разрешено использовать как сервера.

5. После установления соединения можно начинать передачу основных сообщений.

Очевидным минусом такого подхода является необходимость наличия общих знакомых серверов, при этом минимум один из этих серверов должен быть активным в текущий момент времени. Также, пока сеть является недостаточно большой, нужны служебные сервера, которые позволят сети вырасти. Рассмотрим решение, которое позволяет избегать использования серверов.

[13]Привычный и приватный адреса и порты.

### Birthday Attack

Существует способ[14] связи устройств[15], находящихся за NAT, без использования служебных серверов. Этот способ требует большого числа исходящих пакетов, а также адреса должны быть известны. То есть способ позволяет угадать лишь порты.

Положим, Node1 знает адрес Node2, тогда нам остаётся лишь угадать, какой из 65,535 портов используется для соединения. Даже при скорости отправки в 100 пакетов в секунду понадобится более 10 минут на угадывание порта.

Здесь на помощь приходит парадокс дней рождения[16]. Так, пусть теперь у устройства Node2 будет открыто, например, 256 портов, а Node1 будет пытаться угадать один из них. Таким образом, Node1 угадает нужный порт с вероятностью 99.9% за 2048 попыток.

Важное замечание: Symmetric NATs всё так же остаются непреступными.

## 2.4 Коммуникация в сети

В этом разделе опишем виды коммуникации на уровне "ContactNet". YAP2P рассматривает пересылку любой информации на данном уровне как пересылку сообщений и предоставляет примитивы для хранения истории сообщений. Далее в этом разделе будем рассматривать YAP2P как библиотеку для разработки peer-to-peer мессенджеров.

Напомним, что на этом уровне происходит основное общение в сети. Выделим следующие виды общения людей: "диалог" — общение двух людей, "канал" — один человек (группа людей) передаёт информацию, а остальные люди эту информацию потребляют, и "беседа" — равноправное общение группы людей. В случае клиент-серверной архитектуры сложностей никаких нет: вся информация хранится на сервере, где и происходит её дополнительная обработка, такая как синхронизация истории сообщений. В одноранговых сетях, ввиду отсутствия серверов, проблема хранения и синхронизации сообщений ложится на устройства участников чата. Проблема хранения сообщений решается автоматическим

[14] `https://tailscale.com/blog/how-nat-traversal-works#the-benefits-of-birthdays`

[15] Будем использовать обозначения из 2.3.4.

[16] `https://shorturl.at/ah146`

удалением старых сообщений и ограничением истории чата максимальным количеством сообщений. Проблема передачи сообщений на сеансовом уровне решается протоколом связи в сети (Symmetric Datagram Protocol (2.3.2)), однако их синхронизация происходит на уровень выше.

Рассмотрим основные структуры, связанные с общением пользователей.

Как было упомянуто выше, выделяются три вида чатов.

```
enum Chat {
    OneToOne,
    Channel,
    Group,
}
```

- OneToOne — диалог, общение двух пользователей;
- Channel — лекционная модель общения;
- Group — беседа группа пользователей.

В ‘Chat::OneToOne‘ участвуют двое пользователей, следовательно, сообщения просто передаются от одного к другому. В случае ‘Chat::Channel‘ и ‘Chat::Group‘ получать сообщения от одного пользователя, через которого осуществлялось подключение к чату, не удобно, так как этот пользователь может быть отключен от сети долгое время, либо просто покинуть чат. В качестве решения данной проблемы предлагается отправлять в чат служебное сообщение с ‘PeerId‘ нового участника чата. Каждый из участников чата проверяет, знает ли он данного пользователя, и, если знает, уведомляет его об этом.

Единицей общения является сообщение, основными характеристиками которого являются отправитель, время отправки и сами данные:

```
struct Message {
    sender: Peer,
    timestamp: u64,
    key: [u8; 32],
    is_encrypted: bool,
    data: Vec<u8>
}
```

Сообщение также опционально шифруются при пересылке, однако ключи шифрования хранятся в любом случае (2.5).

