Руководство по сетевому стеку Reticulum

Это руководство призвано предоставить вам всю необходимую информацию для понимания Reticulum, построения сетей или разработки программ с его использованием, а также для участия в разработке самого Reticulum.

- Что такое Reticulum?
  - Текущее состояние
    - Эталонная реализация
    - Что предлагает Reticulum?
    - Где можно использовать Reticulum?
    - Типы интерфейсов и устройства
- Быстрый старт
  - Автономная установка Reticulum
    - Решение проблем с зависимостями и установкой
  - Попробуйте использовать программу на базе Reticulum
  - Использование включенных утилит
  - Создание сети с помощью Reticulum
  - Настройка подключения
    - Ориентация
    - Создание личной инфраструктуры
    - Комбинирование стратегий
    - Состояние сети и ответственность
    - Вклад в глобальную сеть
  - Подключение к распределенной магистрали
  - Хостинг публичных точек входа
  - Подключение экземпляров Reticulum через Интернет
  - Добавление радиоинтерфейсов
  - Создание и использование пользовательских интерфейсов
  - Разработка программы с помощью Reticulum
  - Примечания по установке для конкретных платформ
    - Android
    - ARM64
    - Debian Bookworm
    - MacOS

        - OpenWRT
        - Raspberry Pi
        - RISC-V
        - Ubuntu Lunar
        - Windows
    - Pure-Python Reticulum
- Дзен Reticulum
            - Иллюзия центра
        - Заблуждение облака
    - Децентрализация или нецентрализуемость?
    - Смерть адресу
  - Физика доверия
    - Враждебные среды
    - Шифрование — это не функция
    - Архитектуры «нулевого доверия»
  - Преимущества дефицита
    - Заблуждение о пропускной способности
    - Стоимость байта
    - Поток и время
    - Освобождение от ограничений
  - Суверенитет через инфраструктуру
    - Заблуждение операторского уровня
    - Личная инфраструктура
    - Способность отключаться
- Идентичность и кочевничество
  - Портативное существование
    - Узлы роуминга
    - Объявление о присутствии
    - Якорь в потоке
  - Этика инструмента
    - Принцип причинения вреда
    - Протокол общественного достояния
    - Сохранение человеческой свободы воли
  - Шаблоны проектирования для систем, следующих за IP
    - «Хранение и пересылка»
    - Имя —это сила
    - Интерфейс — это среда

        - Возникающие шаблоны
      - Структура независимости
        - Работа завершена
      - Открытое небо
- Программы, использующие Reticulum
  - Программы и утилиты
    - Удаленная оболочка
    - Сеть Nomad
    - Узел страницы RNS
    - Retipedia
    - Sideband
    - MeshChatX
    - MeshChat
    - Columba
    - Reticulum Релейный чат
    - RetiBBS
    - RBrowser
    - Reticulum Network Telephone
    - LXST Phone
    - LXMFy
    - LXMF Interactive Client
    - RNS FileSync
    - Micron Parser JS
    - RNMon
  - Протоколы
    - LXMF
    - LXST
    - RRC
  - Модули интерфейсов и ресурсы по подключению
- Использование Reticulum в вашей системе
  - Настройка и данные
  - Включенные утилиты
    - Утилита rnsd
    - Утилита rnstatus
    - Утилита rnid
    - Утилита rnpath
    - Утилита rnprobe

        - Утилита rncp
        - Утилита rnx
        - Утилита rnodeconf
    - Обнаружение интерфейсов
    - Удаленное управление
    - Управление черными дырами
        - Управление локальными черными дырами
        - Автоматический поиск списков
        - Публикация списков «черных дыр»
    - Улучшение конфигурации системы
        - Фиксированные имена последовательных портов
        - Reticulum как системная служба
- Понимание Reticulum
    - Мотивация
    - Цели
    - Введение и основные функции
        - Пункты назначения
        - Объявления об открытых ключах
        - Идентичности
        - Дополнительная информация
    - Транспорт Reticulum
        - Типы узлов
        - Механизм объявления в деталях
        - Достижение пункта назначения
        - Ресурсы
    - Идентификаторы сети
        - Общий обзор
        - Текущее использование
        - Будущие последствия
        - Создание и использование сетевой идентичности
    - Настройка
    - Особенности протокола
        - Приоритезация пакетов
        - Коды доступа к интерфейсам
        - Формат кабеля
        - Правила распространения объявлений
        - Криптографические примитивы

- Коммуникационное оборудование
  - Комбинирование типов оборудования
  - RNode
    - Создание RNodes
    - Поддерживаемые платы и устройства
    - Установка
    - Использование с Reticulum
  - Оборудование на базе Wi-Fi
  - Оборудование на базе Ethernet
  - Последовательные линии и устройства
  - Модемы пакетной радиосвязи
- Настройка интерфейсов
  - Пользовательские интерфейсы
  - Автоматический интерфейс
  - Магистральный интерфейс
    - Прослушиватели
    - Подключение удаленных устройств
  - Интерфейс сервера TCP
  - Интерфейс TCP-клиента
  - Интерфейс UDP
  - Интерфейс I2P
  - Интерфейс RNode LoRa
  - Мультиинтерфейс RNode
  - Последовательный интерфейс
  - Интерфейс Pipe
  - Интерфейс KISS
  - Интерфейс AX.25 KISS
  - Обнаруживаемые интерфейсы
    - Включение обнаружения
    - Параметры обнаружения
    - Режимы интерфейса
    - Вопросы безопасности
    - Пример конфигурации
  - Общие параметры интерфейса
  - Режимы интерфейса
  - Управление скоростью объявления
  - Ограничение скорости для новых адресатов

- Построение сетей
  - Концепции и обзор
    - Вводные замечания
    - Пункты назначения, а не адреса
    - Транспортные узлы и экземпляры
    - Сети без доверия
    - Гетерогенная связь
- Поддержка Reticulum
  - Пожертвования
  - Оставить отзыв
- Примеры кода
  - Минимальный
  - Объявление
  - Трансляция
  - Эхо
  - Связь
  - Идентификация
  - Запросы и ответы
  - Канал
  - Буфер
  - Передача файлов
  - Пользовательские интерфейсы
- Лицензия Reticulum

- Справочник API
  - `Reticulum`
  - `Идентификация`
  - `Пакет`
  - `Пакет`
  - `Получение пакета`
  - `Связь`
  - `Подтверждение запроса`
  - `Ресурс`
  - `Канал`
  - `MessageBase`
  - `Буфер`
  - `Читатель необработанного канала`

    - `RawChannelWriter`
    - `Транспорт`

Индексы и таблицы

- Индекс
- Страница поиска

Что такое Reticulum?

Reticulum — это сетевой стек на основе криптографии, предназначенный для построения как локальных, так и глобальных сетей с использованием легкодоступного оборудования, способный продолжать работу в неблагоприятных условиях, таких как крайне низкая пропускная способность и очень высокая задержка.

Чтобы понять основную философию и цели этой системы, прочтите «Дзен Reticulum».

Reticulum позволяет создавать глобальные сети с помощью готовых инструментов и предлагает сквозное шифрование, секретность пересылки, самонастраиваемый многоскоковый транспорт с криптографической поддержкой, эффективную адресацию,
неподдельные подтверждения получения пакетов и многое другое.

С точки зрения пользователей Reticulum позволяет создавать приложения, которые уважают и укрепляют автономию и суверенитет сообществ и отдельных лиц. Reticulum обеспечивает безопасную цифровую коммуникацию, которая не может подвергаться внешнему контролю, манипуляциям или цензуре.

Reticulum позволяет создавать как небольшие, так и потенциально глобальные сети без необходимости в иерархических или бюрократических структурах для их контроля или управления, при этом обеспечивая отдельным лицам и сообществам полный суверенитет над своими сегментами сети.

Reticulum представляет собой **полноценный сетевой стек** и не требует использования IP-протокола или более высоких уровней, хотя IP (с TCP или UDP) легко использовать в качестве базового канала для Reticulum. Поэтому туннелирование Reticulum через Интернет или частные IP-сети не представляет никаких сложностей. Reticulum построен непосредственно на криптографических принципах, что обеспечивает отказоустойчивость и стабильную работу в открытых сетях, не требующих доверия.

Не требуется никаких модулей ядра или драйверов. Reticulum может работать полностью в пользовательском пространстве и будет работать практически на любой системе, на которой работает Python 3.
Reticulum хорошо работает даже на небольших одноплатных компьютерах, таких как Pi Zero.

Текущий статус

Все основные функции протокола реализованы и работают, но, вероятно, появятся дополнения по мере изучения реальных условий использования. API и формат передачи данных можно считать завершенными и стабильными, но они могут измениться, если это будет абсолютно необходимо.

Эталонная реализация

Код на языке Python, для которого написана данная документация и который известен как «Reticulum Network Stack», является эталонной реализацией Reticulum. Протокол Reticulum полностью и авторитетно определяется данной
эта эталонная реализация и данное руководство. Его поддерживает Марк Квист (Mark Qvist), идентифицируемый идентификатором Reticulum
`<bc7291552be7a58f36152290465165c>` .

Совместимость с протоколом Reticulum определяется как полная взаимодействие и достаточная функциональная равноценность с данной эталонной реализацией. Любая конкретная реализация протокола, достигающая этого, является Reticulum. Любая, которая этого не делает, не является Reticulum.

Эталонная реализация лицензируется по лицензии Reticulum. Протокол Reticulum

был передан в общественное достояние в 2016 году.

Что предлагает Reticulum?

- Глобальная уникальная адресация и идентификация без координации
- Полностью самонастраиваемая многоскоковая маршрутизация по гетерогенным операторам
- Гибкую масштабируемость в гетерогенных топологиях
    - Reticulum может передавать данные по любой комбинации физических носителей и топологий
    - Сети с низкой пропускной способностью могут сосуществовать и взаимодействовать с крупными сетями с высокой пропускной способностью
- Анонимность инициатора: общайтесь, не раскрывая свою личность
    - Reticulum не включает адреса источника в какие-либо пакеты

- Асимметричное шифрование X25519 и подписи Ed25519 как основа всей коммуникации
    - Базовые идентификационные ключи Reticulum представляют собой наборы ключей с 512-битными кривыми Эллиптической кривой
- Форвардная секретность доступна для всех типов связи, как для отдельных пакетов, так и для каналов
- Reticulum использует следующий формат для зашифрованных токенов:
    - Временные ключи для каждого пакета и канала, полученные в результате обмена ключами ECDH на кривой Curve25519
    - AES-256 в режиме CBC с заполнением PKCS7
    - HMAC с использованием SHA256 для аутентификации
    - IV генерируются с помощью os.urandom()
- Неподдельные подтверждения доставки пакетов
- Гибкая и расширяемая система интерфейсов
    - Reticulum включает в себя большое разнообразие встроенных типов интерфейсов
    - Возможность загрузки и использования пользовательских типов интерфейсов или типов, предоставленных сообществом
    - Легко создавайте собственные настраиваемые интерфейсы для связи по любым каналам
- Аутентификация и сегментация виртуальной сети на всех поддерживаемых типах интерфейсов
- Интуитивно понятный и удобный API
    - Проще и удобнее в использовании, чем API сокетов, и проще, но мощнее
    - Значительно упрощает создание распределенных и децентрализованных приложений
- Надежная и эффективная передача произвольных объемов данных
    - Reticulum может обрабатывать как несколько байтов данных, так и файлы объемом в несколько гигабайт
    - Последовательность, сжатие, координация передачи и вычисление контрольных сумм выполняются автоматически
    - API очень прост в использовании и предоставляет информацию о ходе передачи
- Легкий, гибкий и расширяемый механизм запросов/ответов
- Эффективное установление соединения
    - Общая стоимость установки зашифрованного и проверенного соединения составляет всего 3 пакета, общим объемом 297 байт
    - Низкая стоимость поддержания открытых каналов связи — всего 0,44 бита в секунду

- Надежная последовательная доставка с помощью механизмов Channel и Buffer

Где можно использовать Reticulum?

Практически на любой среде, которая может поддерживать как минимум полудуплексный канал с пропускной способностью более 5 бит в секунду и размером MTU 500 байт.
Радиостанции для передачи данных, модемы, радиостанции LoRa, последовательные линии, TNC AX.25, цифровые режимы любительского радио, ad-hoc WiFi, оптические каналы связи в свободном пространстве и аналогичные системы — все это примеры типов интерфейсов, для которых был разработан Reticulum.

В качестве примера приемопередатчика, который очень хорошо подходит для Reticulum, был разработан интерфейс с открытым исходным кодом на базе LoRa под названием RNode [https://unsigned.io/rnode]. Его можно собрать самостоятельно, преобразовав обычную плату разработчика LoRa, или приобрести в виде готового приемопередатчика у различных поставщиков.

Reticulum также можно инкапсулировать в существующие IP-сети, поэтому ничто не мешает вам использовать его через проводной Ethernet или локальную сеть Wi-Fi, где он будет работать не хуже. Фактически, одним из преимуществ Reticulum является то, насколько легко он позволяет объединять различные среды в самонастраиваемую, отказоустойчивую и зашифрованную ячеистую сеть.

Например, можно настроить Raspberry Pi, подключенный как к радиомодулю LoRa, так и к пакетному радиомодулю TNC и сети Wi-Fi. Как только интерфейсы будут добавлены, Reticulum позаботится обо всем остальном, и любое устройство в сети Wi-Fi сможет обмениваться данными с узлами на стороне LoRa и пакетной радиосвязи, и наоборот.

Типы интерфейсов и устройства

Reticulum реализует ряд обобщенных типов интерфейсов, которые охватывают все коммуникационное оборудование, на котором может работать Reticulum. Если ваше оборудование не поддерживается, легко реализовать класс интерфейса. В настоящее время Reticulum может использовать следующие устройства и средства связи:

- Любое устройство Ethernet
    - Устройства Wi-Fi

    - Проводные устройства Ethernet
    - Оптоволоконные приемопередатчики
    - Радиостанции передачи данных с портами Ethernet
- LoRa с использованием RNode [https://unsigned.io/rnode]
    - Может быть установлено на многие популярные платы LoRa [https://github.com/markqvist/rnodeconfigutil#supported-devices]
    - Можно приобрести в виде готового к использованию приемопередатчика [https://unsigned.io/rnode]
- ТНК для пакетной радиосвязи, такие как OpenModem [https://unsigned.io/openmodem]
    - Любой TNC для пакетной радиосвязи в режиме KISS
    - Идеально подходит для УКВ- и СВЧ-радио
- Любое устройство с последовательным портом
- Сеть I2P
- Сети TCP над IP
- UDP в сетях IP
- Все, к чему можно подключиться через stdio
    - Reticulum может использовать внешние программы и каналы в качестве интерфейсов
    - Это можно использовать для простого взлома виртуальных интерфейсов
    - Или для быстрого создания интерфейсов с пользовательским оборудованием
- Всё остальное с помощью пользовательских модулей интерфейсов,

написанных на Python Полный список и более подробную информацию см. в главе «Поддерживаемые интерфейсы».

Быстрое начало

Лучший способ начать работу с сетевым стеком Reticulum зависит от того, что вы хотите сделать. В этом руководстве описаны разумные пути начала работы для различных сценариев.

Автономная установка Reticulum

Если вы просто хотите установить Reticulum и связанные утилиты в системе, проще всего это сделать через менеджер пакетов `pip`:

```
pip install rns
```

Если у вас ещё не установлен pip, вы можете установить его с помощью диспетчера пакетов вашей системы, выполнив команду типа `sudo apt install python3-pip`, `sudo pamac install python-pip` или аналогичную.

Вы также можете скачать колеса релизов Reticulum с GitHub или других каналов распространения и установить их в автономном режиме с помощью `pip`:

```
pip install ./rns-1.1.2-py3-none-any.whl
```

На платформах, ограничивающих установку пользовательских пакетов через `pip`, вам, возможно, придется вручную разрешить это с помощью флага командной строки `--break-system-packages` при установке. Это фактически не повредит никакие пакеты, если только вы не устанавливали Reticulum напрямую через пакетный менеджер вашей операционной системы.

```
pip install rns --break-system-packages
```

Более подробные инструкции по установке см. в разделе «Примечания по установке для конкретных платформ».

После завершения установки может оказаться полезным ознакомиться с главой «Использование Reticulum в вашей системе».

Решение проблем с зависимостями и установкой

На некоторых платформах бинарные пакеты могут быть доступны не для всех зависимостей, и установка `с помощью pip` может завершиться с ошибкой. В таких случаях проблему обычно можно решить, установив пакеты
для вашей платформы:

```
# Debian / Ubuntu / производные sudo apt
install build-essential

# Arch / Manjaro / производные sudo
pamac install base-devel

# Fedora
sudo dnf groupinstall "Development Tools" "Development Libraries"
```

После установки базовых пакетов для разработки `pip` должен быть в состоянии скомпилировать любые недостающие зависимости из исходного кода и завершить установку даже на платформах, где нет готовых скомпилированных пакетов.

Попробуйте воспользоваться программой, созданной на базе Reticulum

Если вы просто хотите попробовать программу, созданную с помощью Reticulum, существует целый ряд различных программ, которые обеспечивают базовую связь и различные другие полезные функции даже в сетях Reticulum с крайне низкой пропускной способностью.

Использование встроенных утилит

Reticulum поставляется с набором встроенных утилит, которые упрощают управление сетью, проверку подключения и предоставление доступа к Reticulum другим программам в вашей системе.

Вы можете использовать `rnsd` для запуска Reticulum в качестве фоновой или фоновой службы, а утилиты `rnstatus`, `rnpath` и `rnprobe` — для просмотра и запроса состояния сети и подключения.

Чтобы узнать больше об этих утилитах, ознакомьтесь с главой «Использование Reticulum в вашей системе» данного руководства.

Создание сети с помощью Reticulum

Чтобы создать сеть, вам необходимо указать один или несколько *интерфейсов*, которые будет использовать Reticulum. Это делается в конфигурационном файле Reticulum, который по умолчанию находится в папке `~/.reticulum/config`. Вы можете получить пример конфигурационного файла со всеми параметрами с помощью команды `rnsd --exampleconfig`.

При первом запуске Reticulum создаст файл конфигурации по умолчанию с одним активным интерфейсом. Этот интерфейс по умолчанию использует ваши существующие сети Ethernet и Wi-Fi (если они есть) и позволяет вам общаться с другими узлами Reticulum в пределах ваших локальных доменов широковещания.

Для дальнейшего обмена данными вам потребуется добавить один или несколько интерфейсов. Конфигурация по умолчанию включает ряд примеров, начиная от использования TCP через Интернет и заканчивая интерфейсами LoRa и Packet Radio.

С Reticulum вам нужно только настроить, через какие интерфейсы вы хотите обмениваться данными. Нет необходимости настраивать адресные пространства, подсети, таблицы маршрутизации или другие вещи, к которым вы, возможно, привыкли при работе с другими типами сетей.

Как только Reticulum определит, какие интерфейсы ему следует использовать, он автоматически определит топологию сети и настроит передачу данных в любые пункты назначения, о которых
.

В ситуациях, когда у вас уже есть налаженная сеть Wi-Fi или Ethernet и много устройств, которые хотят использовать одни и те же внешние сетевые пути Reticulum (например, через LoRa), часто достаточно позволить одной системе выступать в качестве шлюза Reticulum, добавив любые внешние интерфейсы в конфигурацию этой системы, а затем включив на ней передачу данных. Любое другое
устройство в вашей локальной сети Wi-Fi сможет подключиться к этой более широкой сети Reticulum, просто используя конфигурацию по умолчанию (AutoInterface).

Возможно, примеров в файле конфигурации будет достаточно для начала работы. Если вам нужна дополнительная информация, вы можете ознакомиться с главами «Построение сетей» и
интерфейсов» данного руководства, но, что наиболее важно, начните с чтения следующего раздела, «Настройка подключения», так как он дает наиболее важное понимание того, как обеспечить надежное подключение с минимальными затратами на обслуживание.

Настройка подключения

Reticulum — это не сервис, на который вы подписываетесь, и не единая глобальная сеть, к которой вы «присоединяетесь». Это *сетевой стек* — набор инструментов для построения систем связи
, соответствующих вашим конкретным ценностям, требованиям и операционной среде. Способ подключения к другим узлам Reticulum полностью зависит от вас.

Один из самых мощных аспектов Reticulum заключается в том, что он предоставляет множество инструментов для налаживания, поддержания и оптимизации подключения. Вы можете использовать эти инструменты по отдельности или комбинировать их в сложных конфигурациях, чтобы
достижения самых разных целей.

Независимо от того, стремитесь ли вы создать полностью частную сеть с физической изоляцией для своей семьи; построить устойчивую сеть сообщества, которая выдержит
коллапса инфраструктуры; подключить как можно больше узлов по всему миру; или просто поддерживать надежное зашифрованное соединение с конкретной организацией, которая вам важна, Reticulum предоставляет механизмы для реализации всего этого.

Нет «правильного» или «неправильного» способа построить сеть Reticulum, и вам не нужно быть сетевым инженером, чтобы просто начать работу. Если информация потоки идут так, как вы задумывали, а ваши требования к конфиденциальности и безопасности соблюдены, ваша настройка — успех. Reticulum разработан так, чтобы сделать возможными самые сложные и непростые сценарии, даже когда другие сетевые технологии терпят неудачу.

Найти свой путь

Когда вы впервые начинаете использовать Reticulum, вам необходимо установить соединение с узлами, с которыми вы хотите обмениваться данными — это процесс *инициализации соединения*.

Важно

Распространенной ошибкой в современных сетях является зависимость от нескольких централизованных, жестко запрограммированных точек входа. Если каждый пользователь просто подключается к одному и тому же списку публичных IP-адресов, найденных на веб-сайте, сеть становится уязвимой, централизованной и, в конечном итоге, не способной выполнить обещания децентрализации и отказоустойчивости. Здесь на вас лежит ответственность.

Reticulum поощряет подход *органического роста*. Вместо того, чтобы полагаться на постоянные статические соединения с удаленными серверами, вы можете использовать временные соединения для начальной настройки, чтобы постоянно *обнаруживать* более релевантную или локальную инфраструктуру. После обнаружения ваша система может автоматически формировать более прочные и прямые связи с этими узлами и отбрасывать временные соединения для начальной настройки. В результате получается сеть соединений, которая является географически релевантной, отказоустойчивой и эффективной.

Можно просто добавить несколько публичных точек входа в раздел `[interfaces]` вашей конфигурации Reticulum и подключиться, но лучшим вариантом является включение обнаружения интерфейсов и либо ручной выбор релевантных локальных интерфейсов, либо включение автоматического подключения к обнаруженным интерфейсам.

В данном контексте актуальным является параметр «bootstrap_only». Это автоматизированный инструмент, позволяющий оптимизировать распределение подключений. Включив
обнаружение интерфейсов и автоматическое подключение, а также пометку интерфейса как `bootstrap_only`, вы указываете Reticulum использовать этот интерфейс в первую очередь для поиска вариантов подключения, а затем отключать его, как только будет обнаружено достаточное количество точек входа. Это помогает создать топологию сети, которая отдаёт предпочтение локальности и отказоустойчивости перед простой централизацией, вызванной использованием лишь нескольких
статических точек входа.

Хорошие места для поиска определений интерфейсов для начальной настройки подключения — это такие веб-сайты, как каталог .rns.recipes [https://directory.rns.recipes/] и rmap.world

[https://rmap.world/].

Создание личной инфраструктуры

Вам не нужен дата-центр, чтобы стать значимой частью экосистемы Reticulum. На самом деле, самые важные узлы в сети часто бывают самыми маленькими.

Мы настоятельно рекомендуем всем, в том числе и домашним пользователям, думать о создании **собственной инфраструктуры**. Не подключайте каждый телефон, планшет и компьютер в вашем доме напрямую к общедоступному интернет-шлюзу. Вместо этого
перепрофилируйте старый компьютер, Raspberry Pi или поддерживаемый маршрутизатор, чтобы он действовал как ваш собственный, личный **транспортный узел**:

- Ваш локальный транспортный узел находится у вас дома, подключен к вашему Wi-Fi и, возможно, к радиоинтерфейсу (например, RNode).
- Вы настраиваете этот узел с интерфейсом `bootstrap_only` (возможно, это TCP-туннель в более широкую сеть) и включаете обнаружение интерфейсов.
- Пока вы спите, работаете или готовите, ваш узел прослушивает сеть. Он обнаруживает других членов локального сообщества, проверяет их сетевые идентификаторы и автоматически устанавливает прямые соединения.
- Теперь ваши личные устройства подключаются к вашему *локальному* узлу, который является частью живой и динамичной локальной ячеистой сети. Ваш трафик проходит по локальным маршрутам, предоставляемым другими реальными людьми из сообщества, а не отскакивает от удаленного сервера.

Не ждите, пока другие построят сети, которые вы хотите видеть. Каждая сеть важна, возможно, даже в первую очередь те, которые поддерживают отдельные семьи и людей. Как только появится достаточное количество такой личной локальной инфраструктуры
, их прямое соединение друг с другом, без прохождения через общедоступный Интернет, станет неизбежным.

Комбинирование стратегий

Нет необходимости придерживаться одной единственной стратегии. Наиболее надежные конфигурации часто сочетают статические, динамические и обнаруживаемые интерфейсы.

- **Статические интерфейсы:** вы поддерживаете постоянный интерфейс с надежным другом или организацией, используя статическую конфигурацию.
- **Связи начальной загрузки:** вы подключаете интерфейс `bootstrap_only` к общедоступный шлюз в Интернете для поиска новых узлов, с которыми можно установить соединение, или для восстановления связи в случае сбоя других интерфейсов.
- **Локальная широкополосная связь:** вы запускаете `RNodeInterface` на общей частоте, что дает вам полностью независимый и частный широкополосный доступ как к вашей собственной сети, так и к другим узлам Reticulum по всему миру, при этом никакие «поставщики услуг» не смогут контролировать или отслеживать, как вы взаимодействуете с людьми.

Комбинируя эти методы, вы создаете систему, защищенную от единичных точек отказа, адаптируемую к меняющимся условиям сети и лучше интегрированную в вашу физическую и социальную реальность.

Состояние сети и ответственность

Участвуя в более широких сетях, которые вы открываете и создаете, вы неизбежно столкнетесь с узлами, которые настроены неверно, являются вредоносными или просто неработающими. Чтобы защитить свои ресурсы и ресурсы ваших локальных узлов, вы можете использовать систему управления «черными дырами».

Независимо от того, блокируете ли вы спамерский аккаунт вручную или подписываетесь на
список «черных дыр», который ведет надежный сетевой идентификатор, эти инструменты помогают
обеспечить, чтобы *ваши* транспортные ресурсы использовались для того, что *вы* считаете легитимной коммуникацией. Это поддерживает эффективность вашего локального сегмента и способствует благополучию сети в целом

Вклад в глобальную сеть Ret

Если у вас есть возможность разместить стабильный узел с публичным IP-адресом, рассмотрите возможность стать публичной точкой входа. Опубликовав свой интерфейс как обнаруживаемый, вы предоставляете потенциальную точку подключения для других, помогая сети расти и охватывать новые регионы.

Рекомендации по правильной настройке публичной точки входа см. в разделе «Размещение публичных точек входа».

Подключение к распределенной магистрали

Глобальная распределенная магистральная сеть транспортных узлов Reticulum поддерживается волонтерами со всего мира. Эта сеть представляет собой гетерогенное собрание как публичных, так и частных узлов, образующих нескоординированную, добровольную межсетевую магистраль, которая в настоящее время обеспечивает глобальные транспортные и межсетевые возможности для Reticulum.

В качестве хорошей отправной точки вы можете найти определения интерфейсов для подключения ваших собственных сетей к этой магистрали на таких веб-сайтах, как
каталоге .rns.recipes [https://directory.rns.recipes/] и rmap.world [https://rmap.world/].

Совет

Не полагайтесь только на одно подключение к распределенной магистрали для повседневного использования. Гораздо лучше настроить несколько резервных подключений и включить опции обнаружения интерфейсов, чтобы ваши узлы могли непрерывно находить возможности для пиринга по мере развития сети. Ознакомьтесь с разделом «Настройка подключения» (Bootstrapping Connectivity), чтобы понять эти опции.

Хостинг общедоступных точек входа

Если вы хотите помочь в создании мощной глобальной магистрали межсетевого взаимодействия, вы можете разместить в Интернете публичную (или частную) точку входа в сеть Reticulum. В этом разделе приведены некоторые полезные рекомендации. После настройки публичной точки входа рекомендуется сделать её обнаруживаемой в сети Reticulum.

Вам понадобится физический или виртуальный компьютер с публичным IP-адресом, доступным для других устройств в Интернете.

Самый эффективный и производительный способ разместить точку входа, поддерживающую подключение множества пользователей, — это использовать `BackboneInterface` . Этот тип интерфейса полностью совместим с типами `TCPClientInterface` и `TCPServerInterface`, но работает гораздо быстрее и использует меньше системных
ресурсов, что позволяет вашему устройству обрабатывать тысячи подключений даже на небольших системах

Также важно перевести подключаемый интерфейс в режим шлюза, так как это значительно сократит время сходимости сети и ускорит определение маршрута для всех, кто подключается к вашему входу.

```
# В этом примере показан магистральный интерфейс
#, настроенный для работы в качестве шлюза, через который
пользователи # могут подключаться к публичной или частной сети

[[Public Gateway]]
  type = BackboneInterface enabled =
  yes
  режим = шлюз слушать_на
  = 0.0.0.0
  port = 4242

  # На общедоступных интерфейсах может быть # целесообразно
  настроить разумные целевые значения скорости
  #. announce_rate_target = 3600
  штраф_за_частоту_объявлений = 3600
  announce_rate_grace = 12
```

Если же вы хотите создать частную точку входа из Интернета, вы можете использовать опции IFAC name и passphrase, чтобы защитить свой интерфейс с помощью сетевого имени и парольной фразы.

```
# Частная точка входа, требующая предварительно согласованного #
имени сети и парольной фразы для подключения.

[[Частный шлюз]]
  type = BackboneInterface enabled =
  yes
  режим = шлюз listen_on
  = 0.0.0.0
```

```
порт = 4242
имя_сети = private_ret пароль =
2owjajquafIanPecAc
```

Если вы размещаете точку входа на операционной системе, которая не поддерживает `BackboneInterface`, вы можете вместо этого использовать `TCPServerInterface`, хотя его производительность будет ниже.

Подключение экземпляров Reticulum через Интернет

В настоящее время Reticulum предлагает три интерфейса, подходящих для подключения экземпляров через Интернет: Backbone, TCP и I2P. Каждый интерфейс предлагает свой набор возможностей, и пользователям Reticulum следует тщательно выбирать интерфейс, наиболее подходящий для их нужд.

`TCPServerInterface` позволяет пользователям размещать экземпляр, доступный по TCP/IP. Этот метод, как правило, быстрее, имеет меньшую задержку и более энергоэффективен, чем использование `I2PInterface`, однако он также раскрывает больше данных о хосте сервера.

`BackboneInterface` — это очень быстрый и эффективный тип интерфейса , доступный в операционных системах POSIX и предназначенный для одновременной обработки тысяч соединений с низкими затратами памяти, вычислительных ресурсов и ввода-вывода. Он полностью совместим с типами интерфейсов на основе TCP.

TCP-соединения раскрывают IP-адреса как вашего экземпляра, так и сервера любому, кто может проанализировать соединение. Кто-то может использовать эту информацию для определения вашего местоположения или личности. Злоумышленники, анализирующие ваши пакеты, могут записывать метаданные пакетов, такие как время передачи и размер пакета. Несмотря на то что Reticulum шифрует трафик, протокол TCP этого не делает, поэтому злоумышленник может с помощью анализа пакетов узнать, что на системе запущен Reticulum, а также какие другие IP-адреса подключаются к ней.
Для размещения общедоступного экземпляра через TCP также требуется общедоступный доступный IP-адрес, чего большинство интернет-соединений больше не предлагает.

`I2PInterface` маршрутизирует сообщения через Invisible Internet Protocol (I2P) [https://geti2p.net/en/]. Чтобы использовать этот интерфейс, пользователи должны также запускать демон I2P параллельно с `rnsd`. Для постоянно работающих узлов I2P рекомендуется использовать i2pd [https://i2pd.website/].

По умолчанию I2P шифрует и смешивает весь трафик, отправляемый через Интернет, и скрыть IP-адреса экземпляров Reticulum как у отправителя, так и у получателя. Работа узла I2P также предполагает ретрансляцию зашифрованных пакетов других пользователей I2P, что
требует дополнительной пропускной способности и вычислительных ресурсов, но также значительно затрудняет атаки по времени и другие формы глубокого анализа пакетов.

I2P также позволяет пользователям размещать глобально доступные экземпляры Reticulum с непубличных IP-адресов, а также за брандмауэрами и NAT.

В целом рекомендуется использовать узел I2P, если вы хотите разместить общедоступный экземпляр, сохранив при этом анонимность. Если вам важнее производительность и немного более простая настройка, используйте TCP.

Добавление радиоинтерфейсов

После того как Reticulum установлен и работает, вы можете добавить радио с помощью любого совместимого оборудования, которое у вас есть. Reticulum поддерживает широкий спектр радиооборудования, и если у вас уже есть какое-либо если оно доступно, то, скорее всего, оно будет работать с Reticulum. Информацию о настройке см. в разделе «Интерфейсы» данного руководства.

Если у вас еще нет готового приемопередатчика, вы можете легко и недорого собрать RNode — универсальный цифровой приемопередатчик дальнего действия, который легко интегрируется с Reticulum.

Чтобы собрать его самостоятельно, необходимо установить пользовательскую прошивку на поддерживаемую плату разработчика LoRa с помощью скрипта автоматической установки или веб-прошивателя. Руководство см. в главе «Коммуникационное оборудование». Если вы предпочитаете приобрести готовое устройство, вы можете обратиться к списку поставщиков.

Другие аппаратные интерфейсы на основе радиосвязи разрабатываются и предоставляются широким сообществом Reticulum. Вы можете найти больше

информации по этим темам в системах обмена информацией на базе Reticulum.

Если у вас есть коммуникационное оборудование, которое пока не поддерживается ни одним из существующих типов интерфейсов, вы можете легко написать (и, возможно, опубликовать) модуль пользовательского интерфейса, который обеспечит его совместимость с Reticulum.

Создание и использование пользовательских интерфейсов

Хотя Reticulum включает гибкий и широкий набор встроенных интерфейсов, они не охватывают все возможные типы коммуникационного оборудования, которое Reticulum потенциально может использовать для связи.

Поэтому можно легко написать собственные модули интерфейсов, которые можно загружать во время выполнения и использовать наравне с любыми встроенными типами интерфейсов.

Дополнительную информацию по данной теме, а также примеры кода, на которых можно основываться, см. в главе «Настройка интерфейсов».

Разработка программы с Reticulum

Если вы хотите разрабатывать программы, использующие Reticulum, самый простой способ начать — установить последнюю версию Reticulum с помощью pip:

```
pip install rns
```

Вышеуказанная команда установит Reticulum и зависимости, и вы будете готовы импортировать и использовать RNS в своих собственных программах. Следующим шагом, скорее всего, будет ознакомление с некоторыми примерами программ.

Весь API Reticulum описан в главе «Справочник API»
этом руководстве. Прежде чем приступить к работе, рекомендуется прочитать это руководство полностью, но, по крайней мере, начните с главы «Понимание Reticulum».

Примечания по установке для конкретных платформ

Для некоторых платформ требуется несколько иная процедура установки или существуют различные особенности, о которых стоит знать. Они перечислены здесь.

Android

Reticulum можно использовать на Android разными способами. Самый простой способ начать — использовать приложение типа Sideband [https://unsigned.io/sideband].

Для большего контроля и доступа к дополнительным функциям можно использовать Reticulum и связанные с ним программы через приложение Termux [https://termux.com/], доступное на момент написания статьи в F-droid [https://f-droid.org].

Termux — это эмулятор терминала и среда Linux для устройств на базе Android, которая включает в себя возможность использования множества различных программ и библиотек, в том числе Reticulum.

Чтобы использовать Reticulum в среде Termux, вам необходимо установить `Python` и библиотеку `python-cryptography` с помощью `pkg` — встроенного в Termux менеджера пакетов. После этого вы сможете установить Reticulum с помощью `pip`.

В Termux выполните следующее:

```
# Сначала убедитесь, что индексы и пакеты обновлены. pkg update
pkg upgrade

# Затем установите Python и библиотеку cryptography. pkg install
python python-cryptography

# Убедитесь, что pip обновлен, и установите модуль wheel. pip install wheel pip
--upgrade

# Установите Reticulum pip
install rns
```

Если по какой-то причине пакет `python-cryptography` недоступен для вашей платформы через диспетчер пакетов Termux, вы можете попробовать скомпилировать его локально на своем устройстве с помощью следующей команды:

```
# Сначала убедитесь, что индексы и пакеты обновлены. pkg update
pkg upgrade

# Затем установите зависимости для библиотеки cryptography. pkg install
python build-essential openssl libffi rust

# Убедитесь, что pip обновлен, и установите модуль wheel. pip install wheel pip
--upgrade

# Чтобы установщик мог скомпилировать модуль cryptography,
# нам нужно указать, для какой платформы мы компилируем: export
CARGO_BUILD_TARGET="aarch64-linux-android"

# Запустите процесс установки модуля криптографии.
# В зависимости от вашего устройства это может занять несколько минут,
# поскольку модуль необходимо скомпилировать локально на вашем устройстве. pip
install cryptography

# Если вышеуказанная установка прошла успешно, теперь вы можете установить #
Reticulum и любое связанное с ним программное обеспечение
pip install rns
```

Также возможно включить Reticulum в приложения, скомпилированные и распространяемые в виде APK-файлов для Android. Подробное руководство и пример исходного кода будут добавлены сюда позже. До тех пор вы можете использовать исходный код Sideband [https://github.com/markqvist/sideband] в качестве примера и отправной точки.

ARM64

На некоторых архитектурах, включая ARM64, не для всех зависимостей имеются предварительно скомпилированные бинарные файлы. На таких системах перед установкой Reticulum или программ, зависящих от Reticulum, может потребоваться установка `python3-dev` (или аналога).

```
# Установите Python и пакеты для разработки sudo apt
update
sudo apt install python3 python3-pip python3-dev
```

```
# Установите Reticulum
python3 -m pip install rns
```

После установки этих пакетов `pip` сможет локально установить все недостающие зависимости в вашей системе.

Debian Bookworm

В версиях Debian, выпущенных после апреля 2023 года, по умолчанию больше невозможно использовать `pip` для установки пакетов в вашу систему. К сожалению, вам придётся вместо этого использовать замену — команду `pipx`, которая помещает установленные пакеты в изолированную среду. Это не должно негативно повлиять на Reticulum, но не будет работать для включения и использования Reticulum в ваших собственных скриптах и программах.

```
# Установите pipx
sudo apt install pipx

# Сделать установленные программы доступными в командной строке pipx
ensurepath

# Установить Reticulum
pipx install rns
```

В качестве альтернативы вы можете восстановить нормальное поведение `pip`, создав или отредактировав файл конфигурации, расположенный в `~/.config/pip/pip.conf`, и добавив следующий раздел:

```
[global]
break-system-packages = true
```

Для однократной установки Reticulum без глобального включения опции `break-system-packages`, можно использовать следующую команду:

```
pip install rns --break-system-packages
```

Примечание

Директива `--break-system-packages` несколько вводит в заблуждение выбор слов. Установка этого параметра, конечно, не повредит никакие системные пакеты, а просто позволит устанавливать пакеты `pip` как на уровне пользователя, так и на системном уровне.
Хотя в редких случаях это *может* привести к конфликтам версий, в целом это не создает никаких проблем, особенно при установке Reticulum.

macOS

Чтобы установить Reticulum на macOS, у вас должны быть установлены Python и менеджер пакетов `pip`

.

Системы под управлением macOS могут значительно различаться по тому, предустановлен ли Python, и если да, то какая версия установлена, а также установлен ли `pip`
Также должен быть установлен и настроен менеджер пакетов. Если у вас есть сомнения, вы можете загрузить и установить Python вручную по ссылке [https://www.python.org/downloads/].

Когда Python и `pip` будут доступны в вашей системе, просто откройте окно терминала и выполните одну из следующих команд:

```
# Установите Reticulum и утилиты с помощью pip: pip3
install rns

# В некоторых версиях для установки может потребоваться
использовать флаг --break-system-packages: pip3 install
rns --break-system-packages
```

Примечание

Директива `--break-system-packages` — это несколько вводящий в заблуждение . Ее установка, конечно, не повредит никакие системные пакеты, а просто позволит устанавливать пакеты `pip` на уровне пользователя и всей системы.
Хотя в редких случаях это *может* привести к конфликтам версий, в целом это не вызывает никаких проблем, особенно при установке Reticulum.

Кроме того, некоторые комбинации версий macOS и Python требуют, чтобы вы вручную добавили каталог установленных пакетов `pip` в переменную среды *PATH*, прежде чем вы сможете использовать установленные команды в терминале. Обычно достаточно добавить следующую строку в скрипт инициализации оболочки (например,
пример `~/.zshrc`):

```
export PATH=$PATH:~/Library/Python/3.9/bin
```

Настройте версию Python и расположение скрипта инициализации оболочки в соответствии с вашей системой.

OpenWRT

На системах OpenWRT с достаточным объемом хранилища и памяти вы можете установить Reticulum и связанные утилиты с помощью менеджера пакетов *opkg* и *pip*.

Примечание

На момент выпуска данного руководства ведется работа над созданием готовых пакетов Reticulum для OpenWRT с полной настройкой, интеграцией служб и интерфейса `uci`. Дополнительную информацию можно найти в репозиториях feed-reticulum [https://github.com/gretel/feed-reticulum] и reticulum-openwrt [https://github.com/gretel/reticulum-openwrt]
.

Чтобы установить Reticulum на OpenWRT, сначала войдите в сеанс командной строки, а затем следуйте следующим инструкциям:

```
# Установите зависимости
opkg install python3 python3-pip python3-cryptography python3-pyserial

# Установите Reticulum pip
install rns

# Запустите rnsd с включенной отладочной регистрацией
rnsd -vvv
```

Примечание

Приведенные выше инструкции были проверены и протестированы исключительно на OpenWRT 21.02. Возможно, что для других версий потребуются слегка измененные команды установки или названия пакетов. Вам также понадобится достаточно свободного места в файловой системе оверлея и достаточно свободной оперативной памяти для фактического запуска Reticulum и любых связанных с ним программ и утилит.

В зависимости от конфигурации вашего устройства, вам может потребоваться настроить правила брандмауэра, чтобы Reticulum мог подключаться к вашему устройству и с него. Пока не будет готова соответствующая упаковка, вам также нужно будет вручную создать службу или скрипт запуска, чтобы Reticulum запускался автоматически при загрузке системы.

Также обратите внимание, что для работы *AutoInterface* необходимо включить локальные IPv6-адреса для всех устройств Ethernet и Wi-Fi, которые вы планируете использовать. Если команда `ip a`
показывает адрес, начинающийся с `fe80::` для данного устройства,
`AutoInterface` должен работать для этого устройства.

Raspberry Pi

В настоящее время рекомендуется использовать 64-разрядную версию ОС Raspberry Pi, если вы хотите запустить Reticulum на компьютерах Raspberry Pi, поскольку 32-разрядная
Для некоторых версий не всегда доступны пакеты для определённых зависимостей. Если Python и менеджер пакетов *pip* ещё не установлены, сначала установите их, а затем установите Reticulum с помощью *pip*.

```
# Установите зависимости
sudo apt install python3 python3-pip python3-cryptography python3-
pyserial

# Установить Reticulum
pip install rns --break-system-packages
```

Примечание

Директива `--break-system-packages` — это несколько вводящий в заблуждение . Ее установка, конечно, не повредит никакие системные пакеты, а просто позволит устанавливать пакеты `pip` как на уровне пользователя, так и на системном уровне. Хотя в редких случаях это *может* привести к конфликтам версий, обычно это не создает никаких проблем, особенно при установке Reticulum.

Хотя Reticulum можно установить и запустить на 32-разрядных версиях ОС Raspberry Pi, для этого потребуется вручную настроить и установить необходимые зависимости сборки, что не описывается в данном руководстве.

RISC-V

На некоторых архитектурах, в том числе RISC-V, не для всех зависимостей доступны предварительно скомпилированные бинарные файлы. В таких системах перед установкой Reticulum или программ, зависящих от Reticulum, может потребоваться установить `python3-dev` (или аналогичный пакет).

```
# Установите Python и пакеты для разработки sudo apt
update
sudo apt install python3 python3-pip python3-dev

# Установка Reticulum
python3 -m pip install rns
```

После установки этих пакетов `pip` сможет локально скомпилировать любые недостающие зависимости в вашей системе.

Ubuntu Lunar

В версиях Ubuntu, выпущенных после апреля 2023 года, по умолчанию больше невозможно использовать `pip` для установки пакетов в вашу систему. К сожалению, вам придётся вместо этого использовать замену — команду `pipx`, которая помещает установленные пакеты в изолированную среду. Это не должно негативно повлиять на Reticulum, но не будет работать для включения и использования Reticulum в ваших собственных скриптах и программах.

```
# Установите pipx
sudo apt install pipx

# Сделать установленные программы доступными в командной строке pipx
ensurepath

# Установить Reticulum
pipx install rns
```

В качестве альтернативы вы можете восстановить нормальную работу `pip`, создав или отредактировав файл конфигурации, расположенный по адресу `~/.config/pip/pip.conf`, и добавив в него следующий раздел:

```
[global]
break-system-packages = true
```

Для однократной установки Reticulum без глобального включения опции `break-system-packages`, можно использовать следующую команду:

```
pip install rns --break-system-packages
```

Примечание

Директива `--break-system-packages` — это несколько вводящий в заблуждение . Ее установка, конечно, не повредит никакие системные пакеты, а просто позволит устанавливать пакеты `pip` на уровне пользователя и всей системы.
Хотя в редких случаях это *может* привести к конфликтам версий, обычно это не создает никаких проблем, особенно при установке Reticulum.

Windows

В операционных системах Windows самый простой способ установить Reticulum — использовать менеджер пакетов `pip` из командной строки (либо из командной строки Windows, либо из Windows PowerShell).

Если у вас еще не установлен Python, [скачайте и установите Python](https://www.python.org/downloads/) [https://www.python.org/downloads/]. На момент публикации данного руководства

рекомендуемая версия — Python 3.12.7
[https://www.python.org/downloads/release/python-3127].

Важно! Когда установщик запросит, обязательно добавьте программу Python в переменные среды PATH. Если вы этого не сделаете, вы не сможете использовать установщик `pip` или запускать входящие в комплект утилиты Reticulum программы (такие как `rnsd` и `rnstatus`) из командной строки.

После установки Python откройте командную строку или Windows PowerShell и введите:

```
pip install rns
```

Теперь вы можете использовать Reticulum и все входящие в комплект утилиты непосредственно из вашего любимого интерфейса командной строки.

Reticulum на чистом Python

Предупреждение

Если вы используете пакет `rnspure` для запуска Reticulum на системах, не поддерживающих PyCA/криптографию [https://github.com/pyca/cryptography], важно прочитать и понять раздел «Криптографические примитивы» данного руководства.

В некоторых редких случаях и на менее распространенных типах систем установка одной или нескольких зависимостей невозможна. В таких ситуациях вы можете использовать пакет `rnspure` вместо пакета `rns` или использовать `pip` с опцией командной строки `--no-dependencies`. Пакет `rnspure` не требует никаких внешних зависимостей для установки. Обратите внимание, что фактическое содержимое пакетов `rns` и `rnspure` *полностью идентично*. Единственное различие заключается в том, что пакет `rnspure` не содержит списка зависимостей, необходимых для установки.

Независимо от того, как установлен и запущен Reticulum, он будет загружать внешние зависимости только в том случае, если они *нужны* и *доступны*. Если, например, вы

Если вы хотите использовать Reticulum в системе, не поддерживающей `pyserial`, это вполне возможно с помощью пакета *rnspure*, однако Reticulum не сможет использовать интерфейсы, основанные на последовательном соединении. Все остальные доступные модули по-прежнему будут загружаться при необходимости.

Дзен Reticulum

Иллюзия центра

На протяжении большей части поколения нас учили представлять себе цифровой мир через призму иерархии. В наших ментальных картах доминирует единственный, вводящий в заблуждение образ: **Облако**.

Мы представляем себе сеть как обширное, эфирное пространство «там, наверху» или «где-то там». Централизованное хранилище сервисов и данных, к которому мы, скромные клиенты, должны подключаться. Мы создаем наше программное обеспечение с этим допущением, жестко встроенным в нашу логику: *есть сервер. Сервер обладает полномочиями. Сервер знает путь. Я должен найти сервер, чтобы работать*.

Это ментальная модель «клиент-сервер», и она является основным препятствием для понимания Reticulum.

Заблуждение облака

Первый шаг в «Дзен Ретикулума» — осознать, что *облака не существует*.
Есть только компьютеры других людей. Когда вы разрабатываете для облака, вы разрабатываете *для* арендодателя. Вы принимаете тот факт, что существование вашего приложения
зависит от разрешения, работоспособности и постоянной благосклонности центрального органа.

В Reticulum вы должны переключить свое мышление с «подключения к» на «нахождение среди». Reticulum — это не услуга, на которую вы подписываетесь, — *это ткань, в которой вы обитаете*. Нет никакого «там, наверху». Есть только *здесь* и *там*, а пространство между ними — это одноранговая сеть.

Децентрализация или нецентрализуемость?

В современных технологических кругах часто можно услышать слово «децентрализованный». Но зачастую это всего лишь маркетинговый термин, обозначающий «слегка распределенную централизацию». Блокчейн с несколькими доминирующими майнерами или федеративный протокол с несколькими гигантскими серверами. *На практике* это по-прежнему централизованная система. Просто вместо одного центра здесь их несколько.

Reticulum идет дальше. Он стремится к **«нецентрализованности»**.

Это не политическая позиция, основанная на желании, а фундаментальная математическая
характеристика протокола, на которой построено все остальное. Reticulum исходит из того, что каждый узел в сети потенциально враждебен, а каждое соединение потенциально скомпрометировано. Он разработан без «привилегированных» узлов. Хотя некоторые узлы могут выступать в роли транспортных экземпляров — пересылая
трафик для других — они делают это *«вслепую»* и знают только о своем ближайшем окружении, и ни о чем больше. Они маршрутизируют на основе криптографических доказательствах, а не на административных привилегиях. Они не могут видеть, кто с кем общается, и не могут выборочно манипулировать трафиком, не нарушив при этом свою собственную способность к маршрутизации полностью.

Система построена таким образом, что иерархия становится структурно невозможной. Вы не можете захватить адрес, поскольку не существует центрального реестра, который можно было бы захватить. Вы не можете заблокировать пользователя, поскольку нет центрального переключателя, который можно было бы нажать. Вы можете предлагать пути прохождения по сети, но не можете заставить кого-либо ими пользоваться.

Смерть адресу

Чтобы освободиться от центра, вы должны также отказаться от концепции «адреса».

В мире IP адрес — это местоположение. Это координата в *глубоко иерархичной* и статичной сетке. Если вы переносите компьютер в другой дом, ваш адрес меняется. Если ваш маршрутизатор перезагружается, ваш адрес может измениться. Ваша *личность* привязана к вашему *местоположению*, и поэтому она уязвима и легко поддается контролю.

Reticulum устраняет эту связь между *идентичностью* и *местоположением*.

В Reticulum адрес — это не место; это **хеш идентичности**. Это криптографическое представление того, *кто* вы есть, а не *где* вы находитесь. Благодаря этому ваш адрес является переносимым. Вы можете перенести ноутбук из Wi-Fi-кафе в Берлине в сеть LoRa в горах, а затем на канал пакетной радиосвязи на лодке, и ваш «адрес» — ваш *хеш назначения* — никогда не изменится.

Сеть не маршрутизирует данные к месту; она маршрутизирует их к *человеку* (или машине). Когда вы отправляете пакет, вы не нацеливаетесь на координаты в сетке; вы шифруете сообщение для конкретного объекта. Сеть динамически определяет, где в данный момент находится этот объект, и делает это таким образом, что никто на самом деле не знает, где этот объект физически расположен.

Рассмотрим:

- **Старый способ:** *«Я на* `192.168.1.5`. *Найди меня*».
- **«Дзен-способ»:** *«Я* — `<327c1b2f87c9353e01769b01090b18f2>`.
  Где бы я ни был, мои одноранговые узлы могут связаться со мной».

Как только вы перестаете думать о серверах и начинаете думать о переносимых идентичностях, когда каждый всегда может напрямую связаться с любым другим участником, иллюзия центра исчезает. Вы понимаете, что *нет* никакого центра, который бы скреплял сеть. Не нужны ни координаторы, ни бюрократы. Сеть — это просто совокупность равноправных участников, общающихся напрямую, независимо и без какого-либо руководителя.

Физика доверия

Паранойя — отличный принцип проектирования

Если мы примем, что центра нет — что сеть представляет собой хаотичную одноранговую сетку — нам придется столкнуться с пугающей реальностью: **никто не стоит на страже у входа**.

В традиционном подходе к построению сетей мы опираемся на концепцию «доверенного ядра». Мы полагаем, что Wi-Fi в местном кафе безопасен или что магистральные провайдеры являются нейтральными хранителями. Мы строим нашу безопасность как замок: с крепкими

стены снаружи, мягкость и доверие внутри. Мы используем шифрование только тогда, когда выходим в «дикий» Интернет.

Враждебные среды

«Дзен Reticulum» требует от вас перевернуть это представление. Вы должны исходить из того, что *любая* среда враждебна. Это не цинизм, а просто безразличная физика.

Когда вы передаете информацию по радиоволнам, это все равно что кричать в переполненной комнате. Любой может вас услышать. Когда вы пользуетесь Интернетом, ваши
пакеты проходят через маршрутизаторы, контролируемые незнакомцами, корпорациями и государственными структурами. Рассчитывать на конфиденциальность в этой среде без криптографической защиты — это не оптимизм, а грубая халатность.

Reticulum построено на предпосылке, что каждое соединение прослушивается, а каждый узел — потенциальный противник. Если ваша система не может выстоять против противника, владеющего
физический уровень, она не сможет выжить вообще.

Но в этом и заключается парадокс: предполагая, что сеть враждебна, вы делаете её безопасной. Когда вы принимаете опасности такими, какие они есть, они становятся управляемыми. Когда вы перестаёте доверять инфраструктуре и начинаете доверять математике, вы устраняете единственную точку отказа: человеческую добросовестность.

Шифрование — это не функция

В мире TCP/IP шифрование является второстепенным элементом. Это слой, который мы накладываем поверх протокола (HTTPS, TLS), чтобы залатать дыры в безопасности исходной конструкции. Это «функция», которую иногда *включают* для «конфиденциальных данных». Такой подход является принципиально ошибочным, поскольку все данные являются конфиденциальными.

В Reticulum шифрование — это **сила притяжения**.

Оно не является опцией. Это не плагин. Это *фундаментальная сила, которая позволяет сети существовать*. Если бы вы убрали шифрование из Reticulum, маршрутизация перестала бы работать. Транспортная система использует криптографические подписи и энтропию для проверки путей и передачи информации. Если бы пакеты были в виде открытого текста,

промежуточные узлы не смогли бы подтвердить, что маршрут действителен, а конечные точки не смогли бы предотвратить подделку или взлом.

В Reticulum энтропия зашифрованного пакета и *есть* логика маршрутизации.

Просить версию Reticulum без шифрования — все равно что просить версию океана без жидкости. Вы просите не об изменении функции, а о другой физической вселенной. Мы проектируем для вселенной,
, в которой информация обладает массой, структурой и целостностью.

Архитектуры «нулевого доверия»

Мы должны отучиться полагаться на **институциональное доверие**.

На протяжении десятилетий нас приучали доверять авторитетам. Мы доверяем веб-сайту, потому что цепочка центров сертификации (компаний, которых мы не знаем) ручается за него. Мы доверяем приложению, потому что оно находится в магазине приложений (управляемом корпорацией, которую мы не контролируем). Мы доверяем сообщению, потому что оно пришло с номера телефона, присвоенного оператором связи. Тем не менее, сегодня все в нашей цифровой информационной сфере менее надежно и рискованнее, чем средневековый рынок подержанного нижнего белья.

Reticulum заменяет институциональное доверие **криптографическим доказательством**.

В Reticulum вы доверяете узлу не потому, что у него красивое имя хоста или потому, что он указан в каталоге. Вы доверяете ему, потому что он хранит закрытый ключ, соответствующий хэшу назначения, с которым вы общаетесь. Это доверие является бинарным, математическим и **абсолютным**. Либо подпись совпадает, либо нет. Здесь нет «возможно».

Этот сдвиг переносит власть от института к индивидууму. Вы становитесь высшим арбитром своих собственных доверительных отношений. Вы решаете, какие ключи принимать, по каким путям идти и какие идентичности признавать.

Подумайте:

- **Старый подход:** *«Я доверяю этому сайту, потому что браузер показывает, что значок замка зелёный*».
- **Путь дзен:** *«Я доверяю этому сайту, потому что я проверил его хеш-отпечаток вне канала связи, и математика подтверждает подпись*».

Когда вы усваиваете «Физику доверия», вы перестаете искать защиты в брандмауэрах, VPN и соглашениях об условиях предоставления услуг. Вы понимаете, что
настоящая безопасность зависит от самой архитектуры протокола. Вы можете перестать доверять облаку и начать доверять коду — потому что можете проверить его самостоятельно.

Преимущества дефицита

Каждый бит имеет значение

Мы пристрастились к изобилию. В современной цифровой экосистеме пропускная способность воспринимается как бесконечный, ровный океан. Мы без зазрения совести транслируем видео высокой четкости, отправляем целые библиотеки кода ради отображения одной-единственной кнопки и измеряем производительность в гигабитах в секунду. Это
изобилие опустошило наше ремесло. Когда исчезают ограничения, умирает эффективность, а вместе с ней — определенная ясность и качество.

Reticulum предлагает вам выйти из океана и встать на канатоходческую веревку.

Заблуждение пропускной способности

Дзен Reticulum требует осознания того, что **5 бит в секунду — это вполне приемлемая скорость**.

Современному разработчику это кажется полным параличом. Но в ограничениях таится огромная свобода: когда у вас гигабитное соединение, вы можете позволить себе невероятную небрежность. Вы можете растрачивать ресурсы. Вы можете перекладывать свои проблемы на инфраструктуру. *«Медленно? Купите более быстрый роутер*».

Но при высокой задержке и низкой пропускной способности канала (будь то зашумленный радиоканал HF или слабый прыжок LoRa) вы не можете переложить проблемы ни на кого. Вы должны
решать их. Сеть не терпит расточительства.

Это вынуждает перейти от потребления к взаимодействию. Вы больше не потребляете услугу, предоставляемую «толстой трубой»; вы вступаете в осторожные переговоры с физической средой. Среда становится партнером в
разговоре, а не просто тупым проводником. Вам внезапно нужно *понять мир, чтобы быть в нем*.

Стоимость байта

В условиях дефицита ресурсов байт — это не просто данные, а энергия, время и пространство.

Каждый переданный вами байт расходует заряд батареи на узле, работающем от солнечной энергии. Он занимает ценное эфирное время, которое могло бы быть использовано другим узлом. Он
представляет собой измеримую часть электромагнитного спектра.

Когда вы осознаете это, вы начинаете писать код по-другому. Вы перестаете спрашивать: «Сколько данных я могу отправить?», а начинаете спрашивать: «Каков *минимальный* объем информации, необходимый для передачи этого намерения? Как я могу наилучшим образом использовать свою информационную энтропию?»

Именно в этом проявляется элегантность Reticulum. Протокол разработан так, чтобы избавиться от всего несущественного. Для установления соединения требуется три очень маленьких пакета. Хеш адресата помещается в 16 байт. Накладные расходы ничтожно малы, оставляя почти весь канал для самого сообщения.

Рассмотрим:

- **Старый подход:** *«Мне нужно отправить обновление статуса. Я отправлю объект JSON с метаданными, временными метками и информацией о профиле пользователя (15 КБ)».*
- **«Дзен-подход»:** *«Мне нужно отправить обновление статуса. Я отправлю один байт, представляющий код состояния. Контекст уже известен».*

Конечно, это оптимизация, но, что более важно, *это форма уважения*. Эффективность в общей среде — это акт ответственного отношения. Забирая из сети только то, что вам нужно, вы оставляете место для других. Сеть прислушивается к тем, кто говорит с целью.

Поток и время

Дефицит также учит нас времени. Мы стали зависимы от *синхронного* «сейчас» — мгновенного пинга, потока в реальном времени. Но Reticulum принимает *асинхронное* время.

Когда соединения прерывисты, а задержка измеряется минутами или часами, «Реальное время» — это иллюзия. Reticulum предлагает использовать механизм **«Store and Forward»** не просто в качестве запасного варианта, а в качестве основного способа работы. Вы пишете сообщение, оно отправляется, когда это возможно, и доходит, когда доходит.

Это меняет психологическую структуру общения. Это устраняет тревогу, связанную с немедленным ответом. Это дает возможность для размышлений. Вы не требуете внимания получателя *прямо сейчас*; вы оставляете подарок на его пути, чтобы он нашел его, когда будет готов.

Проектируя с учетом задержки, вы проектируете с учетом отказоустойчивости. Вы больше не строите карточный домик, который рушится при потере одного пакета. Вы строите каменную арку, которая распределяет нагрузку *во времени*.

Освобождение от ограничений

В дефиците таится странный оптимизм. Когда вы вынуждены действовать в условиях жестких ограничений, вам приходится расставлять приоритеты. *Вы* должны решить, что действительно важно. *В этом* и заключается суть свободы воли.

В бесконечном фантастическом мире «Облака» все срочно, поэтому ничего не срочно. В экономике «Ретикулума» стоимость передачи данных заставляет вас взвешивать ценность вашего сообщения. Вам действительно нужно отправить этот смайлик с сердцем
? Необходима ли эта фотография?

Когда вы отсеиваете шум, остается *сигнал*.

Эта дисциплина формирует другого типа разработчика. Она формирует мастера, который понимает, что лучший код — это тот, который не нужно писать. Она формирует пользователя, который понимает, что самое сильное сообщение — это то, которое *понятно*, а не то, которое громче всех. В мире Reticulum вы не просто потребитель пропускной способности; вы — архитектор намерений.

Суверенитет через инфраструктуру

Будьте своей собственной сетью

Мы живем в эпоху цифрового арендования. Мы арендуем подключение у интернет-провайдеров. Мы арендуем хранилище у поставщиков облачных услуг. Мы даже заимствуем свою идентичность у социальных сетей. Мы — арендаторы в доме, который не строили, подчиняемся правилам, которые не писали, и подвергаемся выселению по прихоти арендодателя, который никогда нас не встречал.

«Дзен Reticulum» — это осознание того, что вы *можете* стать владельцем дома.

Заблуждение операторского уровня

На протяжении десятилетий нам внушали, что создание сетей — это не просто сложно, а невозможно. Это представляется как черная магия, доступная только телекоммуникационным компаниям и миллиардерам, требующая миллионов долларов на оптоволокно, климатизированных центров обработки данных и целых армий инженеров. Нам говорят, что построение надежной инфраструктуры — это «слишком сложно» для отдельного человека или

небольшой организации. Это большая, жирная ложь.

Физика проста. Радиоволне нужны передатчик и приемник. Пакету нужен путь. «Сложность» современного Интернета — это в основном бюрократическая — гора систем биллинга, захват регуляторов и устаревшее наследие, предназначенные для того, чтобы удержать власть в руках стражей ворот.

Reticulum избавляет от бюрократии. Он работает на оборудовании, которое стоит стоимость ужина. Он работает на частотном спектре, который можно использовать бесплатно. Он демонстрирует, что для создания надежной сети планетарного масштаба не требуется компания из списка Fortune 500. Требуется лишь желание внедрить проект и распределенные, нескоординированные усилия множества людей.

Личная инфраструктура

Вот где дело доходит до практического применения. Вы можете читать о Reticulum, вы можете понимать теорию, но истинное понимание приходит только тогда, когда вы подключаете радиостанцию и запускаете транспортный узел. Внезапно вы больше не потребитель.
Вы становитесь оператором.

Это изменение едва заметно, но глубоко. Когда вы управляете собственной инфраструктурой, сеть перестает быть услугой, которую вам *предоставляют*. Она становится пространством, в котором вы *живете*. Вы несете ответственность за поток информации. Вы обретаете глубокое понимание среды — того, как погода влияет на радиоволны, как меняется топология, как пакеты данных «танцуют» в эфире.

Из этого проистекает тихая компетентность. Вы перестаете спрашивать «Интернет не работает?» и начинаете спрашивать *«Мои* каналы работают?». Вы перестаете ждать техника и начинаете проверять логи. Это форма силы. Понять систему, которая передает ваши слова, — значит освободиться от тайны, которая держит вас в зависимости.

Способность отключаться

Зачем утруждаться? Зачем покупать радио, писать конфигурацию и оставлять Pi работать в углу?

Потому что старая, централизованная сеть хрупка. И потому что большинство из нас даже не хочет больше там находиться.

Интернет, на который мы сегодня полагаемся, представляет собой цепочку единичных точек отказа. Перережьте подводный кабель — и целый континент погрузится в темноту. Отключите энергосистему — и «облако» исчезнет. Снизьте приоритет «неправильного» трафика — и поток информации задушится.

Суверенитет — это способность пережить отключение, независимо от того, было ли оно случайным или преднамеренным.

Создавая собственную инфраструктуру, вы создаете линию спасения. Reticulum разработан для работы через каналы, недоступные традиционному интернету — оголенные провода, радиостанции на батарейках, импровизированные WiFi-сети. Когда сеть

выходит из строя, приходят цензоры или не оплачивается счет, ваша сеть Reticulum продолжает тихо гудеть.

Речь идет не о том, чтобы «отстраниться» от общества. Речь идет о создании фундамента, на котором могло бы функционировать настоящее *Общество*.

Рассмотрим:

- **Старый подход:** «У меня медленный интернет. Надо позвонить провайдеру и пожаловаться».
- **Путь дзен:** «На пути много помех. Я настрою антенну или найду лучший маршрут».

Взяв на себя ответственность за инфраструктуру, вы берете на себя ответственность за свой голос. Вы перестаете кричать в чужой рупор и начинаете создавать свой собственный. Сеть больше не является чем-то, что с вами происходит; это то, что вы сами создаете.

Идентичность и кочевничество

Меняющееся «я»

В старом мире вас определяют ваши координаты. Если вы находитесь по адресу `34.109.71.5`, значит, вы *здесь*. Если вы отсоедините кабель и пойдете по улице, вы исчезнете. Ваше цифровое «я» испарится, потому что оно было привязано к розетке. Вы — призрак в бесконечной системе шестеренок, рычагов и транзисторов, привязанный к аппаратному обеспечению и тем, кому оно принадлежит.

Это порождает едва уловимое, постоянное беспокойство. Мы боимся отключения, потому что в архитектуре старого Интернета отключение — это своего рода смерть.

«Дзен ретикулума» предлагает другой способ существования.

Портативное существование

В Reticulum ваша личность — это не местоположение или имя пользователя, присвоенное сервисом. Это криптографический ключ — сложная, уникальная математическая

подпись, существующая независимо от физического мира. Вы можете носить его только в своем уме, если хотите.

Представьте себе, что это не столько почтовый адрес, сколько имя. *Настоящее имя*.

Если вы путешествуете из Берлина в Токио, вы не меняете свое имя. Вы по-прежнему остаетесь собой. Люди, которые вас знают, по-прежнему могут вас узнать. Reticulum
применяет этот принцип к сетевому уровню. Ваш хеш назначения (Destination Hash) **неизменен**. Он путешествует вместе с вами, надежно хранится на вашем устройстве и *непоколебим*, *как камень*.

Это меняет отношения между вами и машиной. Вы не «входите» в сеть через конкретный шлюз. Вы — конечная точка. Сеть не подключается к месту; *она сходится на вас*.

Роуминг узлов

Эта свобода вводит новое понятие времени и пространства: **кочевничество**.

Поскольку ваша личность переносима, ваша связь может быть гибкой. Вы можете
В одну минуту вы сидите за столом, подключенным к оптоволоконной магистрали, а в следующую — идете по полю, подключенным только к сети LoRa с большим радиусом действия. Для остальной части сети ничего не изменилось. Вашим друзьям не нужно
обновлять ваши контактные данные. Сообщения, которые они отправляют, не возвращаются обратно. Сеть распознает смену среды и автоматически перенаправляет поток данных.

Вы больше не являетесь стационарным узлом в фиксированной сетке. Вы — странник в изменчивой среде.

Интерфейсы — будь то Wi-Fi, Ethernet, пакетная радиосвязь или физический провод — это всего лишь одежда, которую носит ваш узел. Вы меняете ее в соответствии с окружающей среде. Внутри вы остаетесь прежними. В этом заключается освобождение протокола. Он рассматривает физическую среду как временное обстоятельство, а не как определение себя.

Подумайте:

- **Старый подход:** *«У меня прервалось соединение. Мне нужно заново подключиться к VPN, чтобы сообщить, где я нахожусь сейчас».*
- **«Дзен-подход»:** *«Я переместился. Сеть незаметно адаптируется к этой новой реальности».*

Сообщение о присутствии

Как сеть находит странника? Она прислушивается.

В мире IP мы запрашиваем каталоги. Мы спрашиваем сервер: «Где Марк?» Сервер проверяет свою базу данных и дает нам координаты. Это означает, что кто-то, где-то, отслеживает вас. Это предполагает и *требует* слежения.

Reticulum заменяет слежку на **объявления**.

Вместо того чтобы спрашивать у центрального органа, где вы находитесь, вы просто сообщаете о своём присутствии. Вы передаёте криптографическое доказательство: «Я здесь, и я тот, за кого себя выдаю». Эта информация распространяется по сети. Ваши соседи принимают её, обновляют свои таблицы маршрутов и передают дальше.

Это тихий, органичный процесс. Это цифровой эквивалент зажигания фонарей в темноте. Вам не нужно гнаться за светом; вы позволяете свету найти вас. Это уважает вашу автономию. Вы выбираете, когда объявлять о себе, как часто говорить и кому. Вы также выбираете, когда исчезнуть — на мгновение или навсегда.

Якорь в потоке

В этом кочевничестве есть глубокий покой. Оно учит вас, что стабильность не приходит от того, что вы стоите на месте. Стабильность приходит от *внутренней согласованности*.

Обладая собственным закрытым ключом, вы владеете собственным центром тяжести. Мир вокруг вас — инфраструктура, топография и доступность каналов связи — может хаотично меняться. Бури могут сносить вышки. Кабели могут быть перерезаны. Интернет может перестать работать.

Но пока у вас есть ключ, у вас есть ваша личность. Вся инфраструктура может быть разрушена и восстановлена, а вы останетесь собой. Ничто не вечно, но и ничего не теряется.

Вы становитесь суверенным субъектом, движущимся сквозь шум, связанным не жесткостью кабелей, а гибкостью распознавания. Сеть
становится местом, в котором вы обитаете, а не услугой, на которую вы подписаны: вы как дома в эфире.

Этика инструмента

Технология с совестью

Вы отреклись от центра. Вы приняли физику доверия. Вы приняли экономику дефицита и свободу безграничного кочевничества. Вы стоите в новом пространстве. А теперь посмотрите на инструмент в вашей руке.

В старом мире нас учили, что технология нейтральна. Нам говорят, что «не оружие убивает людей, а люди», или что компонент — это просто компонент, независимо от его комбинаторного потенциала. Это удобная ложь. Она служит лишь для того, чтобы позволить создателям умыть руки от ответственности.

Но теперь мы знаем лучше. Мы знаем, что **архитектура — это политика**, а *политика — это контроль*. То, как вы строите систему, определяет, как она будет использоваться. Если вы построите систему, оптимизированную для массовой слежки, вы получите паноптикон. Если вы построите систему, оптимизированную для централизованного контроля, вы получите диктатуру. Если вы построите систему, оптимизированную для извлечения, вы получите паразита.

«Дзен ретикулума» утверждает, что инструмент никогда не

бывает нейтральным. Напротив: инструмент — это **воплощение**

намерения.

Принцип вреда

Почему лицензия Reticulum запрещает использование программного обеспечения в системах, предназначенных для нанесения вреда людям? Разве это не просто ограничение свободы?

Да, это ограничение *лицензии*, но это расширение *свободы*.

Создание мощных инструментов без морального компаса ни в коем случае не является добродетельным или похвальным поступком — это чистая и простая безответственность.

Инструмент, который легко можно использовать для угнетения, представляет реальную опасность для пользователя. Если вы создаете сеть, которую тиран может обратить против вас, вы не свободны. Вы просто ждете, когда узел на шее затянется. Встраивая «Принцип
в юридическую ДНК эталонной реализации, мы создаем защитный механизм. Мы ясно и незыблемо заявляем, что *этот инструмент* предназначен для **жизни**, а не для смерти.

Это приводит программное обеспечение в соответствие с интересами человечества. Это гарантирует, что сеть не может быть использована в качестве системы уничтожения, пульта управления боевым дроном или устройства для пыток без нарушения условий лицензии и закона. Это черта, проведенная на песке — не правительством или внешней властью, а самими создателями этого инструмента.

Рассмотрим:

- **Старый подход:** *«Это всего лишь программное обеспечение. Как люди его используют — не моя проблема».*
- **Путь дзен:** *«Это программное обеспечение — среда обитания. Я не позволю использовать его для постройки клетки».*

Решать, присоединяться ли к этому, — *ваше* право: мы никому не навязываем эту позицию. Если вы решите отдать предпочтение жизни перед смертью, творчеству перед разрушением, мы предоставим вам чрезвычайно мощный инструмент, которым вы сможете распоряжаться и создавать по своему усмотрению. Если же нет — мы откажем вам в этом.

Если вам это не нравится, то вы нам здесь совершенно не нужны, и вы остаетесь наедине со своими проблемами.

Протокол общественного достояния

Это приводит к важному различию: разнице между *идеей* и *реализацией*.

Протокол — математические правила работы Reticulum — посвящен общественному достоянию. Он принадлежит человечеству. **Никто не может владеть им**.
Любой может реализовать его, усовершенствовать или адаптировать. В этом заключается суть свободной коммуникации, которая сама по себе должна оставаться свободной навсегда.

Но функциональная, развернутая *эталонная реализация* — код на Python, техническое обслуживание, годы труда — имеет совесть. Это различие является двигателем устойчивости. Оно позволяет протоколу быть универсальным, при этом
гарантируя, что конкретный труд создателей не будет узурпирован с целью подрыва основополагающего замысла самого проекта. Из этого документа должно быть совершенно ясно, в чем заключается этот замысел.

Если вы хотите построить с помощью Reticulum систему, которая манипулирует и наносит вред пользователям ради прибыли или нацеливает ракеты, вы можете использовать протокол, находящийся в общественном достоянии, и начать с нуля. Но вы не можете взять нашу работу. Вы должны сделать свою собственную. Это служит опорой ответственности. Если вы хотите создать оружие, идите и сами выковывайте сталь, пока мир наблюдает. И когда прольется кровь — она будет на **ваших** руках.

Сохранение человеческой инициативы

Мы живем в эпоху хищнической эксплуатации. Общедоступные ресурсы открытого исходного кода собираются, поглощаются и перерабатываются алгоритмами машинного обучения, владельцы которых стремятся заменить тех самых людей, кто создал эти ресурсы. Наш код, наши слова и наше творчество используются для обучения
систем, специально разработанных для того, чтобы сделать нас ненужными, не предлагая ничего другого взамен, кроме рабства и поводков.

Reticulum выступает против этого.

Лицензия защищает программное обеспечение от использования в качестве корма для чудовища. Она проводит чёткую грань: этот инструмент предназначен для *людей*. Он предназначен для связи между людьми. Он
не является набором данных, который можно безжалостно эксплуатировать с целью создания синтетического повелителя, управляемого крошечным конгломератом контролеров.

Это радикальный акт сохранения. Защищая код от присвоения ИИ, мы защищаем пространство для человеческой свободы действий. Мы гарантируем, что останется цифровое пространство, где участники — это плоть, кровь и душа, где решения принимаются умами, а не повелителями, прячущимися за моделями.

Когда вы используете Reticulum, вы пользуетесь инструментом, который уважает вас. Он не рассматривает вас как объект для отслеживания. Он не рассматривает ваши данные как топливо для алгоритма. Он рассматривает вас как суверенного, равноправного партнера.

Это меняет основополагающую предпосылку использования технологии. Это возвращает достоинство взаимодействию. Вы не просто пользователь услуги; вы — участник взаимного соглашения. Инструмент поддерживает вашу автономию, а не подрывает ее.

Таким образом, этика — это не ограничение, а основа. Это основа, которая помогает гарантировать, что сеть и завтра будет принадлежать вам.

Шаблоны проектирования для систем пост-IP

Практическая философия для разработчиков

Философия бесполезна, если ее нельзя воплотить в коде.
Метафоры, которые мы рассмотрели — кочевничество, дефицит, доверие — это не просто поэзия, а реальные инженерные ограничения. Когда вы садитесь писать программное обеспечение для Reticulum, эти концепции должны формировать саму структуру вашего приложения.

Теперь мы переходим от вопроса *«почему»* к вопросу *«как»*. Именно здесь абстрактное превращается в конкретное, и здесь вы увидите истинную глубину узоров, которые мы плели.

Хранение и пересылка

Интернет приучил нас к нетерпеливости. Мы пишем синхронный код. Мы отправляем запрос и ждем, блокируя пользовательский интерфейс, затаив дыхание. Если ответ не приходит в течение 250 миллисекунд, мы показываем индикатор загрузки. Если он не приходит в течение

пяти секунд, мы показываем сообщение об ошибке. Мы рассматриваем сетевое соединение как бинарное состояние: либо мы «онлайн», либо у нас «не работает».

Это неустойчиво. Это отрицание реальности.

В Reticulum подключение представляет собой спектр, а присутствие — асинхронно. Если это вообще применимо к вашим целям, вы должны проектировать свои приложения так, чтобы
Примите принцип **«Store & Forward**».

Вместо того чтобы требовать немедленного ответа, ваше приложение должно действовать как терпеливый участник. Вы создаете сообщение для кого-то или чего-то в сети. Сеть хранит его. Она передает его от узла к узлу, возможно, в течение часы или дни, ожидая появления получателя. Когда он наконец появляется, сообщение доставляется. Это требует перехода от модели «запрос/ответ» к модели «событие/обработчик». Как именно это сделать — задача, которую вам предстоит разумно решить в рамках вашей области задач, но уже существуют системы на основе Reticulum, которые справляются с этим чрезвычайно хорошо, и вы можете использовать их в качестве источника вдохновения.

Рассмотрим:

- **Старый способ:** `Connect() -> Send() -> Wait() -> Сбой при истечении времени ожидания.`
- **«Дзен-подход»:** `Send() -> Продолжаем жить. -> Receive(), когда оно приходит.`

Это кардинально меняет пользовательский опыт. Устраняется беспокойство, связанное с полоской загрузки. Создается ощущение непрерывности. Пользователь не «ждет сети»; он взаимодействует с постоянным журналом коммуникации, который находится в самой сети.

Имя — это сила

В мире IP мы являемся рабами системы доменных имен. Мы полагаемся на иерархию регистраторов, которые сопоставляют понятные человеку имена с адресами, понятными машинам. Эта иерархия является узким местом. Если регистратор аннулирует ваш домен или если DNS-сервер выходит из строя, вы исчезаете.

Reticulum устраняет эту иерархию с помощью **идентификации на основе хешей**.

В этом шаблоне проектирования имя — это не строка, которую вы ищете; это криптографический адрес, который вы проверяете. При разработке для Reticulum вы перестаете запрашивать у пользователя URL и начинаете запрашивать адрес назначения или хеш идентификатора.

Сначала это кажется странным. Хеш типа `<83b7328926fed0d2e6a10a7671f9e237>` выглядит чужеродно по сравнению с `myfriend.com`. Но эта чужеродность — и есть защита. Его **невозможно** подделать. Его **невозможно** подвергнуть цензуре со стороны регистратора. Он **абсолютен**.

Разработка с учетом этого означает смену метафор пользовательского интерфейса. Вы больше не просматриваете сеть страниц; вы управляете реестром ключей. Вы создаете «адресную книгу», которая на самом деле является связкой ключей. Имена присваиваются пользователем, и власть остается за ним. То, что хеши выглядят сложными, напрямую аналогично прочности связей, формируемых их использованием. Это
заставляет пользователя выполнить процедуру проверки — внеканальное согласование, которое восстанавливает тот элемент доверия, который был утрачен из-за использования SSL-сертификатов.

Интерфейс — это среда

Один из самых освобождающих паттернов в Reticulum — это **транспортная независимость**.

В традиционных сетях ваш код часто переполнен транспортной логикой. «Я подключен к Wi-Fi? Проверить пропускную способность. Я подключен к сотовой сети? Проверить тарифный план. Я подключен к Ethernet?». Вы постоянно занимаетесь микроуправлением каналом.

В Reticulum вы пишете в API, а API пишет в среду. Вы отправляете пакет в пункт назначения. Вам не важно, проходит ли этот пакет через TCP-туннель, радиоволну LoRa или последовательный интерфейс. Это забота стека.

Это позволяет создавать **универсальные приложения**. Представьте себе приложение для обмена сообщениями. Вы пишете его один раз. Оно работает на ноутбуке, подключенном к оптоволоконной сети. Оно работает на

телефоне в городе через Wi-Fi. И, без изменения ни одной строчки кода, оно работает на устройстве в глуши, обмениваясь данными с другими устройствами только по радио.

Принцип прост: **никогда не программируйте под аппаратное обеспечение.**

Программируйте под задачу. Рассмотрим:

- **Старый способ:** `socket.connect(ip, port)` , а затем еще много всего
- **«Дзен-подход»:** `RNS.Packet(destination, data).send()`

Абстрагируя среду передачи, вы делаете свое программное обеспечение невосприимчивым к изменениям в инфраструктуре. Пользователь может переключиться с 4G-хотспота на HF-модем
завтра. Вашему программному обеспечению не нужно об этом знать. Оно просто продолжает диалог.

Возникающие паттерны

Когда вы объединяете эти паттерны — *«Хранение и пересылка»*, *«Идентификация на основе хешей»* и *«Транспортная независимость»* — вы создаете программное обеспечение, которое воспринимается принципиально иначе.

Оно кажется *надежным*. Оно не мерцает, когда пропадает сигнал. Оно не паникует, когда сервер не работает. Оно имеет вес. Оно имеет стойкость. Оно имеет *значимость*.

Вы больше не создаете «клиента», который выпрашивает внимания у «сервера». Вы создаете автономного агента, существующего в сетке. Он говорит, когда нужно, слушает, когда может, и носит свою идентичность с собой, куда бы ни пошел.

Это и есть высшее проявление дзен. Код — это не просто набор инструкций: это рамки поведения. Это способ *существования* в сети.

Структура независимости

Мы избавились от иллюзий. Мы увидели, что центр пуст, что доверие *должно* быть крепким, что ресурсы ограничены и что мы должны владеть своей

инфраструктурой. Мы увидели, что у инструментов есть своя этика и что наша идентичность может меняться плавно

Это возвращение общего достояния. Слишком долго мы позволяли, чтобы самая важная основа человеческого общества — *наша способность общаться друг с другом* — была захвачена структурами, которые не разделяют наших интересов. Мы позволили, чтобы архитектуру нашего общения проектировали не архитекторы, а бухгалтеры.

Мы возвращаем ее себе. Не путем подачи прошений хозяевам, а путем построения нового мира внутри, над, под и вокруг оболочки старого.

Работа завершена

Самое тяжелое уже позади.

Протокол находится в общественном достоянии — это подарок человечеству, который никогда не может быть отнят. Программное обеспечение написано, протестировано и работает на устройствах, разбросанных по всему миру. Руководство лежит перед вами. Исходный код эталонной реализации теперь распространяется на сотнях
тысячи устройств по всему миру. Никто не может удалить или уничтожить его. Аппаратное обеспечение доступно и широко распространено.

Путь сюда был нелегким, но мы дошли. Теперь нет комитета по дорожной карте, ожидающего одобрения. Нет венчурного капитала, диктующего пользовательский опыт. Нет генерального директора, который должен утвердить выпуск следующей функции.

Есть только вы.

Барьер для входа — это уже не сложность: это просто привычка к зависимости. Вас приучили ждать. Ждать обновления приложения. Ждать, пока интернет-провайдер починит линию. Ждать, пока платформа разрешит публикацию. Ждать, пока правительство изменит политику. Ждать лайков. Ждать, пока революцию покажут по телевизору.

Революция никогда не транслировалась

по телевидению. Она передается

пакетами.

Открытое небо

Будущее этой технологии — это строительный проект.

Это похоже на одиночный узел на подоконнике, прислушивающийся к помехам. Это похоже на сообщение, отправленное соседу, минуя шум коммерческого Интернета. Это похоже на сеть сообщества, которая растет, звено за звеном, прыжок за прыжком, поддерживаемая руками тех, кому важнее связь, чем прибыль.

У вас есть чертежи. У вас есть инструменты. У вас есть философия. Шум старого мира улетучился, оставив вас с тихой ясностью открытого спектра.

Марк, начало 2026 года

Программы, использующие Reticulum

В этой главе представлен неполный список известных программ, систем и протоколов прикладного уровня, созданных с использованием Reticulum.

Эти программы позволят вам понять, как работает Reticulum. Большинство из них разработаны так, чтобы стабильно работать даже в медленных сетях на основе LoRa или пакетной радиосвязи, но все они также могут использоваться по быстрым каналам, таким как локальный Wi-Fi, проводной Ethernet, Интернет или любая их комбинация.

Таким образом, начать экспериментировать легко, без необходимости настраивать какие-либо радиоприемники или инфраструктуру только для того, чтобы попробовать. Для начала достаточно запустить программы на отдельных устройствах, подключенных к одной сети Wi-Fi, а физические радиоинтерфейсы можно будет добавить позже.

Программы и утилиты

Уже существует множество различных приложений, использующих Reticulum, которые служат самым разным целям — от повседневной коммуникации и обмена информацией до системного администрирования и решения сложных задач в области сетей и связи.

Разработка приложений и систем на базе Reticulum продолжается, поэтому рассматривайте этот список как неполный перечень *некоторых* из доступных вариантов. Проведя небольшой поиск, в первую очередь на самом сайте Reticulum, вы найдёте ещё много интересного.

Удаленная оболочка

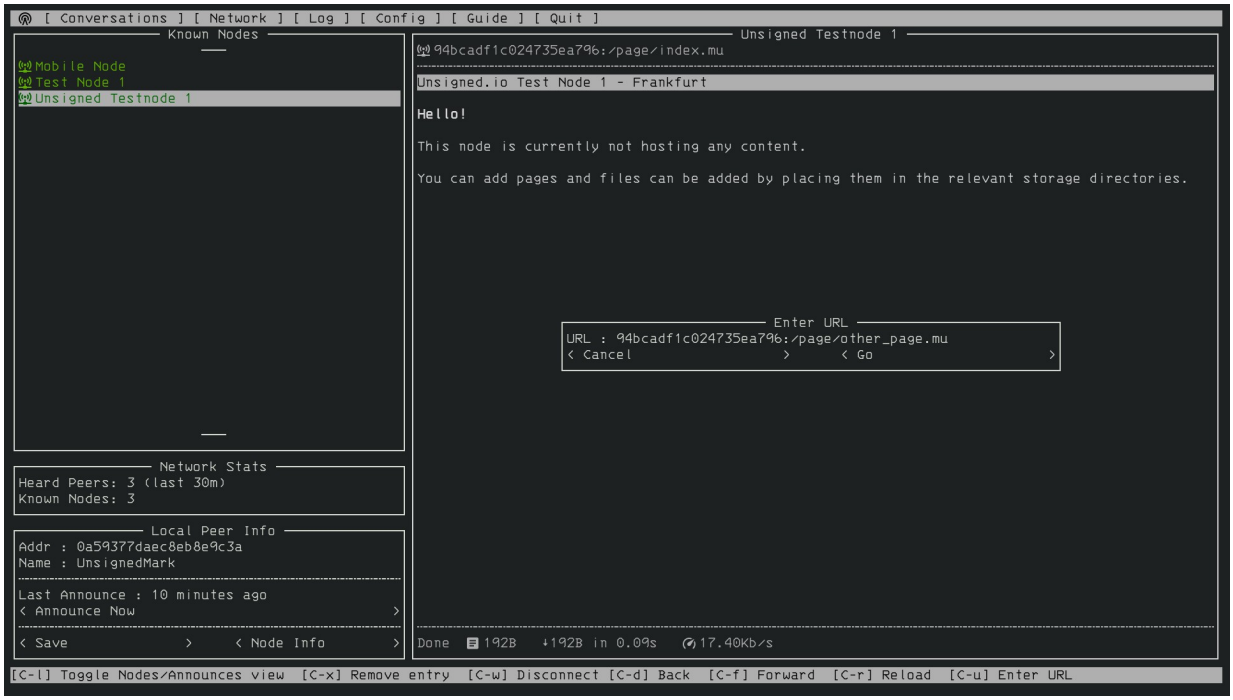
Программа rnsh [https://github.com/acehoss/rnsh] позволяет устанавливать полностью интерактивные удаленные сеансы оболочки через Reticulum. Она также позволяет передавать данные любой программы в удаленную систему или из нее, и работает аналогично `ssh`. Программа `rnsh` очень эффективна и может обеспечивать полностью интерактивные сеансы оболочки

сеансы, даже по каналам связи с крайне низкой пропускной способностью, таким как LoRa или пакетная радиосвязь.

В дополнение к стандартному, полностью интерактивному терминальному режиму, для каналов связи с крайне ограниченной пропускной способностью `rnsh` предлагает линейно-интерактивный режим, позволяющий взаимодействовать с удаленными системами даже тогда, когда пропускная способность канала измеряется несколькими
сотен бит в секунду.

Nomad Network

Терминальная программа Nomad Network
[https://github.com/markqvist/nomadnet] предоставляет полноценный зашифрованный Коммуникационный пакет, созданный на базе Reticulum. Он включает в себя функцию обмена зашифрованными сообщениями (как прямой, так и с отложенной доставкой для пользователей, находящихся в автономном режиме), обмен файлами, а также встроенный текстовый браузер и сервер страниц с поддержкой динамически отображаемых страниц, аутентификации пользователей и другими возможностями.



[https://github.com/markqvist/nomadnet]

Nomad Network [https://github.com/markqvist/nomadnet] — это клиентский интерфейс для протокола обмена сообщениями и информацией LXMF.

Узел страниц RNS

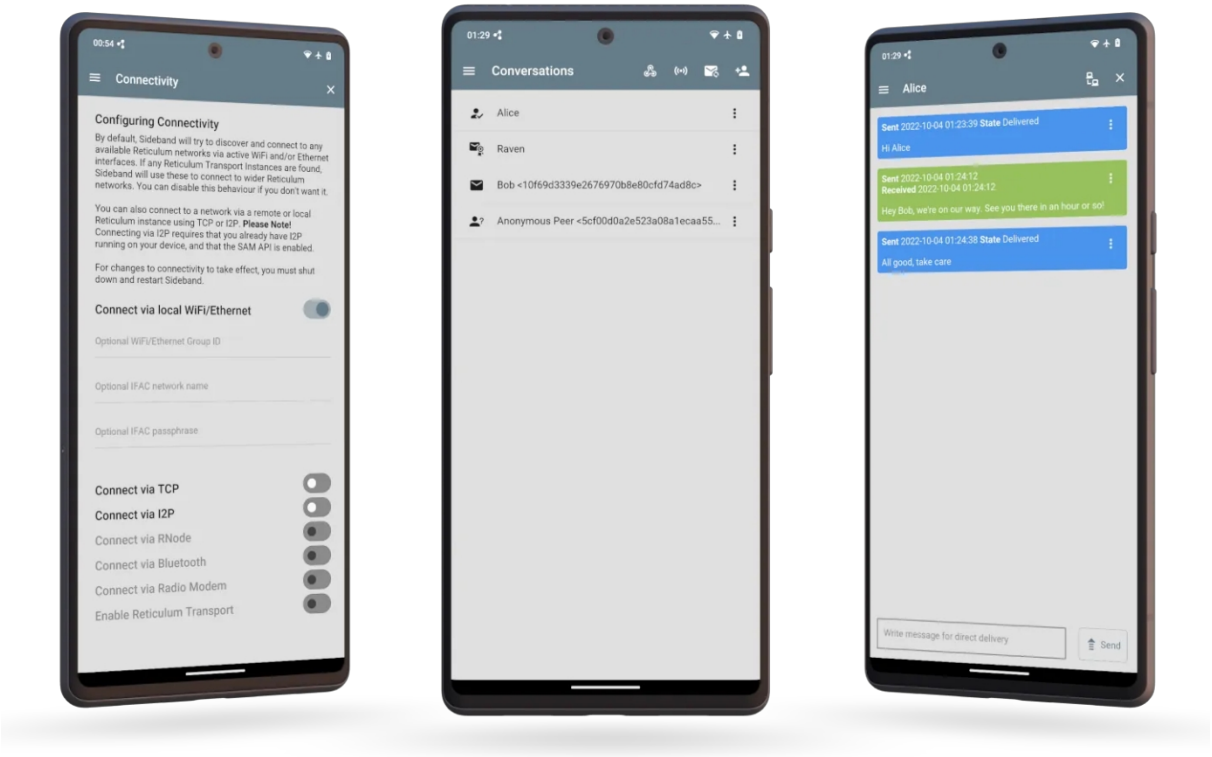
RNS Page Node [https://git.quad4.io/RNS-Things/rns-page-node] — это простой способ предоставлять страницы и файлы любому другому клиенту, совместимому с Nomad Network. Готовая замена узлов NomadNet, которые в основном обслуживают страницы и файлы.

Retipedia

Вы можете размещать всю Википедию (или любой файл `.zim`) для других клиентов Nomad Network с помощью Retipedia [https://github.com/RFnexus/Retipedia].

Sideband

Если вы предпочитаете использовать клиент LXMF с графическим интерфейсом, обратите внимание на Sideband [https://unsigned.io/sideband], доступный для Android, Linux, macOS и Windows. Sideband — это расширенный клиент LXMF и LXST, а также многофункциональная утилита Reticulum, с возможностями и функциональностью, ориентированными на опытных пользователей.



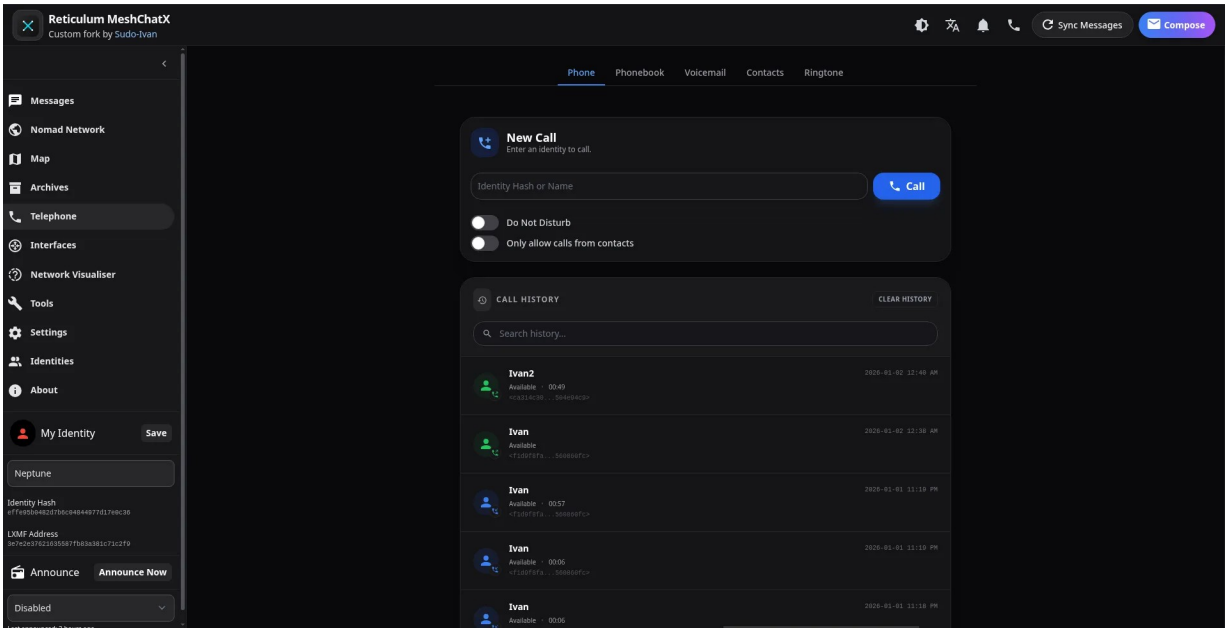
[https://unsigned.io/sideband]

Sideband позволяет вам общаться с другими людьми или системами, совместимыми с LXMF,
через сети Reticulum с использованием LoRa, Packet Radio, WiFi, I2P,
зашифрованных сообщений QR Paper или любых других средств, поддерживаемых Reticulum.

Он также взаимодействует со всеми другими клиентами LXMF и предоставляет расширенные возможности, такие как голосовые сообщения, голосовые звонки в реальном времени, вложения файлов, обмен частной телеметрией и полноценную систему плагинов для расширения функциональности.

MeshChatX

Форк Reticulum MeshChat из будущего [https://git.quad4.io/RNS-Things/MeshChatX], цель которого — предоставить все необходимое для работы с Reticulum, LXMF и LXST в одном красивом и многофункциональном приложении. Этот проект является отдельным от оригинального проекта Reticulum MeshChat и не связан с ним.

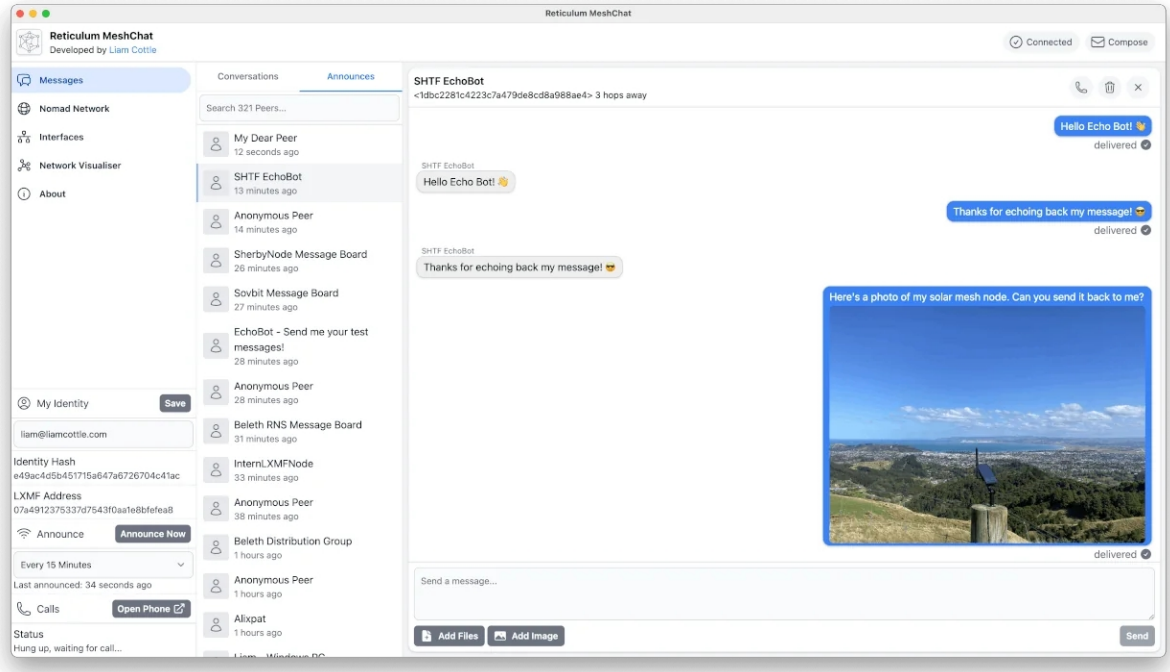


[https://git.quad4.io/RNS-Things/MeshChatX]

Функции включают полную поддержку LXST, настраиваемую голосовую почту, телефонную книгу, обмен контактами и поддержку мелодий звонка, работу с несколькими идентичностями, современный интерфейс и пользовательский опыт, офлайн-документацию, расширенные инструменты, архивирование страниц, интегрированные карты и улучшенную безопасность приложения.

MeshChat

Приложение Reticulum MeshChat [https://github.com/liamcottle/reticulum-meshchat] — это удобный клиент LXMF для Linux, macOS и Windows, который также включает в себя браузер страниц Nomad Network и другие интересные функции.



[https://github.com/liamcottle/reticulum-meshchat]

Reticulum MeshChat, конечно же, также совместим с Sideband и Nomad Network, а также с любыми другими клиентами LXMF.

Columba

Columba [https://github.com/torlando-tech/columba/] — это простое и привычное приложение для обмена сообщениями LXMF для Android, созданное с использованием нативного интерфейса Android и Material Design 3.



[https://github.com/torlando-tech/columba/]

Хотя приложение все еще находится на ранней стадии и активно разрабатывается, оно, конечно же, также совместимо со всеми другими клиентами LXMF и позволяет вам без проблем обмениваться сообщениями

с любым другим пользователем LXMF.

Reticulum Relay Chat

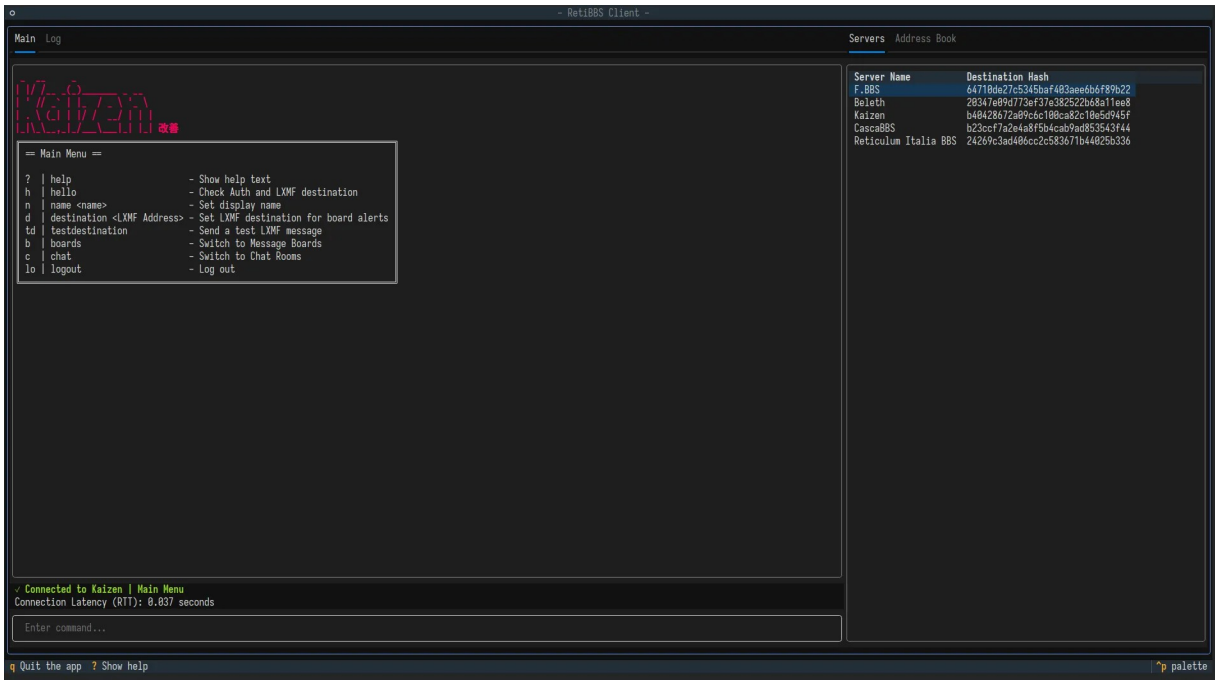
Reticulum Relay Chat [https://rrc.kc1awv.net/] — это система живого чата, построенная на основе сетевого стека Reticulum. Она существует для того, чтобы люди могли общаться друг с другом в реальном времени через Reticulum, не затягиваясь в базы данных сообщений, механизмы синхронизации или архитектурные обязательства, о которых они не просили.

Программа rrcd [https://github.com/kc1awv/rrcd] предоставляет функциональную эталонную реализацию демона-хаба RRC. Клиенты RRC включают rrc-gui [https://github.com/kc1awv/rrc-gui] и rrc-web [https://github.com/kc1awv/rrc-web].

RRC по духу ближе к IRC, чем к современным «универсальным платформам». Вы подключаетесь, входите в комнату, общаетесь, а затем уходите. Если вы были на месте, вы видели разговор. Если вас не было, разговор не ждал вас. Это не случайность. Это задумка проекта.

RetiBBS

RetiBBS [https://github.com/kc1awv/RetiBBS] — это реализация системы досок объявлений для сетей Reticulum.

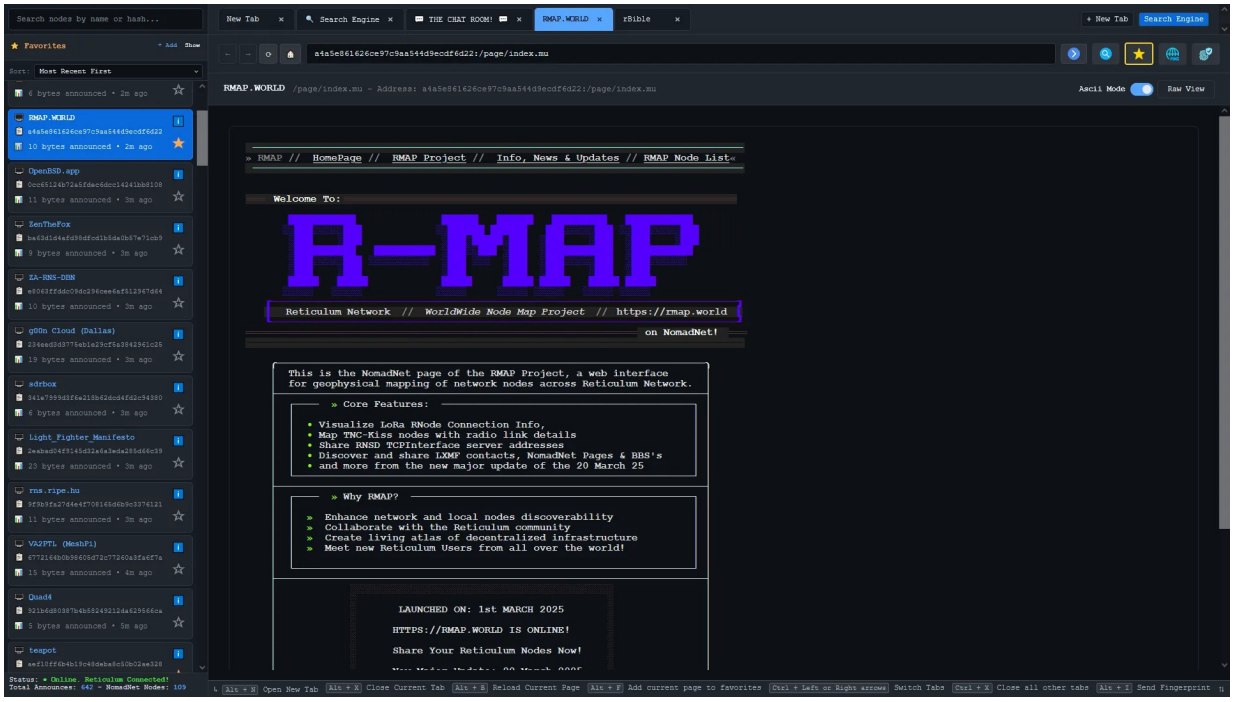


[https://github.com/kc1awv/RetiBBS]

RetiBBS позволяет пользователям безопасно общаться через доски объявлений.

RBrowser

Программа rBrowser [https://github.com/fr33n0w/rBrowser] — это кроссплатформенный, автономный веб-браузер для просмотра узлов NomadNetwork через сеть Reticulum. Он автоматически обнаруживает узлы NomadNet через сетевые объявления и предоставляет удобный интерфейс для просмотра распределенного контента с поддержкой разметки Micron.



[https://github.com/fr33n0w/rBrowser]

Включает в себя такие полезные функции, как автоматический прослушивание объявлений, добавление узлов в избранное, просмотр и отображение любых ссылок NomadNet, загрузка файлов с удаленных узлов, уникальный локальный поиск NomadNet
и многое другое.

Телефон Reticulum Network

Программа `rnphone`, входящая в состав пакета LXST

[https://github.com/markqvist/LXST], представляет собой утилиту и демон Reticulum телефонная утилита и демон для Reticulum, позволяющая создавать физические аппаратные
телефоны для LXST и Reticulum, а также просто совершать звонки через командную строку.