История сообщений представляет собой список сообщений чата, а также характеристики чата и накладываемые на него ограничения:

```
struct History {
    chat_t: Chat,
    chat_id: [u8; 32],
    is_encrypted: bool,
    top_key: KeyChain,
    top_timestamp: Mutex<u64>,
    constraint: usize,
    soft_ttl: u64,
    hard_ttl: u64,
    messages: Mutex<Vec<Message>>,
    other_members: Mutex<Vec<Peer>>
}
```

Рассмотрим некоторые поля структур:

- ‘chat_id‘ — идентификатор чата;
- ‘is_encrypted‘ — шифруются ли сообщения при пересылке;
- ‘top_key‘ — структура для хранения и обновления ключей;
- ‘top_timestamp‘ — время последнего сообщения в чате;
- ‘constraint‘ — максимальное число сообщений в чате до истечения ‘hard_ttl‘;
- ‘soft_ttl‘ — time to live сообщений, во время которого может храниться любое количество сообщений;
- ‘hard_ttl‘ — максимальный time to live сообщений, все сообщения старше удаляются из истории сообщений;
- ‘messages‘ — все сообщения в чате;
- ‘other_members‘ — участники чата, кроме текущего пользователя.

### 2.4.1 One-to-one

Самый популярный и простой вид коммуникации: один пользователь просто отправляет сообщения другому пользователю. Пользователи заранее знакомы и знают ключи шифрования, истории инициализированы. Этот вариант является тривиальным для протокола общения в сети (2.3.2).

### 2.4.2 One-to-many

Лекционный вид коммуникации: существует так называемый администратор канала, который передаёт всю информацию в чате. У "лектора" нет проблем с синхронизацией ключей шифрования, а "слушателям" необходимо лишь получать сообщения в нужном порядке, чтобы избегать ошибок при дешифровке.

### 2.4.3 Many-to-many

Беседы натуральнее рассматривать как диалог двух и более пользователей. Однако такой подход может вызвать проблемы при защищённой передаче сообщений из-за постоянной смены ключа шифрования. Как альтернативный подход предлагается представлять беседы как объединение каналов, администраторами которых являются члены беседы.

Предложенный альтернативный подход требует либо хранения собственной истории для каждого участка беседы, либо отслеживания ключа шифрования и времени последнего сообщения каждого участника беседы. Первый вариант строится на основе описанной выше структуры, когда для второго варианта требуется модификация. Обе структуры носят одно название и в одном приложении может быть использован только одна из них. Выбрать, какую структуру использовать, можно при подключении библиотеки выбором нужной feature[17].

Разберём безопасную для синхронизации структуру ‘History‘ подробнее. Она отличается от обычной одним полем: ‘top_key‘ заменён на ‘top_encryptor‘. Новое поле является словарём, ставящим в соответствие каждому участнику беседы свой контейнер ‘TopEncryptor‘, в котором отслеживается ключ шифрования и время последнего сообщения.

[17] `https://doc.rust-lang.org/cargo/reference/features.html`

```rust
struct TopEncryptor {
    key: KeyChain,
    timestamp: u64
}
```

Структура является чисто внутренней и не является частью API библиотеки. Выбор структуры ‘History‘ происходит через условную компиляцию[18].

```rust
struct History {
    chat_t: Chat,
    chat_id: [u8; 32],
    is_encrypted: bool,
    top_encryptor: Mutex<HashMap<PeerId, TopEncryptor>>,
    top_timestamp: Mutex<u64>,
    constraint: usize,
    soft_ttl: u64,
    hard_ttl: u64,
    messages: Mutex<Vec<Message>>,
    other_members: Mutex<Vec<Peer>>
}
```

## 2.5 Безопасность в сети

При разработке протокола безопасности в сети в первую очередь внимание уделялось общению пользователей ‘OneToOne‘, то есть прямому обмену сообщений между двумя пользователями. Однако должна была быть поддержка других видов общения без существенных изменений в логике будущих приложений. Разработанный протокол также должен иметь возможность расширения в будущем.