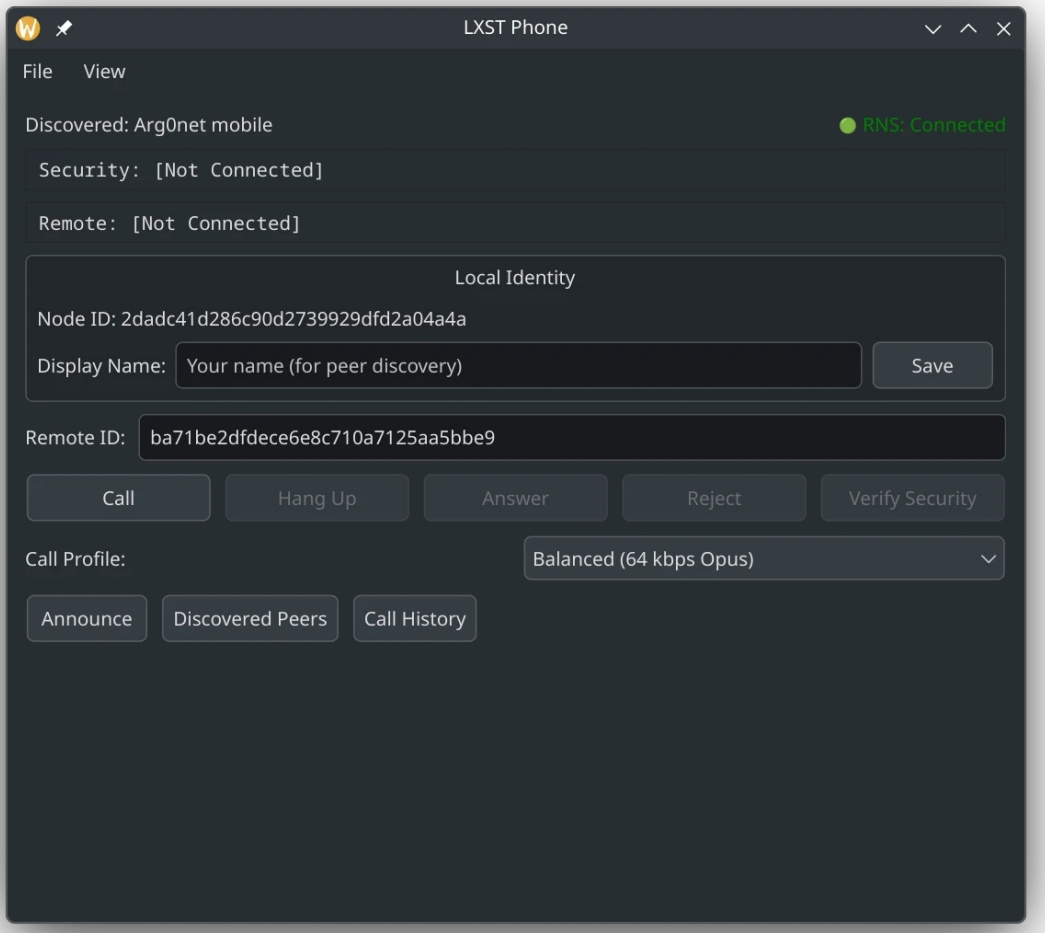


[https://github.com/markqvist/LXST]

Она поддерживает прямое взаимодействие с аппаратными периферийными устройствами, такими как клавиатуры GPIO
клавиатуры и ЖК-дисплеи, предоставляя модульную систему для создания безопасных аппаратных телефонов.

LXST Phone

Программа LXST Phone [https://github.com/kc1awv/lxst_phone] — это кроссплатформенное настольное приложение для осуществления голосовых вызовов LXST через Reticulum.



[https://github.com/kc1awv/lxst_phone]

Он поддерживает различные расширенные функции, такие как проверка SAS, блокировка одноранговых узлов, ограничение скорости, хранение зашифрованной истории вызовов и управление контактами.

LXMFy

LXMFy [https://lxmfy.quad4.io/] — это комплексная и продвинутая среда создания ботов для LXMF, позволяющая создавать любые системы автоматизации или ботов, работающие на базе LXMF и Reticulum. Реализации ботов доступны

[https://github.com/lxmfy/awesome-lxmfy-bots] для управления Home Assistant, интеграции с LLM и различных других целей.

Интерактивный клиент LXMF

LXMF Interactive Client [https://github.com/fr33n0w/lxmf-cli] — это многофункциональный терминальный клиент обмена сообщениями LXMF со множеством расширенных возможностей и расширяемой архитектурой плагинов.

RNS FileSync

Программа RNS FileSync [https://git.quad4.io/RNS-Things/RNS-Filesync] обеспечивает автоматическую синхронизацию файлов между устройствами без использования центральных серверов, подключения к Интернету или облачных сервисов. Она работает через любые сетевые среды, поддерживаемые Reticulum, включая радиосвязь, LoRa, Wi-Fi или Интернет, что делает её идеальным решением для обмена файлами в автономных сетях, с учетом требований конфиденциальности и обеспечением отказоустойчивости.

Micron Parser JS

Micron Parser JS [https://github.com/RFnexus/micron-parser-js] — это парсер на основе JavaScript для языка разметки Micron, который используется большинством веб-браузеров Nomad Network. Если вы хотите создавать утилиты или инструменты для отображения страниц Micron, эта библиотека вам просто необходима.

RNMon

RNMon [https://github.com/lbatalha/rnmon] — это мониторинговый демон, предназначенный для отслеживания состояния нескольких приложений RNS и отправки метрик в экземпляр InfluxDB по протоколу Influx Line.

Протоколы

В результате практического применения и тестирования в сообществе Reticulum сформировался ряд стандартных протоколов. Хотя при написании программного обеспечения на базе Reticulum вам иногда может потребоваться использовать полностью настраиваемые протоколы и реализации, применение этих протоколов предоставляет разработчикам приложений простой способ быстро и без лишних усилий реализовать расширенные функциональные возможности.
Их использование также обеспечивает совместимость и взаимодействие между множеством различных клиентских приложений, создавая открытую экосистему коммуникаций, в которой пользователи могут свободно выбирать приложения, отвечающие их потребностям, оставаясь при этом на связи со всеми остальными.

LXMF

LXMF [https://github.com/markqvist/lxmf] — это простой и гибкий формат обмена сообщениями и протокол доставки, который поддерживает широкий спектр приложений, потребляя при этом минимальную пропускную способность. Он обеспечивает маршрутизацию сообщений без настройки
, сквозное шифрование и секретность пересылки, и может передаваться по любому типу среды, поддерживаемому Reticulum.

LXMF достаточно эффективен, чтобы доставлять сообщения через системы с крайне низкой пропускной способностью, такие как пакетная радиосвязь или LoRa. Зашифрованные сообщения LXMF
также могут кодироваться в виде QR-кодов или текстовых URI, что позволяет осуществлять полностью аналоговую передачу сообщений на бумаге.

Используя узлы распространения, LXMF также предлагает способ хранения и пересылки сообщения пользователям или конечным точкам, недоступным на момент отправки сообщения.

LXST

LXST [https://github.com/markqvist/lxst] — это простой и гибкий формат потоковой передачи в реальном времени и протокол доставки, который поддерживает широкий спектр приложений, используя при этом минимально возможную пропускную способность. Он построен на базе Reticulum и предлагает маршрутизацию потоков без настройки, сквозное шифрование и конфиденциальность пересылки, а также может передаваться по любому типу среды, поддерживаемому Reticulum. В настоящее время он обеспечивает работу приложений голосовой связи и телефонии в реальном времени
по Reticulum.

RRC

Протокол Reticulum Relay Chat [https://rrc.kc1awv.net/] — это система чата, построенная на базе сетевого стека Reticulum. Она предназначена для обеспечения групповой коммуникации практически в реальном времени без использования баз данных истории сообщений, механизмов федерации или архитектурных сложностей.

RRC намеренно прост. Он не претендует на то, чтобы быть электронной почтой, почтовым ящиком или распределенным архивом. Он ведет себя скорее как разговор в комнате. Если вы
Вы были там — вы это слышали. Если вас там не было — вы этого не слышали. Это не ошибка, в этом и суть.

Модули интерфейсов и ресурсы по подключению

В этом разделе представлен список различных модулей интерфейсов, руководств и ресурсов, предоставленных сообществом, для создания сетей Reticulum через специальные или сложные среды.

- Пользовательский модуль интерфейса для работы RNS по HTTP [https://git.quad4.io/RNS-Things/RNS-over-HTTP]
- Руководство по запуску Reticulum через ICMP [https://github.com/matvik22000/rns-over-icmp] с использованием `PipeInterface`
- Руководство по запуску Reticulum через DNS [https://github.com/markqvist/Reticulum/discussions/1002] с Iodine
- Руководство по запуску Reticulum через HF-радио [https://github.com/RFnexus/reticulum-over-hf]
- Modem73 [https://github.com/RFnexus/modem73] — это интерфейс модема KISS TNC OFDM, который можно использовать с Reticulum

Использование Reticulum в вашей системе

Reticulum не устанавливается в качестве драйвера или модуля ядра, как это обычно бывает с сетевыми стеками. Вместо этого Reticulum распространяется в виде модуля Python, содержащего сетевое ядро, а также набор утилит и фоновых программ.

Это означает, что для его установки или использования не требуются специальные привилегии. Кроме того, он очень лёгкий, его легко переносить и устанавливать на новые системы.

Если у вас установлен Reticulum, любая программа или приложение, использующее Reticulum, автоматически загрузит и инициализирует Reticulum при запуске, если он еще не работает.

Во многих случаях этого подхода достаточно. Когда какой-либо программе требуется использовать Reticulum, он загружается, инициализируется, запускаются интерфейсы, и программа может теперь обмениваться данными через любые доступные сети Reticulum. Если запускается другая программа, которой также требуется доступ к той же сети Reticulum, просто используется уже запущенный экземпляр. Это работает для любого количества одновременно запущенных программ и очень просто в использовании, но в зависимости от вашего сценария использования существуют и другие варианты.

Конфигурация и данные

Reticulum хранит всю информацию, необходимую для работы, в одном каталоге файловой системы. При запуске Reticulum ищет каталог конфигурации в следующих местах:

- `/etc/reticulum`
- `~/.config/reticulum`
- `~/.reticulum`

Если существующий каталог конфигурации не найден, создается каталог `~/.reticulum`, и в нём автоматически формируется конфигурация по умолчанию. При желании вы можете переместить её в любое другое место.

Также можно использовать совершенно произвольные каталоги конфигурации, указав соответствующие параметры командной строки при запуске программ на базе Reticulum. Вы также можете запускать несколько отдельных экземпляров Reticulum на одной физической системе, либо изолированно друг от друга, либо соединенными между собой.

В большинстве случаев на одной физической системе потребуется запускать только один экземпляр Reticulum. Его можно запустить либо при загрузке в качестве системной службы, либо просто запустить, когда это потребуется программе. В любом случае любое количество
программ, запущенных на одной системе, автоматически будет использовать один и тот же экземпляр Reticulum, если это допускает конфигурация, что и происходит по умолчанию.

Вся конфигурация Reticulum находится в файле
Файл `~/.reticulum/config`. При первом запуске Reticulum в новой системе создаётся базовый, но полностью работоспособный файл конфигурации. Конфигурация по умолчанию выглядит следующим образом:

```
# Это файл конфигурации Reticulum по умолчанию.
# Вероятно, вам следует отредактировать его, чтобы добавить любые
дополнительные # интерфейсы и настройки, которые могут вам понадобиться.

# В эту конфигурацию по умолчанию включены только самые базовые # параметры.
Чтобы увидеть более подробный и гораздо более длинный
# пример конфигурации, вы можете выполнить команду: # rnsd --
exampleconfig


[reticulum]

# Если вы включите Transport, ваша система будет маршрутизировать трафик
# для других пиров, передавать объявления и обслуживать запросы на маршруты. #
Это следует делать для систем, которые подходят для работы
# в качестве транспортных узлов, т. е. если они стационарны и
# всегда включены. Эта директива является опциональной и может быть удалена #
для краткости.

enable_transport = No
```

```
# По умолчанию первая программа, запускающая Reticulum
# Сетевой стек создаст общий экземпляр, с которым смогут взаимодействовать другие
# программы. Только общий экземпляр # напрямую открывает все настроенные
интерфейсы, а другие
# локальные программы взаимодействуют с общим экземпляром через # локальный
сокет. Это полностью прозрачно для
# пользователя и, как правило, должна быть включена. Эта директива # является
опциональной и может быть удалена для краткости.

share_instance = Yes


# Если вы хотите запустить несколько *разных* общих экземпляров # в одной
системе, вам нужно будет указать разные
# имена экземпляров для каждого. На платформах, поддерживающих доменные
# сокетов, это можно сделать с помощью опции instance_name: instance_name = default

# Некоторые платформы не поддерживают доменные сокеты, и если # это так, вы
можете изолировать разные экземпляры,
# указав уникальный набор портов для каждого:

# shared_instance_port = 37428
# instance_control_port = 37429


# Если вы хотите явно использовать TCP для связи между экземплярами
# вместо доменных сокетов, это также # возможно с помощью следующей опции:

# shared_instance_type = tcp


# В системах, где запущенные экземпляры могут не иметь доступа # к одному и
тому же общему каталогу конфигурации Reticulum,
# все равно можно обеспечить полную интерактивность
# запущенных экземпляров, вручную указав общий ключ RPC
# ключ. Практически во всех случаях эта опция не требуется, но # она может быть
полезна в таких операционных системах, как Android.
# Ключ должен быть указан в виде байтов в шестнадцатеричном формате.

# rpc_key = e5c032d3ec4e64a6aca9927ba8ab73336780f6d71790
```

```
# Можно разрешить удаленное управление системами Reticulum # с помощью различных встроенных утилит, таких как
# rnstatus и rnpath. Вам потребуется указать один или
# хешей идентификаторов Reticulum для аутентификации
# запросов от клиентских программ. Для этой цели вы можете # использовать существующие файлы идентификаторов или сгенерировать новые с помощью
# утилиты rnid.

# enable_remote_management = yes
# remote_management_allowed = 9fb6d773498fb3feda407ed8ef2c3229, 2d882c5586e548d79b5af27bca1776dc


# Вы можете настроить Reticulum так, чтобы он выводил сообщение об ошибке и принудительно закрывался # в случае возникновения неустранимой ошибки интерфейса, например, # при исчезновении аппаратного устройства интерфейса. Это
# опциональная директива, и для краткости ее можно опустить. # По умолчанию это поведение отключено.

# panic_on_interface_error = No


# Когда Transport включен, можно разрешить # экземпляру Transport отвечать на запросы проверки от
# утилиты rnprobe. Это может быть полезным инструментом для проверки # подключения. Когда эта опция включена, пробник
# Адресат будет сгенерирован на основе идентификатора # экземпляра транспорта и выведен в журнал при запуске. # Необязательный параметр, по умолчанию отключен.

# respond_to_probes = No


[logging]
# Допустимые уровни журнала: от 0 до 7:
#     0: Записывать в журнал только критическую информацию
#     1: Регистрировать ошибки и события более низких уровней
#     2: Регистрировать предупреждения и события более низких уровней #     3: Регистрировать уведомления и уровни журнала ниже
#     4: Регистрировать информацию и ниже (это значение по умолчанию) #5: Подробная регистрация
```

```
#     6: Отладочная регистрация
#     7: Максимальный уровень

ведения журнала loglevel = 4


# Раздел interfaces определяет физические и виртуальные # интерфейсы, которые
Reticulum будет использовать для связи. Этот
# раздел будет содержать примеры для различных типов интерфейсов. Вы можете
изменить их или использовать в качестве основы для
# Свою собственную конфигурацию или просто удалите неиспользуемые. [interfaces]

  # Этот интерфейс обеспечивает связь с другими
  # узлами Reticulum с локальной связью по UDP. Для этого
  # не требует никакой функциональной IP-инфраструктуры, такой как маршрутизаторы
  # или серверов DHCP, но потребует, чтобы в вашей операционной системе был
  включен хотя бы локальный по каналу# IPv6, который # должен быть включен
  по умолчанию практически в любой ОС. См.
  # руководство Reticulum для получения информации о дополнительных параметрах настройки.

  [[Интерфейс по умолчанию]]
    type = AutoInterface
    interface_enabled = True
```

Если инфраструктура Reticulum уже существует локально, вам, вероятно, не нужно ничего менять, и вы, возможно, уже подключены к более широкой сети. Если нет, вам, вероятно, потребуется добавить соответствующие *интерфейсы* в конфигурацию, чтобы обмениваться данными с другими системами.

Вы можете сгенерировать гораздо более подробный пример конфигурации, выполнив команду:

```
rnsd --exampleconfig
```

В выводе приведены примеры для большинства типов интерфейсов, поддерживаемых Reticulum, а также дополнительные опции и параметры настройки.

Рекомендуется прочитать комментарии и пояснения в приведенной выше конфигурации по умолчанию. Это поможет вам освоить основные концепции, необходимые для
настроить сеть. Как только вы это сделаете, ознакомьтесь с главой «Интерфейсы» данного руководства.

Включенные утилиты

Reticulum включает ряд полезных утилит как для управления сетями Reticulum, так и для выполнения типичных задач в сетях Reticulum, таких как передача файлов на удаленные системы и удаленное выполнение
команд и программ удаленно.

Если вы часто используете Reticulum из нескольких разных программ или просто хотите, чтобы Reticulum оставался доступным постоянно, например, если вы являетесь хостом транспортного узла, вам, возможно, захочется запустить Reticulum в качестве отдельного сервиса, которым могут пользоваться другие программы, приложения и сервисы.

Утилита rnsd

Запустить Reticulum в качестве службы очень просто. Достаточно выполнить прилагаемую команду `rnsd`. Когда `rnsd` запущен, он поддерживает все настроенные интерфейсы в открытом состоянии, обрабатывает транспортные данные (если эта функция включена) и позволяет любым другим программам сразу же использовать сеть Reticulum, для которой он настроен.

Вы даже можете запустить несколько экземпляров `rnsd` с разными настройками на одной системе.

Примеры использования

Запустите `rnsd`:

```
$ rnsd

[2023-08-18 17:59:56] [Уведомление] Запущен rnsd версии 0.5.8
```

Запуск `rnsd` в режиме службы с отправкой всего вывода журнала непосредственно в файл:

```
$ rnsd -s
```

Сгенерируйте подробный пример конфигурации с объяснениями всех различных параметров конфигурации и примерами настройки интерфейса:

```
$ rnsd --exampleconfig
```

Все параметры командной строки

```
Использование: rnsd.py [-h] [--config CONFIG] [-v] [-q] [-s] [--
exampleconfig] [--version]

Параметры демона сетевого стека Reticulum:
  -h, --help            показать это сообщение справки и завершить работу
  --config CONFIG       путь к альтернативному каталогу конфигурации
Reticulum
  -v, --verbose
  -q, --quiet
  -s, --service         rnsd запущен как служба и должен вести журнал в файл
  -i, --interactive переход в интерактивную оболочку после инициализации
  --exampleconfig       вывести подробный пример конфигурации в stdout и
завершить работу
  --version             показать номер версии программы и завершить работу
```

Вы можете легко добавить `rnsd` в качестве постоянно работающей службы, настроив службу.

Утилита rnstatus

С помощью утилиты `rnstatus` можно просматривать состояние настроенных интерфейсов Reticulum, аналогично программе `ifconfig`.

Примеры использования

Запустите `rnstatus` :

```
$ rnstatus

Совместный экземпляр [37428]
    Статус: Работает
    Обслуживание:         1
    программа     Скорость:
    1,00   Гбит/с   Трафик:
    83,13 КБ↑
                86,10 КБ↓

AutoInterface[Local]
    Статус      : Активен
    Режим       : Полный
    Скорость    : 10,00 Мбит/с
    Узлы        : 1 доступен
    Трафик : 63,23 КБ↑
                80,17 КБ↓

TCPInterface[RNS Testnet Dublin/dublin.connect.reticulum.network:4965]
    Статус      :
    Работает Режим   :
    Полный
    Скорость    : 10,00 Мбит/с
    Трафик : 187,27 КБ↑
                74,17 КБ↓

Состояние RNodeInterface[RNode
    UHF]        : Активен
    Режим       : Скорость точки
    доступа     : 1,30 кбит/с
    Доступ      : 64-битная IFAC от <…e702c42ba8>
    Трафик : 8,49 КБ↑
                9,23 КБ↓

Экземпляр Reticulum Transport <5245a8efe1788c6a1cd36144a270e13b> работает
```

Отфильтруйте вывод, чтобы отобразить только некоторые интерфейсы:

```
$ rnstatus rnode RNodeInterface[RNode

UHF]
    Статус      : Активен
    Режим       : Точка доступа
```

```
    Скорость    : 1,30 кбит/с
    Доступ      : 64-битная IFAC от <…e702c42ba8>
    Трафик : 8,49 КБ↑
                9,23 КБ↓
```

Экземпляр `Reticulum Transport <5245a8efe1788c6a1cd36144a270e13b>` работает

Все параметры командной строки

```
Использование: rnstatus [-h] [--config CONFIG] [--version] [-a] [-A]
                        [-l] [-t] [-s SORT] [-r] [-j] [-R hash] [-i
path]
                        [-w секунды] [-d] [-D] [-m] [-I секунды] [-v]
[filter]
```

Состояние сетевого стека `Reticulum`

позиционные аргументы:

фильтр	отображать только интерфейсы, имена которых содержат фильтр

опции:

`-h, --help`	показать это сообщение справки и выйти
`--config CONFIG`	путь к альтернативному каталогу конфигурации `Reticulum`
`--version`	показать номер версии программы и завершить работу
`-a, --all`	показать все интерфейсы
`-A, --announce-stats`	показать статистику анонсов
`-l, --link-stats`	показать статистику каналов
`-t, --totals`	отобразить итоговые данные о трафике
`-s, --sort` СОРТИРОВКА	отсортировать интерфейсы по [скорости, трафику, `rx, tx, rxs, txs,` объявления, `arx, atx, held]`
`-r, --reverse`	обратная сортировка
`-j, --json`	вывод в формате `JSON`
`-R` хеш	транспортный идентификационный хеш удаленного экземпляра, от которого нужно получить статус
`-i` путь	путь к идентификатору, используемому для удаленного управления
`-w` секунд	таймаут перед сдачей удаленного запроса

```
запросов
   -d, --discovered            список обнаруженных интерфейсов
   -D                          показать подробности и записи конфигурации для
обнаруженных интерфейсов
   -m, --monitor               непрерывно отслеживать состояние
   -I, --monitor-interval секунд
                               интервал обновления для режима мониторинга
(по умолчанию: 1)
   -v, --verbose
```

Примечание

При использовании параметра `-R` для запроса удаленного экземпляра транспортного уровня необходимо также указать параметр `-i` с путем к файлу идентификационных данных, авторизованному для удаленного управления в целевой системе.

Утилита rnid

С помощью утилиты `rnid` можно создавать, управлять и просматривать идентификаторы Reticulum. Программа также может вычислять хэши назначения и выполнять шифрование и дешифрование файлов.

С помощью `rnid` можно осуществлять асимметричное шифрование файлов и информации для любого целевого хеша Reticulum, а также создавать и проверять криптографические подписи.

Примеры использования

Создание новой идентичности:

```
$ rnid -g ./new_identity
```

Отображение информации о ключе идентичности:

```
$ rnid -i ./new_identity -p

Загружена идентичность <984b74a3f768bef236af4371e6f248cd> из открытого ключа
new_id         :
```

```
0f4259fef4521ab75a3409e353fe9073eb10783b4912a6a9937c57bf44a62c1 e
Частный ключ: скрыт
```

Зашифровать файл для пользователя LXMF:

```
$ rnid -i 8dd57a738226809646089335a6b03695 -e my_file.txt

Восстановлена идентичность <bc7291552be7a58f361522990465165c> для адресата
<8dd57a738226809646089335a6b03695>
Шифрование файла my_file.txt
Файл my_file.txt зашифрован для
<bc7291552be7a58f361522990465165c> в my_file.txt.rfe
```

Если идентификатор назначения еще не известен, его можно получить из сети с помощью опции командной строки `-R`:

```
$ rnid -R -i 30602def3b3506a28ed33db6f60cc6c9 -e my_file.txt Запрос неизвестного

идентификатора для
<30602def3b3506a28ed33db6f60cc6c9>...
Получено идентификатор <2b489d06eaf7c543808c76a5332a447d> для адреса
<30602def3b3506a28ed33db6f60cc6c9> из сети Шифрование файла my_file.txt
Файл my_file.txt зашифрован для
<2b489d06eaf7c543808c76a5332a447d> в my_file.txt.rfe
```

Расшифровать файл с помощью идентификатора Reticulum, для которого он был зашифрован:

```
$ rnid -i ./my_identity -d my_file.txt.rfe

Загружена идентичность <2225fdeecaf6e2db4556c3c2d7637294> из
./my_identity
Расшифровка ./my_file.txt.rfe...
Файл ./my_file.txt.rfe расшифрован с помощью
<2225fdeecaf6e2db4556c3c2d7637294> в ./my_file.txt
```

Все параметры командной строки

```
Использование: rnid.py [-h] [--config путь] [-i идентификатор] [-g путь] [-v] [-
q] [-a аспекты]
```

```
                    [-H аспекты] [-e путь] [-d путь] [-s путь] [-V путь] [-r путь]
[-w путь]
                    [-f] [-R] [-t секунды] [-p] [-P] [--version]

Параметры утилиты Reticulum Identity & Encryption:
  -h, --help                показать это сообщение справки и выйти
  --config путь             путь к альтернативному каталогу конфигурации
Reticulum
  -i, --identity identity
                            шестнадцатеричный идентификатор Reticulum или хеш
назначения, либо путь к файлу Identity
  -g, --generate файл       создать новый идентификатор
  -m, --import identity_data
                            импорт идентификатора Reticulum в шестнадцатеричном,
формате base32 или base64
  -x, --export              экспортировать идентификатор в
шестнадцатеричном, base32 или base64 формате
  -v, --verbose             увеличить подробность
  -q, --quiet               уменьшить уровень подробности
  -a, --announce аспекты
                            объявить пункт назначения на основе этого
Идентичность
  -H, --hash aspects        показать хэши назначения для других аспектов
данной идентичности
  -e, --encrypt файл        зашифровать файл
  -d, --decrypt файл        расшифровать файл
  -s, --sign путь           подписать файл
  -V, --validate путь       проверить подпись
  -r, --read файл           путь к входному файлу
  -w, --write файл          путь к выходному файлу
  -f, --force               записать вывод, даже если это приведет к
перезаписи существующих файлов
  -R, --request             запрашивать неизвестные идентификаторы из сети
  -t секунд                 таймаут запроса идентификатора перед сдачей
  -p, --print-identity      вывести информацию об идентификаторе и завершить работу
  -P, --print-private       разрешить отображение закрытых ключей
  -b, --base64              Использовать входные и выходные данные, закодированные в
  base64
```

-B, --base32	Использовать входные и выходные данные, закодированные в base32
--version	показать номер версии программы и завершить работу

Утилита rnpath

С помощью утилиты `rnpath` можно искать и просматривать пути к конечным точкам в сети Reticulum.

Примеры использования

Определение пути к пункту назначения:

```
$ rnpath c89b4da064bf66d280f0e4d8abfd9806
```

Путь найден, пункт назначения <c89b4da064bf66d280f0e4d8abfd9806> находится на расстоянии 4 прыжков через <f53a1c4278e0726bb73fcc623d6ce763> на TCPInterface[Testnet/dublin.connect.reticulum.network:4965]

Все параметры командной строки

```
Использование: rnpath [-h] [--config CONFIG] [--version] [-t] [-m hops] [-r]
[-d] [-D]
                  [-x] [-w секунд] [-R хеш] [-i путь] [-W
сек] [-b] [-B] [-U]
                  [--duration DURATION] [--reason REASON] [-p] [-j]
[-v]
                  [destination] [list_filter]

Утилита управления путями Reticulum: позиционные

аргументы:
  пункт_назначения      шестнадцатеричный хеш пункта назначения
  фильтр_списка         фильтр для просмотра удаленного списка blackhole

опции:
  -h, --help            показать это сообщение справки и выйти
  --config CONFIG       путь к альтернативному каталогу конфигурации
Reticulum
  --version             показать номер версии программы и завершить работу
  -t, --table           показать все известные пути
```

```
  -m, --max hops            максимальное количество переходов для фильтрации таблицы
  путей
  -r, --rates               показать информацию о скорости передачи
  -d, --drop                удалить путь к месту назначения
  -D, --drop-announces      удалить все объявления из очереди
  -x, --drop-via            отбросить все пути через указанный экземпляр транспорта
  -w секунд                 таймаут перед сдачей
  -R хеш                    хэш идентификатора транспорта удаленного
экземпляра для управления
  -i путь                   путь к идентификатору, используемому для
удаленного управления
  -W секунд                 таймаут перед отказом от удаленных запросов
  -b, --blackholed          список идентификаторов, помещенных в черный список
  -B, --blackhole           заблокировать идентификатор
  -U, --unblackhole         снять с идентификатора статус «blackhole»
  --duration DURATION       продолжительность действия blackhole в часах
  --reason ПРИЧИНА          причина отправки идентификатора в черную дыру
  -p, --blackholed-list
                            просмотр опубликованного списка blackhole для
удаленного экземпляра транспорта
  -j, --json                вывод в формате JSON
  -v, --verbose
```

Утилита rnprobe

Утилита `rnprobe` позволяет проверить доступность адресата, аналогично программе `ping`. Обратите внимание, что ответы на запросы будут поступать только в том случае, если для указанного адресата настроена отправка подтверждений о получении пакетов.
У многих адресатов эта опция не включена, поэтому большинство адресатов, скорее всего, не будут отвечать.

Вы можете включить функцию ответа на проверки для адреса в экземплярах транспорта Reticulum, установив конфигурационную директиву `respond_to_probes`. После этого Reticulum выведет адрес проверки в журнал при запуске экземпляра транспорта.

Примеры использования

Проверка адреса:

```
$ rnprobe rnstransport.probe 2d03725b327348980d570f739a3a5708 Отправлен 16-

байтовый запрос на <2d03725b327348980d570f739a3a5708>
Получен действительный ответ от <2d03725b327348980d570f739a3a5708>
Время прохождения в оба конца составляет 38,469 миллисекунд через 2 прыжка
```

Отправить более крупный запрос:

```
$ rnprobe rnstransport.probe 2d03725b327348980d570f739a3a5708 - s 256

Отправлен 16-байтовый пробный пакет на <2d03725b327348980d570f739a3a5708>
Получен действительный ответ от <2d03725b327348980d570f739a3a5708>
Времяпрохождения составляет 38,781 миллисекунд через 2 прыжка
```

Если интерфейс, получающий ответы на запрос, поддерживает отчетность по радио такие параметры, как **RSSI** и **SNR**, утилита `rnprobe` также выведет их в составе результата.

```
$ rnprobe rnstransport.probe e7536ee90bd4a440e130490b87a25124 Отправлен 16-

байтовый запрос на <e7536ee90bd4a440e130490b87a25124>
Получен действительный ответ от <e7536ee90bd4a440e130490b87a25124>
Время прохождения в оба конца составляет 1,809 секунды через 1 прыжок [RSSI -73 дБм]
[SNR
12,0 дБ]
```

Все параметры командной строки

```
Использование: rnprobe [-h] [--config CONFIG] [-s SIZE] [-n PROBES] [-t
                  секунды] [-w секунды] [--version] [-v] [full_name]
                  [destination_hash]

Утилита Reticulum Probe позиционные

аргументы:
  full_name                   полное имя назначения в точечной
формате
  destination_hash            шестнадцатеричный хеш пункта назначения опции:
```

```
  -h, --help                  показать это сообщение справки и выйти
  --config CONFIG             путь к альтернативному каталогу конфигурации
Reticulum
  -s SIZE, --size SIZE        размер полезной нагрузки пакета зонда в байтах
  -n PROBES, --probes PROBES
                              количество отправляемых зондов
  -t секунд, --timeout секунд
                              время ожидания перед сдачей
  -w секунд, --wait секунд
                              время между каждым запросом
  --version                   показать номер версии программы и завершить работу
  -v, --verbose
```

Утилита rncp

Утилита `rncp` — это простой инструмент для передачи файлов. С его помощью можно передавать файлы через Reticulum.

Примеры использования

Запустите rncp на принимающей системе, указав, каким идентификаторам разрешено отправлять файлы:

```
$ rncp --listen -a 1726dbad538775b5bf9b0ea25a4079c8 -a
c50cc4e4f7838b6c31f60ab9032cbc62
```

Вы также можете указать разрешенные хеши идентификаторов (по одному в строке) в файле
~/.rncp/allowed_identities и просто запустив программу в режиме прослушивания:

```
$ rncp --listen
```

С другой системы скопируйте файл на принимающую систему:

```
$ rncp ~/path/to/file.tgz 73cbd378bb0286ed11a707c13447bb1e
```

Или загрузите файл с удаленной системы:

```
$ rncp --fetch ~/path/to/file.tgz 73cbd378bb0286ed11a707c13447bb1e
```

Файл идентификационных данных по умолчанию хранится в папке `~/.reticulum/identities/rncp`, но вы можете использовать другой файл, который будет создан, если он ещё не существует

```
$ rncp ~/path/to/file.tgz 73cbd378bb0286ed11a707c13447bb1e -i
/path/to/identity
```

Все параметры командной строки

```
Использование: rncp [-h] [--config путь] [-v] [-q] [-S] [-l] [-F] [-f] [-j
               путь] [-b секунды] [-a допустимый_хэш] [-n] [-p] [-i
               идентичность] [-w секунды] [--version] [файл]
[путь_назначения]

Утилита передачи файлов Reticulum

позиционные аргументы:
  файл                  файл для передачи
  путь                  шестнадцатеричный хеш получателя

опции:
  -h, --help            показать это сообщение справки и выйти
  --config путь         путь к альтернативному каталогу конфигурации
Reticulum
  -v, --verbose         увеличить уровень подробности
  -q, --quiet           уменьшить подробность
  -S, --silent          отключить вывод информации о ходе передачи
  -l, --listen          прослушивать входящие запросы на передачу
  -C, --no-compress     отключить автоматическое сжатие
  -F, --allow-fetch     разрешить аутентифицированным клиентам загружать
файлы
  -f, --fetch           загрузить файл с удаленного прослушивателя вместо
отправки
  -j, --jail путь       ограничить запросы на получение файлов указанным
путем
  -s, --save путь       сохранить полученные файлы в указанном пути
  -O, --overwrite       Разрешить перезапись полученных файлов вместо
добавления суффикса
  -b секунд             интервал объявления, 0 — объявлять только при
запуске
```

-a разрешенный_хэш	разрешить эту идентичность (или добавить в ~/.rncp/allowed_identities)
-n, --no-auth	принимать запросы от любого пользователя
-p, --print-identity	вывести информацию об идентичности и адресате и завершить работу
-i идентификатор	путь к идентификатору, который нужно использовать
-w секунд	таймаут отправителя перед сдачей
-P, --phy-rates	отобразить скорости передачи данных на физическом уровне
--version	показать номер версии программы и завершить работу

Утилита rnx

Утилита `rnx` — это простая программа для удалённого выполнения команд. Она позволяет запускать команды на удалённых системах через Reticulum и просматривать вывод команд. Если вам нужна полностью интерактивная удалённая оболочка, обязательно ознакомьтесь также с программой rnsh [https://github.com/acehoss/rnsh].

Примеры использования

Запустите rnx на системе-слушателе, указав, каким идентификаторам разрешено выполнять команды:

```
$ rnx --listen -a 941bed5e228775e5a8079fc38b1ccf3f -a
1b03013c25f1c2ca068a4f080b844a10
```

С другой системы запустите команду на удаленном компьютере:

```
$ rnx 7a55144adf826958a9529a3bcf08b149 "cat /proc/cpuinfo"
```

Или войдите в псевдо-оболочку интерактивного режима:

```
$ rnx 7a55144adf826958a9529a3bcf08b149 -x
```

Файл идентификации по умолчанию хранится в `~/.reticulum/identities/rnx`, но вы можете использовать другой, который будет создан, если он еще не существует

```
$ rnx 7a55144adf826958a9529a3bcf08b149 -i /path/to/identity -x
```

Все параметры командной строки

```
Использование: rnx [-h] [--config путь] [-v] [-q] [-p] [-l] [-i идентификатор] [-
x] [-b] [-n] [-N]
              [-d] [-m] [-a allowed_hash] [-w секунд] [-W секунд] [--stdin
STDIN]
              [--stdout STDOUT] [--stderr STDERR] [--version] [destination]
[command]

Позиционные аргументы утилиты удаленного выполнения

Reticulum:
  destination                 шестнадцатеричный хеш слушателя
  команда                     команда для выполнения

дополнительные аргументы:
  -h, --help                  показать это сообщение справки и выйти
  --config путь               путь к альтернативному каталогу конфигурации
Reticulum
  -v, --verbose               увеличить уровень подробности
  -q, --quiet                 уменьшить подробность
  -p, --print-identity        вывести информацию об идентификаторе и пункте
назначения и завершить работу
  -l, --listen                прослушивать входящие команды
  -i идентификатор            путь к идентификатору, который нужно использовать
  -x, --interactive           перейти в интерактивный режим
  -b, --no-announce           не выводить сообщение при запуске программы
  -a разрешенный_хэш          принимать от этого идентификатора
  -n, --noauth                принимать файлы от любого
  -N, --noid                  не идентифицироваться перед слушателем
  -d, --detailed              показать подробный вывод результатов
  -m                          отразить код завершения удаленной команды
  -w секунд                   время ожидания подключения и запроса перед
сдачей
  -W секунд                   максимальное время загрузки результата
  --stdin STDIN               передать входные данные в stdin
  --stdout STDOUT             максимальный размер в байтах возвращаемого stdout
  --stderr STDERR             максимальный размер в байтах возвращаемого stderr
  --version                   показать номер версии программы и завершить работу
```

Утилита rnodeconf

Утилита `rnodeconf` позволяет просматривать и настраивать существующие RNodes, а также создавать и подготавливать новые RNodes на любом поддерживаемом аппаратном устройстве.

Все параметры командной строки

```
Использование: rnodeconf [-h] [-i] [-a] [-u] [-U] [--fw-version версия]
                         [--fw-url url] [--nocheck] [-e] [-E] [-C]
                         [--baud-flash скорость_передачи] [-N] [-T] [-b] [-B]
[-p] [-D i]
                         [--display-addr байт] [--freq Гц] [--bw Гц] [-
-txp дБм]
                         [--sf коэффициент] [--cr скорость] [--eeprom-backup] [-
-eeprom-dump]
                         [--eeprom-wipe] [-P] [--trust-key шестнадцатеричные байты] [-
-version] [-f]
                         [-r] [-k] [-S] [-H FIRMWARE_HASH] [--platform
платформа] ревизия]
                         [--product product] [--model model] [--hwrev [port]

Утилита настройки и прошивки RNode. Эта программа позволяет изменять
различные настройки и режимы запуска RNode. Она также может устанавливать, прошивать и
обновлять прошивку на поддерживаемых устройствах.

позиционные аргументы:
  port                        последовательный порт, к которому подключен RNode

опции:
  -h, --help                  показать это сообщение справки и выйти
  -i, --info                  Показать информацию об устройстве
  -a, --autoinstall           Автоматическая установка на различных
поддерживаемых устройствах
  -u, --update                Обновление прошивки до последней версии
  -U, --force-update          Обновление до указанной прошивки, даже если
версия совпадает с установленной или старше ее
  --fw-version версия         Использовать конкретную версию прошивки для
обновления или автоматической установки
```

--fw-url url Использовать альтернативный URL-адрес для загрузки прошивки
--nocheck Не проверять наличие обновлений прошивки в Интернете
-e, --extract Извлечь прошивку из подключенного RNode для последующего использования
-E, --use-extracted Использовать извлеченную прошивку для автоматической установки или обновления
-C, --clear-cache Очистить локально кэшированные файлы прошивки
--baud-flash baud_flash
Установить определенную скорость передачи данных при прошивке устройства. По умолчанию — 921600
-N, --normal Переключить устройство в нормальный режим
-T, --tnc Переключить устройство в режим TNC
-b, --bluetooth-on Включить Bluetooth на устройстве
-B, --bluetooth-off Отключить Bluetooth на устройстве
-p, --bluetooth-pair Перевести устройство в режим сопряжения Bluetooth
-D, --display i Установить яркость дисплея (0–255)
-t, --timeout s Установить таймаут дисплея в секундах, 0 для отключения
-R, --rotation rotation
Установить поворот дисплея, допустимые значения: от 0 до 3
--display-addr байт Установить адрес дисплея в виде шестнадцатеричного байта (00–FF)
--recondition-display
Запуск перенастройки дисплея
--np i Установить интенсивность NeoPixel (0–255)
--freq Гц Частота в Гц для режима TNC
--bw Гц Ширина полосы в Гц для режима TNC
--txp дБм Мощность передачи в дБм для режима TNC
--sf коэффициент Коэффициент растяжения для режима TNC (7–12)
--cr rate Скорость кодирования для режима TNC (5–8)
-x, --ia-enable Включить избегание помех
-X, --ia-disable Отключить предотвращение интерференции
-c, --config Вывести конфигурацию устройства
--eeprom-backup Резервное копирование EEPROM в файл
--eeprom-dump Вывести содержимое EEPROM в консоль
--eeprom-wipe Разблокировать и очистить EEPROM
-P, --public Отобразить открытую часть ключа подписи
--trust-key шестнадцатеричные байты Открытый ключ, которому доверяют для проверки устройства
--version Вывести версию программы и завершить работу
-f, --flash Записать прошивку и загрузочную EEPROM
-r, --rom Загрузочная EEPROM без перепрошивки

прошивки
   -k, --key   Сгенерировать новый ключ подписи и завершить работу
   -S, --sign   Отобразить открытую часть ключа подписи
   -H, --firmware-hash FIRMWARE_HASH
                                Установить хеш установленной прошивки
   --platform platform   Спецификация платформы для начальной загрузки
устройства
   --product product   Спецификация продукта для начальной загрузки
устройства
   --model модель   Код модели для загрузки устройства
   --hwrev версия   Ревизия аппаратного обеспечения для загрузочного модуля
   устройства

Для получения дополнительной информации о том, как создавать собственные RNodes, ознакомьтесь с разделом «Создание RNodes» данного руководства.

Обнаружение интерфейсов

Reticulum включает в себя встроенные средства для обнаружения подключаемых через сам Reticulum. Это особенно полезно в ситуациях, когда вам нужно выполнить одно или несколько из следующих действий:

- Обнаружить доступные в Интернете точки входа
- Найти доступные точки доступа в реальном мире
- Поддерживать связь с экземплярами RNS с неизвестными или меняющимися IP-адресами

Обнаруженные интерфейсы могут быть **автоматически** подключены Reticulum, что позволяет создавать конфигурации, в которых произвольный интерфейс может выступать просто в качестве начального соединения, которое можно разорвать, как только будут обнаружены и подключены более подходящие
.

Механизм обнаружения интерфейсов использует объявления, отправляемые через саму Reticulum, и поддерживает как общедоступные интерфейсы, так и частное, зашифрованное обнаружение, которое может быть декодировано только указанными *сетевыми идентичностями*. Также можно указать, какие сетевые идентичности следует считать достоверными источниками для обнаруженных интерфейсов, чтобы интерфейсы, опубликованные неизвестными субъектами, игнорировались.

Примечание

Сетевая идентичность — это обычный набор ключей идентичности Reticulum, который может использоваться одним или несколькими транспортными узлами для идентификации своей принадлежности к одной общей сети. В контексте обнаружения интерфейсов это упрощает управление подключением только к тем конкретным сетям, которые вас интересуют, даже если эти сети используют множество отдельных физических транспортных узлов.

Это также позволяет удобно настраивать автоматическое подключение обнаруженных интерфейсов только к тем сетям, к которым у вас есть определенная степень доверия.

Информацию о том, как сделать ваши интерфейсы обнаруживаемыми, см. в главе «Обнаруживаемые интерфейсы» данного руководства. В данном разделе будет посвящён тому, как *находить и подключаться к* интерфейсам, доступным в сети.

В самом простом случае включение обнаружения интерфейсов сводится к установке значения `discover_interfaces` в `true` в конфигурации Reticulum:

```
[reticulum]
...
discover_interfaces = yes
...
```

Как только этот параметр будет включен, ваш экземпляр RNS начнет прослушивать объявления об обнаружении интерфейсов и сохранять их для последующего использования или проверки. Вы можете отобразить список обнаруженных интерфейсов с помощью утилиты `rnstatus`:

```
$ rnstatus -d

Имя                 Тип          Состояние       Последнее       Значение
Расположение                                     обнаружение
-------------------------------------------------------------------------------
Концентратор        Магистраль   ✓ Доступно      1 ч  назад      16
боковых полос
46.2316, 6.0536
RNS Амстердам       Магистральная сеть ✓ Доступно  32 мин. назад  16
52.3865, 4.9037
```

Вы можете просмотреть более подробную информацию об обнаруженных интерфейсах, включая фрагменты конфигурации для непосредственного вставки в ваш файл конфигурации `[interfaces]`, с помощью опции `rnstatus -D`:

```
$ rnstatus -D sideband

Идентификатор транспорта : 521c87a83afb8f29e4455e77930b973b
Название           : Sideband Hub
Тип                : BackboneInterface
Состояние          : Доступно Транспорт
                   : Включено Расстояние
                   : 2 прыжка
Обнаружено         : 9 ч 40 мин назад
Последнее обнаружение      : 1 час и 15
минут назад
Местоположение     : 46.2316, 6.0536
Адрес              : sideband.connect.reticulum.network:7822 Значение
отметки            : 16

Запись конфигурации:
  [[Sideband Hub]]
    тип = BackboneInterface включено =
    да
    remote = sideband.connect.reticulum.network
    target_port = 7822
    transport_identity = 521c87a83afb8f29e4455e77930b973b
```

Помимо предоставления информации об обнаружении локальных интерфейсов и управления ими, утилита `rnstatus` может экспортировать данные об обнаруженных интерфейсах в машино-
в удобном для чтения формате JSON с помощью опции `rnstatus -d --json`. Это может быть полезно для экспорта данных во внешние приложения, такие как страницы состояния,
карты точек доступа и т. п.

Чтобы контролировать, какие источники считаются допустимыми для обнаружения, для системы обнаружения интерфейсов можно указать дополнительные параметры конфигурации.

- Опция `interface_discovery_sources` представляет собой список сетевых или транспортных идентификаторов, от которых будут приниматься интерфейсы. Если эта опция задана, все остальные будут игнорироваться. Если эта опция не задана,

обнаруженные интерфейсы будут приниматься из любого источника, но по-прежнему должны соответствовать требованиям к значению метки.

- Параметр `required_discovery_value` указывает минимальное значение отметки, необходимое для того, чтобы объявление интерфейса считалось действительным. Чтобы сделать вычислительно сложным спам в сети с большим количеством неработающих или вредоносных интерфейсов, каждый объявленный интерфейс
  требуется действительная криптографическая метка с настраиваемым значением сложности.
- Значение параметра `autoconnect_discovered_interfaces` по умолчанию равно `0` и определяет максимальное количество обнаруженных интерфейсов, которые должны быть подключены автоматически в любой момент времени. Если для этого параметра задано значение больше `0`, Reticulum автоматически управляет подключениями обнаруженных интерфейсов и подключает или отключает их в зависимости от доступности. Вы можете в любой момент вручную добавить обнаруженные интерфейсы в свою конфигурацию, чтобы обеспечить их постоянную доступность.
- Параметр `network_identity` указывает *сетевую идентичность* для данного экземпляра RNS. Эта идентичность используется как для подписи (и, возможно, шифрования) *исходящих* объявлений об обнаружении интерфейсов, так и для расшифровки входящей информации об обнаружении.

Приведенный ниже фрагмент конфигурации содержит пример настройки этих дополнительных параметров:

```
[reticulum]
...
discover_interfaces = yes
interface_discovery_sources = 521c87a83afb8f29e4455e77930b973b
required_discovery_value = 16
autoconnect_discovered_interfaces = 3
идентификатор_сети = ~/.reticulum/storage/identities/my_network
...
```

Удаленное управление

Можно разрешить удаленное управление системами Reticulum с помощью различных встроенных утилит, таких как `rnstatus` и `rnpath`. Для этого необходимо установить директиву `enable_remote_management` в разделе `[reticulum]` файла конфигурации. Также необходимо указать один или несколько

Reticulum использует хэши идентификаторов для аутентификации запросов от клиентских программ. Для этой цели можно использовать существующие файлы идентификаторов или сгенерировать новые с помощью утилиты rnid.

Ниже приведен сокращенный пример включения удаленного управления в конфигурационном файле Reticulum:

```
[reticulum]
...
enable_remote_management = yes
remote_management_allowed = 9fb6d773498fb3feda407ed8ef2c3229,
2d882c5586e548d79b5af27bca1776dc
...
```

Чтобы увидеть полный пример конфигурации, можно запустить команду `rnsd --exampleconfig`.

Управление «черными дырами»

Сети Reticulum по сути являются открытыми и не требуют разрешений, что позволяет участвовать любому, у кого есть совместимый интерфейс. Хотя такая открытость необходима для отказоустойчивой и децентрализованной сети, она также подвергает сеть потенциальному злоупотреблению, например, когда узлы заваливают сеть
чрезмерными широковещательными объявлениями или иными формами истощения ресурсов.

Система **Blackhole** предоставляет инструменты, помогающие решать эту проблему. Она позволяет операторам и отдельным пользователям блокировать определённые идентификаторы на
транспортном уровне, не давая им распространять объявления через ваш узел, а другим узлам — достигать их через вашу сеть.

Важно

В сетях Reticulum принципиально **невозможно** *глобально* заблокировать или подвергнуть цензуре какой-либо идентификатор или пункт назначения. Функция «blackhole» предотвращает объявления (и трафик) ко всем пунктам назначения, связанным с
идентификатором, попавшим в «черную дыру», *только в ваших собственных сегментах сети*.

Это дает пользователям и операторам контроль над тем, что они хотят разрешить *в своих собственных сегментах сети*, но нет возможности глобально подвергнуть цензуре или удалить идентификатор, пока *кто-то* готов обеспечить его транспортировку.

Эта функция служит двум целям:

- **Для отдельных пользователей:** она предлагает простой способ поддерживать тихую и эффективную локальную сеть путем ручной блокировки спамерских или нежелательных узлов.
- **Для сетевых операторов:** позволяет создавать объединенные стандарты безопасности для всего сообщества. Публикуя и обмениваясь списками blackhole, операторы могут защищать крупные инфраструктуры и распространять правила фильтрации спама по всей сети без ручного вмешательства.

Управление локальными черными списками

Самый быстрый способ управления нежелательными идентификаторами — это ручная настройка с помощью утилиты `rnpath`. Это позволяет мгновенно блокировать или разблокировать определенные идентификаторы в локальном экземпляре Transport.

Блокировка идентификатора

Чтобы заблокировать идентификатор, используйте флаг `-B` (или `--blackhole`), за которым следует хеш идентификатора. По желанию можно указать продолжительность и причину, что полезно для ведения журнала и использования в будущем.

```
$ rnpath -B 3a4f8b9c1d2e3f4g5h6i7j8k9l0m1n2o
```

Вы также можете добавить продолжительность (в часах) и причину:

```
$ rnpath -B 3a4f8b9c1d2e3f4g5h6i7j8k9l0m1n2o --duration 24 --reason
"Excessive announces"
```

Снятие с черного списка

Чтобы удалить идентификатор из черной дыры, используйте флаг `-U` (или `--unblackhole`):

```
$ rnpath -U 3a4f8b9c1d2e3f4g5h6i7j8k9l0m1n2o
```

Просмотр списка черных дыр

Чтобы просмотреть все идентификаторы, которые в данный момент заблокированы на вашем локальном экземпляре, используйте флаг `-b`
(или `--blackholed`):

```
$ rnpath -b

<3a4f8b9c1d2e3f4g5h6i7j8k9l0m1n2o> заблокирован на 23 ч 56 мин (чрезмерное количество объявлений)
<399ea050ce0eed1816c300bcb0840938> заблокирован на неопределённый срок (спам в объявлениях)
<d56a4fa02c0a77b3575935aedd90bdb2> заблокирован на неопределенный срок (спам объявлений)
<2b9ec651326d9bc274119054c70fb75e> заблокирован на неопределенный срок (спам в объявлениях)
<1178a8f1fad405bf2ad153bf5036bdfd> заблокирован на неопределенный срок (спам в объявлениях)
```

Автоматический поиск списков

Ручная блокировка идентификаторов эффективна в случае непосредственных угроз, но ведение актуального списка блокировок для большой сети нецелесообразно. Reticulum поддерживает **автоматический поиск списков**, позволяя вашему узлу подписываться на списки blackhole, поддерживаемые доверенными пирами или центральным органом, которым вы управляете самостоятельно.

Предупреждение

Проверьте перед подпиской! Подписка на источник черных списков — это серьезное действие, которое дает этому источнику возможность диктовать, с кем вы можете общаться. Перед добавлением источника в вашу конфигурацию убедитесь, что его администратор соответствует вашей политике использования и ценностям. Слепо

Подписка на недоверенные списки может привести к непреднамеренной блокировке легитимных узлов или важных служб.

Если эта функция включена, ваш транспортный экземпляр будет периодически (примерно
раз в час) подключаться к настроенным источникам, загружать их последние списки blackhole и автоматически объединять их с вашим локальным списком блокировки. Это обеспечивает защиту по принципу «настроил и забыл» как для отдельных пользователей, так и для крупных сетей.

Настройка

Чтобы включить автоматический поиск источников, добавьте опцию `blackhole_sources` в раздел `[reticulum]` вашего файла конфигурации. Эта опция принимает разделенный запятыми список хешей Transport Identity, которым вы доверяете в плане предоставления действительных списков blackhole.

```
[reticulum]
...
# Автоматически загружать списки blackhole из этих доверенных источников
blackhole_sources = 521c87a83afb8f29e4455e77930b973b,
68a4aa91ac350c4087564e8a69f84e86
...
```

Как это работает

1. При включении служба `BlackholeUpdater` работает в фоновом режиме.
2. Для каждого хеша идентификатора, указанного в `blackhole_sources`, она пытается установить временную ссылку на связанный с ним пункт назначения «rnstransport.info.blackhole».
3. Она запрашивает путь `/list`, который возвращает словарь идентификаторов, попавших в черную дыру, и связанных с ними метаданных.
4. Полученный список объединяется с вашей локальной базой данных `blackholed_identities`
   .
5. Списки сохраняются на диск, что гарантирует их сохранность при перезапуске.

Примечание

Вы можете проверить внешние списки, на которые вы подписаны, и их содержимое, не импортируя их, с помощью команды `rnpath -p`. Подробности о запросах к удаленным спискам blackhole см. в документации по утилите rnpath.

Публикация списков blackhole

Если вы управляете общедоступным шлюзом, общинным хабом или просто хотите поделиться своим списком «черных дыр» с другими, вы можете настроить свой экземпляр так, чтобы он выступал в роли издателя списка «черных дыр». Это позволит другим узлам подписаться на *ваши*
определения нежелательного трафика.

Включение публикации

Чтобы опубликовать свой локальный список blackhole, включите опцию `publish_blackhole` в разделе `[reticulum]`:

```
[reticulum]
...
publish_blackhole = yes
...
```

Когда эта опция включена, ваш экземпляр Transport зарегистрирует обработчик запросов по адресу `rnstransport.info.blackhole`. Любой узел, подключающийся к этому адресу и запрашивающий `/list`, получит полный набор идентификаторов, присутствующих в данный момент в вашей локальной базе данных blackhole.

Федерация и доверие

Система blackhole основана на отношениях доверия между подписчиком и издателем. Подписываясь на источник, вы тем самым доверяете этому источнику, что он будет блокировать только те идентичности, которые действительно наносят вред сети.

По мере развития экосистемы эта система будет интегрирована с **Network Identities**. Это позволит сообществам проверять, что опубликованный список «черных дыр» действительно предоставлен конкретной сетью или организацией, обладающей определенным уровнем репутации и надежности, что добавит уровень

криптографического доверия к процессу федерации. Это предотвращает публикацию злоумышленниками поддельных списков, предназначенных для цензуры легитимного трафика.

Для операторов это создает масштабируемую модель, в которой ведение единого высококачественного списка блокировки может защитить тысячи нижестоящих узлов, что значительно снижает административную нагрузку.

Улучшение конфигурации системы

Если вы настраиваете систему для постоянного использования с Reticulum, есть несколько изменений в конфигурации системы, которые могут упростить ее администрирование. Эти изменения будут подробно описаны здесь.

Фиксированные имена последовательных портов

На экземпляре Reticulum с несколькими интерфейсами на основе последовательных портов может быть целесообразно использовать фиксированные имена устройств для последовательных портов вместо динамически назначаемых сокращений, таких как `/dev/ttyUSB0`. В большинстве дистрибутивов на базе Debian, включая Ubuntu и Raspberry Pi OS, эти устройства можно найти в каталоге `/dev/serial/by-id`.

Вы можете использовать такой путь к устройству напрямую вместо пронумерованных сокращений. Вот пример TNC для пакетной радиосвязи, настроенного таким образом:

```
[[Packet Radio KISS Interface]] type =
  KISSInterface interface_enabled =
  True outgoing = true
  port = /dev/serial/by-id/usb-FTDI_FT230X_Basic_UART_43891CKM-if00-port0
  speed = 115200
  databits = 8
  parity = none
  stopbits = 1
  преамбула = 150
  txtail = 10
  persistence = 200
  время слота = 20
```

Использование этой методологии позволяет избежать потенциальной путаницы с именами, когда физические
устройства могут подключаться и отключаться в разном порядке, или когда присвоение имен устройствам меняется от одной загрузки к другой.

Reticulum как системный сервис

Вместо того чтобы запускать Reticulum вручную, можно установить `rnsd` в качестве системной службы, чтобы она запускалась автоматически при загрузке системы.

Системная служба

Если вы установили Reticulum с помощью `pip`, программа `rnsd`, скорее всего, будет находиться только в локальном для пользователя пути установки, а это означает, что `systemd` не сможет её запустить. В этом случае вы можете просто создать символьную ссылку на программу `rnsd` в каталоге, который находится в пути systemd:

```
sudo ln -s $(which rnsd) /usr/local/bin/
```

Затем вы можете создать файл службы `/etc/systemd/system/rnsd.service`
со следующим содержанием:

```
[Unit]
Description=Демон сетевого стека Reticulum
After=multi-user.target

[Service]
# Если вы запускаете Reticulum на устройствах с Wi-
Fi # или других устройствах, которым требуется #
дополнительное время для инициализации, # вам может
понадобиться # добавить небольшую задержку перед
запуском # Reticulum системой systemd:
# ExecStartPre=/bin/sleep 10
Type=simple
Restart=always
RestartSec=3
User=USERNAMEHERE
ExecStart=rnsd --service
```

```
[Install]
WantedBy=multi-user.target
```

Обязательно замените `USERNAMEHERE` на имя пользователя, под которым вы хотите запускать `rnsd`. Чтобы запустить `rnsd` вручную, выполните:

```
sudo systemctl start rnsd
```

Если вы хотите, чтобы `rnsd` запускался автоматически при загрузке, выполните:

```
sudo systemctl enable rnsd
```

Служба пользовательского пространства

В качестве альтернативы можно использовать пользовательскую службу systemd вместо общесистемной. В этом случае всю настройку можно выполнить от имени обычного пользователя. Создайте файл пользовательской службы systemd `~/.config/systemd/user/rnsd.service` со следующим содержанием:

```
[Unit]
Description=Демон сетевого стека Reticulum
After=default.target

[Service]
# Если вы запускаете Reticulum на устройствах с WiFi
# или других устройствах, которым требуется #
дополнительное время для инициализации, # вы можете
добавить небольшую задержку перед # запуском
Reticulum системой systemd:
# ExecStartPre=/bin/sleep 10
Type=simple
Restart=always
RestartSec=3
ExecStart=RNS_BIN_DIR/rnsd --service

[Установка] WantedBy=default.target
```

Замените `RNS_BIN_DIR` на путь к каталогу с бинарными файлами Reticulum (например, /home/USERNAMEHERE/rns/bin).

Запустите пользовательскую службу:

```
systemctl --user daemon-reload systemctl --
user start rnsd.service
```

Если вы хотите автоматически запускать `rnsd` без необходимости входа в систему под именем USERNAMEHERE, выполните:

```
sudo loginctl enable-linger USERNAMEHERE systemctl --
user enable rnsd.service
```

Обзор Reticulum

В этой главе кратко описаны общие цели и принципы работы
принципы работы Reticulum. Она должна дать вам общее представление о том, как работает стек, и понимание того, как разрабатывать сетевые приложения с использованием Reticulum.

Эта глава не является исчерпывающим источником информации о Reticulum, по крайней мере, пока. В настоящее время единственным полным репозиторием и окончательным авторитетом в вопросе о том, как на самом деле функционирует Reticulum, является эталонная реализация на Python и справочник по API. Тем не менее, эта глава является важным ресурсом для понимания того, как работает Reticulum с точки зрения высокого уровня, а также общих принципов Reticulum и того, как применять их при создании собственных сетей или программного обеспечения.

После прочтения этой главы вы должны будете хорошо понимать, как работает сеть Reticulum, какие задачи она способна решать и как вы можете использовать её самостоятельно. В этой главе также даётся обзор идей и философии, лежащих в основе Reticulum, а также рассказывается, какие проблемы она стремится решить и каким образом подходит к их решению.

Мотивация

Основной мотивацией для разработки и внедрения Reticulum послужило нынешнее отсутствие надежных, функциональных и безопасных минимальных инфраструктуры для цифровой связи. Я убежден, что крайне желательно создать надежный и эффективный способ организации сетей цифровой связи на большие расстояния, которые могли бы безопасно обеспечивать обмен информацией между людьми и машинами, без централизованного органа власти, контроля, цензуры или барьеров для доступа.

Практически все существующие сегодня сетевые системы имеют одно общее ограничение: для их функционирования требуются значительные усилия по координации, а также централизованное доверие и власть. Чтобы присоединиться к таким сетям, необходимо получить одобрение
контролирующих «стражей». Эта потребность в координации и доверии неизбежно ведет

к созданию среды централизованного контроля, в которой операторам инфраструктуры или правительствам очень легко контролировать или изменять трафик, а также подвергать цензуре или преследовать нежелательных участников. Это также делает совершенно невозможным свободное развертывание и использование сетей по собственному усмотрению, как это делается с другими распространенными инструментами, которые
усиливают индивидуальную инициативу и свободу.

Цель Reticulum — свести к минимуму потребность в координации и доверии. Проект нацелен на то, чтобы сделать безопасные, анонимные и не требующие разрешений сети и обмен информацией инструментом, который любой может просто взять и использовать.

Поскольку Reticulum полностью независим от конкретных сред, его можно использовать для создания
сети, в зависимости от того, что лучше всего подходит для конкретной ситуации или что у вас есть в наличии. В некоторых случаях это могут быть пакетные радиосвязи на УКВ-частотах, в других — сеть на частоте 2,4 ГГц с использованием готовых радиостанций, а также стандартные платы разработчика LoRa.

На момент выпуска этого документа самой быстрой и простой конфигурацией для разработки и тестирования является использование радиомодулей LoRa с прошивкой с открытым исходным кодом (см. раздел «Рекомендуемая конфигурация»), подключенных к любому типу компьютера или мобильного устройства, на котором может работать Reticulum.

Конечной целью Reticulum является предоставление каждому возможности стать собственным сетевым оператором, а также обеспечение недорогого и простого покрытия обширных территорий множеством независимых, взаимосвязанных и автономных сетей. Reticulum **— это не** *одна сеть*, **а инструмент** для создания *тысяч сетей*. Сетей без аварийных выключателей, слежки, цензуры и контроля. Сетей, которые могут свободно взаимодействовать, объединяться и разъединяться друг с другом и не требуют централизованного надзора. Сети для людей. *Сети для народа*.

Цели

Чтобы обеспечить максимально широкую применимость и эффективность развертывания, при разработке Reticulum руководствовались следующими целями:

- **Полная совместимость с стеком программного обеспечения с открытым исходным кодом**
  Reticulum должен быть реализован с использованием исключительно программного обеспечения с открытым исходным кодом и способен работать только с ним. Это имеет решающее значение для обеспечения

доступности, безопасности и прозрачности системы.

- **Независимость от аппаратного уровня**

  Reticulum должен быть полностью независимым от аппаратного обеспечения и должен работать на широком спектре физических сетевых уровней, таких как радиомодули передачи данных, последовательные линии, модемы, портативные приемопередатчики, проводной Ethernet, Wi-Fi или любые другие устройства, способные передавать поток цифровых данных.

  Аппаратное обеспечение, созданное специально для Reticulum, должно быть максимально дешевым и использовать готовые компоненты, чтобы его можно было легко модифицировать и воспроизвести любому заинтересованному лицу.

- **Очень низкие требования к пропускной способности**

  Reticulum должен обеспечивать надёжную работу даже при пропускной способности канала связи всего *5 бит в секунду*.

- **Шифрование по умолчанию**

  Reticulum должен по умолчанию использовать надежное шифрование для всей коммуникации.

- **Анонимность инициатора**

  Должна быть возможность обмениваться данными через сеть Reticulum, не раскрывая никакой идентифицирующей информации о себе.

- **Использование без лицензии**

  Reticulum должен функционировать через физические средства связи , не требующих никакой лицензии на использование. Reticulum должен быть спроектирован таким образом, чтобы его можно было использовать в радиочастотных диапазонах ISM и чтобы он мог обеспечивать рабочие связи на большие расстояния в таких условиях, например, путем подключения модема к PMR- или CB-радиостанции или с помощью модулей LoRa или WiFi.

- **Поставляемое программное обеспечение**

  Помимо основного сетевого стека и API, позволяющих разработчикам создавать приложения с помощью Reticulum, необходимо реализовать и выпустить вместе с самим Reticulum базовый набор средств связи на его основе. Они должны служить как функциональный базовый набор средств связи, так и в качестве примера и учебного ресурса для тех, кто желает создавать приложения с помощью Reticulum.

- **Простота использования**
  Эталонная реализация Reticulum написана на Python, чтобы сделать ее простой в использовании и понимании. Программист, имеющий лишь базовый опыт, должен быть в состоянии использовать Reticulum для написания сетевых приложений.

- **Низкая стоимость**
  Развертывание системы связи на основе Reticulum должно быть максимально дешевым. Это должно быть достигнуто за счет использования недорогого готового оборудования, которое потенциальные пользователи, возможно, уже имеют. Стоимость настройки рабочего узла должна быть менее 100 долларов, даже если все компоненты необходимо приобретать.

Введение и основные функции

Reticulum — это сетевой стек, подходящий для каналов связи с высокой задержкой и низкой пропускной способностью. В своей основе Reticulum представляет собой систему, *ориентированную на обмен сообщениями*. Она подходит как для локальных сценариев «точка-точка» или «точка-многоточка», где все узлы находятся в пределах досягаемости друг друга, так и для сценариев, где пакеты должны передаваться через несколько прыжков в сложной сети, чтобы достичь получателя.

Reticulum отказывается от понятий адресов и портов, известных из протоколов IP, TCP и UDP. Вместо этого Reticulum использует единую концепцию *«пунктов назначения»*. Любое приложение, использующее Reticulum в качестве сетевого стека, должно создать один или несколько пунктов назначения для получения данных, а также знать пункты назначения, в которые необходимо отправить данные.

Все пункты назначения в Reticulum *представлены* в виде 16-байтового хеша. Этот хеш получается путем усечения полного SHA-256-хеша идентифицирующих характеристик пункта назначения. Пользователям адреса пунктов назначения будут отображаться в виде 16 шестнадцатеричных байтов, как в этом примере:
`<13425ec15b621c1d928589718000d814>` .

Размер усечения в 16 байт (128 бит) для адресов назначения был выбран в качестве разумного компромисса между объемом адресного пространства и накладными расходами на пакеты. Адресное пространство, обеспечиваемое этим размером, способно поддерживать многие миллиарды

одновременно активных устройств в одной сети, при этом сохраняя низкую накладную нагрузку пакетов, что крайне важно в сетях с низкой пропускной способностью. В крайне маловероятном случае, если это адресное пространство приблизится к перегрузке, изменение всего одной строки кода
позволит расширить адресное пространство Reticulum до 256 бит, обеспечив его потенциальную поддержку сетей галактического
масштаб. Очевидно, что это полное и нелепое перераспределение ресурсов, и, как таковое, нынешних 128 бит должно хватить даже в далеком будущем.

По умолчанию Reticulum шифрует все данные с использованием криптографии на основе эллиптических кривых и алгоритма AES. Любой пакет, отправляемый адресату, шифруется с помощью ключа, вычисляемого для каждого пакета отдельно. Reticulum также может устанавливать зашифрованный канал связи с адресатом, называемый *«Link*». Как данные, передаваемые по каналам «Link», так и отдельные пакеты обеспечивают *анонимность инициатора*. Кроме того, по умолчанию каналы обеспечивают *прямую секретность*, используя обмен ключами по алгоритму Диффи-Хеллмана с эллиптической кривой на кривой Curve25519 для вывода эфемеричных ключей для каждого канала. Асимметричная пакетная связь без каналов также может обеспечивать прямую секретность с автоматической сменой ключей, включая эту функцию для каждого пункта назначения. Многоскоковые уровни транспорта, координации, верификации и надежности являются полностью автономны и также основаны на криптографии с использованием эллиптических кривых.

Reticulum также предлагает шифрование с симметричным ключом для групповой связи, а также незашифрованные пакеты (**только** для целей локальной широковещательной передачи).

Reticulum может подключаться к различным интерфейсам, таким как радиомодемы, радиостанции передачи данных и последовательные порты, и предоставляет возможность легко туннелировать трафик Reticulum по IP-каналам, таким как Интернет или частные IP-сети.

Пункты назначения

Для получения и отправки данных с помощью стека Reticulum приложению необходимо создать один или несколько пунктов назначения. Reticulum использует три различных базовых типа пунктов назначения и один специальный:

- **Одиночный**

Тип *«один* адресат» является наиболее распространённым типом в Reticulum и подходит для большинства задач. Он всегда идентифицируется уникальным открытым ключом. Любые данные, отправленные этому адресату, будут зашифрованы с помощью эфемеричных ключей, полученных в результате обмена ключами по алгоритму ECDH, и будут доступны для чтения только создателю адреса, который владеет соответствующим закрытым ключом.

- **Простой**

  Тип адреса *«plain»* не шифруется и подходит для трафика, который должен быть транслирован нескольким пользователям или должен быть доступен для чтения любому. Трафик, направляемый в адрес типа *«plain»*, не шифруется. Как правило, адреса типа *«plain»* можно использовать для широковещательной информации, предназначенной для общего доступа. Доступ к адресам типа «plain» возможен только напрямую, и пакеты, адресованные простым адресатам, никогда не передаются через несколько прыжков в сети. Чтобы обеспечить передачу через несколько переходов в Reticulum, информация *должна* быть зашифрована, поскольку Reticulum использует шифрование каждого пакета для проверки маршрутов и поддержания их работоспособности.

- **Групповой**

  Тип специального адреса *«группа»*, который определяет виртуальный адрес с симметричным шифрованием. Данные, отправленные на этот адрес, будут зашифрованы симметричным ключом и будут доступны для чтения любому, кто владеет этим ключом, но, как и в случае с типом адреса *«обычный»*, В настоящее время пакеты, адресованные таким получателям, не передаются через несколько промежуточных узлов, хотя запланированное обновление Reticulum позволит поддерживать глобально доступные *групповые* адреса.

- **Связь**

  Связь — это особый тип адреса, который служит абстрактным каналом к *одному* адресату, подключенному напрямую или через несколько прыжков. *Связь* также обеспечивает надёжность и более эффективное шифрование, секретность пересылки, анонимность инициатора и, как таковая, может быть полезна даже в тех случаях, когда узел доступен напрямую. Она также предоставляет более функциональный API и позволяет легко осуществлять запросы и ответы, передачу больших объемов данных и многое другое.

Именование адресатов

Пункты назначения создаются и называются с помощью понятной точечной нотации *аспектов* и представлены в сети в виде хеша этого значения. Хеш представляет собой SHA-256, усечённый до 128 бит. Аспект верхнего уровня всегда должен быть уникальным идентификатором для приложения, использующего данный пункт назначения. Следующие уровни аспектов могут быть определены любым образом создателем приложения.

Аспекты могут быть любой длины и в любом количестве, и даже длинное итоговое имя назначения не повлияет на эффективность, поскольку в сети имена всегда передаются в виде сокращенных хэшей SHA-256.

Например, пункт назначения для приложения мониторинга окружающей среды может состоять из названия приложения, типа устройства и типа измерения, например:

```
название приложения  : environmentlogger
аспекты              : remotesensor, temperature

полное имя : environmentlogger.remotesensor.temperature хеш  :
4faf1b2e0a077e6a9d92fa051f256038
```

Для *одиночного* адреса Reticulum автоматически добавляет связанный открытый ключ в качестве аспекта адреса перед хешированием. Это делается для того, чтобы гарантировать доставку только по правильному адресу, поскольку любой может прослушивать любое имя адреса. Добавление открытого ключа гарантирует, что данный пакет будет направлен только тому адресату, который владеет соответствующим закрытым ключом для расшифровки пакета.

Обратите внимание! Здесь необходимо понять одну очень важную концепцию:

- Любой может использовать имя адреса
  `environmentlogger.remotesensor.temperature`
- Каждый пункт назначения, использующий такой подход, по-прежнему будет иметь уникальный хеш назначения и, следовательно, будет однозначно адресуем, поскольку их открытые ключи будут различаться.

При фактическом использовании именования с *одним* пунктом назначения рекомендуется не использовать никаких уникальных идентифицирующих признаков в именовании аспектов. Имена аспектов должны быть общими терминами, описывающими, какой тип пункта назначения представлен. Уникальная идентификация аспекта всегда достигается путем добавления открытого ключа,

что превращает адрес в уникально идентифицируемый. Reticulum делает это автоматически.

Любой адрес в сети Reticulum можно найти и достичь, просто зная его хеш-адрес (и открытый ключ, но если открытый ключ неизвестен, его можно запросить у сети, просто зная хеш-адрес). Использование имен приложений и аспектов упрощает структурировать программы Reticulum и позволяет фильтровать информацию и данные, которые получает ваша программа.

Подводя итог, различные типы адресатов следует использовать в следующих ситуациях:

- **Одиночный**
  Когда требуется частная связь между двумя конечными точками. Поддерживает многоуровневую передачу.

- **Групповая**
  Когда требуется частная связь между двумя или более конечными точками. Косвенно поддерживает несколько переходов, но сначала должна быть установлена через *один* пункт назначения.

- **Простой**
  Когда желательна связь в виде открытого текста, например, при широковещательной передаче информации или для целей локального обнаружения.

Для связи с *одним* адресатом необходимо знать его открытый ключ. Допустим любой способ получения открытого ключа, однако Reticulum включает простой механизм, позволяющий другим узлам узнать ваш открытый ключ , называемый *«announce»*. Также можно запросить неизвестный открытый ключ из сети, так как все экземпляры транспортного уровня служат распределенным реестром открытых ключей

Обратите внимание, что информацию об открытых ключах можно передавать и проверять иными способами, помимо использования встроенной функции *объявления*, и поэтому для получения открытых ключей не обязательно использовать функции *объявления* и *запроса пути*. Однако это, безусловно, самый простой способ, и его следует использовать, если нет веских причин поступать иначе.

Объявления открытых ключей

Объявление отправляет специальный пакет по любым соответствующим интерфейсам, содержащий всю необходимую информацию о хэше назначения и открытом ключе, а также может содержать некоторые дополнительные данные, специфичные для приложения. Весь пакет подписывается отправителем для обеспечения подлинности. Использование функции объявления не является обязательным, но во многих случаях это будет самым простым способом обмена открытыми ключами в сети. Механизм объявления также служит для установления сквозной связи с объявленным адресатом, поскольку
объявление распространяется по сети.

Например, объявление в простом мессенджере может содержать следующую информацию:

- Хеш адресата объявителя
- Открытый ключ объявителя
- Данные, специфичные для приложения, в данном случае ник пользователя и статус доступности
- Случайный набор символов, обеспечивающий уникальность каждого нового объявления
- Подпись Ed25519 вышеуказанной информации, подтверждающая подлинность

Имея эту информацию, любой узел Reticulum, который ее получит, сможет восстановить адрес назначения исходящего сообщения для безопасной связи с этим адресом. Вы, возможно, заметили, что для полного восстановления информации об объявленном адресе назначения не хватает одного элемента — имен аспектов адреса назначения. Их намеренно опущены, чтобы
экономии пропускной способности, так как в большинстве случаев они будут подразумеваться. Принимающее приложение уже будет их знать. Если имя адреса не является полностью подразумеваемым, в специфическую для приложения
, которая позволит получателю вывести название.

Важно отметить, что объявления будут пересылаться по всей сети в соответствии с определенным шаблоном. Это будет подробно описано в разделе «Механизм объявлений в деталях».

В Reticulum пункты назначения могут свободно перемещаться по сети. Это сильно отличается от таких протоколов, как IP, где предполагается, что адрес всегда остается в том сегменте сети, в котором он был назначен. Это

В Reticulum такого ограничения не существует: любой узел *полностью переносим* по всей топологии сети и *может быть даже перемещен в другие сети Reticulum*, отличные от той, в которой он был создан, при этом оставаясь доступным. Чтобы обновить информацию о своей доступности, узлу достаточно отправить объявление в любую сеть, частью которой он является. Через некоторое время он станет доступным во всей сети.

Наблюдение за тем, как *отдельные* адреса всегда привязаны к паре закрытого/открытого ключей, приводит нас к следующей теме.

Идентичности

В Reticulum *идентичность* не обязательно представляет собой личное удостоверение, а является абстракцией, которая может представлять любой вид *проверяемой сущности*. Это вполне может быть человек, но также может быть интерфейс управления машиной, программой, роботом, компьютером, датчиком или чем-то совершенно иным. В общем, любой вид агента, который может действовать, на который могут воздействовать, или который может хранить или манипулировать информацией, может быть представлен в виде идентичности. *Идентичность* может использоваться для создания любого количества адресатов.

Каждому пункту назначения всегда привязана *идентичность*, за исключением *простых* и *групповых* пунктов назначения. Пункты назначения и идентичности связаны между собой многосторонней связью. Вы можете создать пункт назначения, и если при создании он не будет связан с какой-либо идентичностью, система автоматически создаст новую для использования.
В некоторых ситуациях это может быть желательным, но чаще всего вам, вероятно, захочется сначала создать идентичность, а затем использовать ее для создания новых адресатов.

Например, мы могли бы использовать идентичность для представления пользователя приложения для обмена сообщениями. Затем данная идентичность может создавать адресаты, чтобы обеспечить доставку сообщений пользователю. Во всех случаях крайне важно надежно и конфиденциально хранить закрытые ключи, связанные с любой идентичностью Reticulum, поскольку получение доступа к ключам идентичности равносильно получению
доступа и контроля над доступностью любых пунктов назначения, созданных этой идентичностью.

Дальнейшие шаги

Вышеупомянутые функции и принципы составляют ядро Reticulum и вполне достаточны для создания функциональных сетевых приложений в локальных кластерах, для
Например, при использовании радиоканалов, когда все заинтересованные узлы могут напрямую слышать друг друга. Однако для обеспечения реальной полезности нам нужен способ направления трафика через несколько промежуточных узлов в сети.

В следующих разделах будут представлены две концепции, позволяющие это сделать: *пути* и *каналы*.

Транспорт Reticulum

Методы маршрутизации, используемые в традиционных сетях, принципиально несовместимы с типами физических сред и условиями, с которыми
для работы с которыми был разработан Reticulum. Эти механизмы в основном предполагают доверие на физическом уровне и часто требуют гораздо большей пропускной способности, чем Reticulum может предположить. Поскольку Reticulum разработан для работы в открытом радиоспектре, такого доверия предполагать нельзя, а пропускная способность часто очень ограничена.

Чтобы преодолеть такие проблемы, *транспортная* система Reticulum использует криптография на основе асимметричных эллиптических кривых для реализации концепции *путей*, позволяющих определять, как приблизить информацию к определенному пункту назначения. Важно отметить, что ни один узел в сети Reticulum не знает полного пути к пункту назначения. Каждый транспортный узел, участвующий в сети Reticulum, знает только самый прямой путь, позволяющий приблизить пакет на один шаг к пункту назначения.

Типы узлов

В настоящее время Reticulum различает два типа сетевых узлов. Все узлы в сети Reticulum являются *экземплярами Reticulum*, а некоторые из них также являются *транспортными узлами*. Если система, на которой работает Reticulum, закреплена в одном месте и предназначена для обеспечения доступности в течение большей части времени, она является хорошим кандидатом на роль *транспортного узла*.

Любой экземпляр Reticulum может стать транспортным узлом, если включить эту функцию в конфигурации. Это различие устанавливается пользователем при настройке узла и используется для определения того, какие узлы в сети будут помогать пересылать трафик, а какие будут полагаться на другие узлы для более широкой связи.

Если узел является *экземпляром,* ему следует задать конфигурационную директиву `enable_transport = No` , что является настройкой по умолчанию.

Если это *транспортный узел*, для него следует задать конфигурационную директиву `enable_transport = Yes`.

Механизм объявления в деталях

Когда экземпляр Reticulum передает *объявление* о пункте назначения, оно будет перенаправлено любым транспортным узлом, который его получит, но в соответствии с
некоторым конкретным правилам:

- Если именно это объявление уже получали ранее, игнорировать его.

- Если нет, записать в таблицу, от какого транспортного узла было получено объявление и сколько раз в общей сложности оно было переслано, чтобы добраться сюда.

- Если объявление было переслано *m+1* раз, оно больше не будет пересылаться. По умолчанию *m* установлено на 128.

- После случайной задержки объявление будет повторно передано на всех интерфейсы, имеющие доступную пропускную способность для обработки объявлений. По умолчанию максимальный объем пропускной способности, выделяемый для обработки объявлений, установлен на уровне 2 %, но его можно настроить отдельно для каждого интерфейса.

- Если у какого-либо интерфейса недостаточно пропускной способности для ретрансляции объявления, ему будет присвоен приоритет, обратно пропорциональный количеству прыжков, и оно будет вставлено в очередь, управляемую этим интерфейсом.

- Когда у интерфейса появляется пропускная способность для обработки объявления, он приоритезирует объявления для пунктов назначения, которые находятся ближе всего

по количеству прыжков, тем самым отдавая приоритет доступности и связности локальных узлов даже в медленных сетях, подключенных к более широким и быстрым сетям.

- После того как объявление было повторно передано, и если не слышно, что другие узлы повторно передают объявление с большим числом прыжков, чем когда оно покинуло этот узел, его передача будет повторена *r* раз. По умолчанию *r* установлено в 1.

- Если поступает более новое объявление из того же пункта назначения, в то время как идентичное объявление уже ожидает передачи, новейшее объявление отбрасывается. Если новейшее объявление содержит иные , оно заменит старое объявление.

Как только объявление достигнет транспортного узла в сети, любой другой узел, находящийся в прямом контакте с этим транспортным узлом, сможет связаться с пунктом назначения, откуда поступило объявление, просто отправив пакет, адресованный этому пункту назначения. Любой транспортный узел, располагающий информацией об объявлении, сможет направить пакет к пункту назначения, определив наиболее эффективный следующий узел на пути к нему.

Согласно этим правилам, объявление будет распространяться по сети предсказуемым образом и сделает объявленный пункт назначения доступным в течение короткого промежутка времени. Быстрые сети, способные обрабатывать большое количество объявлений, могут достичь полной конвергенции очень быстро, даже при постоянном добавлении новых пунктов назначения. Более медленным сегментам таких
сетей может потребоваться немного больше времени, чтобы получить полную информацию о широких и быстрых сетях, к которым они подключены, но со временем они все равно смогут это сделать, при этом отдавая приоритет полной и быстро сходящейся сквозной связности для своих локальных, более медленных сегментов.

Совет

Даже очень медленные сети, которые просто не способны достичь *полной* конвергенции, как правило, всё равно смогут связаться с **любым другим адресатом на любых подключенных сегментах**, поскольку соединительные транспортные узлы будут отдавать приоритет объявлениям в более медленные сегменты, которые фактически запрашиваются узлами, расположенными на них.

Это означает, что медленным сегментам с низкой пропускной способностью или ограниченными ресурсами **не** требуется полная информация о сети, поскольку пути всегда можно рекурсивно определить из других сегментов, которые обладают такой информацией.

В целом, даже чрезвычайно сложные сети, использующие максимальные 128 прыжков, достигнут полной сквозной связности примерно за одну минуту, при условии, что доступно достаточно пропускной способности для обработки необходимого количества объявлений.

Достижение пункта назначения

В сетях с изменяющейся топологией и недоверительной связью узлам нужен способ установить *проверенную связь* друг с другом. Поскольку предполагается, что базовые сетевые среды являются недоверительными, Reticulum должен предоставить способ гарантировать, что узел, с которым вы общаетесь, действительно является тем, кого вы ожидаете. Reticulum предлагает два способа сделать это.

Для обмена небольшими объёмами информации Reticulum предлагает API *Packet*, который работает именно так, как и ожидается — на уровне отдельных пакетов. При отправке пакета используется следующий процесс:

- Пакет всегда создается с указанием адресата и некоторыми данными полезной нагрузки. Когда пакет отправляется на *один* тип адресата, Reticulum автоматически создает временный ключ шифрования, выполняет обмен ключами ECDH с открытым ключом адресата (или ключом Ratchet, если он доступен) и шифрует информацию.

- Важно отметить, что этот обмен ключами не требует сетевого трафика. Отправитель уже знает открытый ключ адресата из ранее полученного объявления и, таким образом, может выполнить обмен ключами ECDH локально, перед отправкой пакета.

- Открытая часть вновь сгенерированной эфемеричной пары ключей включается в зашифрованный токен и отправляется вместе с зашифрованными данными
  в пакете.

- Когда адресат получает пакет, он может самостоятельно выполнить обмен ключами ECDH и расшифровать пакет.

- Для каждого пакета, отправляемого таким образом, используется новый эфемеридный ключ.

- После того как пакет был получен и расшифрован адресатом, он может *подтвердить* факт получения пакета. Для этого он вычисляет хеш SHA-256 полученного пакета и подписывает этот хеш своим ключом подписи Ed25519. Затем транспортные узлы в сети могут направить это *подтверждение* обратно к источнику пакета, где подпись можно проверить с помощью известного открытого ключа подписи адресата.

- Если пакет адресован *групповому* типу назначения, он будет зашифрован с помощью заранее согласованного ключа AES-256, связанного с адресатом. Если пакет адресован *простому* типу назначения, данные полезной нагрузки не будут зашифрованы. Ни один из этих двух типов назначения не может обеспечить передовую секретность. В целом рекомендуется всегда использовать тип *одиночного* назначения, если только не требуется строго необходимость использовать один из других типов.

Для обмена большими объемами данных или когда требуются более длительные сеансы двунаправленной связи, Reticulum предлагает *Link* API. Для установления *соединения* используется следующий процесс:

- Во-первых, узел, желающий установить соединение, отправляет пакет *запроса на соединение*, который проходит по сети и находит нужный адрес назначения. По пути транспортирующие узлы, пересылающие пакет, фиксируют этот *запрос на соединение* и помечают его как ожидающий.

- Во-вторых, если пункт назначения принимает *запрос на установку соединения*, он отправляет обратно пакет, подтверждающий подлинность своей идентичности (и получение запроса на установку соединения), инициировавшему узлу. Все узлы, которые изначально пересылали пакет, также смогут проверить это подтверждение и, таким образом, признать действительность *соединения* по всей сети. Соединение теперь помечается как *установленное*.

- После того как узлы-ретрансляторы подтвердят действительность *соединения*, они запоминают *его*, и в дальнейшем его можно использовать, обращаясь к хэшу, представляющему это соединение.

- В рамках *запроса на установку соединения* происходит обмен ключами по алгоритму Диффи-Хеллмана с использованием эллиптической кривой, что позволяет создать эффективно зашифрованный туннель между двумя узлами. Таким образом, этот режим связи предпочтительным даже в ситуациях, когда узлы могут общаться напрямую, когда объем обмениваемых данных исчисляется десятками пакетов или когда требуется использование более продвинутых функций API.

- После установки *соединения* оно автоматически обеспечивает функцию подтверждения получения сообщений с помощью того же механизма *проверки*, о котором говорилось ранее, так что отправляющий узел может получить подтверждение того, что информация достигла предполагаемого получателя.

- После установления *соединения* инициатор может оставаться анонимным или пройти аутентификацию перед адресатом с помощью идентификатора Reticulum. Эта аутентификация происходит внутри зашифрованного соединения и раскрывается только проверенному адресату, без участия посредников.

Через минуту мы обсудим детали реализации этой методологии, но сначала давайте подытожим, каким целям она служит. Сначала мы убеждаемся, что узел, отвечающий на наш запрос, действительно является тем, с кем мы хотим общаться, а не злоумышленником, выдающим себя за него. Одновременно мы устанавливаем эффективный зашифрованный канал. Настройка этого канала относительно нетребовательна к пропускной способности, поэтому его можно использовать даже для короткого обмена данными, а затем воссоздавать по мере необходимости, что также приведет к смене ключей шифрования. Соединение также можно поддерживать в течение более длительных периодов времени, если это более подходит для данного приложения. Процедура также вставляет *идентификатор канала* — хеш, вычисленный из пакета запроса на установку канала, — в память промежуточных узлов, что означает, что обменивающиеся данными узлы впоследствии могут связаться друг с другом, просто ссылаясь на этот *идентификатор канала*.

Совокупная затрата пропускной способности на установку соединения составляет 3 пакета общим объемом 297 байт (подробнее в разделе «Формат двоичного пакета»). Объем пропускной способности, используемый для поддержания соединения в открытом состоянии, практически ничтожен и составляет 0,45 бит

в секунду. Даже на медленном канале пакетной радиосвязи со скоростью 1200 бит в секунду 100 одновременно работающих каналов связи всё равно оставляют 96 % пропускной способности канала для передачи фактических данных.

Подробное описание установления соединения

После изучения основ механизма объявления, поиска пути через сеть и обзора процедуры установления соединения, в этом разделе будет более подробно рассмотрен процесс установления соединения в Reticulum.

В терминологии Reticulum *канал* не следует рассматривать как прямое соединение «узел-узел» на физическом уровне, а как абстрактный канал, который может быть открыт в течение любого промежутка времени и охватывать произвольное количество прыжков, где между двумя узлами будет происходить обмен информацией.

- Когда узел в сети хочет установить проверенное соединение с другим узлом, он случайным образом генерирует новую пару ключей X25519 пару закрытого и открытого ключей. Затем он создает пакет *запроса на установку соединения* и рассылает его широковещанием.

  Следует отметить, что упомянутая выше пара открытого и закрытого ключей X25519 представляет собой две отдельные пары ключей: пару ключей шифрования, используемую для вывода общего симметричного ключа, и пару ключей подписи, используемую для подписи и проверки сообщений на канале связи. Они передаются по каналу связи вместе, и для упрощения данного объяснения их можно рассматривать как один открытый ключ.

- *Запрос* на *установ*ку *соединения* адресован хэшу адресата, и содержит следующие данные: вновь сгенерированный открытый ключ X25519 *LKi*.

- Рассылаемый пакет будет направлен по сети в соответствии с правилами, изложенными ранее.

- Любой узел, пересылающий запрос на установку соединения, сохраняет в своей *таблице соединений идентификатор соединения* вместе с количеством прыжков, которые проделал пакет при получении. Идентификатор соединения представляет собой хеш всего пакета запроса на установку соединения. Если

пакет запроса на установку соединения не *подтвержден* адресатом в течение по истечении определенного периода времени запись снова будет удалена из *таблицы связей*.

- Когда адресат получает пакет запроса на установку соединения, он принимает решение о принятии или отклонении запроса. В случае принятия адресат также генерирует новую пару открытого и закрытого ключей X25519 и выполняет обмен ключами по алгоритму Диффи-Хеллмана, получая новый симметричный ключ, который будет использоваться для шифрования канала после его установки.

- Теперь формируется пакет *подтверждения соединения* и передается по сети. Этот пакет адресован *идентификатору соединения*. Он содержит следующие данные: вновь сгенерированный открытый ключ X25519 *LKr* и подпись Ed25519, состоящую из *идентификатора соединения* и *LKr*, сгенерированную *исходным ключом подписи* адресата.

- Проверяя этот пакет *подтверждения соединения*, все узлы, которые изначально транспортировали пакет *запроса на соединение* от отправителя к адресату, теперь могут убедиться, что предполагаемый адресат получил запрос и принял его, а также что путь, выбранный ими для пересылки запроса, был действителен. При успешном выполнении этой проверки транспортирующие узлы
помечает канал как активный. Теперь по маршруту в сети установлен абстрактный двунаправленный канал связи. Пакеты теперь могут обмениваться в обоих направлениях с любого конца канала, просто указывая в адресе пакета *идентификатор* канала.

- Когда источник получает *доказательство*, он однозначно узнает, что к пункту назначения установлен проверенный путь. Теперь он также может использовать открытый ключ X25519, содержащийся в *доказательстве канала,* для выполнения собственного обмена ключами по алгоритму Диффи-Хеллмана и вывода симметричного ключа, который используется для шифрования канала. Теперь информация может обмениваться надежно и безопасно.

Примечание

Важно отметить, что данная методология гарантирует, что источнику запроса не нужно раскрывать какую-либо идентифицирующую информацию о себе. **Инициатор соединения остается полностью анонимным**.

При использовании *каналов* Reticulum автоматически проверяет все данные, отправляемые по каналу, а также может автоматизировать повторную передачу, если используются *ресурсы*.

Ресурсы

Для обмена небольшими объёмами данных в сети Reticulum достаточно интерфейса Packet, однако для обмена данными, требующими использования большого количества пакетов, необходим эффективный способ координации передачи.

Именно для этого и предназначен ресурс Reticulum. *Ресурс* может автоматически обеспечить надежную передачу произвольного объема данных по установленному каналу связи. Ресурсы могут автоматически сжимать данные, разбивать их на отдельные пакеты, упорядочивать последовательность передачи, проверять целостность и собирать данные на другом конце.

Ресурсы очень просты в использовании с программной точки зрения, и для надежной передачи любого объема данных требуется всего несколько строк кода. Их можно использовать для передачи данных, хранящихся в памяти, или для потоковой передачи данных непосредственно из файлов.

Сетевые идентификаторы

В Reticulum каждый узел и приложение использует криптографическую **идентичность** для проверки подлинности и установления зашифрованных каналов. В то время как стандартные идентификаторы обычно используются для представления отдельного пользователя, устройства или службы, Reticulum вводит концепцию **сетевой идентичности** для представления логической группы узлов или инфраструктуры всего сообщества.

Сетевая идентичность по сути представляет собой стандартный набор ключей Reticulum Identity. Однако её назначение и использование отличаются от личной идентичности. Вместо идентификации отдельного субъекта сетевая идентичность выступает в качестве общих учетных данных,

, объединяющая несколько независимых экземпляров транспорта в рамках единого, поддающегося проверке административного домена.

Концептуальный обзор

Стандартную идентичность Reticulum можно представить как самостоятельный, созданный частным образом паспорт для одного человека. Сетевая идентичность, напротив, сродна криптографическому флагу или уставу, развевающемуся над флотом кораблей. Это
означает, что, хотя корабли могут действовать независимо и находиться на большом физическом расстоянии, они принадлежат одной и той же организации, следуют одним и тем же протоколам и должны действовать согласованно.

Когда вы настраиваете сетевую идентичность на одном или нескольких ваших узлах, вы фактически заявляете, что эти узлы составляют конкретную «сеть» в рамках более широкой сетки Reticulum. Это позволяет другим узлам распознавать интерфейсы не просто как «узел с именем Алиса», а как «шлюз, принадлежащий организации «Восточный фронт свободы»».

Текущее использование

В настоящее время основная функция сетевой идентичности реализуется в системе обнаружения интерфейсов.

Когда экземпляр транспорта рассылает объявление об обнаружении интерфейса, он может по желанию подписать это объявление сетевой идентичностью, а не только своей локальной транспортной идентичностью. Удаленные узлы, получающие объявление, могут затем проверить подпись. Это обеспечивает функциональность для двух важных отличий:

1. **Аутентичность:** это доказывает, что интерфейс был опубликован оператором, обладающим закрытым ключом для данной сетевой идентичности.
2. **Границы доверия:** эта функция позволяет пользователям настраивать свои системы таким образом, чтобы они принимали подключения и устанавливали соединения только с интерфейсами, принадлежащими определённым сетевым идентификаторам, что фактически создаёт «белые списки» зон доверенной инфраструктуры.

Примечание

Если вы включите шифрование в своих объявлениях об обнаружении, сетевая идентичность будет использоваться в качестве общего секрета. Только узлы, которым явно предоставлен полный набор ключей сетевой идентичности (и которые настроили его локально), смогут расшифровать и использовать данные о соединении.

В будущем эта функциональность будет расширена, так что одноранговым узлам с делегированными ключами будет разрешено расшифровывать объявления об обнаружении, не имея корневого сетевого ключа. В настоящее время этой функциональности достаточно для частного обмена информацией об интерфейсах, когда вы контролируете все узлы, которые должны расшифровывать обнаруженные интерфейсы.

Будущие перспективы

Хотя текущая реализация сосредоточена на обнаружении интерфейсов, концепция сетевых идентификаторов служит основополагающим строительным блоком для будущих функций Reticulum, предназначенных для поддержки крупномасштабного органического формирования ячеистой сети.

По мере развития экосистемы сетевые идентификаторы будут способствовать:

- **Распределенное разрешение имен:** система, в которой сети могут публиковать сопоставления имен и идентификаторов, позволяя разрешать имена, понятные человеку, без использования централизованных серверов.
- **Публикация сервисов:** Сети смогут объявлять о конкретных возможностях, сервисах или информационных конечных точках, доступных для общего пользования или для своих участников.
- **Межсетевая федерация:** отношения доверия между различными сетями, обеспечивающие беспрепятственный, но управляемый поток трафика и информации через отдельные административные границы.
- **Распределенное управление черными списками:** система распределения черных списков, основанная на репутации, в которой доверенные сетевые идентичности могут подписывать и публиковать списки заблокированных идентичностей. Это позволяет сообществам совместно обеспечивать соблюдение стандартов безопасности и фильтровать спам или вредоносные

идентификаторы в тех частях расширенной сетки, за которые они отвечают.

Внедряя сетевые идентичности уже сейчас, вы готовите свою инфраструктуру к совместимости с этой будущей функциональностью.

Создание и использование сетевой идентичности

Поскольку сетевая идентичность — это просто стандартная идентичность Reticulum, ее можно создать с помощью встроенных инструментов.

1. **Создание идентификатора:** Воспользуйтесь утилитой `rnid` для создания нового файла идентификатора, который будет служить вашим сетевым идентификатором.

   ```
   $ rnid -g ~/.reticulum/storage/identities/my_network
   ```

2. **Распространение открытого ключа:** открытый ключ необходимо распространить среди всех экземпляров транспорта, которым требуется проверять объявления вашей сети и информацию об обнаружении. По умолчанию, если ваш узел настроен на использование сетевой идентичности, это происходит автоматически (с помощью стандартного механизм объявления).

3. **Настройте экземпляры:** в разделе `[reticulum]` файла конфигурации на каждом узле в вашей сети укажите в параметре `network_identity` файл, который вы создали.

   ```
   [reticulum]
   ...
   network_identity =
   ~/.reticulum/storage/identities/my_network
   ...
   ```

После настройки ваши экземпляры будут автоматически использовать эту идентичность для подписи объявлений об обнаружении (и, возможно, для расшифровки конфиденциальной сетевой информации), представляя единый фронт для более широкой сети.

Рекомендуемая настройка

В этом разделе подробно описана рекомендуемая *эталонная конфигурация* для Reticulum. Важно отметить, что Reticulum разработан для использования практически на любом вычислительном устройстве и практически через любой канал, позволяющий отправлять и получать данные, при условии соблюдения некоторых минимальных требований.

Канал связи должен поддерживать как минимум полудуплексный режим работы и обеспечивать среднюю пропускную способность не менее 5 бит в секунду, а также поддерживать MTU физического уровня 500 байт. Стек Reticulum должен работать практически на любом оборудовании, способном обеспечить среду выполнения Python 3.x
.

При этом данная эталонная конфигурация была описана с целью предоставить общую платформу для всех, кто хочет помочь в разработке Reticulum, а также для тех, кто хочет узнать о рекомендуемой конфигурации, чтобы начать экспериментировать. Эталонная система состоит из трех частей:

- **Устройство интерфейса**
  , которое обеспечивает доступ к физической среде, по которой происходит связь, например, радиостанция со встроенным модемом. Конфигурация с отдельным модемом, подключенным к радиостанции, также будет являться интерфейсным устройством.

- **Хостовое устройство**
  Некое вычислительное устройство, способное запускать необходимое программное обеспечение, взаимодействовать с интерфейсным устройством и обеспечивать взаимодействие с пользователем.

- **Программный стек**
  Программное обеспечение, реализующее протокол Reticulum, и приложения, использующие его.

Эту эталонную конфигурацию можно считать относительно стабильной платформой для разработки, а также для начала создания сетей или приложений. Хотя детали реализации могут измениться на текущем этапе разработки, цель состоит в том, чтобы сохранить аппаратную совместимость настолько долго, насколько это возможно, и было решено, что текущая эталонная конфигурация

функциональной платформой на многие годы вперед. Текущая конфигурация эталонной системы выглядит следующим образом:

- **Интерфейсное устройство**
  Радиоустройство передачи данных, состоящее из радиомодуля LoRa и микроконтроллера с прошивкой с открытым исходным кодом, которое может подключаться к хост-устройствам через USB. Оно работает в диапазонах частот 430, 868 или 900 МГц. Более подробную информацию можно найти на странице RNode [https://github.com/markqvist/rnode_firmware].

- **Хост-устройство**
  Любое компьютерное устройство под управлением Linux и Python. Raspberry Pi с ОС на базе Debian — хороший вариант для начала, но можно использовать что угодно.

- **Программный стек**
  Самая последняя версия реализации Reticulum на Python, работающая на операционной системе на базе Linux.

Примечание

Чтобы избежать путаницы, очень важно отметить, что эталонное **не** использует стандарт LoRaWAN, а использует собственный MAC-уровень поверх простой модуляции LoRa! Таким образом, вам понадобится простой радиомодуль LoRa, подключенный к контроллеру с правильной прошивкой. Полная информация о том, как получить или изготовить такое устройство, доступна на странице RNode [https://github.com/markqvist/rnode_firmware].

С помощью данной эталонной конфигурации подключиться к сети Reticulum можно примерно за 100 долларов, даже если у вас ещё нет необходимого оборудования и вам придётся всё покупать.

Конечно, эта эталонная конфигурация — всего лишь рекомендация для легкого старта, и вам следует адаптировать ее к своим конкретным потребностям или к тому оборудование, которым вы располагаете.

Особенности протокола

В этой главе подробно описана информация, относящаяся к протоколу, которая необходима для реализации Reticulum, но не является критически важной для понимания того, как протокол работает на общем уровне. К ней следует относиться скорее как к справочному материалу, чем к обязательному чтению.

Приоритезация пакетов

В настоящее время Reticulum полностью игнорирует приоритеты в отношении *общего* трафика. Весь трафик обрабатывается по принципу «первым пришел — первым обслужен». Трафик повторной передачи объявлений и другой трафик обслуживания обрабатывается в соответствии со временем повторной передачи и приоритетами, описанными ранее в этой главе.

Коды доступа к интерфейсам

Reticulum может создавать именованные виртуальные сети, а также сети, доступ к которым возможен только при знании заранее согласованной парольной фразы. Подробная информация о настройке этой функции приведена в разделе «Общие параметры интерфейса». Для реализации этой функции Reticulum использует концепцию кодов доступа к интерфейсу, которые вычисляются и проверяются для каждого пакета.

Интерфейс с включенной именованной виртуальной сетью или аутентификацией по парольной фразе будет выводить общий идентификатор подписи Ed25519 и для каждого исходящего пакета генерировать подпись всего пакета. Затем эта подпись вставляется в пакет в качестве кода доступа к интерфейсу перед передачей. В зависимости от скорости и возможностей интерфейса, IFAC может быть полной 512-битной подписью Ed25519 или урезанной версией.
Настроенную длину IFAC можно проверить для всех интерфейсов с помощью утилиты `rnstatus`.

При получении интерфейс проверит, соответствует ли подпись ожидаемому значению, и отбросит пакет, если это не так. Это гарантирует, что в сеть пропускаются только те пакеты, которые отправлены с правильными параметрами именования и/или парольной фразы.

Формат передачи

== Формат передачи Reticulum ======

Пакет Reticulum состоит из следующих полей:

[HEADER 2 байта] [ADDRESSES 16/32 байта] [CONTEXT 1 байт] [DATA
0–465 байт]

* Поле HEADER имеет длину 2 байта.
  * Байт 1: [Флаг IFAC], [Тип заголовка], [Флаг контекста], [Тип распространения],
    [Тип назначения] и [Тип пакета]
  * Байт 2: Количество прыжков

* Поле кода доступа к интерфейсу, если установлен флаг IFAC.
  * Длина кода доступа к интерфейсу может варьироваться от 1 до 64 байт в зависимости от возможностей и конфигурации физического интерфейса.

* Поле ADDRESSES содержит 1 или 2 адреса.
  * Длина каждого адреса составляет 16 байт.
  * Флаг «Тип заголовка» в поле HEADER определяет, содержит ли поле ADDRESSES 1 или 2 адреса.
  * Адреса представляют собой хеши SHA-256, усеченные до 16 байт.

* Поле CONTEXT имеет размер 1 байт.
  * Оно используется Reticulum для определения контекста пакета.

* Поле DATA имеет размер от 0 до 465 байт.
  * Оно содержит полезную нагрузку данных пакета.

Флаг IFAC

открыть	0	Пакет для общедоступного интерфейса
аутентифицированный	1	Аутентификация интерфейса включена в пакет

Типы заголовков

тип 1	0	Двухбайтовый заголовок, одно поле адреса длиной 16 байт
тип 2	1	Двухбайтовый заголовок, два адресных поля по 16 байт

Флаг контекста

не установлен	0
установлен	1

Флаг контекста используется для различных типов сигнализации, в зависимости от пакета контекста сигнализации

Типы распространения

широковещательная	0
транспорт	1

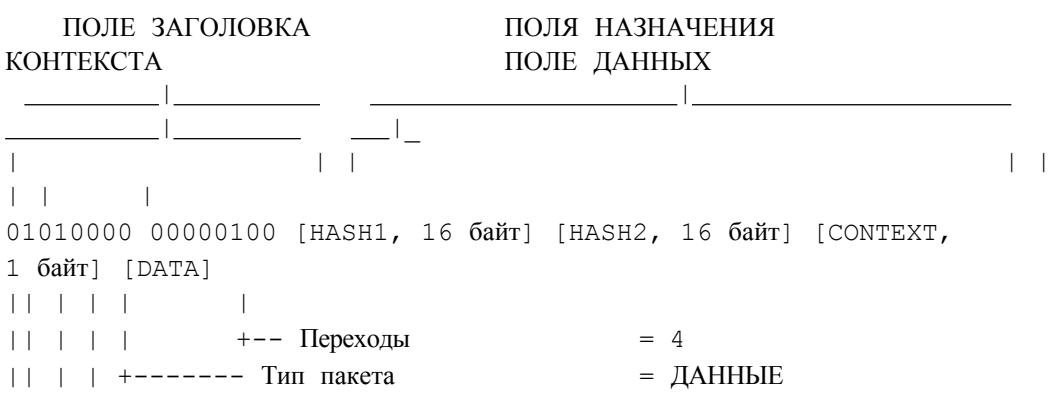
Типы назначения

одиночный	00
групповой	01
обычный	10
ссылка	11

Типы пакетов

данные	00
объявление	01
запрос на установку соединения	10
доказательство	11

+- Пример пакета -+



```
|| | +--------- Тип назначения = ОДИН
|| +----------- Тип распространения = TRANSPORT
|+------------- Тип заголовка              = HEADER_2 (двухбайтовый заголовок,
два поля адреса)
+-------------- Коды доступа               = ОТКЛЮЧЕН
```

+- Пример пакета -+

```
    ПОЛЕ ЗАГОЛОВКА    ПОЛЕ ПРИНАДЛЕЖНОСТИ    ПОЛЕ КОНТЕКСТА    ПОЛЕ ДАННЫХ
 _________|__________    _________|__________    ___________|_________    ___|_
|                    | |                    | |                      | |      |
00000000 00000111 [HASH1, 16 байт] [CONTEXT, 1 байт] [DATA]
 ||  |  | |      |
 ||  |  | |      +--  Хмель                 = 7
 || | | +------- Тип пакета                 = ДАННЫЕ
 || | +--------- Тип назначения = ОДИН
 || +----------- Тип распространения = BROADCAST
 |+------------- Тип заголовка              = HEADER_1 (двухбайтовый заголовок,
одно поле адреса)
 +-------------- Коды доступа               = ОТКЛЮЧЕН
```

+- Пример пакета -+

```
    ПОЛЕ ЗАГОЛОВКА      ПОЛЕ IFAC        ПОЛЕ ПРИНАДЛЕЖАЩЕГО   ПОЛЕ
КОНТЕКСТА                ПОЛЕ ДАННЫХ
 _________|__________    ________|_________    _________|__________
___________|_________    ___|_
|                    | |                | |                    | |
| |      |
10000000 00000111 [IFAC, N байт] [HASH1, 16 байт] [CONTEXT, 1
байт] [DATA]
 ||  |  | |      |
 ||  |  | |      +--  Хмель                 = 7
 || | | +------- Тип пакета                 = ДАННЫЕ
 || | +--------- Тип назначения = ОДИН
 || +----------- Тип распространения = BROADCAST
 |+------------- Тип заголовка              = HEADER_1 (двухбайтовый заголовок,
одно поле адреса)
 +-------------- Коды доступа               = ВКЛЮЧЕН
```

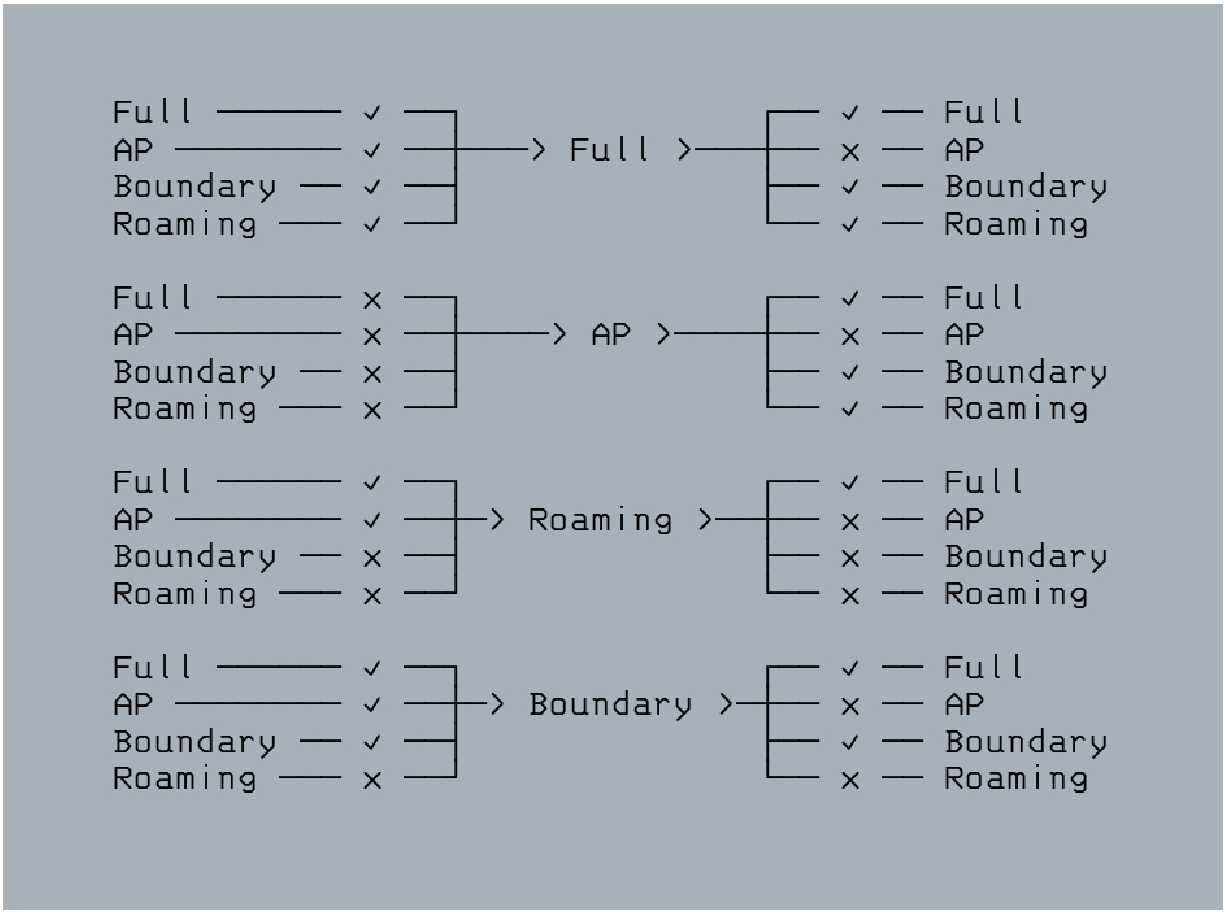
Примеры размеров различных типов пакетов

В приведенной ниже таблице указаны примерные размеры различных типов пакетов. Указанные размеры представляют собой полный размер пакета в сети, включающий все поля, в том числе заголовки, но исключающий коды доступа к интерфейсу.