Решением стало применение симметричного алгоритма шифрования с 256-битным ключом (по умолчанию предлагается AES-256). Обмен ключами при установке контакта между двумя пользователями происходит посредством протокола обмена ключей Diffie-Hellman. При обмене сообщениями предлагается

[18] https://doc.rust-lang.org/reference/conditional-compilation.html

постоянно обновлять ключ шифрования. Для обеспечения непрерывного синхронного обновления ключей шифрования используется технология blockchain.

Рассмотрим алгоритмы построения, обмена и обновления ключей шифрования более подробно.

### 2.5.1 Diffie-Hellman

Diffie-Hellman — криптографический протокол, позволяющий пользователям получить секретный ключ по незащищённому каналу связи.

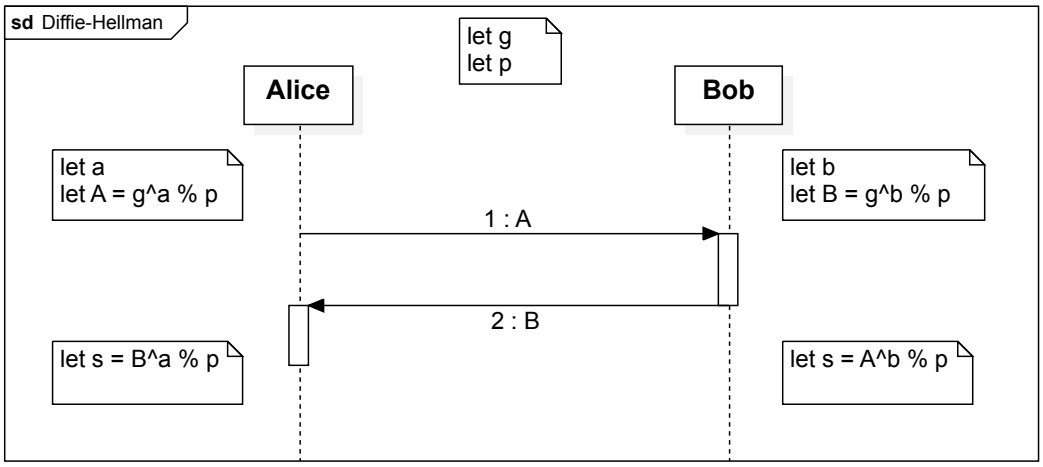



Рисунок 2.1 Алгоритм обмена ключей Diffie-Hellman

Рассмотрим подробнее структуры, используемые для этого алгоритма.

```
struct DHConfig {
    base: u64,
    modulus: u64
}
```

В алгоритме используются две общие для всех константы: ‘base‘ и ‘modulus‘ (‘g‘ и ‘p‘ на рисунке 2.1 соответственно) — основание и модуль (модуль должен быть простым числом). ‘DHConfig‘ используется для хранения этих констант.

```
struct DH {
    config: DHConfig,
    private_key: u32
}
```

‘DH‘ — основная структура, определяющая логику алгоритма Diffie-Hellman, хранящая конфигурацию алгоритма и сгенерированный приватный ключ (‘a‘ и ‘b‘ на рисунке 2.1). Эта структура позволяет получить публичный ключ, а также приватный ключ.

Публичный ключ при обмене передаётся неявно: как было описано в 2.1, ‘PeerId‘ состоит из 16 байтов, семь из которых — со второго по восьмой включительно — представляют собой 56-битный публичный ключ (то есть ‘p‘ — 56 битовое простое число).

Приватный ключ получается из ‘DH‘ и ‘Peer‘ пользователя, с которым устанавливается контакт. Так как приватный ключ также имеет длину 56 бит, он на прямую не применим в AES256. В шифровании используется хэш полученного публичного ключа, вычисляемый алгоритмом SHA256.