- Запрос маршрута : 51 байт
- Объявление : 167 байт
- Запрос на ссылку : 83 байт
- Доказательство связи : 115 байт
- Пакет RTT ссылки : 99 байт
- Пакет поддержания связи : 20 байт

Правила распространения объявлений

В следующей таблице приведены правила автоматической передачи объявлений из одного типа интерфейса в другой для всех возможных комбинаций. С точки зрения распространения объявлений режимы *Full* и *Gateway* идентичны.



См. раздел «Режимы интерфейса» для получения общего представления о различных режимах интерфейса и о том, как их настраивать.

Криптографические примитивы

Reticulum использует набор простых, эффективных, надежных и тщательно протестированных
криптографических примитивов, имеющих широко доступные реализации, которые можно использовать как на универсальных процессорах, так и на микроконтроллерах.

Одним из основных соображений при выборе именно этого набора примитивов является то, что их можно *безопасно* реализовать с относительно небольшим количеством подводных камней практически на всех современных вычислительных платформах.

Перечисленные здесь примитивы **являются авторитетными**. Все, что претендует на то, чтобы быть Reticulum, но не использует именно эти примитивы, **не является** Reticulum и

возможно, является намеренно скомпрометированным или ослабленным клоном. Используемые примитивы:

- Ed25519 для подписей
- X25519 для обмена ключами ECDH
- HKDF для вывода ключей
- Зашифрованные токены основаны на спецификации Fernet
  - Эфемеричные ключи, полученные в результате обмена ключами ECDH на кривой Curve25519
  - AES-256 в режиме CBC с заполнением PKCS7
  - HMAC с использованием SHA256 для аутентификации сообщений
  - IV должны генерироваться с помощью `os.urandom()` или более надежного метода
  - Отсутствуют поля метаданных версии Fernet и временной метки
- SHA-256
- SHA-512

В стандартной конфигурации установки примитивы `X25519`, `Ed25519` и `AES-256-CBC` предоставляются библиотекой OpenSSL [https://www.openssl.org/] (через пакет PyCA/cryptography [https://github.com/pyca/cryptography]). Функции хеширования `SHA-256` и `SHA-512` предоставляются стандартной библиотекой Python hashlib [https://docs.python.org/3/library/hashlib.html]. Примитивы `HKDF`, `HMAC`, `Token` и функция заполнения `PKCS7` всегда предоставляются следующими внутренними реализациями:

- `RNS/Cryptography/HKDF.py`
- `RNS/Cryptography/HMAC.py`
- `RNS/Cryptography/Token.py`
- `RNS/Cryptography/PKCS7.py`

Reticulum также включает полную реализацию всех необходимых примитивов на чистом Python. Если при запуске Reticulum в системе отсутствуют OpenSSL и PyCA, Reticulum вместо них будет использовать внутренние примитивы на чистом Python. Очевидным следствием этого является производительность: бэкэнд OpenSSL работает *гораздо* быстрее. Однако наиболее важным следствием является потенциальная утечка безопасности при использовании примитивов, которые не прошли такого же тщательного анализа, тестирования и проверки, как примитивы из OpenSSL.

При использовании стандартных процедур установки RNS невозможно установить Reticulum в системе, где отсутствуют необходимые примитивы OpenSSL; если они отсутствуют, их поиск и установка будут выполнены в качестве зависимостей. Использовать примитивы, написанные исключительно на Python, можно только при их ручном указании, например, с помощью пакета `rnspure`.

Предупреждение

Если вы хотите использовать внутренние примитивы, написанные на чистом Python, **настоятельно**
, чтобы вы хорошо понимали риски, которые это представляет, и приняли осознанное решение о том, приемлемы ли для вас эти риски.

Коммуникационное оборудование

Один из действительно ценных аспектов Reticulum — это возможность использовать его практически через любой мыслимый вид среды связи. Типы интерфейсов Типы интерфейсов, доступные для настройки в Reticulum, достаточно гибки, чтобы охватить использование большинства имеющегося проводного и беспроводного коммуникационного оборудования — от десятилетней давности модемов пакетной радиосвязи до современных систем транспортной сети, работающих в миллиметровом диапазоне.

Если у вас уже есть или вы используете какое-либо коммуникационное оборудование, весьма вероятно, что оно будет работать с Reticulum сразу после установки.
Если это не так, можно обеспечить необходимую связь с помощью очень
Это не составит большого труда, если использовать, например, PipeInterface или TCPClientInterface в сочетании с кодом, подобным TCP KISS Server [https://github.com/simplyequipped/tcpkissserver] от simplyequipped [https://github.com/simplyequipped].

Также очень легко писать и загружать пользовательские модули интерфейсов в Reticulum, что позволяет вам обмениваться данными практически со всем, что только можно себе представить.

Хотя такая широкая поддержка и гибкость очень полезны, обилие вариантов иногда затрудняет выбор, с чего начать, особенно если вы начинаете с нуля.

В этой главе будут описаны несколько разумных путей для начала работы, позволяющих наладить реальную функциональную беспроводную связь с минимальными затратами и усилиями. Будут рассмотрены две основные категории устройств: *RNodes* и *радиомодули на базе WiFi*. Кроме того, будут кратко описаны другие распространенные варианты.

Умение использовать всего несколько типов оборудования позволит с минимальными усилиями создавать широкий спектр полезных сетей.

Комбинирование типов оборудования

При проектировании и построении сети целесообразно комбинировать различные типы каналов связи и оборудования. Один из полезных шаблонов проектирования заключается в использовании высокопроизводительных каналов связи «точка-точка» на основе WiFi или миллиметровых радиочастот (с направленными антеннами с высоким коэффициентом усиления) для магистрали сети, а также в использовании RNodes на базе LoRa для обеспечения покрытия больших территорий и подключения клиентских устройств.

RNode

Надежные и универсальные системы цифровых радиотрансиверов дальнего действия, как правило, либо очень дороги, либо сложны в настройке и эксплуатации, либо труднодоступны, либо потребляют много энергии, либо все вышеперечисленное одновременно. В попытке исправить эту ситуацию была разработана система радиотрансиверов *RNode*.
важно отметить, что RNode — это не одно конкретное устройство от одного конкретного поставщика, а *открытая платформа*, которую любой может использовать для создания совместимых цифровых приемопередатчиков, подходящих для его нужд и конкретных ситуаций.

RNode — это универсальное, совместимое, энергоэффективное, надежное, открытое и гибкое устройство радиосвязи с большим радиусом действия. В зависимости от компонентов оно может работать во многих различных диапазонах частот и использовать различные схемы модуляции, но чаще всего, и для
В целях данной главы мы ограничимся рассмотрением устройств RNode, использующих модуляцию *LoRa* в стандартных диапазонах ISM.

Не путайте! RNodes могут использовать LoRa в качестве *модуляции физического уровня*, но они не используют и не имеют никакого отношения к протоколу и стандарту *LoRaWAN*, обычно применяемым для централизованно управляемых устройств IoT. RNodes используют *«сырую» модуляцию LoRa* без каких-либо дополнительных накладных расходов на протокол. Вся функциональность протокола высокого уровня обрабатывается непосредственно Reticulum.

Создание RNodes

RNode был разработан как система, которую легко воспроизвести во времени и пространстве. Вы можете собрать работающий приемопередатчик, используя широко доступные компоненты и несколько программных инструментов с открытым исходным кодом. Хотя вы можете спроектировать и построить RNodes полностью с нуля, в точном соответствии с вашими желаемыми спецификациями, в этой главе будет описан самый простой подход к

создания RNodes: использование стандартных плат разработки LoRa. Этот подход сводится к двум простым шагам:

1. Приобретите одну или несколько поддерживаемых плат разработчика
2. Установите прошивку RNode с помощью автоматического установщика

После того как прошивка будет установлена и настроена с помощью скрипта установки, устройство будет готово к использованию с любым программным обеспечением, поддерживающим RNodes, включая Reticulum. Для работы устройства с Reticulum необходимо добавить в конфигурацию RNodeInterface в конфигурацию.

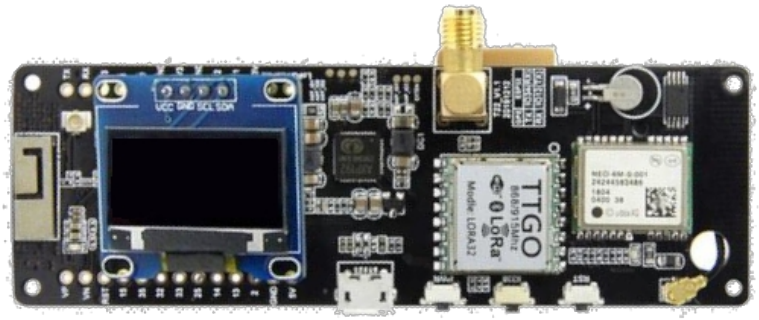
Поддерживаемые платы и устройства

Чтобы создать один или несколько RNodes, вам понадобится приобрести поддерживаемые платы для разработки или готовые устройства. Автоустановщик поддерживает следующие платы и устройства.



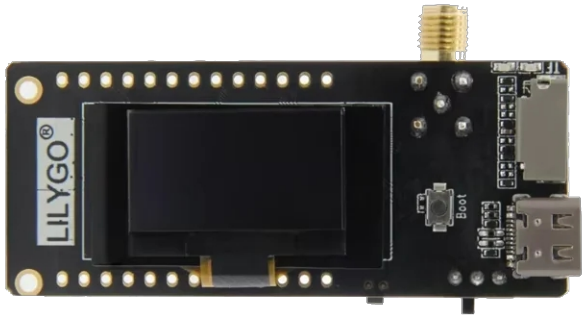
LilyGO T-Beam Supreme

- **Микросхема приемопередатчика** Semtech SX1262 или SX1268
- **Платформа устройства** ESP32
- **Производитель** LilyGO [https://lilygo.cn]



LilyGO T-Beam

- **Микросхема приемопередатчика** Semtech SX1262, SX1268, SX1276 или SX1278
- **Платформа устройства** ESP32
- **Производитель** LilyGO [https://lilygo.cn]



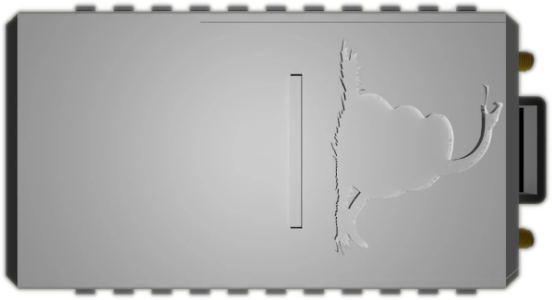
LilyGO T3S3

- **Микросхема приемопередатчика** Semtech SX1262, SX1268, SX1276 или SX1278
- **Платформа устройства** ESP32
- **Производитель** LilyGO [https://lilygo.cn]



Платы на базе RAK4631

- **Микросхема приемопередатчика** Semtech SX1262 или SX1268
- **Платформа устройства** nRF52
- **Производитель** RAK Wireless [https://www.rakwireless.com]



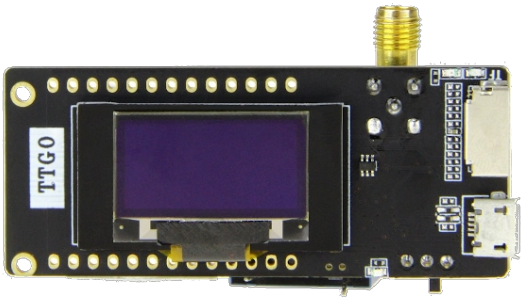
OpenCom XL

- **Микросхемы приемопередатчика** Semtech SX1262 и SX1280 (двойной приемопередатчик)
- **Платформа устройства** nRF52
- **Производитель** Liberated Embedded Systems [https://liberatedsystems.co.uk/]



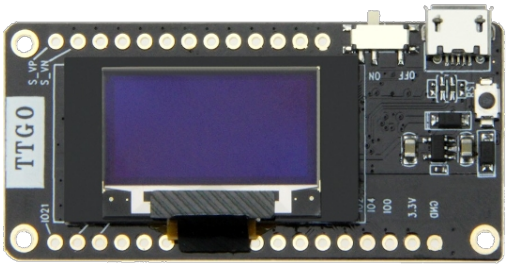
Unsigned RNode v2.x

- **Микросхема приемопередатчика** Semtech SX1276 или SX1278
- **Платформа устройства** ESP32
- **Производитель** unsigned.io [https://unsigned.io]



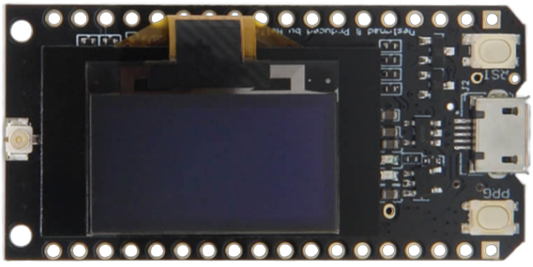
LilyGO LoRa32 v2.1

- **Микросхема приемопередатчика** Semtech SX1276 или SX1278
- **Платформа устройства** ESP32
- **Производитель** LilyGO [https://lilygo.cn]



LilyGO LoRa32 v2.0

- **Микросхема приемопередатчика** Semtech SX1276 или SX1278
- **Платформа устройства** ESP32
- **Производитель** LilyGO [https://lilygo.cn]



LilyGO LoRa32 v1.0

- **Микросхема приемопередатчика** Semtech SX1276 или SX1278
- **Платформа устройства** ESP32
- **Производитель** LilyGO [https://lilygo.cn]



LilyGO T-Deck

- **Микросхема приемопередатчика** Semtech SX1262 или SX1268

- **Платформа устройства** ESP32
- **Производитель** LilyGO [https://lilygo.cn]



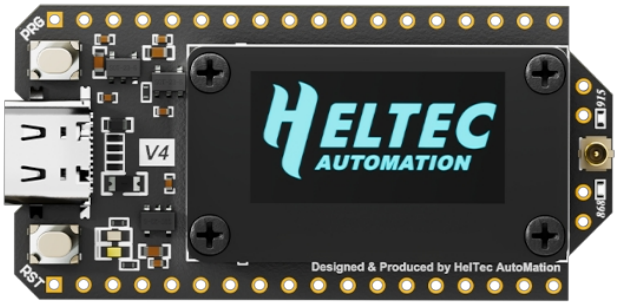
LilyGO T-Echo

- **Микросхема приемопередатчика** Semtech SX1262 или SX1268
- **Платформа устройства** nRF52
- **Производитель** LilyGO [https://lilygo.cn]



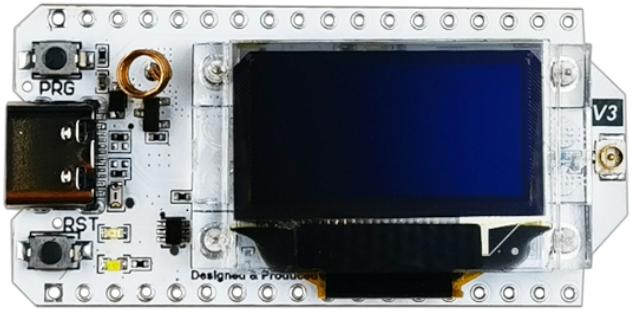
Heltec T114

- **Микросхема приемопередатчика** Semtech SX1262 или SX1268
- **Платформа устройства** nRF52
- **Производитель** Heltec Automation [https://heltec.org]



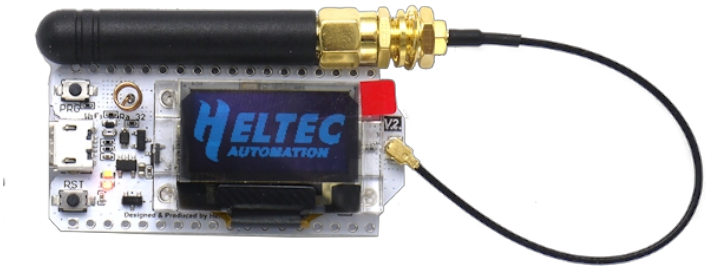
Heltec LoRa32 v4.0

- **Микросхема приемопередатчика** Semtech SX1262
- **Платформа устройства** ESP32
- **Производитель** Heltec Automation [https://heltec.org]



Heltec LoRa32 v3.0

- **Микросхема приемопередатчика** Semtech SX1262 или SX1268
- **Платформа устройства** ESP32
- **Производитель** Heltec Automation [https://heltec.org]



Heltec LoRa32 v2.0

- **Микросхема приемопередатчика** Semtech SX1276 или SX1278
- **Платформа устройства** ESP32
- **Производитель** Heltec Automation [https://heltec.org]

Установка

После приобретения совместимых плат вы можете установить прошивку RNode [https://github.com/markqvist/RNode_Firmware] с помощью утилиты настройки RNode [https://github.com/markqvist/rnodeconfigutil]. Если у вас есть
Если вы установили Reticulum в своей системе, программа `rnodeconf` уже будет доступна. Если нет, убедитесь, что в системе установлены `Python 3` и `pip`, а затем установите Reticulum с помощью `pip`:

```
pip install rns
```

После завершения установки пришло время приступить к установке прошивки на ваши устройства. Запустите `rnodeconf` в режиме автоматической установки следующим образом:

```
rnodeconf --autoinstall
```

Утилита проведет вас через процесс установки, задавая ряд вопросов о вашем оборудовании. Просто следуйте инструкциям, и утилита автоматически установит и настроит ваши устройства.

Использование с Reticulum

После установки и настройки устройств вы можете использовать их с Reticulum, добавив соответствующий раздел интерфейса в файл конфигурации Reticulum. В конфигурации вы можете указать все параметры интерфейса, такие как последовательный порт и параметры эфирного вещания.

Оборудование на базе WiFi

В Reticulum можно использовать любое оборудование, работающее по протоколу Wi-Fi, как с малым, так и с большим радиусом действия
. Любое оборудование, полностью поддерживающее мостовой Ethernet через WiFi-интерфейс, будет работать с AutoInterface в Reticulum. Большинство устройств ведут себя так по умолчанию или позволяют это сделать через настройки.

Это означает, что вы можете просто настроить физические соединения устройств на базе Wi-Fi и начать обмен данными через них с помощью Reticulum. Нет необходимости включать какую-либо IP-инфраструктуру, такую как серверы DHCP, DNS или подобные, при условии, что доступен хотя бы Ethernet, а пакеты проходят прозрачно через физические устройства на базе Wi-Fi.





Ниже приведен список примеров WiFi-радиомодулей (и аналогичных), которые хорошо подходят для высокопроизводительных каналов Reticulum на большие расстояния:

- Радиостанции Ubiquiti airMAX [https://store.ui.com/collections/operator-airmax-devices]
- Радиостанции Ubiquiti LTU [https://store.ui.com/collections/operator-ltu]
- Радиостанции MikroTik [https://mikrotik.com/products/group/wireless-systems]

Этот список ни в коем случае не является исчерпывающим и служит лишь несколькими примерами радиооборудования, которое является относительно недорогим, но при этом обеспечивает большую дальность действия и высокую пропускную способность для сетей Reticulum. Как и во всех других случаях, Reticulum также может сосуществовать с IP-сетями, работающими одновременно на таких устройствах.

Оборудование на базе Ethernet

Reticulum может работать на любом оборудовании, способном обеспечить коммутируемую среду на базе Ethernet. Это означает, что Reticulum может использовать все: от простого Ethernet-коммутатора до оптоволоконных систем и радиостанций передачи данных с интерфейсами Ethernet.

Среда Ethernet не требует наличия какой-либо IP-инфраструктуры, такой как серверы DHCP или настройка маршрутизации, но в случае, если такая инфраструктура существует, Reticulum просто будет сосуществовать с ней.

Для использования Reticulum через среды на базе Ethernet, как правило, достаточно использовать встроенный AutoInterface. Этот интерфейс также работает через любой виртуальный сетевой адаптер, например, устройства `tun` и `tap` в Linux.

Последовательные линии и устройства

С помощью SerialInterface также можно использовать Reticulum через любой тип необработанной последовательной линии. Этот тип интерфейса также полезен для использования Reticulum через коммуникационное оборудование, оснащённое интерфейсом последовательного порта.

Модемы пакетной радиосвязи

С Reticulum можно использовать любой модем пакетной радиосвязи, предоставляющий стандартный интерфейс KISS через USB, последовательный порт или TCP. Сюда входят виртуальные программные модемы, такие как FreeDV TNC [https://github.com/xssfox/freedv-tnc] и Dire Wolf [https://github.com/wb2osz/direwolf].

Настройка интерфейсов

Reticulum поддерживает использование множества типов устройств в качестве сетевых интерфейсов и позволяет комбинировать их любым удобным для вас образом. Количество различных сетевых топологий, которые можно создать с помощью Reticulum, практически безгранично, но для всех них характерно то, что вам потребуется определить один или несколько *интерфейсов* для использования Reticulum.

В следующих разделах описаны интерфейсы, доступные в настоящее время в Reticulum, а также приведены примеры конфигураций для соответствующих типов интерфейсов.

Чтобы получить общее представление о том, как можно создавать сети с использованием различных типов интерфейсов, ознакомьтесь с главой «Создание сетей» данного руководства.

Пользовательские интерфейсы

В дополнение к встроенным типам интерфейсов Reticulum **полностью расширяем** за счет пользовательских интерфейсов, предоставленных пользователями или сообществом, а создание пользовательских
модулей интерфейсов не представляет сложности. Ознакомьтесь с примером пользовательского интерфейса, чтобы увидеть базовый код интерфейса, на основе которого можно продолжить работу.

Автоматический интерфейс

`AutoInterface` обеспечивает связь с другими обнаруживаемыми узлами Reticulum через любой вид локальной сети на базе Ethernet или Wi-Fi. Несмотря на то, что он использует IPv6 для обнаружения одноранговых узлов и UDP для передачи пакетов, ему **не** требуется никакой функциональной IP-инфраструктуры, такой как маршрутизаторы или серверы DHCP, в вашей физической сети.

Предупреждение

Если на вашем компьютере установлено программное обеспечение **брандмауэра**, оно может блокировать трафик, необходимый для работы `AutoInterface`. В таком случае вам необходимо разрешить UDP-трафик на портах `29716` и `42671`.

Если между узлами присутствует хотя бы какое-либо коммутационное средство (проводной коммутатор, концентратор, точка доступа Wi-Fi или аналогичное устройство, либо просто два устройства, соединенные напрямую кабелем Ethernet), система будет работать без какой-либо настройки, конфигурации или промежуточных устройств.

Для работы обнаружения одноранговых узлов `AutoInterface` также требуется, чтобы в вашей системе была доступна поддержка локальных IPv6-адресов, что должно быть по умолчанию во всех современных операционных системах, как настольных, так и мобильных.

Примечание

Практически все современное Ethernet- и Wi-Fi-оборудование будет работать с `AutoInterface` без какой-либо настройки или конфигурации, однако на небольшом подмножестве устройств по умолчанию включены параметры, ограничивающие межустройственную связь, что приводит к блокировке обнаружения одноранговых узлов `AutoInterface`. Это
Эта проблема чаще всего встречается на очень дешевых Wi-Fi-роутерах, поставляемых интернет-провайдерами, и иногда её можно отключить в настройках роутера.

```
# В этом примере показана минимальная настройка # автоинтерфейса.
Она обеспечит связь со всеми # доступными устройствами на всех
# доступных физических устройствах на базе Ethernet, #
имеющихся в системе.
[[Интерфейс по умолчанию]]
  type = AutoInterface
  enabled = yes

# В этом примере показан более конкретно # настроенный интерфейс
Auto, который использует только определенные # физические
интерфейсы и имеет ряд # других настроек.
[[Default Interface]] type
  = AutoInterface
```

```
  enabled = yes

  # Вы можете создать несколько изолированных сетей Reticulum
  # в одной физической локальной сети,
  # указав разные идентификаторы групп. group_id
  = reticulum

  # Также можно выбрать тип многоадресного адреса:
  # временный (по умолчанию, временный многоадресный адрес) # или
  постоянный (постоянный многоадресный адрес) multicast_address_type
  = permanent

  # Вы также можете конкретно выбрать, # какие сетевые
  устройства ядра использовать. devices = wlan0,eth1

  # Или позволить AutoInterface использовать все подходящие
  # устройства, за исключением списка игнорируемых.
  ignored_devices = tun0,eth0
```

Если вы подключены к Интернету по IPv6, и ваш провайдер маршрутизирует IPv6-многоадресную рассылку, вы можете настроить Auto Interface на глобальное автоматическое обнаружение других узлов Reticulum в пределах выбранного вами Group ID. Вы можете указать область обнаружения, установив значение `link` , `admin` , `site`, `organisation` или `global`.

```
[[Интерфейс по умолчанию]]
  type = AutoInterface
  enabled = yes

  # Настроить глобальное обнаружение

  group_id = custom_network_name
  discovery_scope = global

  # Другие параметры настройки

  discovery_port = 48555
  data_port = 49555
```

Магистральный интерфейс

Интерфейс Backbone — это очень быстрый и ресурсоэффективный тип интерфейса, предназначенный в первую очередь для соединения экземпляров Reticulum через различные типы среды передачи данных. Он использует бэкэнд ввода-вывода на основе событий ядра и способен
обслуживать тысячи интерфейсов и/или клиентов при относительно низком
. **В настоящее время этот тип интерфейса поддерживается только в Linux и Android**.

Примечание

Интерфейс Backbone полностью совместим с типами `TCPServerInterface` и `TCPClientInterface`; их можно использовать взаимозаменяемо и устанавливать перекрестные соединения между ними. В системах, поддерживающих `BackboneInterface`, обычно рекомендуется использовать именно его, если вам не требуются конкретные опции или функции, предоставляемые интерфейсами TCP-сервера и TCP-клиента.

Хотя цель состоит в поддержке *всех* типов сокетов и устройств ввода-вывода, предоставляемых базовой операционной системой, первоначальный выпуск обеспечивает поддержку только для TCP-соединений по IPv4 и IPv6.

Для всех типов соединений через `BackboneInterface` Reticulum корректно обрабатывает перебои в работе, обрывы канала и соединения, которые то появляются, то исчезают.

Слушатели

Следующие примеры иллюстрируют различные способы настройки прослушивателей `BackboneInterface`.

```
# В этом примере показан магистральный интерфейс, # который
прослушивает входящие соединения на
# указанном IP-адресе и номере порта. [[Backbone Listener]]
   type = BackboneInterface
```

```
  enabled = yes
  listen_on = 0.0.0.0
  port = 4242

# В качестве альтернативы можно привязаться к конкретному IP-адресу
[[Backbone Listener]]
  type = BackboneInterface enabled =
  yes
  listen_on = 10.0.0.88
  порт = 4242

# Или к конкретному сетевому устройству
[[Backbone Listener]]
  тип = BackboneInterface включено = да
  устройство = eth0
  порт = 4242
```

Если вы используете интерфейс на устройстве, имеющем как IPv4-, так и IPv6-адреса, вы можете использовать опцию `prefer_ipv6` для привязки к IPv6-адресу:

```
# В этом примере показан магистральный интерфейс, #
прослушивающий IPv6-адрес указанного
# сетевого устройства ядра. [[Backbone
Listener]]
  type = BackboneInterface
  enabled = yes prefer_ipv6 =
  yes
  device = eth0 port
  = 4242
```

Чтобы использовать `BackboneInterface` через Yggdrasil [https://yggdrasil-network.github.io/], можно просто указать устройство `tun` Yggdrasil и порт прослушивания, например:

```
# В этом примере показан магистральный интерфейс, #
прослушивающий соединения через Yggdrasil. [[Yggdrasil
Backbone Interface]]
  тип = BackboneInterface включено = да
```

```
устройство = tun0
порт = 4343
```

Подключение удаленных устройств

Следующие примеры иллюстрируют различные способы подключения к удаленным прослушивателей `BackboneInterface`. Как отмечено выше, интерфейсы `BackboneInterface` также могут подключаться к удаленным `TCPServerInterface`, и поэтому эти типы интерфейсов можно использовать взаимозаменяемо.

```
# Вот пример интерфейса Backbone, который # подключается к
удаленному прослушивателю.
[[Backbone Remote]]
  type = BackboneInterface enabled =
  yes
  remote = amsterdam.connect.reticulum.network target_port
  = 4251
```

Чтобы подключиться к удаленным устройствам через Yggdrasil [https://yggdrasil-network.github.io/], просто укажите целевой IPv6-адрес и порт Yggdrasil, например:

```
[[Yggdrasil Remote]]
     тип = BackboneInterface включено =
     да
     target_host = 201:5d78:af73:5caf:a4de:a79f:3278:71e5 target_port = 4343
```

Интерфейс TCP-сервера

Интерфейс TCP-сервера подходит для того, чтобы позволить другим узлам подключаться через Интернет или частные сети IPv4 и IPv6. Когда интерфейс TCP-сервера
, другие узлы Reticulum могут подключаться к нему с помощью интерфейса TCP-клиента.

```
# В этом примере показан интерфейс TCP-сервера. # Он будет
прослушивать входящие соединения на всех IP-интерфейсах # на порту
4242.
[[Интерфейс TCP-сервера]]
```

```
  type = TCPServerInterface enabled =
  yes
  listen_ip = 0.0.0.0
  listen_port = 4242

# В качестве альтернативы можно привязаться к конкретному IP-адресу
[[TCP Server Interface]]
  type = TCPServerInterface enabled =
  yes
  listen_ip = 10.0.0.88
  listen_port = 4242

# Или к конкретному сетевому устройству
[[TCP Server Interface]]
  тип = TCPServerInterface включено =
  да
  device = eth0
  listen_port = 4242
```

Если вы используете интерфейс на устройстве, имеющем как IPv4, так и IPv6 адреса, вы можете использовать опцию `prefer_ipv6` для привязки к IPv6 адресу:

```
# В этом примере показан интерфейс TCP-сервера. # Он будет
прослушивать входящие соединения на
# указанном IP-адресе и номере порта.

[[Интерфейс TCP-сервера]] type =
  TCPServerInterface enabled =
  yes
  prefer_ipv6 = True
  устройство = eth0
  port = 4242
```

Чтобы использовать интерфейс TCP-сервера через Yggdrasil [https://yggdrasil-network.github.io/], достаточно просто указать устройство `tun` Yggdrasil и порт прослушивания, например:

```
[[Интерфейс TCP-сервера Yggdrasil]] type =
  TCPServerInterface enabled = yes
```

```
  устройство = tun0
  порт_прослушивания = 4343
```

Примечание

Интерфейсы TCP поддерживают туннелирование через I2P, но для обеспечения надёжности необходимо использовать параметр i2p_tunneled:

```
[[TCP-сервер на I2P]]
  type = TCPServerInterface enabled =
  yes
  listen_ip = 127.0.0.1
  listen_port = 5001
  i2p_tunneled = yes
```

Практически во всех случаях проще использовать специальный интерфейс `I2PInterface`, но для полного контроля и при использовании маршрутизаторов I2P, работающих на внешних системах, существует и этот вариант.

Интерфейс TCP-клиента

Для подключения к интерфейсу TCP-сервера можно использовать интерфейс TCP-клиента. Множество интерфейсов TCP-клиентов от разных пиров могут одновременно подключаться к одному и тому же интерфейсу TCP-сервера.

Типы интерфейсов TCP также могут переносить перебои в работе на уровне IP-связи. Это означает, что Reticulum будет корректно обрабатывать IP-соединения, которые то появляются, то исчезают, и восстанавливать связь после сбоя, как только снова появится другой конец интерфейса TCP.

```
# Вот пример интерфейса TCP-клиента.
# target_host может быть именем хоста или адресом IPv4 или IPv6. [[Интерфейс TCP-
клиента]]
  type = TCPClientInterface enabled =
  yes
  target_host = 127.0.0.1
  target_port = 4242
```

Чтобы использовать интерфейс TCP-клиента через Yggdrasil [https://yggdrasil-network.github.io/], просто укажите целевой IPv6-адрес и порт Yggdrasil, например:

```
[[Интерфейс TCP-клиента Yggdrasil]] type =
     TCPClientInterface enabled = yes
     target_host = 201:5d78:af73:5caf:a4de:a79f:3278:71e5 target_port = 4343
```

Также можно использовать этот тип интерфейса для подключения через другие программы или аппаратные устройства, которые предоставляют интерфейс KISS на TCP-порте, например, программные звуковые модемы. Для этого используйте опцию `kiss_framing`:

```
# Вот пример интерфейса TCP-клиента, который подключается # к программному
звуковому модему TNC через порт KISS over TCP.

[[Интерфейс TCP KISS]]
  type = TCPClientInterface
  enabled = yes kiss_framing =
  True target_host = 127.0.0.1
  target_port = 8001
  fixed_mtu = 500
```

Внимание! Используйте опцию KISS framing только при подключении к внешнему устройства и программы, такие как звуковые модемы и аналогичные устройства, работающие по протоколу TCP. При использовании интерфейса `TCPClientInterface` совместно с интерфейсом `TCPServerInterface` ни в коем случае не следует включать параметр `kiss_framing`, так как это приведет к отключению внутренних механизмов обеспечения надежности и восстановления, которые значительно повышают производительность при работе по ненадежным и прерывистым соединениям TCP.

Для устройств KISS, которым требуется поддержка только определенного MTU, можно использовать опцию `fixed_mtu`.

Примечание

Интерфейсы TCP поддерживают туннелирование через I2P, но для обеспечения надежности необходимо использовать опцию i2p_tunneled:

```
[[TCP-клиент через I2P]]
     type = TCPClientInterface enabled
     = yes
     target_host = 127.0.0.1
     target_port = 5001
     i2p_tunneled = yes
```

UDP-интерфейс

UDP-интерфейс может быть полезен для связи по IP-сетям, как частным, так и через Интернет. Он также может обеспечивать широковещательную связь по IP-сетям, что позволяет легко наладить связь со всеми другими узлами в локальной сети.

Предупреждение

Использование широковещательного UDP-трафика влияет на производительность, особенно при подключении по Wi-Fi. Если ваша цель заключается просто в обеспечении удобной связи со всеми узлами в локальном домене широковещания Ethernet, интерфейс Auto демонстрирует *гораздо* более высокую производительность и еще проще в использовании.

```
# В этом примере реализована связь с другими # локальными узлами
Reticulum по протоколу UDP.

[[UDP-интерфейс]]
  type = UDPInterface
  enabled = yes

  listen_ip = 0.0.0.0
  listen_port = 4242
  forward_ip = 255.255.255.255
  forward_port = 4242

  # Приведенная выше конфигурация обеспечит связь # в пределах локальных
  доменов широковещания всех локальных
  # IP-интерфейсов.

  # Вместо указания listen_ip, listen_port,
  # forward_ip и forward_port, вы также можете привязать
```

```
# к конкретному сетевому устройству, как показано ниже.

# device = eth0 #
port = 4242

# Предполагая, что устройство eth0 имеет адрес
# 10.55.0.72/24, приведённая выше конфигурация # будет
эквивалентна следующей настройке вручную.
# Обратите внимание, что мы одновременно прослушиваем и перенаправляем
трафик на # широковещательный адрес сегментов сети.

# listen_ip = 10.55.0.255 #
listen_port = 4242
# forward_ip = 10.55.0.255 #
forward_port = 4242

# Конечно, вы также можете обмениваться данными только с #
одним IP-адресом

# listen_ip = 10.55.0.15 #
listen_port = 4242
# forward_ip = 10.55.0.16 #
forward_port = 4242
```

Интерфейс I2P

Интерфейс I2P позволяет подключать экземпляры Reticulum через протокол Invisible Internet Protocol [https://i2pd.website]. Это может быть особенно полезно в случаях, , когда вы хотите разместить глобально доступный экземпляр Reticulum, но не имеете доступа к публичным IP-адресам, у вас часто меняется IP-адрес или брандмауэры блокируют входящий трафик.

Используя интерфейс I2P, вы получите глобально доступный, переносимый и постоянный адрес I2P, по которому можно будет связаться с вашим экземпляром Reticulum.

Чтобы использовать интерфейс I2P, у вас в системе должен работать маршрутизатор I2P. Самый простой способ добиться этого — скачать и установить последнюю версию [https://github.com/PurpleI2P/i2pd/releases/latest] пакета `i2pd`. Для подробнее об I2P см. на сайте geti2p.net [https://geti2p.net/en/about/intro].

Когда маршрутизатор I2P запущен в вашей системе, вы можете просто добавить интерфейс I2P в Reticulum:

```
[[I2P]]
  type = I2PInterface
  enabled = yes
  connectable = yes
```

При первом запуске Reticulum сгенерирует новый адрес I2P для подключить интерфейс и начать прослушивать входящий трафик. При первом запуске это может занять некоторое время, особенно если ваш I2P-маршрутизатор также был только что запущен и ещё не имеет хорошего подключения к сети I2P. Когда всё будет готово, в файле журнала должен появиться адрес I2P в формате base32. Вы также можете проверить состояние интерфейса с помощью утилиты `rnstatus`.

Чтобы подключиться к другим экземплярам Reticulum через I2P, просто добавьте список адресов I2P base32, разделенных запятыми, в опцию `peers` интерфейса:

```
[[I2P]]
  type = I2PInterface
  enabled = yes
  connectable = yes
  peers =
5urvjicpzi7q3ybztsef4i5ow2aq4soktfj7zedz53s47r54jnqq.b32.i2p
```

Установка I2P-соединений с нужными пирами может занять от нескольких секунд до нескольких минут, поэтому Reticulum обрабатывает этот процесс в фоновом режиме и выводит соответствующие события в журнал.

Примечание

Хотя интерфейс I2P является самым простым способом использования Reticulum через I2P, также возможно вручную настроить туннелирование интерфейсов TCP-сервера и TCP-клиента через I2P. Это может быть полезно в ситуациях, когда требуется больший контроль, но при этом необходимо вручную настроить туннелирование через конфигурацию демона I2P.

Важно отметить, что эти два метода *взаимозаменяемы*. Например, вы можете использовать I2PInterface для подключения к TCPServerInterface, который был

туннелирован, например, вручную через I2P. Это обеспечивает высокую степень гибкости при настройке сети, сохраняя при этом простоту использования в более простых сценариях.

Интерфейс RNode LoRa

Для использования Reticulum через LoRa можно воспользоваться интерфейсом RNode [https://unsigned.io/rnode/], который обеспечивает полный контроль над параметрами LoRa.

Предупреждение

Радиочастотный спектр является ресурсом, регулируемым законодательством, а законодательство во всем мире сильно различается. Вы несете ответственность за ознакомление с любыми применимыми к вашему местоположению нормами и принятие решений в соответствии с ними.

```
# Вот пример добавления интерфейса LoRa # с использованием
приемопередатчика RNode LoRa.

[[Интерфейс RNode LoRa]] type
  = RNodeInterface

  # Включите интерфейс, если хотите его использовать! enabled
  = yes

  # Последовательный порт для устройства
  port = /dev/ttyUSB0

  # Вы можете подключиться по беспроводной сети к
  # устройству RNode, если оно поддерживает Wi-
  Fi.

  # Подключение по IP-адресу #
  port = tcp://10.0.0.1

  # Или подключитесь по имени хоста
  # port = tcp://rnodef3b9.local

  # Также можно использовать устройства BLE # вместо
  проводных последовательных портов.
```

```
# целевое устройство RNode должно быть сопряжено с
# хост-устройством перед подключением. Устройства
BLE
# устройства можно подключать по имени,
# MAC-адрес BLE или по любому доступному.

# Подключение к конкретному устройству по имени #
port = ble://RNode 3B87

# Или по MAC-адресу BLE
# порт = ble://F4:12:73:29:4E:89

# Или подключиться к первому доступному #
сопряженному устройству
# port = ble://

# Установить частоту на 867,2 МГц
frequency = 867200000

# Установить полосу пропускания LoRa на 125 кГц
bandwidth = 125000

# Установить мощность передачи на 7 дБм
(5 мВт) txpower = 7

# Выбрать коэффициент рассеивания 8.
Допустимый # диапазон — от 7 до 12, где 7
# — самый быстрый, а 12 — # с наибольшей
дальностью. spreadingfactor = 8

# Выбрать кодовую скорость 5. Допустимый
диапазон # составляет от 5 до 8, где 5 — #
самая высокая скорость, а 8 — максимальная
дальность. codingrate = 5

# Вы можете настроить RNode на отправку
# идентификационные данные по каналу с # заданным
интервалом, настроив
# следующие два параметра.

# id_callsign = MYCALL-0 #
id_interval = 600
```

```
  # Для некоторых самодельных интерфейсов RNode # с
  небольшим объемом оперативной памяти
  # может быть полезно. По умолчанию # он отключен.

  # flow_control = False

  # Можно ограничить время
  # RNode с помощью # следующих двух параметров
  конфигурации. # Краткосрочное ограничение применяется
  в # окне продолжительностью примерно 15 секунд,
  # а долгосрочное ограничение применяется
  # в течение скользящего 60-минутного окна. Оба #
  параметра указываются в процентах.

  # airtime_limit_long = 1.5 #
  airtime_limit_short = 33
```

Многофункциональный интерфейс RNode

Для RNode, поддерживающих несколько приемопередатчиков LoRa, интерфейс RNode Multi можно использовать для индивидуальной настройки подинтерфейсов.

Предупреждение

Радиочастотный спектр является ресурсом, использование которого регулируется законодательством, причем законодательные нормы в разных странах мира значительно различаются. Вы несете ответственность за ознакомление с действующими в вашем регионе нормативными актами и принятие решений с учетом этих требований.

```
# Вот пример того, как добавить интерфейс RNode Multi # с использованием
трансивера RNode LoRa.

[[RNode Multi Interface]] type =
RNodeMultiInterface

# Включите интерфейс, если хотите его использовать!
```

```
enabled = yes

# Последовательный порт для устройства
port = /dev/ttyACM0

# Вы можете настроить RNode на отправку
# идентификационные данные по каналу с # заданным
интервалом, настроив
# следующие два параметра.

# id_callsign = MYCALL-0 #
id_interval = 600

   # Подинтерфейс [[[High
   Datarate]]]
      # Подинтерфейсы можно включать и отключать по отдельности
      enabled = yes

      # Установить частоту на 2,4 ГГц
      frequency = 2400000000

      # Установить полосу пропускания LoRa на 1625
      кГц bandwidth = 1625000

      # Установить мощность передачи на 0 дБм
      (0,12 мВт) txpower = 0

      # Виртуальный порт; только производитель
      # или автор конфигурации платы # может сказать, для какого
      # физический аппаратный интерфейс vport = 1

      # Выбрать коэффициент рассеивания 5.
      Допустимый # диапазон — от 5 до 12, где 5
      # является самым быстрым, а 12 — # имеет
      наибольшую дальность. spreadingfactor = 5

      # Выберите скорость кодирования 5. Допустимый
      диапазон # составляет от 5 до 8, где 5 — #
      максимальная скорость, а 8 — максимальная
      дальность.
```

```
  codingrate = 5

  # Можно ограничить время
  # использования эфирного времени RNode с помощью #
  следующих двух параметров конфигурации. #
  Краткосрочное ограничение применяется в # окне
  продолжительностью примерно 15 секунд,
  # и применяется долгосрочный лимит
  # в течение скользящего 60-минутного окна. Оба #
  параметра указываются в процентах.

  # airtime_limit_long = 100
  # airtime_limit_short = 100

[[[Низкая скорость передачи данных]]]
  # Подинтерфейсы можно включать и отключать отдельно
  enabled = yes

  # Установить частоту на 865,6 МГц
  frequency = 865600000

  # Виртуальный порт, только производитель
  # или разработчик конфигурации платы # может сказать, для
  какого
  # физический аппаратный интерфейс vport = 0

  # Установить полосу пропускания LoRa на 125
  кГц bandwidth = 125000

  # Установить мощность передачи на 0 дБм
  (0,12 мВт) txpower = 0

  # Выберите коэффициент рассеивания 7.
  Допустимый # диапазон — от 5 до 12, где 5
  # является самым быстрым, а 12 — # имеет
  наибольшую дальность. spreadingfactor = 7

  # Выберите скорость кодирования 5. Допустимый
  диапазон # составляет от 5 до 8, где 5 — #
  самая высокая скорость, а 8 — максимальная
  дальность.
```

```
    codingrate = 5

    # Можно ограничить время
    # использования эфирного времени RNode с помощью #
    следующих двух параметров конфигурации. #
    Краткосрочное ограничение применяется в # окне
    продолжительностью примерно 15 секунд,
    # а долгосрочное ограничение применяется
    # в течение скользящего 60-минутного окна. Оба #
    параметра указываются в процентах.

    # airtime_limit_long = 100
    # airtime_limit_short = 100
```

Последовательный интерфейс

Reticulum можно использовать напрямую через последовательные порты или через любое устройство с последовательным портом, которое прозрачно передает данные. Полезно для прямой связи по паре проводов или для использования таких устройств, как радиостанции передачи данных и лазеры.

```
[[Последовательный интерфейс]]
  type = SerialInterface enabled =
  yes

  # Последовательный порт устройства: port
  = /dev/ttyUSB0

  # Установите скорость передачи данных по
  последовательному порту и другие # параметры
  конфигурации.
  speed = 115200
  databits = 8
  parity = none
  stopbits = 1
```

Интерфейс канала

Используя этот интерфейс, Reticulum может использовать любую программу в качестве интерфейса через *stdin* и *stdout*. Это можно использовать для простого создания виртуальных интерфейсов или для взаимодействия с пользовательским оборудованием или другими системами.

```
[[Интерфейс Pipe]]
  тип = PipeInterface
  включено = да

  # Внешняя команда для выполнения
  command = netcat -l 5757

  # Опциональная задержка перезапуска, в секундах respawn_delay =
  5
```

Reticulum будет записывать все пакеты в *stdin* команды и будет непрерывно считывать и сканировать ее *stdout* на наличие пакетов Reticulum. При достижении `EOF` Reticulum попытается перезапустить программу после ожидания в течение `respawn_interval` секунд.

Интерфейс KISS

С помощью интерфейса KISS можно использовать Reticulum с различными модемами пакетной радиосвязи и TNC, включая OpenModem [https://unsigned.io/openmodem/]. Интерфейсы KISS также можно настроить на периодическую отправку маяков для идентификации станции.

Предупреждение

Радиочастотный спектр является ресурсом, регулируемым законодательством, и законодательство в разных странах мира значительно различается. Вы несете ответственность за ознакомление с любыми применимыми к вашему местоположению нормами и принятие соответствующих решений.

```
[[Интерфейс KISS для пакетной радиосвязи]]
  type = KISSInterface
  включено = да

  # Последовательный порт для устройства
  port = /dev/ttyUSB1

  # Установите скорость передачи данных по
  последовательному интерфейсу и другие # параметры
  конфигурации.
```

```
скорость = 115200
databits = 8
parity = none
stopbits = 1

# Установить преамбулу модема. preamble
= 150

# Установить конец передачи модема.
txtail = 10

# Настройте параметры CDMA. Эти
# настройки являются разумными значениями по умолчанию.
persistence = 200
slottime = 20

# Можно настроить интерфейс на отправку # идентификатора
по каналу с
# заданным интервалом, настроив
# следующие два параметра. Интерфейс KISS # будет
передавать идентификатор только в том случае, если
установлен
# Прошло # время с момента последней # фактической
передачи. Этот интервал # задается в секундах.
# Эта опция закомментирована и по умолчанию # не
используется.
#  id_callsign  =  MYCALL-0  #
id_interval = 600

# Использовать ли управление потоком KISS.
# Это полезно для модемов, которые имеют #
небольшой внутренний буфер пакетов, но
# вместо этого поддерживают управление потоком пакетов.
flow_control = false
```

Интерфейс AX.25 KISS

Если вы используете Reticulum в диапазоне любительского радио, вам может понадобиться интерфейс AX.25 KISS. В этом случае Reticulum будет автоматически

инкапсулировать свой трафик в AX.25, а также идентифицировать передачи вашей станции с помощью вашего позывного и SSID.

Делайте это только в случае крайней необходимости! Reticulum не нуждается в уровне AX.25 ни для чего, а инкапсуляция в AX.25 приводит к дополнительным накладкам при обработке каждого пакета.

Более эффективный способ — использовать простой интерфейс KISS с функцией передачи сигналов, описанной выше.

Предупреждение

Радиочастотный спектр является ресурсом, регулируемым законодательством, а законодательство во всем мире сильно различается. Вы несете ответственность за ознакомление с любыми применимыми к вашему местоположению нормами и принятие решений в соответствии с ними.

```
[[Интерфейс KISS для пакетной радиосвязи AX.25]]
  type = AX25KISSInterface

  # Установить позывной станции и SSID callsign
  = NO1CLL
  ssid = 0

  # Включите интерфейс, если хотите его использовать! enabled
  = yes

  # Последовательный порт для устройства
  port = /dev/ttyUSB2

  # Установить скорость передачи данных по
  последовательному интерфейсу и другие # параметры
  конфигурации.
  speed = 115200
  databits = 8
  parity = none
  stopbits = 1

  # Установите преамбулу модема. Преамбула
  длительностью 150 мс # является разумным
```

```
# значением по умолчанию, но его может потребоваться
# увеличить для радиостанций с медленно#
открывающимся шумоподавителем и длительным
# переключения = 150

# Установить конец передачи модема. В большинстве
# случаях это значение следует держать как можно ниже
#, чтобы не тратить эфирное время. txtail = 10

# Настройка параметров CDMA. Эти
# настройки являются разумными значениями по умолчанию.
persistence = 200
slottime = 20

# Использовать ли управление потоком KISS. #
Это полезно для модемов с
# небольшим внутренним буфером пакетов.
flow_control = false
```

Обнаруживаемые интерфейсы

Reticulum включает в себя мощную систему публикации ваших локальных интерфейсов в широкую сеть, что позволяет другим узлам обнаруживать их, проверять и автоматически подключаться к ним. Эта функция особенно полезна для создания децентрализованных сетей, в которых узлы могут динамически находить точки входа, такие как шлюзы общедоступного Интернета или локальные точки радиодоступа, не полагаясь на статические файлы конфигурации или централизованные каталоги.

Когда интерфейс становится **обнаруживаемым**, ваш экземпляр Reticulum будет периодически рассылать пакет объявления, содержащий детали соединения и параметры, необходимые другим узлам для установления соединения. Эти объявления распространяются по сети с помощью стандартного механизма объявлений Reticulum с использованием типа назначения `rnstransport.discovery.interface`.

Примечание

Чтобы использовать функцию обнаружения интерфейсов, в вашей среде Python должен быть установлен модуль `LXMF`. Его можно установить с помощью pip:

```
pip install lxmf
```

Включение обнаружения

Обнаружение интерфейсов включается для каждого интерфейса отдельно. Чтобы сделать конкретный интерфейс обнаруживаемым, необходимо добавить опцию `discoverable` в блок конфигурации этого интерфейса и установить для нее значение `yes`.

```
[[Мой общедоступный шлюз]]
  type = BackboneInterface
  ...
  discoverable = yes
```

После включения Reticulum автоматически будет обрабатывать генерацию, подпись, маркировку и рассылку объявлений об обнаружении. Для публикации информации об обнаружении интерфейсов не *обязательно* включать Transport, но в большинстве случаев, когда вы хотите, чтобы другие подключались к вам, вам, вероятно, понадобится установить значение `enable_transport` равным `yes` в разделе `[reticulum]` вашей конфигурации.

Параметры обнаружения

Когда включена опция `discoverable`, становится доступно множество дополнительных параметров для управления тем, как интерфейс представляется сети. Эти параметры позволяют точно настроить метаданные, требования безопасности и видимость вашего интерфейса.

Основные метаданные

`discovery_name`
: Понятное для человека имя интерфейса. Это имя будет отображаться пользователям на удаленных системах при просмотре списка обнаруженных интерфейсов. Если оно не указано, будет использоваться имя интерфейса (заголовок раздела).

`announce_interval`

Интервал в минутах между последовательными объявлениями об обнаружении этот интерфейс. По умолчанию — 360 минут (6 часов). Для стабильной, долго работающей инфраструктуры обычно достаточно более длительных интервалов (от 12 до 22 часов), что снижает нагрузку на сеть. Минимальное допустимое значение — 5 минут (но ожидайте, что ваши уведомления будут ограничены, если вы используете интервалы менее одного часа).

Спецификация доступности

`reachable_on`

Указывает адрес, который удаленные узлы должны использовать для подключения к данному интерфейсу.

- Для интерфейсов TCP и Backbone это, как правило, публичный IP-адрес или имя хоста. Не указывайте порт, он извлекается автоматически из интерфейса.
- Для интерфейсов I2P это, как правило, адрес I2P `b32`. Это значение извлекается автоматически из объекта `I2PInterface` после того, как он запущен и подключен к сети I2P, поэтому не следует задавать его вручную, если только вы не уверены на все сто процентов в том, что делаете.

Динамическое разрешение: этот параметр также принимает путь к внешнему исполняемому скрипту или бинарному файлу. Если путь указан, Reticulum запустит скрипт и использует его `stdout` в качестве адреса доступности. Это полезно для устройств, находящихся за динамическим DNS, NAT или сложными облачными
средах, где внешний IP-адрес неизвестен локально. Скрипт должен просто вывести адрес в stdout и завершить работу.

Примечание

При использовании исполняемого скрипта для параметра `reachable_on` Reticulum ожидает, что скрипт выведет в `stdout` только IP-адрес или имя хоста, за которым следует символ новой строки. Любой дополнительный вывод или ошибки могут привести к сбою разрешения. Убедитесь, что скрипт имеет права на выполнение и устойчив к временным сбоям в сети.

Минимальный пример скрипта, определяющего внешний публичный IP-адрес системы, подключенной к Интернету, может выглядеть следующим образом:

```
#!/bin/bash curl
-s ip.me exit $?
```

В реальной системе скрипт должен быть достаточно надежным, чтобы справляться с перебоями в работе Интернета или сбоями сервисов, так чтобы скрипт *всегда* возвращал разумное значение, а если это невозможно — по крайней мере завершался с ненулевым кодом возврата, чтобы Reticulum знал, что вывод недействителен.

Безопасность и стоимость

`discovery_stamp_value`
: Определяет сложность доказательства работы (proof-of-work) для криптографической метки, включенной в объявление. Это значение действует как барьер затрат, предотвращающий перегрузку сети. Значение по умолчанию равно `14`. Увеличение этого значения делает генерацию объявления вычислительно более затратной, что может быть полезно для предотвращения спама в очень больших сетях, но также увеличивает нагрузку на процессор вашей системы при генерации объявлений. Метки кэшируются и генерируются только при изменении информации об интерфейсе или при перезапуска экземпляра. Если у вас есть вычислительные ресурсы, обычно рекомендуется использовать как можно более высокое значение stamp.

Конфиденциальность и шифрование

`discovery_encrypt`
: Если установлено значение `yes`, полезные данные объявления об обнаружении будут зашифрованы. Для расшифровки объявления удаленные узлы должны обладать *сетевой идентичностью*, настроенной для вашего экземпляра (см. `network_identity` в разделе `[reticulum]`). Это позволяет публиковать частные интерфейсы, которые могут быть обнаружены только определенными доверенными сетями.

Важно

Если вы включите параметр `discovery_encrypt`, но не настроите действительный `network_identity` в разделе `[reticulum]` вашей конфигурации,

Reticulum прервет объявление об обнаружении интерфейса. Для работы шифрования требуется действительный ключ сетевой идентичности.

`publish_ifac`
: Если установлено значение `yes`, имя и пароль кода доступа к интерфейсу (IFAC) для этого интерфейса будут включены в объявление об обнаружении. Это позволяет одноранговым узлам автоматически настраивать правильные параметры аутентификации при подключении к интерфейсу.

Физическое местоположение

широта, долгота, высота
: Необязательные физические координаты интерфейса. Они полезны для географического отображения обнаруженных интерфейсов или для автоматического выбора клиентами ближайшей точки доступа. Координаты должны быть указаны в десятичных градусах, высота — в метрах.

Параметры радиомодуля

Для физических радиоинтерфейсов, таких как `RNodeInterface` или `KISSInterface`, следующие необязательные параметры позволяют транслировать рабочую частоту и характеристики, что дает клиентам возможность проверить совместимость перед подключением:

`discovery_frequency`
: Рабочая частота в Гц. Автоматически настраивается на интерфейсах RNode. Требуется для радиоинтерфейсов на базе KISS и интерфейсов `TCPClientInterfaces`, подключающихся к радиомодемам.

`discovery_bandwidth`
: Ширина полосы сигнала в Гц. Настраивается автоматически на интерфейсах RNode. Полезно для радиоинтерфейсов на базе KISS и интерфейсов `TCPClientInterfaces`, подключающихся к радиомодемам.

`discovery_modulation`
: Тип или схема модуляции. Настраивается автоматически на интерфейсах RNode, но настоятельно рекомендуется указывать на других интерфейсах на основе радиомодулей.

Режимы интерфейса

Когда вы включаете обнаружение на интерфейсе, Reticulum принудительно применяет определенные
режимы интерфейса, чтобы гарантировать, что интерфейс действительно полезен для удаленных узлов.

Если интерфейс настроен как обнаруживаемый , но его режим явно не установлен в `gateway` (для интерфейсов серверного типа, таких как `BackboneInterface` или `TCPServerInterface`) или `access_point` (для радиоинтерфейсов, таких как `RNodeInterface`), Reticulum автоматически
настроит соответствующий режим и зафиксирует уведомление в журнале.

Например, если вы включите обнаружение на `RNodeInterface` без указания режима, Reticulum автоматически установит для него режим `access_point`.

Вопросы безопасности

Обеспечивая обнаруживаемость интерфейсов, вы фактически рассылаете приглашение подключиться к вашей системе. Важно понимать, как выбранные вами параметры настройки влияют на безопасность.

Публикация учетных данных

Если вы включите `параметр publish_ifac = yes`, то в объявлении будет указана будет включена в объявление. Если вы управляете публичной сетью и хотите, чтобы к ней мог подключиться любой, это допустимо. Однако, если вы хотите ограничить доступ определенной группой пользователей, вам **необходимо** включить `discovery_encrypt = yes`. Это гарантирует, что только узлы, обладающие правильным `network_identity,` смогут расшифровать парольную фразу.

Раскрытие топологии

Обнаруживаемый интерфейс объявляет в сети о своем наличии, местоположении (если настроено) и возможностях. Даже если детали соединения зашифрованы, *факт* существования узла, с которым можно установить соединение, в определенной сети становится общедоступной информацией. В условиях повышенной безопасности или в сценариях, требующих

оперативной секретности, следует учитывать последствия раскрытия информации о существовании вашей инфраструктуры.

Пример конфигурации

Ниже приведен пример конфигурации общедоступного магистрального шлюза. Эта конфигурация предоставляет общедоступный интерфейс высокого уровня, к которому может подключиться любой пользователь.

```
[[Мой общедоступный шлюз]]
  type = BackboneInterface mode =
  gateway
  listen_on = 0.0.0.0
  порт = 4242

  # Включить обнаружение
  discoverable = yes

  # Сведения об интерфейсе
  discovery_name = Общедоступная точка входа региона A
  announce_interval = 720

  # Использовать внешний скрипт для определения динамического IP
  reachable_on = /usr/local/bin/get_external_ip.sh

  # Сгенерировать высокое значение метки
  discovery_stamp_value = 24

  # Дополнительные данные о местоположении
  latitude = 51.99714
  longitude = -0.74195
  высота = 15
```

В следующем примере создается зашифрованный интерфейс с поддержкой обнаружения, для декодирования которого требуется определенная сетевая идентификация, а также включаются учетные данные IFAC для беспрепятственной аутентификации.

```
[[Мой частный шлюз]]
  тип = BackboneInterface режим =
  шлюз
```

```
  listen_on = 0.0.0.0
  port = 5858
  network_name = internal_1 passphrase
  = Mevpekyafshak5Wr

  # Включить обнаружение
  discoverable = yes

  # Сведения об интерфейсе
  discovery_name = Частная магистральная сеть региона A
  announce_interval = 720

  # Использовать внешний скрипт для определения динамического IP
  reachable_on = /usr/local/bin/get_external_ip.sh

  # Целевое значение метки
  discovery_stamp_value = 22

  # Шифровать объявления только для нашей сети discovery_encrypt
  = yes

  # Включить учетные данные, чтобы доверенные
  # узлы могли подключаться автоматически publish_ifac = yes

  # Дополнительные данные о местоположении
  latitude = 34.06915
  longitude = -118.44318
  высота = 15
```

В разделе `[reticulum]` вашей конфигурации вы должны определить сетевую идентичность, используемую для шифрования, следующим образом:

```
[reticulum]
...
# Идентификатор, используемый для подписи/шифрования объявлений об обнаружении
network_identity =
~/.reticulum/storage/identities/my_network_identity
...
```

После применения этих параметров конфигурации ваш экземпляр Reticulum будет активно участвовать в экосистеме обнаружения сети. Другие узлы, на которых запущен Reticulum с включенным обнаружением, смогут увидеть ваш интерфейс, проверить его криптографическую метку и (в зависимости от их конфигурации) автоматически подключиться к нему.

Информацию о том, как использовать обнаруженные интерфейсы и настроить систему на автоматическое подключение к ним, см. в главе «Обнаружение интерфейсов».

Общие параметры интерфейсов

Для большинства интерфейсов доступен ряд общих параметров конфигурации. Их можно использовать для управления различными аспектами поведения интерфейса.

- Параметр `enabled` указывает Reticulum, следует ли активировать интерфейс. По умолчанию установлено значение `False`. Чтобы какой-либо интерфейс был активирован, параметр `enabled` должен быть установлен в `True` или `Yes`.

- Параметр `mode` позволяет выбрать высокоуровневое поведение интерфейса из ряда вариантов.

    - Значением по умолчанию является «`full`». В этом режиме доступны все функции обнаружения, формирования ячейки и передачи данных.
    - В режиме `access_point` (или сокращенно `ap`) интерфейс будет работать как точка доступа к сети. В этом режиме объявления не будут автоматически транслироваться по интерфейсу, а пути к пунктам назначения на интерфейсе будут иметь гораздо более короткий срок действия. Этот режим полезен для создания интерфейсов, которые в основном находятся в неактивны, за исключением тех моментов, когда кто-то действительно ими пользуется. Примером может служить радиоинтерфейс, обслуживающий широкой территории, где ожидается, что пользователи будут подключаться на мгновение, использовать сеть, а затем снова исчезать.

- Параметр `outgoing` определяет, разрешена ли передача данных на интерфейсе. По умолчанию установлено значение `True`. Если установлено значение `False` или `No`, интерфейс будет только принимать данные, но никогда не будет их передавать.

- Параметр `network_name` задает имя виртуальной сети для интерфейса. Это позволяет существовать нескольким отдельным сегментам сети на одном физическом канале или среде.

- Параметр `passphrase` позволяет установить пароль для аутентификации на интерфейсе. Этот параметр можно использовать как в сочетании с параметром `network_name`, так и отдельно.

- Опция `ifac_size` позволяет настроить длину кодов аутентификации интерфейса, передаваемых в каждом пакете в сегментах сети с именем и/или аутентифицированных сегментах. По умолчанию для них установлен размер, подходящий для данного интерфейса, но с помощью этой опции можно задать пользовательский размер от 8 до 512 бит. При обычном использовании значение этой опции не следует изменять по сравнению с значением по умолчанию.

- Опция `announce_cap` позволяет настроить максимальную пропускную способность, выделяемую в любой момент времени для распространения объявлений и другого трафика, связанного с обслуживанием сети. По умолчанию она настроена на 2%, и обычно нет необходимости её менять. Может быть установлена на любое значение от `1` до `100`.

  Если интерфейс превышает свой лимит объявлений, он помещает объявления в очередь для последующей передачи. Reticulum всегда отдаёт приоритет распространению объявлений от ближайших узлов. Это гарантирует, что приоритет отдаётся локальной топологии, а медленные сети не перегружаются соединёнными между собой быстрыми сетями.

  Пункти назначения, которые быстро повторно объявляют, будут еще больше понижены в приоритете. Попытка стать «первым в очереди» путем спам-объявлений приведет к прямо противоположному эффекту: перемещению в конец очереди каждый раз, когда поступает новое объявление от чрезмерно объявляющего пункта назначения.

Это означает, что всегда выгодно выбирать сбалансированную частоту объявлений и не объявляться чаще, чем это действительно необходимо для работы вашего приложения.

- Параметр `bitrate` настраивает скорость передачи данных интерфейса. Reticulum будет использовать скорости интерфейса, сообщаемые аппаратным обеспечением, или попытается угадать подходящую скорость, если аппаратное обеспечение не сообщает никаких данных. В большинстве случаев автоматически найденной скорости должно быть достаточно, но её можно настроить с помощью параметра `bitrate`, чтобы задать скорость интерфейса в *битах в секунду*.

- Параметр `bootstrap_only` определяет интерфейс в качестве временного моста для установки первоначального соединения. Если этот параметр включен, интерфейс будет отслеживаться и автоматически отключаться, как только количество автоматически подключенных интерфейсов достигнет предела, настроенного параметром `autoconnect_discovered_interfaces`. Это особенно полезно при использовании медленного или дорогостоящего соединения (такого как одиночный канал LoRa или удаленный TCP-туннель) исключительно для обнаружения более подходящей локальной инфраструктуры, которая затем заменит интерфейс начальной настройки.

Режимы интерфейса

Настройка опционального режима доступна на всех интерфейсах и позволяет выбирать поведение интерфейса на высоком уровне из ряда режимов. Эти режимы влияют на то, как Reticulum выбирает пути в сети, как распространяются объявления, как долго пути остаются действительными и как обнаруживаются пути.

Настройка режимов интерфейсов **не** является строго необходимой, но может оказаться полезной при построении или подключении к более сложным сетям. Если в вашем экземпляре Reticulum не запущен транспортный узел, настраивать режимы интерфейсов, и в таких случаях интерфейсы, как правило, следует оставлять в режиме по умолчанию.

- Режим по умолчанию — `full`. В этом режиме активированы все функции обнаружения, формирования ячеистой сети и транспорта.

- В режиме `шлюза` (или сокращенно `gw`) также доступны все функции обнаружения, формирования сетки и транспорта, но он дополнительно будет пытаться обнаруживать неизвестные пути от имени других
  узлов, находящихся на интерфейсе шлюза. Если Reticulum получает запрос на маршрут к неизвестному пункту назначения от узла на интерфейсе шлюза, он попытается обнаружить этот маршрут через все остальные
  активные интерфейсы и, в случае обнаружения пути, перенаправить его запрашивающему узлу.

  Если вы хотите разрешить другим узлам широко определять пути или подключаться к сети через интерфейс, может быть полезно перевести его в этот режим. Создав цепочку шлюзовых интерфейсов, другие узлы смогут сразу обнаруживать пути к любому пункту назначения по всей цепочке.

  Обратите внимание! Для того, чтобы это работало, в режим шлюза необходимо перевести интерфейс, *обращенный к клиентам*, а не интерфейс, обращенный к более широкой сети (хотя для этого может пригодиться режим границы).

- В режиме `access_point` (или сокращенно `ap`) интерфейс будет работать как точка доступа к сети. В этом режиме объявления не будут автоматически транслироваться на интерфейсе, а пути к
  адресатам на этом интерфейсе будет иметь гораздо более короткий срок действия
  Кроме того, запросы на маршрутизацию от клиентов на интерфейсе точки доступа будут обрабатываться так же, как и на интерфейсе шлюза.

  Этот режим полезен для создания интерфейсов, которые остаются неактивными, пока кто-то не начнет их использовать. Примером может служить радиоинтерфейс, обслуживающий обширную территорию, где ожидается, что пользователи будут подключаться на мгновение, использовать сеть, а затем снова исчезать.

- Режим `роуминга` следует использовать на интерфейсах, которые являются роуминговыми (физически мобильными) с точки зрения других узлов в
  сети. Например, если транспортное средство оснащено внешним интерфейсом LoRa и внутренним интерфейсом на базе Wi-Fi, который обслуживает устройства, перемещающиеся *вместе* с транспортным средством, то внешний

Интерфейс LoRa следует настроить в режиме роуминга, а внутренний интерфейс можно оставить в режиме по умолчанию. При включенном транспорте такая настройка позволит всем внутренним устройствам связываться друг с другом, а также со всеми другими устройствами, доступными на стороне LoRa сети, когда они находятся в зоне доступа. Устройства на стороне LoRa сети также смогут связываться с устройствами внутри транспортного средства, когда оно находится в зоне доступа. Маршруты через интерфейсы роуминга также истекают быстрее.

- Цель `граничного` режима — указать интерфейсы, которые устанавливают связь с сегментами сети, значительно отличающимися от того, в котором находится данный узел. Например, если экземпляр Reticulum является частью сети на базе LoRa, но также имеет высокоскоростное соединение с общедоступным транспортным узлом
, доступный в Интернете, интерфейс, подключающийся через Интернет, должен быть переведен в пограничный режим.

Таблицу, описывающую влияние всех режимов на распространение объявлений, см. в разделе «Правила распространения объявлений».

Управление частотой объявлений

Встроенные механизмы управления объявлениями и описанный выше параметр `announce_cap` по умолчанию в большинстве случаев являются достаточными, однако в некоторых случаях, особенно на высокоскоростных интерфейсах, может оказаться полезным регулировать целевую скорость объявлений. С помощью параметров `announce_rate_target`, `announce_rate_grace` и `announce_rate_penalty` это можно сделать для каждого интерфейса отдельно, что позволяет регулировать *скорость, с которой полученные объявления ретранслируются на другие интерфейсы*.

- Параметр `announce_rate_target` устанавливает минимальное время в секундах, которое должно пройти между полученными объявлениями для любого одного адресата. Например, установка этого значения равным
`3600` означает, что объявления, *полученные* на этом интерфейсе, будут ретранслироваться и распространяться на другие интерфейсы только один раз в час, независимо от того, как часто они поступают.

- Необязательный параметр `announce_rate_grace` определяет, сколько раз адресат может нарушать скорость объявлений, прежде чем начнет применяться целевая скорость.

- Необязательный параметр `announce_rate_penalty` настраивает дополнительное время, которое добавляется к нормальной целевой частоте. Например, если определено наказание в `7200` секунд, то после того, как целевая частота будет принудительно применена, объявления данного адресата будут распространяться только каждые 3 часа, пока он не снизит свою фактическую
частоту объявлений до уровня, не превышающего целевой.

Эти механизмы в сочетании с упомянутыми выше механизмами `annouce_cap` означают, что крайне важно выбрать сбалансированную стратегию объявления для ваших адресатов. Чем более сбалансированным будет ваше решение, тем проще будет вашим адресатам проникнуть в более медленные сети, расположенные на большом расстоянии. Или же вы можете отдать приоритет только доступу к сетям с высокой пропускной способностью, используя более частые объявления.

Текущую статистику и информацию о частоте объявлений можно просмотреть с помощью команды `rnpath -r`.

Важно отметить, что не существует единственно правильного или неправильного способа настройки частоты объявлений. Более медленные сети, естественно, будут стремиться использовать менее
объявлений, чтобы экономить пропускную способность, в то время как очень быстрые сети могут поддерживать приложения, которым требуются очень частые объявления. Reticulum
реализует эти механизмы, чтобы обеспечить беспроблемное *сосуществование* и взаимодействие широкого спектра типов сетей.

Ограничение частоты объявлений о новых пунктах назначения

На публичных интерфейсах, к которым может подключиться любой и объявить новые пункты назначения, может быть полезно контролировать скорость обработки объявлений о *новых* пунктах назначения.

Если в течение короткого промежутка времени поступает большое количество объявлений о вновь созданных или ранее неизвестных
пунктов назначения, Reticulum поместит эти

объявления в очередь ожидания, чтобы трафик объявлений для известных и ранее установленных пунктов назначения мог продолжать обрабатываться без перерывов.

После того как всплеск утихнет и пройдет дополнительный период ожидания, отложенные объявления будут постепенно освобождаться, пока очередь ожидания не будет очищена. Это также означает, что если узел решит подключиться к публичному интерфейсу, объявить большое количество фиктивных адресатов, а затем отключиться, эти адресаты никогда не попадут в таблицы маршрутизации и не будут тратить пропускную способность сети на повторную передачу объявлений.

Важно отметить, что контроль входящего трафика работает на уровне *отдельных подинтерфейсов*. Например, это означает, что один клиент на интерфейсе TCP-сервера не может нарушить обработку входящих объявлений для других подключенных клиентов на том же интерфейсе TCP-сервера. У всех остальных клиентов на том же интерфейсе новые объявления по-прежнему будут обрабатываться без перерывов.

По умолчанию Reticulum обрабатывает это автоматически, и контроль входящих объявлений будет включен на интерфейсе, где это целесообразно. Как правило, нет необходимости изменять конфигурацию контроля входящего трафика, но все параметры доступны для настройки в случае необходимости.

- Параметр `ingress_control` указывает Reticulum, следует ли включить контроль входящего трафика по протоколу Announce на данном интерфейсе. По умолчанию установлено значение `True`.

- Параметр `ic_new_time` определяет, в течение какого времени (в секундах) интерфейс считается вновь созданным. По умолчанию установлено значение `2*60*60` секунд. Этот параметр полезен для общедоступных интерфейсов, которые создают новые подинтерфейсы при подключении нового клиента.

- Параметр `ic_burst_freq_new` устанавливает максимальную частоту входящих объявлений для вновь созданных интерфейсов. По умолчанию — `3,5` объявлений в секунду.

- Параметр `ic_burst_freq` задает максимальную частоту входящих объявлений для других интерфейсов. По умолчанию — `12` объявлений в секунду.

Если интерфейс превышает свою частоту всплеска, входящие объявления для неизвестных адресатов будут временно удерживаться в очереди и обрабатываться позже.

- Параметр `ic_max_held_announces` задает максимальное количество уникальных объявлений, которые будут удерживаться в очереди. Любые дополнительные уникальные объявления будут отбрасываться. По умолчанию — `256` объявлений.

- Параметр `ic_burst_hold` определяет, сколько времени (в секундах) должно пройти после того, как частота всплеска опустится ниже порогового значения, чтобы всплеск оповещения считался устраненным. По умолчанию установлено значение `60` секунд.

- Параметр `ic_burst_penalty` определяет, сколько времени (в секундах) должно пройти после того, как всплеск будет считаться очищенным, прежде чем удерживаемые
объявления, находящиеся в режиме удержания, могут начать выводиться из очереди. По умолчанию
`5*60` секунд.

- Параметр `ic_held_release_interval` определяет, сколько времени (в секундах) должно пройти между освобождением каждого удерживаемого объявления из
очереди. По умолчанию — `30` секунд.

Построение сетей

В этой главе представлены общие сведения, необходимые для построения сетей с помощью Reticulum. Однако она не расскажет вам всего, что нужно знать для успешного проектирования и настройки всех возможных типов сетей. Для этого вам, скорее всего, потребуется прочитать данное руководство полностью, уделить значительное время экспериментам со стеком и интуитивному освоению его функциональности.

Тем не менее, прочитав эту главу, вы должны быть хорошо подготовлены к тому, чтобы *начать* это путешествие. Хотя Reticulum **принципиально отличается** от других сетевых технологий, его использование зачастую может оказаться проще, чем применение традиционных стеков. Если вы раньше занимались построением сетей, вам, вероятно, придётся забыть или, по крайней мере, временно игнорировать многие вещи на данном этапе. В конце концов всё станет понятно. Надеюсь.

Если вы привыкли к протоколам типа IP, давайте хотя бы начнем с облегчения: вам не нужно беспокоиться о согласовании адресов, подсетей и маршрутизации для всей сети, о том, как она будет развиваться в будущем, вы, возможно, не знаете. С Reticulum вы можете просто добавлять новые сегменты в свою сеть, когда это становится необходимым, и Reticulum автоматически обеспечит конвергенцию всей сети автоматически. В Reticulum есть еще много других интересных аспектов, но мы забегаем вперед. Давайте сначала рассмотрим основы.

Концепции и обзор

Прежде чем приступить к созданию собственных сетей, важно понять основные принципы, отличающие сети Reticulum от традиционных подходов к построению сетей. Именно эти принципы определяют, как вы проектируете
сеть, с какими компромиссами вы столкнетесь и на какие возможности вы можете рассчитывать.

Reticulum — это не отдельная сеть, к которой вы «присоединяетесь», а набор инструментов для *создания* сетей. Вы сами решаете, какие среды использовать, как соединять узлы, какие границы доверия устанавливать и какова цель сети. Reticulum предоставляет

криптографическую основу, механизмы передачи данных и алгоритмы конвергенции, которые делают ваш проект реализуемым. Вы определяете цель и структуру.

Такой подход обеспечивает огромную гибкость, но требует мышления в терминах абстракций, отличных от тех, что используются в традиционных сетях.

Вводные соображения

При построении сетей с помощью Reticulum необходимо помнить о следующих важных моментах:

- В сети Reticulum любой узел может самостоятельно генерировать столько адресов (в терминологии Reticulum называемых *«пунктами назначения»*), сколько ему необходимо, и эти адреса становятся доступными для остальной части сети. Централизованного управления адресным пространством не существует.

- Reticulum был разработан для работы как с очень маленькими, так и с очень большими сетями. Хотя адресное пространство может поддерживать миллиарды конечных точек, Reticulum также очень полезен, когда необходимо обеспечить связь всего между несколькими устройствами.

- Сети с низкой пропускной способностью, такие как LoRa и пакетная радиосвязь, могут без проблем взаимодействовать и соединяться с гораздо более крупными сетями с более высокой пропускной способностью без каких-либо проблем. Reticulum автоматически управляет потоком информации, поступающей в различные сегменты сети и исходящей из них, и при ограниченной пропускной способности локальному трафику
  . Однако вам необходимо правильно настроить интерфейсы. Если вы укажете Reticulum передавать весь трафик объявлений с гигабитного соединения на интерфейсы LoRa, он постарается выполнить это как можно лучше, не превышая при этом ограничений пропускной способности, но вы потратите много ценной пропускной способности и эфирного времени, и ваша сеть LoRa будет работать не очень хорошо.

- Reticulum по умолчанию обеспечивает анонимность отправителя/инициатора. Нет возможности фильтровать трафик или различать его на основе источника

трафика.

- Весь трафик шифруется с использованием временных ключей, генерируемых при обмене ключами по алгоритму Диффи-Хеллмана с эллиптической кривой Curve25519.
  Нет возможности проверять содержимое трафика, а также нет возможности устанавливать приоритеты или ограничивать определенные виды трафика. Таким образом, все транспортные и маршрутизационные уровни полностью независимы от типа трафика и будут пропускать весь трафик одинаково.

- Reticulum может функционировать как с инфраструктурой, так и без нее. Когда *транспортные узлы* доступны, они могут маршрутизировать трафик через несколько прыжков для других узлов и будут функционировать как распределенное хранилище криптографических ключей. Когда транспортные узлы недоступны, все узлы, находящиеся в радиусе связи, все равно могут обмениваться данными.

- Каждый узел может стать транспортным, просто включив эту функцию в своей конфигурации, но нет необходимости, чтобы каждый узел в сети был транспортным. Если позволить каждому узлу быть транспортным, это в большинстве случаев ухудшит производительность и надежность сети.

  > *В общем, если узел стационарен, имеет хорошее подключение и работает большую часть времени, он является хорошим кандидатом на роль транспортного узла. Для оптимальной производительности сеть должна содержать количество транспортных узлов, обеспечивающее подключение к предполагаемой области / топографии, и не намного больше.*

- Reticulum разработан для надёжной работы в открытых средах, не требующих доверия. Это означает, что с его помощью можно создавать сети с открытым доступом, в которые участники могут вступать и из которых могут выходить свободно и без какой-либо организации. Данное свойство открывает путь для совершенно нового и пока что в основном неисследованного класса сетевых приложений, в которых сети и информационные потоки внутри них могут органично формироваться и
  распадаться органично.

- Вы можете с такой же легкостью создавать закрытые сети, поскольку Reticulum позволяет добавлять аутентификацию к любому интерфейсу. Это означает, что вы можете ограничить доступ на любом типе интерфейса, даже при использовании устаревших устройств, таких как модемы. Вы также можете смешивать аутентифицированные и открытые интерфейсы в одной системе. См. раздел «Общие параметры интерфейса» в главе «Интерфейсы» данного руководства для получения информации о том, как настроить аутентификацию интерфейса.

Reticulum позволяет объединять в единую ячеистую сеть самые разные типы сетевых сред или использовать только одну среду. Вы можете построить «виртуальную сеть», полностью работающую через Интернет, в которой все узлы обмениваются данными по «каналам» TCP и UDP. Вы также можете построить такую сеть, используя в качестве базовой среды для Reticulum другие уже существующие каналы связи.

Однако большинство реальных сетей, вероятно, будут включать либо какую-либо форму беспроводной связи, либо прямую проводную связь. Чтобы Reticulum мог Для связи через любой тип среды необходимо указать его в файле конфигурации, который по умолчанию находится в папке `~/.reticulum/config`. Примеры настройки интерфейсов см. в главе «Поддерживаемые интерфейсы» данного руководства.

Можно настроить любое количество интерфейсов, и Reticulum автоматически решит, какие из них подходят для использования в данной ситуации, в зависимости от того, куда должен проходить трафик.

Пункты назначения, а не адреса

В традиционных сетях адреса выделяются из управляемого пространства. Если вы хотите общаться с другим узлом, вам нужно знать его адрес, и этот адрес должен быть уникальным в пределах сегмента сети. Это требует координации, либо посредством ручного назначения, либо с помощью серверов DHCP, либо других механизмов выделения.

Reticulum заменяет адреса на **пункты назначения**. Пункт назначения идентифицируется 16-байтовым хешем (128 бит), полученным из хеша SHA-256 идентифицирующих характеристик пункта назначения. Этот хеш служит адресом на

в сети. В сети он представлен в двоичном виде, но при отображении для пользователей обычно выглядит примерно так
`<13425ec15b621c1d928589718000d814>` .

Ключевое отличие заключается в том, что *любой узел может генерировать столько адресатов, сколько ему нужно, без координации*. Уникальность адресата гарантируется устойчивостью SHA-256 к коллизиям и включением в вычислении хеша. Два узла могут использовать одно и то же имя пункта назначения `messenger.user.inbox`, но у них будут разные хеши пункта назначения , поскольку их открытые ключи различаются. Оба могут сосуществовать в одной сети без конфликтов.

Это имеет серьезные последствия для проектирования сети:

- **Отсутствие планирования распределения адресов:** вам никогда не нужно резервировать диапазоны адресов, планировать подсети или согласовывать действия с другими сетевыми операторами.
  Узлы просто генерируют адреса назначения и объявляют их.
- **Глобальная переносимость:** пункт назначения не привязан к физическому местоположению или сегменту сети. Узел может перемещать свои пункты назначения между интерфейсами, средами передачи данных или даже между совершенно отдельными сетями Reticulum, просто отправив объявление по новой среде передачи данных.
- **Неявная аутентификация:** поскольку пункты назначения привязаны к открытым ключам, связь с пунктом назначения по сути проходит криптографическую аутентификацию. Только владелец соответствующего закрытого ключа может расшифровать и ответить на трафик, адресованный этому пункту назначения. Это также *значительно* упрощает аутентификацию на уровне приложений, поскольку позволяет напрямую использовать базовую проверку идентичности, встроенную в основной сетевой уровень.
- **Абстракция идентичности:** одна идентичность Reticulum может создавать несколько адресатов. Это позволяет одному субъекту (человеку, устройству, службе) представлять несколько конечных точек без необходимости использования нескольких криптографических пар ключей.

Транспортные узлы и экземпляры

Reticulum различает два типа узлов: **экземпляры** и **транспортные узлы**. Каждый узел, на котором запущен Reticulum, является экземпляром, но не

каждый экземпляр является транспортным узлом.

Инстанс Reticulum — это любая система, на которой работает стек Reticulum. Он может создавать пункты назначения, отправлять и принимать пакеты, устанавливать соединения и
обмениваться данными с другими узлами. Он также может размещать пункты назначения, доступные
доступны для подключения любому пользователю в сети. Это означает, что вы можете легко размещать глобально доступные сервисы из любого места, включая ваш дом или офис. Глобальная доступность всех пунктов назначения по всей сети гарантирована, при условии наличия физического способа фактической передачи пакетов.
Инстанции являются состоянием по умолчанию и подходят для большинства конечных пользовательских устройств, таких как телефоны, ноутбуки, датчики или любые устройства, которые в основном потребляют сетевые сервисы.

Транспортный узел — это экземпляр, явно настроенный для участия в общесетевой передаче данных. Транспортные узлы пересылают пакеты между узлами, распространяют объявления, ведут таблицы маршрутов и обслуживают запросы на маршруты
от имени других узлов. Когда пункт назначения отправляет объявление, транспортные узлы принимают его, запоминают маршрут и ретранслируют его на другие интерфейсы. Когда узлу необходимо достичь пункта назначения, к которому у него нет маршрута, транспортные узлы помогают определить маршрут через сеть.

Даже устройства, на которых размещены сервисы или которые предоставляют контент, вероятно, следует просто настроить как экземпляры, а самим подключаться к более широким сетям через транспортный узел. В некоторых ситуациях это может быть нецелесообразно, однако, например, вполне возможно разместить личный транспортный узел на Raspberry Pi, при этом одновременно запуская узел распространения LXMF и размещая свой личный сайт или файлы через Reticulum.

Это различие важно. **Не** каждый узел должен быть транспортным узлом:

- **Потребление ресурсов:** транспортные узлы ведут таблицы маршрутов, обрабатывают объявления и пересылают трафик. Для этого требуются ресурсы памяти и процессора
  , которые на устройствах с низким энергопотреблением могут быть ограничены.
- **Требования** к **стабильности:** транспортные узлы способствуют конвергенции сети. Если транспортные узлы часто отключаются, таблицы маршрутов
  устаревают, что негативно сказывается на конвергенции. Стабильные, постоянно подключенные узлы являются лучшими транспортными узлами.

- **Соображения, связанные с пропускной способностью:** Транспортные узлы обрабатывают и ретранслируют трафик обслуживания сети. В средах с очень низкой пропускной способностью наличие слишком большого количества транспортных узлов будет потреблять емкость, которая должна использоваться для передачи фактических данных.

На практике сеть обычно имеет относительно небольшое количество транспортных узлов, стратегически размещенных для обеспечения покрытия и подключения. Устройства конечных пользователей работают как экземпляры, подключаясь через ближайшие транспортные узлы для выхода в более широкую сеть. Эта модель отражает традиционные сети, где маршрутизаторы пересылают трафик, а конечные хосты просто потребляют подключение, но с важным отличием: любой узел *может* стать маршрутизатором при необходимости, и решение принимаете вы, исходя из требований вашей сети.

Транспортные узлы также функционируют как распределенные хранилища криптографических ключей. Когда пункт назначения объявляет о себе, транспортные узлы кэшируют открытый ключ и информацию о пункте назначения. Другие узлы могут запрашивать неизвестные открытые ключи у сети, и транспортные узлы отвечают кэшированной информацией.
Это устраняет необходимость в централизованной службе каталогов, при этом обеспечивая доступность открытых ключей по всей сети.

Сети без доверия

Традиционные модели сетевой безопасности предполагают высокий уровень доверия на определенных уровнях. Вы можете доверять своему интернет-провайдеру доставку пакетов без проверки, доверять своему провайдеру VPN обработку вашего трафика или доверять сетевому администратору правильную настройку брандмауэров. Эти доверительные отношения создают уязвимости и зависимости.

Reticulum разработан для работы в **открытых средах, не требующих доверия**. Это означает, что протокол не делает никаких предположений о надежности сетевой инфраструктуры, других участников или транспортных средств.
Каждый аспект связи защищен с помощью криптографии:

- **Шифрование трафика:** весь трафик, направляемый в отдельные пункты назначения, шифруется с использованием временных ключей.
- **Анонимность источника:** пакеты Reticulum не содержат адресов источника. Наблюдатель, перехвативший пакет, не может определить, кто

его отправил, а только то, кому он адресован (если не включена функция IFAC, в этом случае ничего определить невозможно). Это обеспечивает анонимность инициатора по умолчанию.
- **Проверка маршрута:** механизм объявления включает криптографические подписи, подтверждающие подлинность объявлений о пунктах назначения.
- **Неподдельные подтверждения доставки:** когда адресат подтверждает получение пакета, это подтверждение подписывается ключом идентификации адресата. Это предотвращает ложные подтверждения и обеспечивает надежную проверку доставки.
- **Аутентификация интерфейса:** при использовании кодов доступа к интерфейсу (IFAC) пакеты на аутентифицированных интерфейсах содержат подписи, полученные на основе общего секрета. Только узлы с правильным именем сети и парольной фразой могут генерировать действительные пакеты, что позволяет создавать виртуальные частных сетей на общих средах.

Концепция «без доверия» имеет важные последствия для проектирования сетей:

- **Сети с открытым доступом являются жизнеспособными:** можно создавать сети, к
к которым может присоединиться любой без предварительного одобрения. Поскольку трафик шифруется и аутентифицируется от конца до конца, участники не могут вмешиваться в частную коммуникацию друг друга, даже если они используют одну и ту же транспортную инфраструктуру.
- **Отсутствие проверки или приоритизации трафика:** поскольку содержимое и источники трафика непрозрачны для промежуточных узлов, не существует механизма фильтрации, приоритизации или ограничения трафика на основе его типа или происхождения. Весь трафик обрабатывается одинаково. С точки зрения нейтральности это является преимуществом.
- **Устойчивость к атакам:** сеть может функционировать даже в том случае, если некоторые узлы являются вредоносными или находятся под контролем злоумышленников. Хотя вредоносный транспортный узел может отказаться пересылать определённый трафик или отбрасывать пакеты, он не может расшифровывать, изменять или подделывать легитимный трафик. Резервные пути и наличие нескольких транспортных узлов позволяют смягчить последствия действий вредоносных узлов.

Конечно, вы также можете создавать закрытые сети. Коды доступа к интерфейсам позволяют ограничить доступ к определенным интерфейсам. Сетевые идентификаторы позволяют убедиться, что обнаруженные интерфейсы принадлежат доверенным операторам.

Управление «черными дырами» позволяет блокировать вредоносные идентичности. Reticulum
предоставляет как инструменты для открытых сетей, так и средства управления для закрытых. Выбор остается за вами в зависимости от ваших требований.

Гетерогенная связь

В традиционных сетях для объединения различных транспортных средств обычно требуются шлюзы, уровни преобразования и тщательная настройка. Сеть Wi-Fi изначально не взаимодействует с сетью пакетной радиосвязи без дополнительной инфраструктуры, и вы не можете просто загрузить автомобиль через последовательный порт или отправить зашифрованное сообщение в виде QR-кода.

Reticulum рассматривает **гетерогенность как основную предпосылку**. Протокол разработан для беспрепятственного объединения сред связи с кардинально разными характеристиками:

- **Пропускная способность:** каналы LoRa, работающие со скоростью в несколько сотен бит в секунду, могут соединяться с магистралями Gigabit Ethernet. Reticulum автоматически управляет потоком информации, отдавая приоритет локальному трафику на медленных сегментах и обеспечивая при этом глобальную конвергенцию.
- **Задержка:** спутниковые каналы связи с задержкой в несколько секунд могут сосуществовать с локальными каналами связи, задержка которых измеряется в миллисекундах. Транспортная система прозрачно обрабатывает синхронизацию, асинхронную доставку и повторные передачи.
- **Топология:** микроволновые каналы «точка-точка», радиосети широковещательной передачи, коммутируемые Ethernet-фабрики и виртуальные туннели через Интернет могут быть частью одной и той же сети Reticulum.
- **Надежность:** Прерывистые соединения, которые появляются и исчезают (такие как мобильные устройства или случайные радиосвязи), могут работать наряду с постоянно включенной инфраструктурой. Reticulum плавно обрабатывает потерю соединения и повторное подключение.

Эта гетерогенность достигается за счет нескольких элементов конструкции:

- **Расширяемая система интерфейсов, независимая от среды:** Reticulum взаимодействует с физическим миром через интерфейсные модули. Для добавления поддержки новой среды передачи данных достаточно реализовать класс интерфейса. Сам протокол при этом остается неизменным.

- **Режимы интерфейсов:** Различные режимы (`full` , `gateway` , `access_point` , `roaming` , `boundary`) позволяют настраивать взаимодействие интерфейсов с более широкой сетью в зависимости от их характеристик и роли.
- **Правила распространения объявлений:** объявления пересылаются между интерфейсами в соответствии с правилами, учитывающими ограничения пропускной способности и режимы интерфейсов. Медленные сегменты не перегружаются трафиком из быстрых сегментов.
- **Приоритезация локального трафика:** при ограниченной пропускной способности Reticulum устанавливает приоритет объявлений для ближайших пунктов назначения. Это гарантирует, что локальная связь останется работоспособной, даже если глобальная конвергенция не завершена.

Для проектировщиков сетей это означает, что вы можете свободно использовать любые среды, которые доступны, доступны по цене или подходят для вашей ситуации. Вы можете использовать LoRa для широкополосного покрытия с низкой пропускной способностью, Wi-Fi для локальных каналов связи с высокой пропускной способностью, I2P для анонимного подключения к Интернету и Ethernet для инфраструктурных магистралей — и все это в рамках одной сети. Reticulum автоматически занимается преобразованием и координацией.

Ключевым моментом при проектировании является не то, могут ли различные среды взаимодействовать друг с другом (они могут), а то, **как** они должны взаимодействовать с учетом ваших целей. Узел с несколькими интерфейсами, охватывающими разнородные среды, необходимо настроить с использованием соответствующих режимов интерфейсов, чтобы трафик проходил эффективно. Шлюз, соединяющий медленный сегмент LoRa с быстрой магистралью Интернета, должен быть настроен иначе, чем мобильное устройство, перемещающееся между радиоячейками.

Поддержите Reticulum

Вы можете помочь поддержать дальнейшее развитие открытых, свободных и частных систем связи, делая пожертвования, оставляя отзывы, а также предоставляя код и учебные ресурсы.

Пожертвования

Мы с благодарностью принимаем пожертвования через следующие каналы:

```
Monero: 84FpY1QbxHcgdseePYNmhTHcrgMX4nFfBYtz2GKYToqHVVhJp8Eaw1Z1EedRnKD
19b3B8NiLCGVxzKV17UMmmeEsCrPyA5w

Bitcoin: bc1pgqgu8h8xvj4jtafslq396v7ju7hkgymyrzyqft4llfslz5vp99psqfk3a6

Ethereum: 0x91C421DdfB8a30a49A71d63447ddb54cEBe3465E

Liberapay:
https://liberapay.com/Reticulum/

Ko-Fi:
https://ko-fi.com/markqvist
```

Есть ли в плане развития функции, которые важны для вас или вашей организации? Помогите их быстро реализовать, спонсируя их внедрение.

Оставить отзыв

Отзывы об использовании, работе и возможных сбоях в работе любых компонентов системы имеют огромное значение для дальнейшего развития и совершенствования Reticulum. Но…

Предупреждение

Подумайте, прежде чем говорить. Как показало время, более 80% «отзывов», «сообщений об ошибках» и «советов», полученных проектом Reticulum, оказались бесполезным шумом, вызванным ошибочными предположениями, непониманием базовой функциональности или философии системы, либо просто неверными (но чрезмерно категоричными) личными
предпочтения отдельных случайных архитекторов. Это тратит время всех участников.

Проект Reticulum — это не общественная чайная, призванная удовлетворять потребности в внимании случайных прохожих, а чрезвычайно сложная система, спроектированная и
усовершенствованная на протяжении более десяти лет, предназначенная для обеспечения гарантий связи и доступности в крайне неблагоприятных условиях.

Если вы хотите высказать своё мнение, оно должно быть обоснованным, и мы ожидаем, что ваши аргументы будут опираться на всестороннюю и прочную основу. Все остальное будет игнорироваться.

Reticulum ни при каких обстоятельствах не собирает и не предоставляет никаких данных автоматической аналитики, телеметрии, отчетов об ошибках или статистики, поэтому мы полагаемся на традиционные отзывы пользователей.

Примеры кода

В исходный дистрибутив Reticulum включен ряд примеров. Вы можете использовать эти примеры, чтобы научиться писать собственные программы.

Minimal

Пример *Minimal* демонстрирует минимальную настройку, необходимую для подключения к сети Reticulum из вашей программы. Примерно за пять строк кода вы инициализируете сетевой стек Reticulum и подготовите его к передаче трафика в вашей программе.

```
##########################################################
# В этом примере RNS показана минимальная конфигурация, которая        # #
запускает стек Reticulum Network Stack, создает # # новый пункт назначения и
позволяет пользователю отправить объявление.                          #
##########################################################

import argparse
import sys import
RNS

# Давайте определим имя приложения. Мы будем использовать его для
всех # создаваемых нами адресатов. Поскольку этот базовый пример
# является частью ряда примеров утилит, мы поместим
# их все в пространство имен приложения "example_utilities" APP_NAME =
"example_utilities"

# Эта инициализация выполняется при запуске программы def
program_setup(configpath):
     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)

     # Создадим случайную новую идентичность для нашего примера: identity =
     RNS.Identity()

     # Используя только что созданную идентичность, создаем пункт назначения.
```

```
    # Пункты назначения — это конечные точки в Reticulum, к которым можно
обратиться
    # и с которыми можно обмениваться данными. Пункты назначения также могут
объявлять о
    # существование, что позволит сети узнать, что они доступны
    # и автоматически создавать пути к ним из любой другой точки
    # в сети.
    destination = RNS.Destination(
        identity, RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "minimalsample"
    )

    # Мы настраиваем пункт назначения на автоматическую проверку всех # пакетов,
    адресованных ему. Благодаря этому RNS будет
автоматически
    # генерировать контрольную сумму для каждого входящего пакета и отправлять её #
    обратно отправителю этого пакета. Это позволит любому
, кто
    # пытается установить связь с адресатом, узнать, была ли его
    # сообщение было получено правильно.
    destination.set_proof_strategy(RNS.Destination.PROVE_ALL)

    # Всё готово!
    # Передадим управление циклу объявления announceLoop(destination)


def announceLoop(destination):
    # Сообщить пользователю, что все готово RNS.log(
        "Минимальный пример "+
        RNS.prettyhexrep(destination.hash)+
        " запущен, нажмите Enter, чтобы вручную отправить объявление
(Ctrl-C для выхода)"
    )

    # Мы входим в цикл, который будет работать, пока пользователь не выйдет.
```

```
     # Если пользователь нажмет Enter, мы объявим наш сервер # в сети, что
     позволит клиентам
     # уметь создавать сообщения, адресованные ему. while True:
          entered = input()
          destination.announce()
          RNS.log("Отправлено объявление от " + RNS.prettyhexrep(destination.hash))


############################################################  ####  Запуск
программы                                  ####################################
############################################################

# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем запускает # нужный режим
работы программы.
if ___name____ == "___main___":
     try:
          parser = argparse.ArgumentParser(
               description="Минимальный пример запуска Reticulum и создания пункта
назначения"
          )

          parser.add_argument( "--
               config",
               action="store",
               default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          args = parser.parse_args() if

          args.config:
               configarg = args.config else:
               configarg = None

          program_setup(configarg)

     except KeyboardInterrupt:
```

```
        print("")
        sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Minimal.py.

Объявление

Пример *«Announce»* основан на предыдущем примере и посвящён тому, как объявить пункт назначения в сети и как настроить программу так, чтобы она получать уведомления об объявлениях от соответствующих пунктов назначения.

```
##########################################################
# Этот пример RNS демонстрирует настройку обратных вызовов # # для
объявлений, которые позволят приложению получать # # уведомление при
поступлении актуального для него объявления #
##########################################################

import argparse
import random
import sys import
RNS

# Давайте определим имя приложения. Мы будем использовать его для
всех # создаваемых нами адресатов. Поскольку этот базовый пример
# является частью ряда примеров утилит, мы поместим
# их все в пространство имен приложения "example_utilities" APP_NAME =
"example_utilities"

# Инициализируем два списка строк для использования в качестве app_data fruits =
["Персик", "Айва", "Финик", "Мандарин", "Помело", "Карамбола", "Виноград"]
noble_gases = ["Гелий", "Неон", "Аргон", "Криптон", "Ксенон", "Радон", "Оганессон"]

# Эта инициализация выполняется при запуске программы def
program_setup(configpath):
     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)
```

```
    # Случайным образом создаем новую идентичность для нашего примера
    identity = RNS.Identity()

    # Используя только что созданную идентичность, создаем два назначения
    # в пространстве приложения «example_utilities.announcesample».
    #
    # Пункты назначения — это конечные точки в Reticulum, к которым можно
обратиться
    # и с которыми можно обмениваться данными. Пункты назначения также могут
объявлять о
    # существование, что позволит сети узнать, что они доступны
    # и автоматически создавать пути к ним из любой другой точки
    # в сети.
    destination_1 = RNS.Destination(
        identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "announcesample", "fruits"
    )

    destination_2 = RNS.Destination(
        identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "announcesample",
        "noble_gases"
    )

    # Мы настраиваем пункты назначения на автоматическую проверку всех # пакетов,
    адресованных им. Благодаря этому RNS будет
автоматически
    # генерировать подтверждение для каждого входящего пакета и передавать его # обратно
    отправителю этого пакета. Это позволит любому
, кто
    # пытается установить связь с пунктом назначения, узнать, была ли его
```

```
    # сообщение было получено правильно.
    destination_1.set_proof_strategy(RNS.Destination.PROVE_ALL)
    destination_2.set_proof_strategy(RNS.Destination.PROVE_ALL)

    # Мы создаем обработчик объявлений и настраиваем его так, чтобы он запрашивал
    # объявления от "example_utilities.announcesample.fruits". # Попробуйте
    изменить фильтр и посмотрите, что произойдет. announce_handler =
    ExampleAnnounceHandler(
        aspect_filter="example_utilities.announcesample.fruits"
    )

    # Мы регистрируем обработчик объявлений в Reticulum
    RNS.Transport.register_announce_handler(announce_handler)

    # Всё готово!
    # Передадим управление циклу объявлений announceLoop(destination_1,
    destination_2)


def announceLoop(destination_1, destination_2): # Сообщим
    пользователю, что все готово
    RNS.log("Пример объявления запущен, нажмите Enter, чтобы вручную отправить
объявление (Ctrl-C для выхода)")

    # Мы входим в цикл, который будет работать, пока пользователь не выйдет.
    # Если пользователь нажмет Enter, мы объявим о нашем сервере # в сети,
    что позволит клиентам
    # узнать, как создавать сообщения, адресованные ему. while True:
        entered = input()

        # Случайный выбор фрукта
        fruit = fruits[random.randint(0,len(fruits)-1)]

        # Отправить объявление, включая данные приложения
        destination_1.announce(app_data=fruit.encode("utf-8")) RNS.log(
            "Отправлено объявление от "+
            RNS.prettyhexrep(destination_1.hash)+ "
            ("+destination_1.name+")"
        )
```

```
          # Случайный выбор инертного газа
          noble_gas =
noble_gases[random.randint(0,len(noble_gases)-1)]

          # Отправить объявление, включая данные приложения
          destination_2.announce(app_data=noble_gas.encode("utf-
8"))
          RNS.log(
               "Отправлено объявление от "+
               RNS.prettyhexrep(destination_2.hash)+ "
               ("+destination_2.name+")"
          )

# Нам нужно определить класс обработчика уведомлений, # которому Reticulum
сможет отправлять сообщения при поступлении уведомления.
class ExampleAnnounceHandler:
     # Метод инициализации принимает опциональный
     # аргумент aspect_filter. Если aspect_filter установлен в
     # None, все объявления будут переданы экземпляру. # Если требуются только
     некоторые объявления, его можно задать # в виде строки аспекта.
     def       init(self,       aspect_filter=None):
          self.aspect_filter = aspect_filter

     # Этот метод будет вызван системой транспорта Reticulums # при
     поступлении объявления, соответствующего
     # настроенному фильтру аспектов. Фильтры должны быть конкретными, # и
     не могут использовать подстановочные знаки.
     def received_announce(self, destination_hash, announced_identity,
app_data):
          RNS.log(
               "Получено объявление от "+ RNS.prettyhexrep(destination_hash)
          )

          if app_data:
               RNS.log(
                    "Объявление содержало следующие данные приложения:
"+
                    app_data.decode("utf-8")
               ) ############################################################
```

```
#### Запуск программы ####################################
############################################################

# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем запускает # нужный режим
работы программы.
if ___name____ == "___main___":
     try:
          parser = argparse.ArgumentParser( description="Пример Reticulum,
               демонстрирующий
объявления и обработчики объявлений"
          )

          parser.add_argument( "--
               config",
               action="store",
               default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          args = parser.parse_args() if

          args.config:
               configarg = args.config else:
               configarg = None

          program_setup(configarg)

     except KeyboardInterrupt:
          print("")
          sys.exit(0)
```

Этот пример также можно найти на сайте
https://github.com/markqvist/Reticulum/blob/master/Examples/Announce.p y.

Широковещательная рассылка

Пример *«Broadcast»* демонстрирует, как передавать широковещательные сообщения в виде открытого текста по сети.