### 2.5.2 KeyChain

‘KeyChain‘ — блокчейн без хранения всей цепи ключей. Простой, но эффективный способ обновления ключа шифрования.

```
struct KeyChain {
    top: Mutex<[u8; 32]>
}
```

Новый ключ строится как хэш от предыдущего ключа и переданного нового ключа (32-байтного блока).

Таким образом, ключ шифрования постоянно обновляется. Кроме того, злоумышленнику для взлома сообщения необходимо хранить всю цепь сообщений с момента удачного взлома.

# ЗАКЛЮЧЕНИЕ

В ходе работы были исследованы различные подходы к разработке сетевых приложений, существующие одноранговые сети, а также проблемы, связанные с разработкой peer-to-peer приложений, существующие в современной сети Интернет.

Результатом работы стала библиотека для развёртывания peer-to-peer сети на языке программирования Rust, реализующая функционал для снятия ограничения "один пользователь — одно устройство" и позволяющая реализовывать прикладные приложения на основе предложенной новой топологии. Библиотека разработана с целью упрощения разработки и открытия новых возможностей при проектировании peer-to-peer приложений.

В перспективе ставится задача расширения API библиотеки и развития предложенных в ней концепций и протоколов.

Peer-to-Peer сети являются интересной технологией. С развитием и ростом сети Интернет их использование стало менее удобным. Однако, P2P сети являются полезным базисом для создания различных приложений. Peer-to-peer сети и Blockchain являются основой Web3.0 — будущим поколением сети Интернет — и YAP2P может стать его частью.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ


[1] Jim Blandy, Jason Orendorff, and Leonora F.S. Tindall. *Programming Rust.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2 edition, 2021.

[2] A. Chanda. *Network Programming with Rust.* Packt, 2018.

[3] P.L. Dordal. *An Introduction to Computer Networks.* Peter L Dordal, 2.0.9 edition, 8 2022.

[4] Bryan Ford, Dan Kegel, and Pyda Srisuresh. State of peer-to-peer (p2p) communication across network address translators (nats). RFC 5128, March 2008.

[5] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. 04 2005.

[6] Jon Gjengset. *RUST FOR RUSTACEANS.* William Pollock, 2022.

[7] Cong Phuoc Huynh, Ray Hunt, and Andrew McKenzie. Nat traversal techniques in peer-to-peer networks. 01 2008.

[8] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[9] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* Rust Docs, 2022.

[10] Marten Seemann, Max Inden, and Dimitris Vyzovitis. Decentralized hole punching. In *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 96–98, 2022.

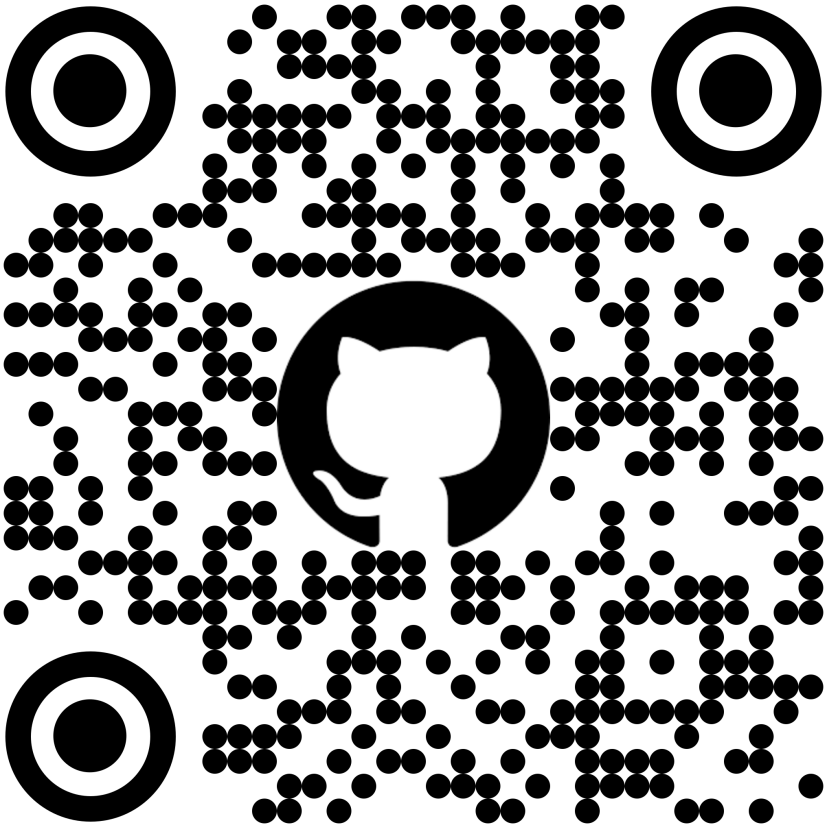
[11] Таненбаум Э. и Уэзеролл Д. *Компьютерные сети.* ООО Издательство «Питер», 5 edition, 2012.