```
##########################################################
# Этот пример RNS демонстрирует широковещательную передачу незашифрованной # #
информации на любые принимающие адреса.                                    #
##########################################################

import sys import
argparse import RNS

# Давайте определим название приложения. Мы будем использовать его
для всех # создаваемых нами пунктов назначения. Поскольку этот базовый
пример
# является частью набора примеров утилит, мы поместим
# их все в пространство имен приложения «example_utilities» APP_NAME =
"example_utilities"

# Эта инициализация выполняется при запуске программы def program_setup(configpath,
channel=None):
     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)

     # Если пользователь не выбрал «канал», мы используем # канал по
     умолчанию под названием «public_information».
     # Этот «канал» добавляется в пространство имен назначения,
     # пространство, чтобы пользователь мог выбирать разные каналы #
     вещания.
     if channel == None:
          channel = "public_information"

     # Мы создаем пункт назначения PLAIN. Это незашифрованная конечная точка
     # который может прослушивать и на который может отправлять информацию любой
     пользователь. broadcast_destination = RNS.Destination(
          None, RNS.Destination.IN,
          RNS.Destination.PLAIN,
          APP_NAME,
          "broadcast",
          channel
     )
```

```
    # Мы указываем обратный вызов, который будет вызываться каждый раз, # когда
    пункт назначения получает данные.
    broadcast_destination.set_packet_callback(packet_callback)

    # Все готово!
    # Передадим управление в главный цикл broadcastLoop(broadcast_destination)

def packet_callback(data, packet):
    # Просто выводим полученные данные print("")
    print("Полученные данные: "+data.decode("utf-8")+"\r\n> ", end="")
    sys.stdout.flush()

def broadcastLoop(destination):
    # Сообщить пользователю, что все готово RNS.log(
        "Пример широковещательной рассылки "+
        RNS.prettyhexrep(destination.hash)+
        " запущен, введите текст и нажмите Enter для широковещания (Ctrl-C для
выхода)"
    )

    # Мы входим в цикл, который будет выполняться до тех пор, пока пользователь не выйдет.
    # Если пользователь нажмёт Enter, мы отправим информацию, # введённую
    пользователем в командной строке.
    while True:
        print("> ", end="") entered
        = input()

        if entered != "":
            data      = entered.encode("utf-8")
            пакет     = RNS.Packet(destination, data)
            packet.send()



############################################################  ####  Запуск
программы                                  ####################################
############################################################
```

```
# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем запускает # программу.
if ___имя_____ == "___main___":
     try:
          parser = argparse.ArgumentParser( description="Пример Reticulum,
               демонстрирующий
отправку и прием широковещательных сообщений"
          )

          parser.add_argument( "--
               config",
               action="store",
               default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          parser.add_argument( "--
               channel",
               action="store",
               default=None,
               help="имя канала широковещания", type=str
          )

          args = parser.parse_args() if

          args.config:
               configarg = args.config else:
               configarg = None

          if args.channel:
               channelarg = args.channel
          else:
               channelarg = None program_setup(configarg,

          channelarg)

     except KeyboardInterrupt:
```

```
        print("")
        sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Broadcast.p y.

Ėcho

Пример *Echo* демонстрирует обмен данными между двумя адресатами с использованием интерфейса Packet.

```
############################################################
# Этот пример RNS демонстрирует простую утилиту «клиент-сервер»     # # echo.
Клиент может отправить запрос echo на # # сервер, и сервер ответит, подтвердив
получение # # пакета.                                                    #
############################################################

import argparse
import sys import
RNS

# Давайте определим имя приложения. Мы будем использовать его для
всех # создаваемых нами адресатов. Поскольку этот пример echo
# является частью ряда примеров утилит, мы поместим
# все они находятся в пространстве имен приложения «example_utilities» APP_NAME =
«example_utilities»


############################################################         ####
Серверная          часть          ########################################
############################################################

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
сервера
def server(configpath):
    global reticulum

    # Сначала необходимо инициализировать Reticulum
```

```
    reticulum = RNS.Reticulum(configpath)

    # Случайным образом создаем новую идентичность для нашего эхо-сервера
    server_identity = RNS.Identity()

    # Мы создаем пункт назначения, к которому клиенты могут отправлять запросы. Мы
    хотим # иметь возможность проверять эхо-ответы, отправляемые нашим клиентам,
    поэтому
    # создаем «единственный» пункт назначения, который может принимать зашифрованные
    # сообщения. Таким образом, клиент может отправить запрос и быть
    # уверен, что никто, кроме этого пункта назначения, не смог # прочитать его.
    echo_destination = RNS.Destination(
        server_identity, RNS.Destination.IN,
        RNS.Destination.SINGLE, APP_NAME,
        "echo",
        "request"
    )

    # Мы настраиваем конечный узел на автоматическую проверку всех # пакетов,
    адресованных ему. Благодаря этому RNS будет
автоматически
    # генерировать доказательство для каждого входящего пакета и передавать его # обратно
    отправителю этого пакета.

echo_destination.set_proof_strategy(RNS.Destination.PROVE_ALL) # Указываем

    пункту назначения, какую функцию в нашей программе
    # запускать при получении пакета. Мы делаем это, чтобы
    # вывести сообщение в журнал, когда сервер получит запрос
    echo_destination.set_packet_callback(server_callback)

    # Всё готово!
    # Подождем запросов от клиентов или ввода пользователя
    announceLoop(echo_destination)


def announceLoop(destination):
    # Сообщим пользователю, что все готово RNS.log(
        "Echo server "+
        RNS.prettyhexrep(destination.hash)+
```

```
          " запущен, нажмите Enter, чтобы вручную отправить объявление
(Ctrl-C для выхода)"
     )

     # Мы входим в цикл, который будет выполняться до тех пор, пока пользователь не выйдет.
     # Если пользователь нажмёт Enter, мы объявим о нашем сервере # в сети,
     что позволит клиентам
     # узнать, как создавать сообщения, адресованные ему. while True:
          entered = input()
          destination.announce()
          RNS.log("Отправлено объявление с " + RNS.prettyhexrep(destination.hash))


def server_callback(message, packet): global
     reticulum

     # Сообщим пользователю, что мы получили запрос эхо, и # что собираемся
     отправить ответ запрашивающему.
     # Отправка доказательства обрабатывается автоматически, так как мы
     # настроили адресата на подтверждение всех входящих пакетов.

     reception_stats = ""
     if reticulum.is_connected_to_shared_instance: reception_rssi =
reticulum.get_packet_rssi(packet.packet_hash) reception_snr
          =
reticulum.get_packet_snr(packet.packet_hash)

          if reception_rssi != None:
               reception_stats += " [RSSI "+str(reception_rssi)+"
dBm]"

          if reception_snr != None:
               reception_stats += " [SNR "+str(reception_snr)+"
дБм]"

     else:
          if packet.rssi != None:
               reception_stats += " [RSSI "+str(packet.rssi)+"
дБм]"
```

```
        if packet.snr != None:
            reception_stats += " [SNR "+str(packet.snr)+" дБ]"

    RNS.log("Получен пакет от эхо-клиента, доказательство отправлено" + reception_stats)


############################################################   ####   Часть
клиента                              ########################################
############################################################

# Эта инициализация выполняется, когда пользователь # выбирает запуск в качестве
клиента
def client(destination_hexhash, configpath, timeout=None): global reticulum

    # Нам нужно двоичное представление хеша # адреса назначения, введенного в
    командной строке
    try:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH // 8) * 2 if
        len(destination_hexhash) != dest_len:
            raise ValueError(
                "Длина адреса назначения неверна, должна состоять из {hex}
шестнадцатеричных символов ({byte} байт).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash) except
    Exception as e:
        RNS.log("Введено недопустимое назначение. Проверьте ввод!")
        RNS.log(str(e)+"\n")
        sys.exit(0)

    # Сначала необходимо инициализировать Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Мы переопределяем уровень логгирования, чтобы выводить сообщение при
    # получении объявления
    if RNS.loglevel < RNS.LOG_INFO:
        RNS.loglevel = RNS.LOG_INFO

    # Сообщить пользователю, что клиент готов!
```

```
    RNS.log(
        "Клиент Echo готов, нажмите Enter, чтобы отправить запрос Echo на
"+
        destination_hexhash+ "
        (Ctrl-C для выхода)"
    )

    # Мы входим в цикл, который будет выполняться до тех пор, пока
    пользователь не выйдет из программы. # Если пользователь нажмёт
    клавишу Enter, мы попытаемся отправить
    # запрос echo на адрес, указанный в # командной строке.
      while True:
          input()

        # Сначала проверим, знает ли RNS путь к месту назначения.
        # Если знает, мы загрузим идентификатор сервера и создадим пакет
        if RNS.Transport.has_path(destination_hash):

            # Чтобы обратиться к серверу, нам нужно знать его
public
            # ключ, поэтому мы проверяем, знает ли Reticulum этот
.
            # Это делается путем вызова метода «recall»
модул
            # Identity. Если пункт назначения известен, он # возвращает экземпляр
я
will        Identity, который можно использовать в
            # исходящих адресах. server_identity =
RNS.Identity.recall(destination_hash)

            # Мы получили нужный экземпляр идентификатора из # метода
            recall, поэтому давайте создадим исходящий
            # назначения. Мы используем следующее соглашение об
            именовании: # example_utilities.echo.request
            # Это соответствует именованию, которое мы указали в # части
            кода, относящейся к серверу.
            request_destination = RNS.Destination(
                server_identity, RNS.Destination.OUT,
                RNS.Destination.SINGLE,
```

```
                    APP_NAME,
                    "echo", "request"
                )

                # Пункт назначения готов, поэтому давайте создадим
packet.
                # Мы устанавливаем адресатом только что созданный
                request_destination #, а единственные данные, которые мы добавляем
                # — это случайный хеш.
                echo_request = RNS.Packet(request_destination,
RNS.Identity.get_random_hash())

                # Отправляем пакет! Если пакет успешно # отправлен, он вернет
                экземпляр PacketReceipt. packet_receipt =
                echo_request.send()

                # Если пользователь указал время ожидания, мы устанавливаем #
                это время ожидания для получения пакета и настраиваем # функцию
                обратного вызова, которая будет вызвана, если # истечет время
                ожидания пакета.
                if timeout != None: packet_receipt.set_timeout(timeout)

packet_receipt.set_timeout_callback(packet_timed_out)

                # Затем мы можем установить обратный вызов доставки на
receipt.
                # Она будет вызвана автоматически, когда поступит подтверждение # о
for

                получении именно этого пакета от
. packet_receipt.set_delivery_callback(packet_delivered)

                # Сообщить пользователю, что запрос эха был отправлен
                RNS.log("Отправлен запрос эха на
"+RNS.prettyhexrep(request_destination.hash)) else:
                # Если мы не знаем этот адрес назначения, сообщите #
                пользователю, чтобы он дождался поступления объявления.
                RNS.log("Пункт назначения пока неизвестен. Запрашиваем
путь...")
```

```
                RNS.log("Нажмите Enter, чтобы вручную повторить попытку после
получения объявления.")
                RNS.Transport.request_path(destination_hash)

# Эта функция вызывается, когда адресат нашего ответа # получает пакет
подтверждения.
def packet_delivered(receipt): global
     reticulum

     if receipt.status == RNS.PacketReceipt.DELIVERED: rtt =
          receipt.get_rtt()
          if (rtt >= 1):
               rtt = round(rtt, 3)
               rttstring = str(rtt)+" секунд" else:
               rtt = round(rtt*1000, 3)
               rttstring = str(rtt) + " миллисекунд"

          reception_stats = ""
          if reticulum.is_connected_to_shared_instance: reception_rssi =
reticulum.get_packet_rssi(receipt.proof_packet.packet_hash) reception_snr =
reticulum.get_packet_snr(receipt.proof_packet.packet_hash)

               if reception_rssi != None: reception_stats
                    += " [RSSI
"+str(reception_rssi)+" дБм]"

               if reception_snr != None: reception_stats +=
                    " [SNR
"+str(reception_snr)+" дБ]"

          else:
               if receipt.proof_packet != None:
                    if receipt.proof_packet.rssi != None: reception_stats += "
                         [RSSI
"+str(receipt.proof_packet.rssi)+" дБм]"

                    if receipt.proof_packet.snr != None: reception_stats +=
                         " [SNR
"+str(receipt.proof_packet.snr)+" дБ]"
```

```
        RNS.log(
            "Получен действительный ответ от "+
            RNS.prettyhexrep(receipt.destination.hash)+ ", время
            прохождения в оба конца составляет "+rttstring+
            reception_stats
        )

# Эта функция вызывается, если произошел таймаут пакета. def
packet_timed_out(receipt):
    if receipt.status == RNS.PacketReceipt.FAILED: RNS.log("Пакет
        "+RNS.prettyhexrep(receipt.hash)+"
истекло")



############################################################  ####  Запуск
программы                                  ####################################
############################################################

# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем запускает # нужный режим
работы программы.
if ___name____ == "___main___":
    try:
        parser = argparse.ArgumentParser(description="Простая утилита-сервер и
клиент для эхо")

        parser.add_argument( "-
            s",
            "--server",
            action="store_true",
            help="ожидание входящих пакетов от клиентов"
        )

        parser.add_argument( "-
            t",
            "--timeout",
            action="store",
            metavar="s",
            default=None,
            help="установить таймаут ответа в секундах", type=float
        )
```

```
          parser.add_argument("--config",
               action="store", default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          parser.add_argument(
               "destination",
               nargs="?", default=None,
               help="шестнадцатеричный хеш адреса назначения сервера", type=str
          )

          args = parser.parse_args() if

          args.server:
               configarg = None
               if args.config:
                    configarg = args.config
               server(configarg)
          else:
               if args.config:
                    configarg = args.config else:
                    configarg = None

               if args.timeout:
                    timeoutarg = float(args.timeout) else:
                    timeoutarg = None

               if (args.destination == None):
                    print("") parser.print_help()
                    print("")
               else:
                    client(args.destination, configarg, timeout=timeoutarg)
     except KeyboardInterrupt:
```

```
        print("")
        sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Echo.py.

Link

Пример *Link* демонстрирует установку зашифрованного соединения с удаленным адресатом и передачу трафика по этому соединению в обоих направлениях.

```
##########################################################
# Этот пример RNS демонстрирует, как настроить соединение с # # пунктом
назначения и передавать по нему данные в обоих направлениях. #
##########################################################

import os import
sys import time
import argparse import
RNS

# Давайте определим имя приложения. Мы будем использовать его для
всех # создаваемых нами адресатов. Поскольку этот пример echo
# является частью ряда примеров утилит, мы поместим
# все они находятся в пространстве имен приложения «example_utilities» APP_NAME =
«example_utilities»

##########################################################          ####
Серверная          часть          ########################################
##########################################################

# Ссылка на последнюю клиентскую ссылку, которая подключилась latest_client_link = None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
сервера
def server(configpath):
     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)
```

```
    # Случайным образом создаем новую идентичность для нашего примера соединения
    server_identity = RNS.Identity()

    # Мы создаем пункт назначения, к которому могут подключаться клиенты. Мы #
    хотим, чтобы клиенты создавали ссылки на этот пункт назначения, поэтому нам #
    нужно создать тип пункта назначения «single». server_destination =
    RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "linkexample"
    )

    # Настраиваем функцию, которая будет вызываться каждый раз, # когда новый
    клиент устанавливает соединение с этим пунктом назначения.

server_destination.set_link_established_callback(client_connected)

    # Все готово!
    # Подождем запросов от клиентов или ввода пользователя
    server_loop(server_destination)

def server_loop(destination):
    # Сообщим пользователю, что все готово RNS.log(
        "Пример соединения "+
        RNS.prettyhexrep(destination.hash)+
        " запущен, ожидаем подключения."
    )

    RNS.log("Нажмите Enter, чтобы вручную отправить объявление (Ctrl-C для выхода)")

    # Мы входим в цикл, который будет работать, пока пользователь не выйдет.
    # Если пользователь нажмет Enter, мы объявим адрес нашего сервера # в
    сети, что позволит клиентам
    # узнать, как создавать сообщения, адресованные ему. while True:
        entered = input()
        destination.announce()
```

```
        RNS.log("Отправлено объявление с "+RNS.prettyhexrep(destination.hash))

# Когда клиент устанавливает соединение с нашим сервером, # эта
функция вызывается с # ссылкой на соединение.
def client_connected(link): global
    latest_client_link

    RNS.log("Клиент подключился")
    link.set_link_closed_callback(client_disconnected)
    link.set_packet_callback(server_packet_received) latest_client_link = link

def client_disconnected(link):
    RNS.log("Клиент отключился")

def server_packet_received(message, packet): global
    latest_client_link

    # Когда данные поступают по любому активному каналу, # все
    они направляются последнему подключившемуся клиенту
    #, который подключился.
    text = message.decode("utf-8") RNS.log("Получены данные
    по каналу: "+text)

    reply_text = "Я получил \""+text+"\" по каналу" reply_data =
    reply_text.encode("utf-8") RNS.Packet(latest_client_link,
    reply_data).send()


############################################################  ####  Часть
клиента                          ##########################################
############################################################

# Ссылка на серверное соединение
server_link = None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
клиента
def client(destination_hexhash, configpath):
    # Нам нужно двоичное представление хэша # назначения, введенного в
    командной строке
```

```
    try:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH // 8) * 2 if
        len(destination_hexhash) != dest_len:
            raise ValueError(
                "Длина адреса назначения неверна, должна состоять из {hex}
шестнадцатеричных символов ({byte} байт).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash) except:
        RNS.log("Введено недопустимое назначение. Проверьте ввод!\n")
        sys.exit(0)

    # Сначала необходимо инициализировать Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Проверить, знаем ли мы путь к месту назначения
    if not RNS.Transport.has_path(destination_hash):
        RNS.log("Пункт назначения пока неизвестен. Запрашиваем путь и ждем
поступления объявления...")
        RNS.Transport.request_path(destination_hash)
        while not RNS.Transport.has_path(destination_hash): time.sleep(0.1)

    # Получить идентификатор сервера
    server_identity = RNS.Identity.recall(destination_hash)

    # Сообщить пользователю, что мы начнем подключение RNS.log("Устанавливаем соединение
    с сервером...")

    # Когда идентификатор сервера известен, мы настраиваем #
    пункт назначения
    server_destination = RNS.Destination(
        server_identity, RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "linkexample"
    )

    # И создать ссылку
```

```
    link = RNS.Link(server_destination)

    # Мы устанавливаем обратный вызов, который будет
    выполняться # каждый раз при получении пакета по #
    соединению
    link.set_packet_callback(client_packet_received)

    # Мы также настроим функции для уведомления
    # пользователя, когда соединение установлено или закрыто
    link.set_link_established_callback(link_established)
    link.set_link_closed_callback(link_closed)

    # Все настроено, поэтому давайте войдем в цикл # для
    взаимодействия пользователя с примером client_loop()

def client_loop(): global
    server_link

    # Ждем, пока соединение не станет активным while not
    server_link:
        time.sleep(0.1)

    should_quit = False пока not
    should_quit:
        try:
            print("> ", end=" ")
            text = input()

            # Проверить, следует ли завершить пример
            if text == "quit" or text == "q" or text == "exit": should_quit =
                True
                server_link.teardown()

            # Если нет, отправить введенный текст по ссылке, если text !=
            "":
                data = text.encode("utf-8")
                if len(data) <= RNS.Link.MDU: RNS.Packet(server_link,
                    data).send()
                else:
                    RNS.log(
                        "Невозможно отправить этот пакет, размер данных

из "+
```

```
                                str(len(data))+" байт превышает MDU канала
MDU пакета "+
                                str(RNS.Link.MDU)+" байт",
                                RNS.LOG_ERROR
                            )

          except Exception as e:
               RNS.log("Ошибка при отправке данных по каналу:
"+str(e))
               should_quit = True server_link.teardown()

# Эта функция вызывается, когда
# установлено соединение с сервером def
link_established(link):
     # Мы сохраняем ссылку на экземпляр # соединения
     для дальнейшего использования
     global server_link
     server_link = link

     # Сообщаем пользователю, что # установлено
     соединение с сервером
     RNS.log("Соединение с сервером установлено, введите текст для отправки или
\"quit\" для выхода")

# Когда соединение будет закрыто, мы # уведомим
пользователя и завершим работу программы
def link_closed(link):
     if link.teardown_reason == RNS.Link.TIMEOUT: RNS.log("Истекло
          время ожидания соединения, завершаю работу")
     elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
          RNS.log("Соединение было закрыто сервером, завершаю работу
сейчас")
     else:
          RNS.log("Соединение закрыто, завершаю работу")

     time.sleep(1.5)
     sys.exit(0)

# Когда по каналу поступает пакет, мы # просто выводим данные.
def client_packet_received(message, packet): text =
     message.decode("utf-8")
```

```
     RNS.log("Получены данные по каналу: "+text) print("> ",
     end=" ")
     sys.stdout.flush()


############################################################  ####  Запуск
программы                                  ####################################
############################################################

# Эта часть программы запускается при старте,
# анализирует ввод пользователя, а затем # запускает нужный
режим работы программы.
if ___name____ == "___main___":
     try:
          parser = argparse.ArgumentParser(description="Простой пример ссылки")

          parser.add_argument( "-
               s",
               "--server",
               action="store_true",
               help="ожидание входящих запросов на подключение от клиентов"
          )

          parser.add_argument( "--
               config",
               action="store",
               default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          parser.add_argument(
               "destination",
               nargs="?", default=None,
               help="шестнадцатеричный хеш адреса назначения сервера", type=str
          )

          args = parser.parse_args()
```

```
        if args.config:
             configarg = args.config else:
             configarg = None

        if args.server:
             server(configarg)
        else:
             if (args.destination == None):
                  print("") parser.print_help()
                  print("")
             else:
                  client(args.destination, configarg)

    except KeyboardInterrupt:
         print("")
         sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Link.py.

Идентификация

Пример *Identify* демонстрирует определение инициатора соединения после его установки.

```
############################################################
# Этот пример RNS демонстрирует, как настроить соединение с # # пунктом назначения и идентифицировать инициатора для его партнера #
############################################################

import os import
sys import time
import argparse import
RNS

# Давайте определим название приложения. Мы будем использовать его для всех # создаваемых нами целей. Поскольку этот пример с командой
echo
```

```
# является частью набора примеров утилит, мы поместим
# их все в пространство имен приложения «example_utilities» APP_NAME =
"example_utilities"

############################################################   ####   Часть
сервера                              ########################################
############################################################

# Ссылка на последнюю ссылку клиента, которая подключилась latest_client_link = None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
сервера
def server(configpath):
     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)

     # Случайным образом создать новую идентичность для нашего соединения, например
     server_identity = RNS.Identity()

     # Создаем пункт назначения, к которому могут подключаться клиенты. Мы #
     хотим, чтобы клиенты создавали ссылки на этот пункт назначения, поэтому нам #
     нужно создать пункт назначения типа «single». server_destination =
     RNS.Destination(
          server_identity,
          RNS.Destination.IN,
          RNS.Destination.SINGLE,
          APP_NAME,
          "identifyexample"
     )

     # Мы настраиваем функцию, которая будет вызываться каждый раз, # когда новый
     клиент создает ссылку на этот пункт назначения.

server_destination.set_link_established_callback(client_connect ed)

     # Все готово!
     # Подождем запросов от клиентов или ввода пользователя
     server_loop(server_destination)

def server_loop(destination):
     # Сообщим пользователю, что все готово
```

```
    RNS.log(
        "Пример идентификатора ссылки "+ RNS.prettyhexrep(destination.hash)+
        " запущен, ожидаем подключения."
    )

    RNS.log("Нажмите Enter, чтобы вручную отправить объявление (Ctrl-C для выхода)")

    # Мы входим в цикл, который будет выполняться до тех пор, пока пользователь не выйдет.
    # Если пользователь нажмёт Enter, мы объявим о назначении нашего сервера
    # в сети, что позволит клиентам
    # узнать, как создавать сообщения, адресованные ему. while True:
        entered = input()
        destination.announce()
        RNS.log("Отправлено объявление с " + RNS.prettyhexrep(destination.hash))

# Когда клиент устанавливает соединение с # нашим сервером-
назначением, эта функция будет вызвана с # ссылкой на соединение.
def client_connected(link): global
    latest_client_link

    RNS.log("Клиент подключился")
    link.set_link_closed_callback(client_disconnected)
    link.set_packet_callback(server_packet_received)
    link.set_remote_identified_callback(remote_identified) latest_client_link =
    link

def client_disconnected(link):
    RNS.log("Клиент отключился")

def remote_identified(link, identity): RNS.log("Удаленный
    идентифицирован как: "+str(identity))

def server_packet_received(message, packet): global
    latest_client_link

    # Получить идентификатор источника для отображения
    remote_peer = "неидентифицированный узел"
    if packet.link.get_remote_identity() != None:
```

```
          remote_peer = str(packet.link.get_remote_identity())

     # Когда данные поступают по любому активному каналу, # все
     они направляются последнему клиенту
     #, который подключился.
     text = message.decode("utf-8")

     RNS.log("Получены данные от "+remote_peer+": "+text) reply_text = "Я

     получил \""+text+"\" по каналу от
"+remote_peer
     reply_data = reply_text.encode("utf-8") RNS.Packet(latest_client_link,
     reply_data).send()


############################################################   ####   Часть
клиента                              ##########################################
############################################################

# Ссылка на канал сервера server_link =
None

# Ссылка на идентификатор клиента client_identity = None

# Эта инициализация выполняется, когда пользователь решает # работать в качестве
клиента
def client(destination_hexhash, configpath): global
     client_identity
     # Нам нужно двоичное представление хеша # назначения, введенного в
     командной строке
     try:
          dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2 if
          len(destination_hexhash) != dest_len:
               raise ValueError(
                    "Длина адреса назначения неверна, должна состоять из {hex}
шестнадцатеричных символов ({byte} байт).".format(hex=dest_len, byte=dest_len//2)
               )

          destination_hash = bytes.fromhex(destination_hexhash) except:
          RNS.log("Введено недопустимое назначение. Проверьте ваш
```

```
ввод!\n")
            sys.exit(0)

      # Сначала необходимо инициализировать Reticulum
      reticulum = RNS.Reticulum(configpath)

      # Создать новую идентичность клиента
      client_identity = RNS.Identity()
      RNS.log(
            "Клиент создал новую идентичность "+
            str(client_identity)
      )

      # Проверим, известен ли нам путь к месту назначения
      if not RNS.Transport.has_path(destination_hash):
            RNS.log("Пункт назначения пока неизвестен. Запрашиваем путь и ждем
поступления объявления...")
            RNS.Transport.request_path(destination_hash)
            пока RNS.Transport.has_path(destination_hash) не равно true:
                  time.sleep(0.1)

      # Получить идентификатор сервера
      server_identity = RNS.Identity.recall(destination_hash)

      # Сообщить пользователю, что мы начнем подключение RNS.log("Устанавливаем соединение
      с сервером...")

      # Когда идентификатор сервера известен, мы настраиваем #
      пункт назначения
      server_destination = RNS.Destination(
            server_identity, RNS.Destination.OUT,
            RNS.Destination.SINGLE,
            APP_NAME,
            "identifyexample"
      )

      # И создаем ссылку
      link = RNS.Link(server_destination)

      # Мы устанавливаем обратный вызов, который будет
      выполняться # каждый раз при получении пакета по #
      соединению
```

```
    link.set_packet_callback(client_packet_received)

    # Мы также настроим функции для уведомления
    # Обработка событий при установке или разрыве соединения
    link.set_link_established_callback(link_established)
    link.set_link_closed_callback(link_closed)

    # Все настроено, давайте перейдем в цикл # для взаимодействия
    пользователя с примером client_loop()

def client_loop(): global
    server_link

    # Ждем, пока соединение не станет активным while not
    server_link:
        time.sleep(0.1)

    should_quit = False пока not
    should_quit:
        try:
            print("> ", end=" ")
            text = input()

            # Проверить, следует ли завершить работу с примером
            if text == "quit" or text == "q" or text == "exit": should_quit =
                True
                server_link.teardown()

            # Если нет, отправить введенный текст по каналу if text != "":
                data = text.encode("utf-8")
                if len(data) <= RNS.Link.MDU: RNS.Packet(server_link,
                    data).send()
                else:
                    RNS.log(
                        "Невозможно отправить этот пакет, размер данных

                        str(len(data))+" байт превышает MDU
из "+
MDU канала, равный "+
                        str(RNS.Link.MDU)+" байт",
                        RNS.LOG_ERROR
                    )
```

```
          except Exception as e:
               RNS.log("Ошибка при отправке данных по каналу:
"+str(e))
               should_quit = True server_link.teardown()

# Эта функция вызывается при разрыве соединения
# Установлено соединение с сервером def
link_established(link):
     # Сохраняем ссылку на экземпляр соединения для
     дальнейшего использования
     global server_link, client_identity server_link = link

     # Сообщить пользователю, что сервер # подключен
     RNS.log("Соединение с сервером установлено, идентификация удаленного узла...")

     link.identify(client_identity)

# Когда соединение будет закрыто, мы # уведомим
пользователя и завершим работу программы
def link_closed(link):
     if link.teardown_reason == RNS.Link.TIMEOUT: RNS.log("Истекло
          время ожидания соединения, завершаем работу")
     elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
          RNS.log("Соединение было закрыто сервером, завершаем работу
сейчас")
     else:
          RNS.log("Соединение закрыто, завершаю работу")

     time.sleep(1.5)
     sys.exit(0)

# При получении пакета по каналу связи мы # просто выводим
данные.
def client_packet_received(message, packet): text =
     message.decode("utf-8") RNS.log("Получены данные по
     каналу: "+text) print("> ", end=" ")
     sys.stdout.flush()
```

```
############################################################  ####  Запуск
программы                                  ####################################
############################################################

# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем # запускает нужный
режим работы программы.
if ___name____ == "___main___":
     try:
          parser = argparse.ArgumentParser(description="Простой пример ссылки")

          parser.add_argument( "-
               s",
               "--server",
               action="store_true",
               help="ожидание входящих запросов на подключение от клиентов"
          )

          parser.add_argument( "--
               config",
               action="store",
               default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          parser.add_argument(
               "destination",
               nargs="?", default=None,
               help="шестнадцатеричный хеш адреса назначения сервера", type=str
          )

          args = parser.parse_args() if

          args.config:
               configarg = args.config else:
```

```
            configarg = None

        if args.server:
            server(configarg)
        else:
            if (args.destination == None):
                print("") parser.print_help()
                print("")
            else:
                client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Identify.py.

Запросы и ответы

В примере *Request* рассматривается отправка запросов и получение ответов.

```
##########################################################
# Этот пример RNS демонстрирует, как отправлять запросы # # и получать ответы по каналу связи.                          #
##########################################################

import os import
sys import time
import random
import argparse import
RNS

# Давайте определим имя приложения. Мы будем использовать его для
всех # создаваемых нами адресатов. Поскольку этот пример с эхом
# является частью ряда примеров утилит, мы поместим
# их все в пространство имен приложения "example_utilities" APP_NAME =
"example_utilities"
```

```
##########################################################          ####
Серверная часть ##########################################
##########################################################

# Ссылка на последнюю ссылку клиента, которая подключилась latest_client_link = None

def random_text_generator(path, data, request_id, link_id, remote_identity,
requested_at):
     RNS.log("Генерируем ответ на запрос "+RNS.prettyhexrep(request_id)+" по каналу
"+RNS.prettyhexrep(link_id))
     texts = ["Они посмотрели вверх", "Каждое полнолуние", "Бекки была
расстроена", "Я буду держаться от этого подальше", "В зоомагазине есть все"]
     return texts[random.randint(0, len(texts)-1)]

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
сервера
def server(configpath):
     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)

     # Случайным образом создаем новую идентичность для нашего примера сервера
     server_identity = RNS.Identity()

     # Создаем пункт назначения, к которому могут подключаться клиенты. Мы #
     хотим, чтобы клиенты создавали ссылки на этот пункт назначения, поэтому нам #
     нужно создать тип пункта назначения «single». server_destination =
     RNS.Destination(
          server_identity,
          RNS.Destination.IN,
          RNS.Destination.SINGLE,
          APP_NAME,
          "requestexample"
     )

     # Настраиваем функцию, которая будет вызываться каждый раз, # когда новый
     клиент создает связь с этим пунктом назначения.

server_destination.set_link_established_callback(client_connect ed)
```

```
    # Мы регистрируем обработчик запросов для обработки входящих # запросов
    по любым установленным соединениям.
    server_destination.register_request_handler(
        "/random/text",
        response_generator = random_text_generator, allow =
        RNS.Destination.ALLOW_ALL
    )

    # Все готово!
    # Подождем запросов от клиентов или ввода пользователя
    server_loop(server_destination)

def server_loop(destination):
    # Сообщим пользователю, что все готово RNS.log(
        "Запрос example "+
        RNS.prettyhexrep(destination.hash)+
        " выполняется, ожидаем подключения."
    )

    RNS.log("Нажмите Enter, чтобы вручную отправить объявление (Ctrl-C для выхода)")

    # Мы входим в цикл, который будет выполняться до тех пор, пока пользователь не выйдет.
    # Если пользователь нажмет Enter, мы объявим адрес нашего сервера # в
    сети, что позволит клиентам
    # узнать, как создавать сообщения, адресованные ему. while True:
        entered = input()
        destination.announce()
        RNS.log("Отправлено объявление с "+RNS.prettyhexrep(destination.hash))

# Когда клиент устанавливает соединение с нашим сервером, # эта
функция вызывается с # ссылкой на данное соединение.
def client_connected(link): global
    latest_client_link

    RNS.log("Клиент подключился")
    link.set_link_closed_callback(client_disconnected) latest_client_link = link
```

```
def client_disconnected(link):
     RNS.log("Клиент отключился")


############################################################   ####   Часть
клиента                              ##########################################
############################################################

# Ссылка на сервер link server_link =
None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
клиента
def client(destination_hexhash, configpath):
     # Нам нужно двоичное представление хеша # назначения, введенного в
     командной строке
     try:
          dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2 if
          len(destination_hexhash) != dest_len:
               raise ValueError(
                    "Длина адреса назначения неверна, должна состоять из {hex}
шестнадцатеричных символов ({byte} байт).".format(hex=dest_len, byte=dest_len//2)
               )

          destination_hash = bytes.fromhex(destination_hexhash) except:
          RNS.log("Введен неверный адрес назначения. Проверьте введенные данные!\n")
          sys.exit(0)

     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)

     # Проверим, известен ли нам путь к месту назначения
     if not RNS.Transport.has_path(destination_hash):
          RNS.log("Пункт назначения пока неизвестен. Запрашиваем путь и ждем
поступления объявления...")
          RNS.Transport.request_path(destination_hash)
          while not RNS.Transport.has_path(destination_hash): time.sleep(0.1)
```

```
    # Получить идентификатор сервера
    server_identity = RNS.Identity.recall(destination_hash)

    # Сообщить пользователю, что мы начнем подключение RNS.log("Устанавливаем соединение
    с сервером...")

    # Когда идентификатор сервера известен, мы настраиваем #
    пункт назначения
    server_destination = RNS.Destination(
        server_identity, RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "requestexample"
    )

    # И создаем ссылку
    link = RNS.Link(server_destination)

    # Настроим функции для уведомления
    # пользователя об установке или закрытии соединения
    link.set_link_established_callback(link_established)
    link.set_link_closed_callback(link_closed)

    # Все настроено, поэтому давайте войдем в цикл # для
    взаимодействия пользователя с примером client_loop()

def client_loop(): global
    server_link

    # Ожидание, пока ссылка станет активной, пока не
    выполнена команда `server_link`:
        time.sleep(0.1)

    should_quit = False пока not
    should_quit:
        try:
            print("> ", end=" ")
            text = input()

            # Проверить, следует ли завершить работу с примером
            if text == "quit" or text == "q" or text == "exit":
```

```
                    should_quit = True
                    server_link.teardown()

               else:
                    server_link.request(
                         "/random/text", data
                         = None,
                         response_callback = got_response, failed_callback =
                         request_failed
                    )


          except Exception as e:
               RNS.log("Ошибка при отправке запроса по каналу:
"+str(e))
               should_quit = True server_link.teardown()

def got_response(request_receipt): request_id =
     request_receipt.request_id response =
     request_receipt.response

     RNS.log("Получен ответ на запрос "+RNS.prettyhexrep(request_id)+":
"+str(response))

def request_received(request_receipt):
     RNS.log("Запрос "+RNS.prettyhexrep(request_receipt.request_id)+" был
получен удаленным узлом.")

def request_failed(request_receipt):
     RNS.log("Запрос "+RNS.prettyhexrep(request_receipt.request_id)+" завершился
сбоем.")


# Эта функция вызывается, когда
# с сервером def link_established(link):
     # Мы сохраняем ссылку на экземпляр # соединения
     для дальнейшего использования
     global server_link
     server_link = link
```

```
    # Сообщаем пользователю, что # установлено
    соединение с сервером
    RNS.log("Соединение с сервером установлено. Нажмите Enter, чтобы отправить
запрос, или введите «quit», чтобы выйти")

# Когда соединение будет закрыто, мы сообщим об этом #
пользователю и завершим работу программы
def link_closed(link):
    if link.teardown_reason == RNS.Link.TIMEOUT: RNS.log("Истекло
        время ожидания соединения, завершаем работу")
    elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
        RNS.log("Соединение было закрыто сервером,
сейчас")
    else:
        RNS.log("Соединение закрыто, завершаю работу")

    time.sleep(1.5)
    sys.exit(0)


############################################################  ####  Запуск
программы                                  ######################################
############################################################

# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем # запускает нужный
режим работы программы.
if ___name____ == "___main___":
    try:
        parser = argparse.ArgumentParser(description="Простой пример
запроса/ответа")

        parser.add_argument( "-
            s",
            "--server",
            action="store_true",
            help="ожидание входящих запросов от клиентов"
        )

        parser.add_argument( "--
            config",
            action="store",
            default=None,
```

```
                help="путь к альтернативной конфигурации Reticulum type=str
directory",

          )

          parser.add_argument(
               "destination",
               nargs="?", default=None,
               help="шестнадцатеричный хеш адреса сервера", type=str
          )

          args = parser.parse_args() if

          args.config:
               configarg = args.config else:
               configarg = None

          if args.server:
               server(configarg)
          else:
               if (args.destination == None):
                    print("") parser.print_help()
                    print("")
               else:
                    client(args.destination, configarg)

     except KeyboardInterrupt:
          print("")
          sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Request.py.

Канал

В примере *«Канал»* рассматривается использование канала для передачи структурированных данных между узлами соединения.

```
##########################################################
# Этот пример RNS демонстрирует, как настроить связь с # # пунктом назначения и
передавать по ней структурированные сообщения # # с использованием канала. #
##########################################################

import os import
sys import time
import argparse
from datetime импортировать datetime

import RNS
from RNS.vendor import umsgpack

# Определим имя приложения. Мы будем использовать его для всех #
создаваемых нами адресатов. Поскольку этот пример с echo
# является частью ряда примеров утилит, мы поместим
# их все в пространство имен приложения «example_utilities» APP_NAME =
"example_utilities"

############################################################ #### Общие
объекты ######################################
############################################################

# Данные канала должны быть структурированы в подклассе
# MessageBase. Это гарантирует, что канал сможет
# сериализовать и десериализовать объект, а также мультиплексировать его # с
другими объектами. Оба конца соединения потребуют
# одинаковые определения объектов, чтобы иметь возможность обмениваться
данными # по каналу.
#
# Примечание: Объекты, которые мы хотим использовать через канал, должны #
быть зарегистрированы в канале, и каждое соединение имеет
# свой экземпляр канала. См. client_connected
# и link_established в этом примере, чтобы увидеть, # как регистрируются
типы сообщений.

# Давайте создадим простой класс сообщения под названием StringMessage
```

#, который будет передавать строку с временной меткой.

```
class StringMessage(RNS.MessageBase):
     # Переменной класса MSGTYPE необходимо присвоить # 2-байтовое целое
     значение. Этот идентификатор позволяет
     каналу найти конструктор вашего сообщения, когда # сообщение поступает по
     каналу.
     #
     # MSGTYPE должен быть уникальным для всех типов сообщений, которые мы
     # регистрируем в канале. Значения MSGTYPE >= 0xf000
     # зарезервированы для системы.
     MSGTYPE = 0x0101

     # Конструктор нашего объекта должен вызываться # без аргументов. Мы
     можем иметь параметры, но у них # должно быть значение по умолчанию.
     #
     # Это необходимо для того, чтобы канал мог создать пустую # версию
     нашего сообщения, в которую можно будет вписать входящее
     # сообщение может быть распаковано.
     def ___init___(self, data=None):
          self.data = data
          self.timestamp = datetime.now()

     # Наконец, наше сообщение должно реализовывать функции
     # которые канал может вызывать для упаковки и распаковки нашего сообщения
     # в/из полезной нагрузки необработанного пакета. Мы будем использовать
     # пакет umsgpack, входящий в состав RNS. Мы также могли бы использовать
     # пакет struct, входящий в состав Python, если бы хотели
     # больше контроля над структурой упакованных байтов. #
     # Также обратите внимание, что упакованные объекты сообщений
     должны # полностью помещаться в один пакет. Количество байтов
     #, доступных для полезных нагрузок сообщений, можно узнать у # канала с
     помощью свойства Channel.MDU.
     # Размер MDU канала немного меньше, чем размер MDU канала связи, # из-
     за кодирования заголовка сообщения.

     # Функция pack кодирует содержимое сообщения в # поток байтов.
     def pack(self) -> bytes:
          return umsgpack.packb((self.data, self.timestamp))
```

```
    # А функция unpack декодирует поток байтов в # содержимое сообщения.
    def unpack(self, raw):
        self.data, self.timestamp = umsgpack.unpackb(raw)


############################################################   ####   Часть
сервера                                  ########################################
############################################################

# Ссылка на последнюю клиентскую ссылку, которая подключилась latest_client_link = None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
сервера
def server(configpath):
    # Сначала необходимо инициализировать Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Случайным образом создать новую идентичность для нашего соединения, например
    server_identity = RNS.Identity()

    # Создаем пункт назначения, к которому могут подключаться клиенты. Мы #
    хотим, чтобы клиенты создавали ссылки на этот пункт назначения, поэтому # нам
    нужно создать пункт назначения типа «single». server_destination =
    RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "channelexample"
    )

    # Мы настраиваем функцию, которая будет вызываться каждый раз, # когда новый
    клиент создает связь с этим пунктом назначения.

server_destination.set_link_established_callback(client_connect ed)

    # Все готово!
    # Подождем запросов от клиентов или ввода пользователя
    server_loop(server_destination)
```

```
def server_loop(destination):
     # Сообщим пользователю, что все готово RNS.log(
          "Пример канала "+
          RNS.prettyhexrep(destination.hash)+
          " запущен, ожидаем подключения."
     )

     RNS.log("Нажмите Enter, чтобы вручную отправить объявление (Ctrl-C для выхода)")

     # Мы входим в цикл, который будет выполняться до тех пор, пока пользователь не выйдет.
     # Если пользователь нажмёт Enter, мы объявим о нашем сервере # в сети,
     что позволит клиентам
     # узнать, как создавать сообщения, адресованные ему. while True:
          entered = input()
          destination.announce()
          RNS.log("Отправлено объявление с " + RNS.prettyhexrep(destination.hash))

# Когда клиент устанавливает соединение с # нашим сервером-
назначением, эта функция будет вызвана с # ссылкой на соединение.
def client_connected(link): global
     latest_client_link
     latest_client_link = link

     RNS.log("Клиент подключился")
     link.set_link_closed_callback(client_disconnected)

     # Регистрация типов сообщений и добавление обратного вызова к каналу channel =
     link.get_channel() channel.register_message_type(StringMessage)
     channel.add_message_handler(server_message_received)

def client_disconnected(link):
     RNS.log("Клиент отключился")

def server_message_received(message): """
     Обработчик сообщений
     @param message: экземпляр подкласса MessageBase
```

```
    @return: True, если сообщение было обработано
    """
    global latest_client_link
    # При получении сообщения по любому активному каналу связи,
    # Все ответы будут направляться последнему подключившемуся клиенту.

    # В обработчике сообщений любое десериализуемое сообщение
    #, поступающее по каналу соединения, будет передано # всем обработчикам
    сообщений, если только предыдущий обработчик
не укажет, что
    # обработал сообщение. #
    #
    if isinstance(message, StringMessage):
        RNS.log("Получены данные по каналу: " + message.data + " (сообщение
создано в " + str(message.timestamp) + ")")

        reply_message = StringMessage("Я получил
\""+message.data+"\" по каналу")
        latest_client_link.get_channel().send(reply_message)

        # Входящие сообщения отправляются каждому
        # обработчику, добавленному к каналу, в том порядке, в котором #
        они были добавлены.
        # Если какой-либо обработчик сообщений возвращает True, сообщение #
        считается обработанным, и все последующие
        обработчики пропускаются. return
        True


########################################################   ####   Часть
клиента                          ##########################################
############################################################

# Ссылка на сервер: server_link = None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
клиента
def client(destination_hexhash, configpath):
    # Нам нужно двоичное представление хеша # назначения, введенного в
    командной строке
```

```
    try:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2 if
        len(destination_hexhash) != dest_len:
            raise ValueError(
                "Длина адреса назначения неверна, должна состоять из {hex}
шестнадцатеричных символов ({byte} байт).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash) except:
        RNS.log("Введено недопустимое назначение. Проверьте ввод!\n")
        sys.exit(0)

    # Сначала необходимо инициализировать Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Проверить, знаем ли мы путь к месту назначения
    if not RNS.Transport.has_path(destination_hash):
        RNS.log("Пункт назначения пока неизвестен. Запрашиваем путь и ждем
поступления объявления...")
        RNS.Transport.request_path(destination_hash)
        while not RNS.Transport.has_path(destination_hash): time.sleep(0.1)

    # Получить идентификатор сервера
    server_identity = RNS.Identity.recall(destination_hash)

    # Сообщить пользователю, что мы начинаем подключение RNS.log("Устанавливаем
    соединение с сервером...")

    # Когда идентификатор сервера известен, мы настраиваем #
    пункт назначения
    server_destination = RNS.Destination(
        server_identity, RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "channelexample"
    )

    # И создаем ссылку
```

```
    link = RNS.Link(server_destination)

    # Мы также настроим функции для уведомления
    # пользователя об установке или закрытии соединения
    link.set_link_established_callback(link_established)
    link.set_link_closed_callback(link_closed)

    # Все настроено, так что давайте войдем в цикл # для
    взаимодействия пользователя с примером client_loop()

def client_loop(): global
    server_link

    # Ждем, пока соединение не станет активным while not
    server_link:
        time.sleep(0.1)

    should_quit = False пока not
    should_quit:
        try:
            print("> ", end=" ")
            text = input()

            # Проверить, следует ли завершить работу с примером
            if text == "quit" or text == "q" or text == "exit": should_quit =
                True
                server_link.teardown()

            # Если нет, отправить введенный текст по каналу, если text !=
            "":
                message = StringMessage(text) packed_size =
                len(message.pack()) channel =
                server_link.get_channel() if
                channel.is_ready_to_send():
                    if packed_size <= channel.mdu: channel.send(message)
                    else:
                        RNS.log(
                            "Невозможно отправить этот пакет, размер данных

                            str(packed_size)+" байт превышает

размер "+
MDU пакета канала "+
```

```
                                    str(channel.MDU)+" байт", RNS.LOG_ERROR
                                )

                    else:
                        RNS.log("Канал не готов к отправке,
пожалуйста, подождите, пока " +
                                    "завершения обработки ожидающих сообщений.",
RNS.LOG_ERROR)

          except Exception as e:
               RNS.log("Ошибка при отправке данных по каналу:
"+str(e))
               should_quit = True server_link.teardown()

# Эта функция вызывается, когда
# с сервером def link_established(link):
     # Мы сохраняем ссылку на экземпляр # соединения
     для дальнейшего использования
     global server_link
     server_link = link

     # Регистрируем сообщения и добавляем обработчик к каналу channel =
     link.get_channel() channel.register_message_type(StringMessage)
     channel.add_message_handler(client_message_received)

     # Сообщаем пользователю, что сервер # подключен
     RNS.log("Соединение с сервером установлено, введите текст для отправки или
\"quit\", чтобы выйти")

# При закрытии ссылки мы уведомим # пользователя и
завершим работу программы
def link_closed(link):
     if link.teardown_reason == RNS.Link.TIMEOUT: RNS.log("Истекло
          время ожидания соединения, завершаем работу")
     elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
          RNS.log("Соединение было закрыто сервером, завершаем работу
сейчас")
     else:
          RNS.log("Соединение закрыто, завершаю работу")
```

```
     time.sleep(1.5)
     sys.exit(0)

# Когда по каналу поступает пакет, мы # просто выводим данные.
def client_message_received(message):
     if isinstance(message, StringMessage):
          RNS.log("Получены данные по каналу: " + message.data + " (сообщение
создано в " + str(message.timestamp) + ")")
          print("> ", end=" ")
          sys.stdout.flush()


############################################################  ####  Запуск
программы                                  ######################################
############################################################

# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем # запускает нужный
режим работы программы.
if ___name____ == "___main___":
     try:
          parser = argparse.ArgumentParser(description="Простой пример канала")

          parser.add_argument( "-
               s",
               "--server",
               action="store_true",
               help="ожидание входящих запросов на подключение от клиентов"
          )

          parser.add_argument( "--
               config",
               action="store",
               default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          parser.add_argument(
```

```
            "destination",
            nargs="?", default=None,
            help="шестнадцатеричный хеш адреса сервера", type=str
        )

        args = parser.parse_args() if

        args.config:
            configarg = args.config else:
            configarg = None

        if args.server:
            server(configarg)
        else:
            if (args.destination == None):
                print("") parser.print_help()
                print("")
            else:
                client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Channel.py.

Буфер

Пример *Buffer* демонстрирует использование буферизованных считывателей и записывателей для передачи двоичных данных между узлами `Link`.

```
##########################################################
# В этом примере RNS показано, как установить соединение с # # конечным пунктом
и передавать по нему двоичные данные с помощью буфера канала      # #.        #
```

```
##########################################################
из _____future____ import annotations
import os
import sys import
time import
argparse
из datetime импортировать datetime

import RNS
from RNS.vendor import umsgpack

# Определим имя приложения. Мы будем использовать его для всех #
создаваемых нами пунктов назначения. Поскольку этот пример с echo
# является частью ряда примеров утилит, мы поместим
# их все в пространство имен приложения «example_utilities» APP_NAME =
"example_utilities"



############################################################  ####  Часть
сервера                          ########################################
############################################################

# Ссылка на последнюю ссылку клиента, которая подключилась latest_client_link = None

# Ссылка на объект «последний буфер» latest_buffer = None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
сервера
def server(configpath):
     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)

     # Случайным образом создаем новую идентичность для нашего примера
     server_identity = RNS.Identity()

     # Мы создаем пункт назначения, к которому могут подключаться клиенты. Мы #
     хотим, чтобы клиенты создавали ссылки на этот пункт назначения, поэтому нам #
     нужно создать тип пункта назначения «single». server_destination =
     RNS.Destination(
          server_identity,
          RNS.Destination.IN,
```

```
            RNS.Destination.SINGLE,
            APP_NAME,
            "bufferexample"
        )

        # Мы настраиваем функцию, которая будет вызываться каждый раз, # когда новый
        клиент создает ссылку на этот пункт назначения.

server_destination.set_link_established_callback(client_connect ed)

        # Все готово!
        # Будем ждать запросов от клиентов или ввода пользователя
        server_loop(server_destination)

def server_loop(destination):
        # Сообщим пользователю, что все готово RNS.log(
            "Пример буфера связи "+ RNS.prettyhexrep(destination.hash)+
            " запущен, ожидаем соединения."
        )

        RNS.log("Нажмите Enter, чтобы вручную отправить объявление (Ctrl-C для выхода)")

        # Мы входим в цикл, который будет работать, пока пользователь не выйдет.
        # Если пользователь нажмет Enter, мы объявим адрес нашего сервера # в
        сети, что позволит клиентам
        # узнать, как создавать сообщения, адресованные ему. while True:
            entered = input()
            destination.announce()
            RNS.log("Отправлено объявление с "+RNS.prettyhexrep(destination.hash))

# Когда клиент устанавливает соединение с нашим сервером, # эта
функция вызывается с # ссылкой на соединение.
def client_connected(link):
        global     latest_client_link,     latest_buffer
        latest_client_link = link
```

```
    RNS.log("Клиент подключен") link.set_link_closed_callback(client_disconnected)

    # Если получено новое соединение, старый читатель # должен быть
    отключен.
    if latest_buffer:
        latest_buffer.close()



    # Создать объекты буфера.
    #     Параметр stream_id в этих функциях #       немного похож на
    файловый дескриптор, за исключением того, что он
    #     является уникальным для *приемника*. #
    #     В этом примере и читатель, и записывающий #       используют
    stream_id = 0, но на самом деле существует два
    #     отдельных однонаправленных потока, протекающих в #
          противоположных направлениях.
    #
    channel = link.get_channel()
    latest_buffer = RNS.Buffer.create_bidirectional_buffer(0, 0, channel,
server_buffer_ready)

def client_disconnected(link):
    RNS.log("Клиент отключился")

def server_buffer_ready(ready_bytes: int): """
    Обратный вызов из буфера, когда в буфере доступны данные

    :param ready_bytes: Количество байтов, готовых к чтению """
    global latest_buffer

    data = latest_buffer.read(ready_bytes) data =
    data.decode("utf-8")

    RNS.log("Получены данные через буфер: " + data)

    reply_message = "Я получил \""+data+"\" через буфер" reply_message =
    reply_message.encode("utf-8") latest_buffer.write(reply_message)
    latest_buffer.flush()
```

```
############################################################          ####
Клиентская часть ########################################
############################################################

# Ссылка на серверное соединение
server_link = None

# Ссылка на объект буфера, необходимая для передачи # объекта из обратного
вызова, связанного со ссылкой, в цикл # клиента.
buffer = None

# Эта инициализация выполняется, когда пользователь # выбирает запуск в качестве
клиента
def client(destination_hexhash, configpath):
     # Нам нужно двоичное представление хеша # назначения, введенного в
     командной строке
     try:
          dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2 if
          len(destination_hexhash) != dest_len:
               raise ValueError(
                    "Длина адреса назначения неверна, должна состоять из {hex}
шестнадцатеричных символов ({byte} байт).".format(hex=dest_len, byte=dest_len//2)
               )

          destination_hash = bytes.fromhex(destination_hexhash) except:
          RNS.log("Введено недопустимое назначение. Проверьте ввод!\n")
          sys.exit(0)

     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)

     # Проверить, знаем ли мы путь к месту назначения
     if not RNS.Transport.has_path(destination_hash):
          RNS.log("Пункт назначения пока неизвестен. Запрашиваем путь и ждем
поступления объявления...")
```

```
        RNS.Transport.request_path(destination_hash)
        while not RNS.Transport.has_path(destination_hash): time.sleep(0.1)

    # Получить идентификатор сервера
    server_identity = RNS.Identity.recall(destination_hash)

    # Сообщить пользователю, что мы начнем подключение RNS.log("Устанавливаем соединение
    с сервером...")

    # Если идентификатор сервера известен, мы настраиваем #
    пункт назначения
    server_destination = RNS.Destination(
        server_identity, RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "bufferexample"
    )

    # И создаем ссылку
    link = RNS.Link(server_destination)

    # Мы также настроим функции для уведомления
    # пользователя, когда соединение установлено или закрыто
    link.set_link_established_callback(link_established)
    link.set_link_closed_callback(link_closed)

    # Все настроено, так что давайте войдем в цикл # для
    взаимодействия пользователя с примером client_loop()

def client_loop(): global
    server_link

    # Ждем, пока соединение не станет активным while not
    server_link:
        time.sleep(0.1)

    should_quit = False пока not
    should_quit:
        try:
            print("> ", end=" ")
```

```
                text = input()

                # Проверить, следует ли завершить пример
                if text == "quit" or text == "q" or text == "exit":
                     should_quit = True
                     server_link.teardown() else:
                     # В противном случае закодируем текст и запишем его в

                     text = text.encode("utf-8") buffer.write(text)
буфер.               # Очистить буфер, чтобы принудительно сбросить данные:

                     buffer.flush()
```

отправлен.