[12] Олифер Виктор и Олифер Наталья. *Компьютерные сети. Принципы, технологии, протоколы.* ООО Издательство «Питер», 2020.

[13] Чарльз Р. Северанс. *Как работают компьютерные сети и интернет.* ДМК Пресс, 2022.

# ПРИЛОЖЕНИЕ А

## Репозиторий проекта на GitHub



YAP2P github repo qr-code

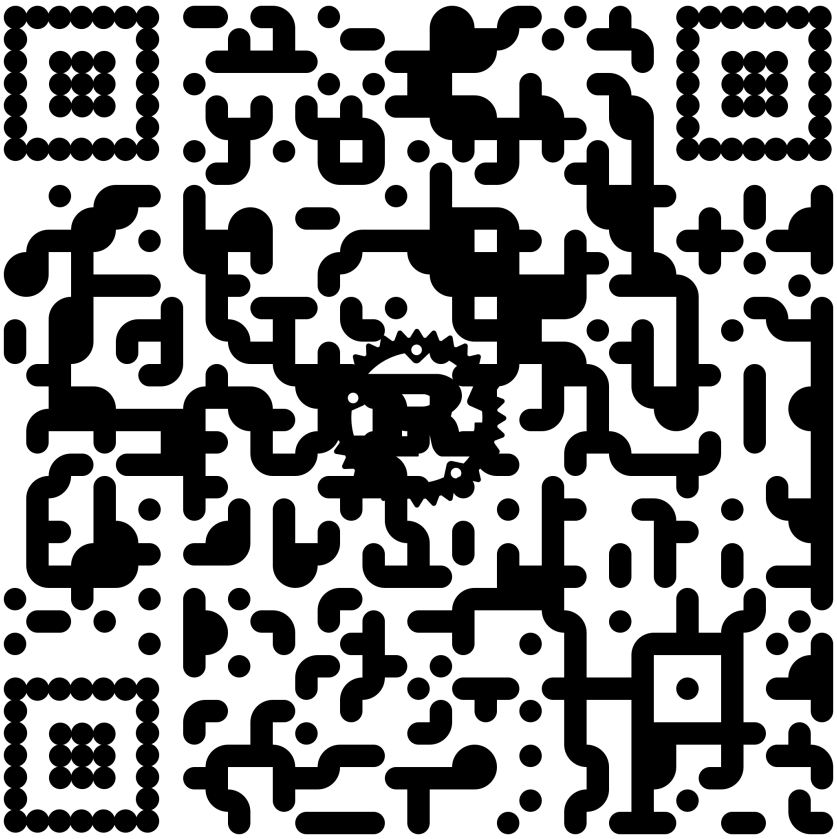
# ПРИЛОЖЕНИЕ Б

## Страница библиотеки на crates.io

YAP2P crates.io page qr-code

# ПРИЛОЖЕНИЕ В

## Документация библиотеки



YAP2P documentation qr-code