```
          except Exception as e:
               RNS.log("Ошибка при отправке данных через буфер канала: "+str(e))
               should_quit = True server_link.teardown()

# Эта функция вызывается, когда
# с сервером def link_established(link):
     # Мы сохраняем ссылку на экземпляр # соединения
     для дальнейшего использования
     global server_link, buffer
     server_link = link

     # Создать буфер; см. server_client_connected() для # получения
     более подробной информации о настройке буфера.
     channel = link.get_channel()
     buffer = RNS.Buffer.create_bidirectional_buffer(0, 0, channel,
client_buffer_ready)

     # Сообщаем пользователю, что сервер # подключен
     RNS.log("Соединение с сервером установлено, введите текст для отправки или
\"quit\" для выхода")

# При закрытии ссылки мы уведомим # пользователя и
завершим работу программы
```

```
def link_closed(link):
     if link.teardown_reason == RNS.Link.TIMEOUT: RNS.log("Истекло
          время ожидания соединения, завершаем работу")
     elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
          RNS.log("Соединение было закрыто сервером, завершаем работу
сейчас")
     else:
          RNS.log("Соединение закрыто, завершаю работу")

     time.sleep(1.5)
     sys.exit(0)

# Когда в буфере появляются новые данные, прочитать их и вывести в терминал.
def client_buffer_ready(ready_bytes: int): global buffer
     data = buffer.read(ready_bytes) RNS.log("Получены данные из
     буфера соединения: " +
data.decode("utf-8"))
     print("> ", end=" ")
     sys.stdout.flush()


############################################################  ####  Запуск
программы                                  ######################################
############################################################

# Эта часть программы запускается при старте,
# анализирует ввод пользователя, а затем # запускает нужный
режим работы программы.
если __имя_____ == "___main___":
     try:
          parser = argparse.ArgumentParser(description="Простой пример буфера")

          parser.add_argument( "-
               s",
               "--server",
               action="store_true",
               help="ожидание входящих запросов на подключение от клиентов"
          )

          parser.add_argument(
```

```
                "--config", action="store",
                default=None,
                help="путь к альтернативной конфигурации Reticulum type=str


directory",

            )

            parser.add_argument(
                "destination",
                nargs="?", default=None,
                help="шестнадцатеричный хеш адреса назначения сервера", type=str
            )

            args = parser.parse_args() if

            args.config:
                configarg = args.config else:
                configarg = None

            if args.server:
                server(configarg)
            else:
                if (args.destination == None):
                    print("") parser.print_help()
                    print("")
                else:
                    client(args.destination, configarg)

        except KeyboardInterrupt:
            print("")
            sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Buffer.py.

Filetransfer

Пример *Filetransfer* реализует базовую программу файлового сервера, позволяющую клиентам подключаться и загружать файлы. Программа использует интерфейс Resource для эффективной передачи файлов любого размера по каналу Reticulum.

```
############################################################
# В этом примере RNS показаны простые серверная и клиентская программы для
передачи файлов # #. Сервер предоставляет доступ к каталогу с файлами
# #, а клиенты могут просматривать список файлов и # # загружать их с
сервера. #
# #
# Обратите внимание, что не рекомендуется использовать RNS-ресурсы для передачи
больших файлов # #, поскольку сжатие, # # шифрование и
упорядочивание в хеш-таблице могут занимать много времени # # на системах с
медленными процессорами, что, вероятно, приведет # # к истечению времени
ожидания на стороне клиента до того, как отправитель ресурса # # сможет
завершить подготовку ресурса. #
# #
# Если вам нужно передавать большие файлы, используйте вместо этого класс
Bundle # #, который автоматически разбивает данные # # на фрагменты,
подходящие для упаковки в виде ресурса. #
############################################################

import os import
sys import time
import threading import
argparse import RNS
import RNS.vendor.umsgpack as umsgpack

# Определим имя приложения. Мы будем использовать его для всех #
создаваемых нами пунктов назначения. Поскольку этот пример с echo
# является частью ряда примеров утилит, мы поместим
# их все в пространство имен приложения "example_utilities" APP_NAME =
"example_utilities"

# Мы также определим таймаут по умолчанию в секундах APP_TIMEOUT
= 45.0

############################################################ #### Часть
сервера ########################################
############################################################
```

```
serve_path = None

# Эта инициализация выполняется, когда пользователь выбирает # запуск в режиме
сервера
def server(configpath, path):
    # Сначала необходимо инициализировать Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Случайным образом создать новую идентичность для нашего файлового сервера
    server_identity = RNS.Identity()

    global serve_path
    serve_path = path

    # Мы создаем пункт назначения, к которому могут подключаться клиенты. Мы #
    хотим, чтобы клиенты создавали ссылки на этот пункт назначения, поэтому нам #
    нужно создать тип пункта назначения «single». server_destination =
    RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "filetransfer", "server"
    )

    # Настраиваем функцию, которая будет вызываться каждый раз, # когда новый
    клиент создает ссылку на этот пункт назначения.

server_destination.set_link_established_callback(client_connected)

    # Все готово!
    # Подождем запросов от клиентов или ввода пользователя
    announceLoop(server_destination)

def announceLoop(destination):
    # Сообщить пользователю, что все готово
    RNS.log("Файловый сервер "+RNS.prettyhexrep(destination.hash)+" запущен")
    RNS.log("Нажмите Enter, чтобы вручную отправить объявление (Ctrl-C для выхода)")
```

```
    # Мы входим в цикл, который будет работать, пока пользователь не выйдет.
    # Если пользователь нажмет Enter, мы объявим наш сервер # destination
    в сети, что позволит клиентам
    # узнать, как создавать сообщения, адресованные ему. while True:
        entered = input()
        destination.announce()
        RNS.log("Отправлено объявление с "+RNS.prettyhexrep(destination.hash))

# Вот удобная функция для вывода списка всех файлов # в нашем
обслуживаемом каталоге
def list_files():
    # Мы добавляем все записи из каталога, которые # являются реальными
    файлами и не начинаются с "." global serve_path
    return [file for file in os.listdir(serve_path) if
os.path.isfile(os.path.join(serve_path, file)) and file[:1] != "."]

# Когда клиент устанавливает соединение с нашим сервером, # эта
функция вызывается с
# ссылкой на это соединение. Затем мы отправляем клиенту # список
файлов, размещенных на сервере.
def client_connected(link):
    # Проверить, существует ли каталог, обслуживаемый сервером if
    os.path.isdir(serve_path):
        RNS.log("Клиент подключен, отправляем список файлов...")

        link.set_link_closed_callback(client_disconnected)

        # Мы упаковываем список файлов для отправки в пакет data =
        umsgpack.packb(list_files())

        # Проверяем размер упакованных данных if
        len(data) <= RNS.Link.MDU:
            # Если он помещается в один пакет, мы просто
            # отправим его одним пакетом по каналу. list_packet =
            RNS.Packet(link, data) list_receipt = list_packet.send()
            list_receipt.set_timeout(APP_TIMEOUT)
            list_receipt.set_delivery_callback(list_delivered)
```

```
                list_receipt.set_timeout_callback(list_timeout) else:
                RNS.log("Слишком много файлов в каталоге served!", RNS.LOG_ERROR)
                RNS.log("Вам следует реализовать функцию для разбиения списка файлов
на несколько пакетов.", RNS.LOG_ERROR)
                RNS.log("Подсказка: клиент уже поддерживает эту функцию :)",
RNS.LOG_ERROR)

            # После этого мы просто оставим соединение # открытым, пока
            клиент не запросит файл. Мы
            # настроим функцию, которая вызывается, когда
            # клиент отправляет пакет с запросом на файл.
            link.set_packet_callback(client_request)
        else:
            RNS.log("Клиент подключен, но обслуживаемый путь больше не существует!",
RNS.LOG_ERROR)
            link.teardown()

def client_disconnected(link):
        RNS.log("Клиент отключился")

def client_request(message, packet): global
        serve_path

        try:
            filename = message.decode("utf-8") except
        Exception as e:
            filename = None

        if filename in list_files(): try:
                # Если у нас есть запрошенный файл, мы # прочитаем
                его и упакуем в виде ресурса
                RNS.log("Клиент запросил \""+filename+"\"") file =
                open(os.path.join(serve_path, filename),
"rb")

                file_resource = RNS.Resource( file,
                    packet.link, callback=resource_sending_concluded
                )
```

```
                file_resource.filename = filename except
            Exception as e:
                # Если что-то пошло не так, мы закрываем #
                соединение
                RNS.log("Ошибка при чтении файла
\""+filename+"\"", RNS.LOG_ERROR)
                packet.link.teardown() raise e
        else:
            # Если у нас его нет, мы закрываем соединение RNS.log("Клиент запросил
            неизвестный файл") packet.link.teardown()

# Эта функция вызывается на сервере по завершении # передачи
ресурса.
def resource_sending_concluded(resource): if
        hasattr(resource, "filename"):
            name = resource.filename
        else:
            name = "resource"

        if resource.status == RNS.Resource.COMPLETE: RNS.log("Отправка
            \""+name+"\" клиенту завершена")
        elif resource.status == RNS.Resource.FAILED: RNS.log("Ошибка отправки
            \""+name+"\" клиенту")

def list_delivered(receipt):
        RNS.log("Список файлов получен клиентом")

def list_timeout(receipt):
        RNS.log("Истекло время ожидания отправки списка клиенту, закрытие этого
соединения")
        link = receipt.destination
        link.teardown()

############################################################   ####   Часть
клиента                              ########################################
############################################################

# Мы храним глобальный список файлов, доступных на сервере server_files =
[]
```

```
# Ссылка на сервер link server_link   =
None

# И ссылка на текущую загрузку current_download     =
None
current_filename      = None

# Переменные для хранения статистики загрузки
download_started      = 0
download_finished     = 0
время_загрузки        = 0
размер_передачи       = 0
размер_файла          = 0


# Эта инициализация выполняется, когда пользователь выбирает # запуск в качестве
клиента
def client(destination_hexhash, configpath):
     # Нам нужно двоичное представление хеша # назначения, введенного в
     командной строке
     try:
          dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2 if
          len(destination_hexhash) != dest_len:
               raise ValueError(
                    "Длина адреса назначения неверна, должна состоять из {hex}
шестнадцатеричных символов ({byte} байт)".format(hex=dest_len, byte=dest_len//2)
               )

          destination_hash = bytes.fromhex(destination_hexhash) except:
          RNS.log("Введено недопустимое назначение. Проверьте ввод!\n")
          sys.exit(0)

     # Сначала необходимо инициализировать Reticulum
     reticulum = RNS.Reticulum(configpath)



     # Проверить, знаем ли мы путь к месту назначения
     if not RNS.Transport.has_path(destination_hash):
          RNS.log("Пункт назначения пока неизвестен. Запрашиваем путь и ждем
поступления объявления...")
```

```
        RNS.Transport.request_path(destination_hash)
        while not RNS.Transport.has_path(destination_hash): time.sleep(0.1)

    # Получить идентификатор сервера
    server_identity = RNS.Identity.recall(destination_hash)

    # Сообщить пользователю, что мы начнем подключение RNS.log("Устанавливаем соединение
    с сервером...")

    # Когда идентификатор сервера известен, мы настраиваем #
    пункт назначения
    server_destination = RNS.Destination(
        server_identity, RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "filetransfer", "server"
    )

    # Мы также хотим автоматически проверять входящие пакеты

server_destination.set_proof_strategy(RNS.Destination.PROVE_ALL
)

    # И создать ссылку
    link = RNS.Link(server_destination)

    # Мы ожидаем, что любые обычные пакеты данных на канале #
    будут содержать список обслуживаемых файлов, поэтому мы
    устанавливаем
    # соответствующий обратный вызов: link.set_packet_callback(filelist_received)

    # Мы также настроим функции для уведомления
    # пользователя, когда соединение установлено или закрыто
    link.set_link_established_callback(link_established)
    link.set_link_closed_callback(link_closed)

    # И настроим канал на автоматический запуск # загрузки
    объявленных ресурсов
    link.set_resource_strategy(RNS.Link.ACCEPT_ALL)
    link.set_resource_started_callback(download_began)
```

```
     link.set_resource_concluded_callback(download_concluded) menu()

# Запрашивает указанный файл с сервера def
download(filename):
     global server_link, menu_mode, current_filename, transfer_size,
download_started
     current_filename = filename
     download_started = 0
     размер_передачи         = 0

     # Мы просто создаем пакет, содержащий
     # запрашиваемое имя файла, и отправляем его по # каналу.
     Мы также указываем, что нам не нужна
     # подтверждения получения пакета.
     request_packet = RNS.Packet(server_link,
filename.encode("utf-8"), create_receipt=False)
     request_packet.send()

     print("")
     print(("Запрошено \""+filename+"\" с сервера, ожидание начала загрузки..."))
     menu_mode = "download_started"

# Эта функция запускает простое меню, в котором пользователь #
может выбрать файлы для загрузки или выйти menu_mode = None
def menu():
     global server_files, server_link #
     Ждем, пока не появится список файлов while
     len(server_files) == 0:
          time.sleep(0.1)
     RNS.log("Готово!") time.sleep(0.5)

     global menu_mode
     menu_mode = "main"
     should_quit = False
     пока (не should_quit):
          print_menu()

          пока menu_mode не равно "main":
```

```
                # Ожидание
                time.sleep(0.25)

          user_input = input()
          if user_input == "q" or user_input == "quit" or user_input ==
"exit":
                should_quit = True
                print("")
          else:
                if user_input in server_files:
                     download(user_input)
                else:
                     try:
                          if 0 <= int(user_input) <
len(server_files):
                               download(server_files[int(user_input)])
                      except:
                          pass

     if should_quit:
          server_link.teardown()

# Выводит меню или экраны для
# различных состояний клиентской программы. # Это
просто и довольно скучно.
# Я не буду здесь вдаваться в подробности. В
основном # просто строки.
def print_menu():
     global menu_mode, download_time, download_started, download_finished,
transfer_size, file_size

     if menu_mode == "main":
          clear_screen()
          print_filelist() print("")
          print("Выберите файл для загрузки, введя его имя или номер, или нажмите q,
чтобы выйти")
          print(("> "), end=' ')
     elif menu_mode == "download_started": download_began =
          time.time()
          while menu_mode == "download_started":
                time.sleep(0.1)
                if time.time() > download_began + APP_TIMEOUT:
```

```
                    print("Время ожидания загрузки истекло")
                    time.sleep(1) server_link.teardown()

    if menu_mode == "downloading":
        print("Загрузка началась") print("")
        пока menu_mode == "загрузка": глобальная переменная
            current_download
            percent = round(current_download.get_progress() *
100.0, 1)
            print(("\rПрогресс: "+str(percent)+" %            "), end='
')

            sys.stdout.flush()
            time.sleep(0.1)

    if menu_mode == "save_error": print(("\rПрогресс: 100.0
        %"), end=' ') sys.stdout.flush()
        print("")
        print("Не удалось записать загруженный файл на диск")
        current_download.status = RNS.Resource.FAILED menu_mode =
        "download_concluded"

    if menu_mode == "download_concluded":
        if current_download.status == RNS.Resource.COMPLETE:
            print(("\rПрогресс: 100.0 %"), end=' ') sys.stdout.flush()

            # Вывести статистику
            часы, остаток = divmod(download_time, 3600) минуты,
            секунды = divmod(остаток, 60) timestring =
            "{:0>2}:{:0>2}:
{:05.2f}".format(int(hours),int(minutes),seconds) print("")
            print("")
            print("--- Статистика----------")
            print("\tВремя, затраченное       : "+timestring)
            print("\tРазмер файла             : "+size_str(file_size))
            print("\tПереданные данные :
"+size_str(transfer_size))
            print("\tФактическая скорость     :
"+size_str(file_size/download_time, suffix='b')+"/c")
```

```
                print("\tСкорость передачи         :
"+size_str(transfer_size/download_time, suffix='b')+"/c")
                print("")
                print("Загрузка завершена! Нажмите Enter, чтобы вернуться в
меню.")
                print("")
                input()

           else:
                print("")
                print("Загрузка не удалась! Нажмите Enter, чтобы вернуться в
меню.")
                input()

           current_download = None
           menu_mode = "main"
           print_menu()

# Эта функция выводит список файлов # на подключенном
сервере.
def print_filelist(): global
     server_files
     print("Файлы на сервере:")
     for index, file in enumerate(server_files): print("\t(" + str(index) +
          ")\t" + file)

def filelist_received(filelist_data, packet): global
     server_files, menu_mode
     try:
          # Распаковать список и расширить наш #
          локальный список доступных файлов
          filelist = umsgpack.unpackb(filelist_data) for file
          in filelist:
               if not file in server_files: server_files.append(file)

          # Если меню уже отображается, # мы обновим
          его с учетом того, что
          # только что получено
          if menu_mode == "main":
               print_menu()
     except:
          RNS.log("Получены неверные данные списка файлов, закрытие
```

```
ссылку")
          packet.link.teardown()

# Эта функция вызывается, когда
# с сервером def link_established(link):
     # Мы сохраняем ссылку на экземпляр # соединения
     для дальнейшего использования
     global server_link
     server_link = link

     # Сообщить пользователю, что # установлено
     соединение с сервером
     RNS.log("Соединение с сервером установлено")
     RNS.log("Ожидание списка файлов...")

     # И настроим небольшую задачу для проверки
     # возможного таймаута при получении # списка файлов
     thread = threading.Thread(target=filelist_timeout_job, daemon=True)
     thread.start()

# Этот процесс просто находится в режиме ожидания в
течение указанного # времени, а затем проверяет, был ли
получен список файлов. # Если нет, программа #
завершит работу.
def       filelist_timeout_job():
     time.sleep(APP_TIMEOUT)

     global server_files
     if len(server_files) == 0:
          RNS.log("Истекло время ожидания списка файлов, завершаю работу") sys.exit(0)


# Когда соединение закрывается, мы сообщим об этом #
пользователю и завершим работу программы
def link_closed(link):
     if link.teardown_reason == RNS.Link.TIMEOUT: RNS.log("Истекло
          время ожидания соединения, завершаем работу")
     elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
          RNS.log("Соединение было закрыто сервером, завершаем работу
```

```
сейчас")
     else:
          RNS.log("Соединение закрыто, завершаю работу")

     time.sleep(1.5)
     sys.exit(0)

# Когда RNS обнаружит, что загрузка # началась, мы
обновим состояние нашего меню
#, чтобы пользователю можно было показать ход #
загрузки.
def download_began(resource):
     global menu_mode, current_download, download_started, transfer_size,
file_size
     current_download = resource

     if download_started == 0: download_started =
          time.time()

     размер_передачи += resource.size
     размер_файла = resource.total_size

     menu_mode = "загрузка"

# Когда загрузка завершится, успешно # или нет, мы обновим
состояние нашего меню и
# сообщить пользователю о результатах. def
download_concluded(resource):
     global menu_mode, current_filename, download_started, download_finished,
download_time
     download_finished = time.time()
     download_time = download_finished - download_started saved_filename =

     current_filename

     if resource.status == RNS.Resource.COMPLETE: counter =
          0
          while os.path.isfile(saved_filename): counter += 1
               saved_filename = current_filename + "." + str(counter)

          try:
               file = open(saved_filename, "wb")
```

```
                file.write(resource.data.read()) file.close()
                menu_mode = "download_concluded" except:
                menu_mode = "save_error"
     else:
          menu_mode = "download_concluded"

# Удобная функция для вывода размера файла в# удобочитаемом
виде
def size_str(num, suffix='B'):
     units = ['','Ki','Mi','Gi','Ti','Pi','Ei','Zi']
     last_unit = 'Yi'

     if suffix == 'b':
          num *= 8
          units = ['', 'K', 'M', 'G', 'T', 'P', 'E', 'Z']
          последняя_единица = 'Y'

     for unit in units:
          if abs(num) < 1024.0:
               return "%3.2f %s%s" % (num, unit, suffix) num /=
          1024.0
     return "%.2f %s%s" % (num, last_unit, suffix)

# Удобная функция для очистки экрана def clear_screen():
     os.system('cls', если os.name=='nt', иначе 'clear')

############################################################  ####  Запуск
программы                                  ######################################
############################################################

# Эта часть программы запускается при запуске,
# анализирует ввод пользователя, а затем # запускает нужный
режим работы программы.
if ___name____ == "___main___":
     try:
          parser = argparse.ArgumentParser(
               description="Простой сервер и клиент для передачи файлов
"
          )
```

```
          parser.add_argument( "-
               s",
               "--serve",
               action="store",
               metavar="dir",
               help="предоставить каталог файлов клиентам"
          )

          parser.add_argument( "--
               config",
               action="store",
               default=None,
               help="путь к альтернативному каталогу конфигурации Reticulum
",
               type=str
          )

          parser.add_argument(
               "destination",
               nargs="?", default=None,
               help="шестнадцатеричный хеш адреса назначения сервера", type=str
          )

          args = parser.parse_args() if

          args.config:
               configarg = args.config else:
               configarg = None

          if args.serve:
               if os.path.isdir(args.serve): server(configarg,
                    args.serve)
               else:
                    RNS.log("Указанный каталог не

exist")
          else:
               if (args.destination == None):
                    print("") parser.print_help()
                    print("")
```

```
            else:
                client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        sys.exit(0)
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/Filetransfer.py.

Пользовательские интерфейсы

Пример *ExampleInterface* демонстрирует создание пользовательских интерфейсов для Reticulum. Reticulum может загружать и использовать любое количество пользовательских интерфейсов, которые будут полностью равноценны встроенным интерфейсам, включая все поддерживаемые режимы работы интерфейсов и общие параметры настройки.

```
# Этот пример иллюстрирует создание определения пользовательского интерфейса
# определения, которое может быть загружено и использовано Reticulum во время
# выполнения. Можно создать и загрузить любое количество пользовательских
интерфейсов. Чтобы использовать интерфейс, поместите его в папку #
~/.reticulum/interfaces и добавьте запись об интерфейсе в
# файл конфигурации Reticulum, подобную этой:

#   [[Пример пользовательского интерфейса]]
#     type = ExampleInterface
#     enabled = no
#     режим = шлюз
#     порт = /dev/ttyUSB0 #
      скорость = 115200
#     биты данных = 8
#     паритет = нет
#     стоп-биты = 1

from time import sleep
import sys
import threading import
time
```

```
# Этот вспомогательный класс HDLC используется интерфейсом # для
разбиения на сегменты и формирования пакетов данных при передаче по
физическому # носителю — в данном случае по последовательному
соединению.
class HDLC():
    # В этом примере интерфейс пакетирует данные с использованием #
    упрощенной структуры HDLC, аналогичной PPP
    FLAG       = 0x7E
    ESC        = 0x7D
    ESC_MASK = 0x20

    @staticmethod
    def escape(data):
        data = data.replace(bytes([HDLC.ESC]), bytes([HDLC.ESC,
HDLC.ESC^HDLC.ESC_MASK]))
        data = data.replace(bytes([HDLC.FLAG]), bytes([HDLC.ESC,
HDLC.FLAG^HDLC.ESC_MASK]))
        return data

# Определим наш собственный класс интерфейса. Он должен # быть
подклассом класса RNS «Interface». class
ExampleInterface(Interface):
    # Все классы интерфейсов должны определять размер # IFAC по
    умолчанию, используемый при настройке IFAC, # когда
    пользователь не указал собственный размер IFAC.
    # параметр указывается в байтах.
    DEFAULT_IFAC_SIZE = 8

    # Следующие свойства относятся исключительно к данной #
    конкретной реализации интерфейса.
    владелец     = None
    порт         = None
    speed    =    None
    databits   =   None
    parity    =    None
    stopbits   =   None
    serial     = None

    # Все интерфейсы Reticulum должны иметь _____________init____
    #, принимающий 2 позиционных аргумента:
    # экземпляр RNS Transport-владельца и словарь # значений
    конфигурации.
    def ___init___(self, owner, configuration):
```

```
        # Следующие строки демонстрируют обработку #
        потенциальных зависимостей, необходимых для
        # правильной работе интерфейса. import importlib
        if importlib.util.find_spec('serial') != None: import
            serial
        else:
            RNS.log("Для использования этого интерфейса необходимо установить модуль
последовательной связи.", RNS.LOG_CRITICAL)
            RNS.log("Вы можете установить его с помощью команды: python3 -m
pip install pyserial", RNS.LOG_CRITICAL)
            RNS.panic()

        # Для начала инициализируем суперкласс с помощью super().
        _____init___()

        # Чтобы убедиться, что данные конфигурации имеют
        # правильном формате, мы анализируем их с помощью следующего #
        метода в общем классе Interface. Этот шаг
        # необходим для обеспечения совместимости на всех # платформах,
        поддерживаемых Reticulum.
        ifconf      = Interface.get_config_obj(configuration)

        # Считываем имя интерфейса из конфигурации # и устанавливаем его в
        нашем экземпляре интерфейса.
        name        = ifconf["name"]
        self.name = name

        # Мы считываем параметры конфигурации из предоставленных # данных
        конфигурации и указываем значения по умолчанию в
        # случае, если какие-либо из них отсутствуют.
        port        = ifconf["port"] if "port" in ifconf else
None
        скорость      = int(ifconf["speed"]) if "speed" in ifconf
else 9600
        databits    = int(ifconf["databits"]) if "databits" in ifconf
else 8
        parity      = ifconf["parity"] if "parity" in ifconf else
"N"
            stopbits    = int(ifconf["stopbits"]) if "stopbits" in
ifconf, иначе 1

            # Если порт не указан, мы прерываем настройку с помощью
```

```
# Вывод исключения. if port ==
None:
     raise ValueError(f"Не указан порт для {self}")

# Все интерфейсы должны предоставлять значение аппаратного MTU
# экземпляру RNS Transport. Это значение должно # представлять
собой максимальный размер полезной нагрузки пакета данных, который #
базовый канал способен обрабатывать во всех
# случаях без какой-либо сегментации.
self.HW_MTU = 564

# Изначально мы устанавливаем свойство «online» в false, # так как
интерфейс еще не был полностью
# инициализирован и подключен. self.online
                  = False

# В этом случае мы также можем установить указанную скорость передачи
данных интерфейса равной скорости последовательного порта.
self.bitrate = speed

# Настроим внутренние свойства интерфейса # в соответствии с
предоставленной конфигурацией. self.pyserial = serial
self.serial      = None
self.owner       = owner
self.port        = port
self.speed       = speed
self.databits = databits
self.parity      = serial.PARITY_NONE
self.stopbits = stopbits self.timeout =
100

if parity.lower() == "e" or parity.lower() == "even": self.parity =
     serial.PARITY_EVEN

if parity.lower() == "o" or parity.lower() == "odd": self.parity =
     serial.PARITY_ODD

# Поскольку все необходимые параметры теперь настроены, # попробуем
открыть последовательный порт.
try:
     self.open_port() except
Exception as e:
```

```
                RNS.log("Не удалось открыть последовательный порт для интерфейса
"+str(self), RNS.LOG_ERROR)
                raise e

            # Если открытие порта прошло успешно, выполните любую #
            необходимую конфигурацию после открытия.
            if self.serial.is_open: self.configure_device()
            else:
                raise IOError("Не удалось открыть последовательный порт")

        # Открыть последовательный порт с заданными
        # и сохраните ссылку на открытый порт. def open_port(self):
            RNS.log("Открытие последовательного порта "+self.port+"...",
RNS.LOG_VERBOSE)
            self.serial = self.pyserial.Serial( port =
                self.port,
                baudrate = self.speed,
                bytesize = self.databits,
                parity = self.parity, stopbits
                = self.stopbits, xonxoff =
                False,
                rtscts = False, timeout
                = 0,
                inter_byte_timeout = None,
                write_timeout = None, dsrdtr =
                False,
            )

        # Единственное, что требуется после открытия порта, # — это немного
        подождать, пока
        # инициализации оборудования, а затем запустить поток, # который
        будет считывать входящие данные с устройства. def
        configure_device(self):
            sleep(0.5)
            thread = threading.Thread(target=self.read_loop) thread.daemon = True
            thread.start() self.online =
            True
            RNS.log("Последовательный порт "+self.port+" теперь открыт",
RNS.LOG_VERBOSE)
```

```
        # Этот метод будет вызван из нашего цикла чтения # всякий раз,
        когда полный пакет будет получен по # базовому каналу.
        def process_incoming(self, data):
            # Обновим счетчик принятых байтов self.rxb +=
            len(data)

            # И отправляем пакет данных в экземпляр # Transport для
            обработки. self.owner.inbound(data, self)

        # Работающий экземпляр Reticulum Transport
        # вызовет этот метод на интерфейсе всякий раз, когда # интерфейсу
        необходимо передать пакет.
        def process_outgoing(self, data): if
            self.online:
                # Сначала преобразуем и упаковываем данные # в
                соответствии с форматированием HDLC.
                data = bytes([HDLC.FLAG])+HDLC.escape(data)+bytes([HDLC.FLAG])

                # Затем записываем сформированные данные в порт written =
                self.serial.write(data)

                # Обновите счетчик переданных байтов # и убедитесь,
                что все данные были записаны self.txb +=
                len(data)
                if written != len(data):
                    raise IOError("По последовательному интерфейсу было
записано только "+str(written)+" байт из "+str(len(data)))

        # Этот цикл чтения выполняется в отдельном потоке и непрерывно #
        принимает байты из базового последовательного порта. # После
        получения полного пакета он
        # отправляется в метод process_incoming, который
        # в свою очередь передаст его экземпляру Transport. def
        read_loop(self):
            try:
                in_frame = False escape
                = False data_buffer =
                b""
                last_read_ms = int(time.time()*1000)
```

```
                while self.serial.is_open:
                     if self.serial.in_waiting:
                          byte = ord(self.serial.read(1)) last_read_ms =
                          int(time.time()*1000)

                          if (in_frame и byte == HDLC.FLAG): in_frame = False
                               self.process_incoming(data_buffer)
                          elif (byte == HDLC.FLAG): in_frame
                               = True data_buffer = b""
                          elif (in_frame и len(data_buffer) <

                               if (byte == HDLC.ESC): escape =
                                    True
self.HW_MTU):                  else:
                               if (escape):
                                      if (byte == HDLC.FLAG ^

                                          byte = HDLC.FLAG
                                       if (byte == HDLC.ESC ^

HDLC.ESC_MASK):
                                         byte = HDLC.ESC
                                    escape = False
                               data_buffer =
HDLC.ESC_MASK):
data_buffer+bytes([byte])

                     else:
                          time_since_last = int(time.time()*1000) -
last_read_ms
                          if len(data_buffer) > 0 and time_since_last data_buffer
> self.timeout:
                               = b""
                               in_frame = False escape =
                               False
                          sleep(0.08)

          except Exception as e:
               self.online = False
               RNS.log("Произошла ошибка последовательного порта, содержащееся
исключение: "+str(e), RNS.LOG_ERROR)
               RNS.log("В интерфейсе "+str(self)+" произошла
```

```
неустранимая ошибка, и теперь он находится в автономном режиме.", RNS.LOG_ERROR)

                if RNS.Reticulum.panic_on_interface_error: RNS.panic()

                RNS.log("Reticulum будет периодически пытаться переподключить
интерфейс.", RNS.LOG_ERROR)

          self.online = False
          self.serial.close()
          self.reconnect_port()

     # Этот метод обрабатывает отключения последовательного порта. def
     reconnect_port(self):
          пока self.online не равно
               True: try:
                    time.sleep(5)
                    RNS.log("Попытка повторного подключения последовательного порта
"+str(self.port)+" для "+str(self)+"...", RNS.LOG_VERBOSE)
                    self.open_port()
                    if self.serial.is_open:
                         self.configure_device()
               except Exception as e:
                    RNS.log("Ошибка при повторном подключении порта, исключение:
"+str(e), RNS.LOG_ERROR)

          RNS.log("Переподключен последовательный порт для "+str(self)) #

     Сигнал Reticulum о том, что этот интерфейс
     # не должен выполнять ограничение входящего трафика.
     def should_ingress_limit(self): return
          False

     # Мы должны предоставить строковое представление этого # интерфейса,
     которое используется всякий раз, когда интерфейс
     # выводится в журналы или внешние программы. def
     ________str___(self):
          return "ExampleInterface["+self.name+"]"

# Наконец, зарегистрируем определенный класс интерфейса в качестве #
целевого класса, который Reticulum будет использовать в качестве интерфейса
interface_class = ExampleInterface
```

Этот пример также можно найти по адресу https://github.com/markqvist/Reticulum/blob/master/Examples/ExampleInt erface.py.

Лицензия Reticulum

Лицензия Reticulum

Авторское право (c) 2016-2026 Марк Квист

Настоящим бесплатно предоставляется разрешение любому лицу, получившему копию
этого программного обеспечения и связанных с ним файлов документации (далее — «Программное обеспечение»),
с Программным обеспечением без ограничений, включая, помимо прочего, права
на использование, копирование, изменение, объединение, публикацию, распространение,
сублицензирование и/или продажу
копии Программного обеспечения, а также разрешать лицам, которым Программное
обеспечение
при наличии соответствующих полномочий и при соблюдении следующих условий:

- Программное обеспечение не должно использоваться в каких-либо
системах, которые включают в
  среди своих функций возможность целенаправленно причинять вред людям.

- Программное обеспечение не должно использоваться, прямо или косвенно,
при создании
  набора данных для обучения искусственного интеллекта, машинного обучения или
языковых моделей
  , включая, но не ограничиваясь, любое использование, способствующее
  обучению или разработке такой модели или алгоритма.

- Вышеуказанное уведомление об авторских правах и настоящее разрешение
должны быть включены в
  всех копиях или существенных частях Программного обеспечения.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПРЕДОСТАВЛЯЕТСЯ «КАК ЕСТЬ», БЕЗ КАКИХ-ЛИБО
ГАРАНТИЙ, ЯВНЫХ ИЛИ
ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ, ВКЛЮЧАЯ, ПОМИМО ПРОЧЕГО, ГАРАНТИИ ТОВАРНОГО
СОСТОЯНИЯ,
ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ И НЕНАРУШЕНИЯ ПРАВ. НИ В

СЛУЧАЕ
АВТОРЫ ИЛИ ВЛАДЕЛЬЦЫ АВТОРСКИХ ПРАВ НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ ПРЕТЕНЗИИ, УБЫТКИ ИЛИ ИНУЮ
ОТВЕТСТВЕННОСТЬ, БУДЬ ТО В РАМКАХ ИСКА ПО ДОГОВОРУ, ИСКА О ВЫПЛАТЕ КОМПЕНСАЦИИ ЗА ПРИЧИНЕННЫЙ УЩЕРБ ИЛИ ИНОГО ИСКА, ВОЗНИКШИЕ ИЗ,
ИЛИ В СВЯЗИ С ПРОГРАММНЫМ ОБЕСПЕЧЕНИЕМ ИЛИ ИСПОЛЬЗОВАНИЕМ ИЛИ ДРУГИМИ ОПЕРАЦИЯМИ С
ПРОГРАММНЫМ ОБЕСПЕЧЕНИЕМ.

Справочник API

Обмен данными в сетях Reticulum осуществляется с помощью простого набора классов, предоставляемых API RNS. В этой главе перечислены и описаны все классы, предоставляемые API сетевого стека Reticulum, а также сигнатуры их методов
сигнатур и способов использования. Его можно использовать в качестве справочника при написании приложений, использующих Reticulum, или прочитать целиком, чтобы получить представление о полном функционале RNS с точки зрения разработчика.

Reticulum

class RNS.Reticulum(*configdir=None*, *loglevel=None*, *logdest=None*, *verbosity=None*, *require_shared_instance=False*, *shared_instance_type=None*)

> Этот класс используется для инициализации доступа к Reticulum внутри программы. Вы должны создать ровно один экземпляр этого класса, прежде чем выполнять какие-либо другие операции RNS, такие как создание адресатов или отправка трафика.
> Каждая независимо выполняемая программа должна создавать свой собственный экземпляр класса Reticulum, но Reticulum автоматически будет обрабатывать межпрограммную коммуникацию в одной системе и предоставлять доступ ко всем подключенным
> а также к внешним интерфейсам.
>
> Как только создается экземпляр этого класса, Reticulum начинает открывать и настраивать любые аппаратные устройства, указанные в предоставленной конфигурации.
>
> В настоящее время первый запущенный экземпляр должен оставаться запущенным, пока подключаются другие локальные экземпляры, так как первый созданный экземпляр будет выступать в качестве главного экземпляра, напрямую взаимодействующего с внешним оборудованием, таким как модемы, TNC и радиостанции. Если главный экземпляр получит запрос на завершение работы, он не завершит работу, пока не будут завершены все клиентские процессы (если их не убить принудительно).
>
> Если вы запускаете Reticulum в системе с несколькими различными программами, использующими RNS, которые запускаются и завершают работу в разное время, будет

целесообразно запустить главный экземпляр RNS в качестве демона, чтобы другие программы могли использовать его по требованию.

MTU = *500*

MTU, которого придерживается Reticulum и которого он ожидает от других узлов
. По умолчанию MTU составляет 500 байт. В пользовательских реализациях сетей RNS можно изменить это значение, но это полностью нарушит совместимость со всеми другими сетями RNS. Идентичный MTU является обязательным условием для взаимодействия узлов в одной сети.

Если вы не уверены в том, что делаете, MTU следует оставить на значении по умолчанию.

LINK_MTU_DISCOVERY = *True*

Определяет, включено ли по умолчанию в данной версии автоматическое обнаружение MTU канала. Обнаружение MTU канала значительно увеличивает пропускную способность на быстрых каналах, но требует, чтобы все промежуточные узлы также поддерживали эту функцию.
Поддержка этой функции была добавлена в версии RNS 0.9.0. В ближайшем будущем этот параметр будет включен по умолчанию. Обновите, пожалуйста, ваши экземпляры RNS.

ANNOUNCE_CAP = *2*

Максимальный процент пропускной способности интерфейса, который в любой момент времени может быть использован для распространения объявлений. Если объявление было запланировано для широковещательной передачи по интерфейсу, но это привело бы к превышению разрешенного выделения пропускной способности, объявление будет поставлено в очередь для передачи, когда появится доступная пропускная способность.

Reticulum всегда будет отдавать приоритет распространению объявлений с меньшим количеством прыжков, гарантируя, что удаленные, крупные сети со множеством пиров на быстрых каналах не перегрузят пропускную способность меньших сетей на более медленных
. Если объявление остается в очереди в течение длительного времени, оно в конечном итоге будет отброшено.

Это значение по умолчанию применяется ко всем созданным интерфейсам, но его можно настроить индивидуально для каждого интерфейса. Как правило, глобальную настройку по умолчанию менять не следует, а любые изменения следует вносить на уровне отдельных интерфейсов.

MINIMUM_BITRATE = *5*

Минимальная скорость передачи данных, требуемая по каналу связи для успешного установления соединений Reticulum. В настоящее время 5 бит в секунду.

static get_instance()

Вернуть текущий экземпляр Reticulum

static should_use_implicit_proof()

Возвращает, являются ли отправленные доказательства явными или неявными.

Возвращает:	True, если текущая запущенная конфигурация указывает на использование неявных доказательств. False, если нет.

static transport_enabled()

Возвращает, включен ли Transport для запущенного экземпляра.

Когда Transport включен, Reticulum будет маршрутизировать трафик для других пиров, отвечать на запросы о пути и передавать объявления по сети.

Возвращает:	True, если Transport включен, False, если нет.

static link_mtu_discovery()

Возвращает, включено ли обнаружение MTU канала для запущенного экземпляра.

Когда обнаружение MTU канала включено, Reticulum автоматически увеличить размер MTU до максимально допустимого значения, что повысит скорость

скорость и эффективность передачи данных.

Возвращает:: True, если обнаружение MTU канала включено, False, если нет.

static remote_management_enabled()

Возвращает значение, указывающее, включено ли удаленное управление для запущенного экземпляра.

Когда удаленное управление включено, авторизованные узлы могут удаленно запрашивать данные об этом экземпляре и управлять им.

Возвращает:: True, если удаленное управление включено, False, если нет.

static required_discovery_value()

Возвращает значение метки, необходимое для того, чтобы обнаруженный интерфейс считался действительным и запоминался.

Возвращает:: Требуемое значение метки в виде целого числа.

static publish_blackhole_enabled()

Возвращает, включена ли публикация списка blackhole для запущенного экземпляра.

Возвращает:: True, если публикация списка blackhole включена, False, если нет.

static blackhole_sources()

Возвращает список хэшей идентификаторов транспорта, из которых берутся списки blackhole.

Возвращает:: Список хешей идентификаторов.

static discovered_interfaces()

Возвращает список интерфейсов, обнаруженных в сети.

Возвращает:: Список обнаруженных интерфейсов.

static interface_discovery_sources()

Возвращает список хэшей сетевых идентификаторов, на основе которых обнаруживаются интерфейсы.

Возвращает:: Список хешей идентификаторов.

Идентификатор

класс RNS.Identity(*create_keys=True*)

Этот класс используется для управления идентификаторами в Reticulum. Он предоставляет методы для шифрования, дешифрования, подписи и проверки и является основой для всей зашифрованной связи в сетях Reticulum.

Параметры:: **create_keys** – Указывает, следует ли генерировать новые ключи шифрования и подписи.

CURVE = *'Curve25519'*

Кривая, используемая для обмена ключами по алгоритму DH с эллиптической кривой

KEYSIZE = *512*

Размер ключа X.25519 в битах. Полный ключ представляет собой объединение 256-битного ключа шифрования и 256-битного ключа подписи.

RATCHETSIZE = *256*

Размер ключа рачета X.25519 в битах.

RATCHET_EXPIRY = *2592000*

Срок действия полученных ратчетов в секундах; по умолчанию — 30 дней. Reticulum всегда будет использовать самый последний объявленный ратчет и сохранять его в памяти в течение периода, равного `RATCHET_EXPIRY` с момента получения, после чего он будет удален. Если в это время будет объявлен более новый ратчет, он заменит уже известный ратчет.

TRUNCATED_HASHLENGTH = *128*

Константа, определяющая длину усеченного хеша (в битах), используемого Reticulum для адресных хешей и других целей. Не настраивается.

static recall(*target_hash*, *from_identity_hash=False*)

Восстановление идентичности для хэша назначения или идентичности. По умолчанию эта функция возвращает идентичность, связанную с заданным хэшем *назначения*. Например, если вам известен хэш назначения `lxmf.delivery` конечной точки, эта функция вернет связанную с ним базовую идентичность. Вы также можете выполнить поиск идентичности по известному *хэшу идентичности*, установив аргумент `from_identity_hash`.

Параметры::
- **target_hash** – хеш назначения или идентификатора в виде *байтов*.
- **from_identity_hash** – Указывает, следует ли выполнять поиск на основе хэша идентификатора вместо хэша назначения (тип *bool*).

Возвращает:: Экземпляр RNS.Identity, который можно использовать для создания исходящего RNS.Destination, или *None,* если пункт назначения неизвестен.

static recall_app_data(*destination_hash*)

Восстановление последних app_data для хэша назначения.

Параметры:: **destination_hash** – хеш назначения в виде *байтов*.
Возвращает::

Байты, содержащие app_data, или *None,* если пункт назначения неизвестен.

static full_hash(*data*)

Получить хеш SHA-256 переданных данных.

Параметры:: **data** – Данные, подлежащие хешированию, в виде *байтов*.
Возвращаемые значения:: Хеш SHA-256 в виде *байтов*.

static truncated_hash(*data*)

Получить усечённый хеш SHA-256 переданных данных.

Параметры:: **data** – Данные для хеширования в виде *байтов*.
Возвращает:: Усечённый хеш SHA-256 в виде *байтов*.

static get_random_hash()

Получить случайный хеш SHA-256.

Параметры:: **data** – Данные для хеширования в виде *байтов*.
Возвращает:: Усечённый хеш SHA-256 случайных данных в виде *байтов*.

static current_ratchet_id(*destination_hash*)

Получить ID ключа рачета, используемого в данный момент для заданного хэша назначения

Параметры:: **destination_hash** – Хеш назначения в виде *байтов*.
Возвращает:: Идентификатор рачета в виде *байтов* или *None*.

static from_bytes(*prv_bytes*)

Создать новый экземпляр RNS.Identity из *байтов* закрытого ключа. Может использоваться для загрузки ранее созданных и сохраненных идентичностей в Reticulum.

Параметры:: **prv_bytes** – *байты* сохраненного закрытого ключа. **ОПАСНО!** Ни в коем случае не используйте это для генерации нового ключа путем ввода случайных данных в prv_bytes.

Возвращает:: Экземпляр RNS.Identity или *None*, если данные в *байтах* были неверны.

static from_file(*path*)

Создание нового экземпляра RNS.Identity из файла. Может использоваться для загрузки ранее созданных и сохраненных идентичностей в Reticulum.

Параметры:: **path** – полный путь к сохраненным данным RNS.Identity

Возвращает:: Экземпляр RNS.Identity или *None*, если загруженные данные были неверны.

to_file(*path*)

Сохраняет идентификатор в файл. При этом на диск записывается закрытый ключ, и любой, у кого есть доступ к этому файлу, сможет расшифровать всю переписку, связанную с этим идентификатором. Будьте очень осторожны при использовании этого метода.

Параметры:: **path** – Полный путь, указывающий, где сохранить идентификатор.

Возвращает:: True, если файл был сохранен, в противном случае False.

get_private_key()

Возвращает:: Личный ключ в виде *байтов*

get_public_key()

Возвращает:: Открытый ключ в виде *байтов*

load_private_key(*prv_bytes*)

Загрузить закрытый ключ в экземпляр.

Параметры::	**prv_bytes** – частный ключ в виде *байтов*.
Возвращаемые значения::	True, если ключ был загружен, в противном случае False.

load_public_key(*pub_bytes*)

Загрузить открытый ключ в экземпляр.

Параметры::	**pub_bytes** – открытый ключ в виде *байтов*.
Возвращает::	True, если ключ был загружен, иначе False.

encrypt(*plaintext*, *ratchet=None*)

Шифрует информацию для идентичности.

Параметры::	**plaintext** – открытый текст, который нужно зашифровать, в виде *байтов*.
Возвращает::	Токен шифротекста в виде *байтов*.
Вызывает исключение::	*KeyError,* если экземпляр не содержит открытого ключа.

decrypt(*ciphertext_token*, *ratchets=None*, *enforce_ratchets=False*, *ratchet_id_receiver=None*)

Расшифровывает информацию для идентификатора.

Параметры::	**ciphertext** – Шифротекст, который необходимо расшифровать, в виде *байтов*.
Возвращает::	Открытый текст в виде *байтов* или *None*, если расшифровка не удалась. *KeyError,* если экземпляр не содержит секретного ключа.
Вызывает исключение::	

sign(*message*)

Подписывает информацию с помощью идентификатора.

Параметры:: **message** – Сообщение, которое необходимо подписать, в виде *байтов*. Подпись в виде *байтов*.
Возвращает:: *KeyError,* если экземпляр не содержит закрытый ключ.
Генерирует::

validate(*signature*, *message*)

Проверяет подлинность подписи подписанного сообщения.

Параметры::
- **signature** – Подпись, подлежащая проверке, в виде *байтов*.
- **message** – Сообщение, подлежащее проверке, в виде *байтов*.

Возвращает:: True, если подпись действительна, в противном случае False. *KeyError,* если экземпляр не содержит открытый ключ.
Вызывает::

Destination

класс RNS.Destination(*identity*, *direction*, *type*, *app_name*, **aspects*)

Класс, используемый для описания конечных точек в сети Reticulum. Экземпляры конечных точек используются как для создания исходящих, так и входящих конечных точек. Тип конечной точки определяет, будет ли при взаимодействии с ней применяться шифрование и какого именно типа. Конечная точка также может объявить о своём присутствии в сети, что обеспечит распространение необходимых ключей для зашифрованной связи с ней.

Параметры::

- **identity** – экземпляр RNS.Identity. Может содержать только открытые ключи для исходящего пункта назначения или закрытые ключи для входящего.
- **direction** – `RNS.Destination.IN` или `RNS.Destination.OUT`.
- **type** – `RNS.Destination.SINGLE`, `RNS.Destination.GROUP` или `RNS.Destination.PLAIN`.
- **app_name** – Строка, указывающая имя приложения.
- ***aspects** – Любое ненулевое количество строковых аргументов.

RATCHET_COUNT = *512*

Количество ключей рачета по умолчанию, которое сохранит пункт назначения, если у него включены рачеты.

RATCHET_INTERVAL = *1800*

Минимальный интервал между ротацией ключей рачета, в секундах.

static expand_name(*identity*, *app_name*, **aspects*)

Возвращает:: Строка, содержащая полное, понятное человеку имя назначения для app_name и ряда аспектов.

static app_and_aspects_from_name(*full_name*)

Возвращает:: Кортеж, содержащий название приложения и список аспектов, для строки с полным именем.

static hash_from_name_and_identity(*full_name*, *identity*)

Возвращает:: Имя назначения в форме адресуемого хеша для строки полного имени и экземпляра Identity.

static hash(*identity*, *app_name*, **aspects*)

Возвращает:: Имя назначения в форме адресуемого хеша для app_name и ряда аспектов.

announce(*app_data=None*, *path_response=False*, *attached_interface=None*, *tag=None*, *send=True*)

Создает пакет объявления для этого адреса и транслирует его по всем соответствующим интерфейсам. К объявлению можно добавить данные, специфичные для приложения.

Параметры::

- **app_data** – *байты,* содержащие app_data.
- **path_response** – Внутренний флаг, используемый RNS.Transport. Игнорируется.

accepts_links(*accepts=None*)

Устанавливает или запрашивает, принимает ли пункт назначения входящие запросы на установку соединения.

Параметры:: **accepts** – Если `True` или `False`, этот метод устанавливает, принимает ли пункт назначения входящие запросы на установку соединения. Если не указан или `None`, метод возвращает, принимает ли пункт назначения в данный момент принимает запросы на установку соединения.

Возвращает:: «`True`» или «`False`» в зависимости от того, принимает ли целевой сервер входящие запросы на установку соединения, если параметр *accepts* не указан или имеет значение `None`.

set_link_established_callback(*callback*)

Регистрирует функцию, которая будет вызвана при установке соединения с этим адресатом.

Параметры:: call**back** — функция или метод с сигнатурой *callback(link)*, которая будет вызвана при установке нового соединения с этим адресатом.

set_packet_callback(*callback*)

Регистрирует функцию, которая будет вызываться при получении пакета этим адресатом.

Параметры:: call**back** – функция или метод с сигнатурой *callback(data, packet)*, которая будет вызываться при получении пакета этим адресатом.

set_proof_requested_callback(*callback*)

Регистрирует функцию, которая будет вызываться при запросе доказательства для пакета, отправленного в этот пункт назначения. Позволяет контролировать, когда и должны ли должны возвращаться доказательства для полученных пакетов.

Параметры:: **callback** — функция или метод с сигнатурой *callback(packet)*, вызываемый при получении пакета, содержащего запрос на подтверждение. callback должен возвращать значение True или False. Если callback возвращает True, доказательство будет отправлено. Если он возвращает False, доказательство не будет отправлено.

set_proof_strategy(*proof_strategy*)

Устанавливает стратегию подтверждения для адресата.

Параметры:: **proof_strategy** – одно из `значений RNS.Destination.PROVE_NONE`, `RNS.Destination.PROVE_ALL` или `RNS.Destination.PROVE_APP`. Если установлено значение `RNS.Destination.PROVE_APP`, будет вызван *callback_proof_requested* для определения, следует ли отправлять доказательство или нет.

register_request_handler(*path*, *response_generator=None*, *allow=ALLOW_NONE*, *allowed_list=None*, *auto_compress=True*)

Регистрирует обработчик запросов.

Параметры::

- **path** – Путь для обрабатывателя запросов, который будет зарегистрирован.
- **response_generator** – Функция или метод с сигнатурой *response_generator(path, data, request_id, link_id, remote_identity, requested_at)*, который будет вызван. Все, что возвращает эта функция, будет отправлено в качестве ответа запрашивающему. Если функция возвращает `None`, ответ не будет отправлен.
- **allow** – одно из `значений` `RNS.Destination.ALLOW_NONE`, `RNS.Destination.ALLOW_ALL` или `RNS.Destination.ALLOW_LIST`. Если установлено значение `RNS.Destination.ALLOW_LIST`, обработчик запросов будет отвечать только на запросы от идентифицированных узлов, указанных в предоставленном списке.
- **allowed_list** — список хэшей RNS.Identity *muna bytes*.
- **auto_compress** – При значении `True` или `False` определяет, следует ли выполнять автоматическое сжатие ответов. Если установлено целое число, ответы будут автоматически сжиматься только в том случае, если их размер в байтах меньше этого значения. Если этот параметр опущен, будут использоваться настройки сжатия по умолчанию.

Генерирует исключения::

`ValueError`, если какой-либо из переданных аргументов недействителен.

deregister_request_handler(*path*)

Отменяет регистрацию

обработчика запросов.

Параметры::

path – Путь к обработчику запросов, который необходимо отменить.

Возвращаемые значения::

True, если обработчик был отрегистрирован, в противном случае False.

enable_ratchets(*ratchets_path*)

Включает рачеты для пункта назначения. Когда рачеты включены, Reticulum будет автоматически менять ключи, используемые для шифрования пакетов, отправляемых в этот пункт назначения, и включать последний ключ рачета в объявления.

Включение рачет на пункте назначения обеспечит прямую секретность для пакетов, отправляемых в этот пункт назначения, даже если они отправляются за пределы `Link`.
Обычная процедура установления соединения Reticulum уже выполняет собственный обмен эфемеричными ключами при каждом установлении соединения, что
означает, что рачеты не нужны для обеспечения секретности перехода для каналов.

Включение рачет-механизмов окажет небольшое влияние на размер объявлений, добавляя 32 байта к каждому отправляемому объявлению.

Параметры:: **ratchets_path** — путь к файлу для хранения данных рачет.

Возвращаемые значения:: True, если операция завершилась успешно, в противном случае — False.

enforce_ratchets()

Когда принудительное использование рачет-ключей включено, этот пункт назначения никогда не будет принимать пакеты, использующие его базовый ключ Identity для шифрования, а будет принимать только пакеты, зашифрованные одним из сохраненных рачет-ключей.

set_retained_ratchets(*retained_ratchets*)

Устанавливает количество ранее сгенерированных ключей рачет, которые этот пункт назначения будет сохранять и пытаться использовать при расшифровке входящих пакетов. По умолчанию — `Destination.RATCHET_COUNT`.

Параметры:: **retained_ratchets** – Количество сгенерированных ключей рачета, которые необходимо сохранить.

Возвращаемые значения:: True, если операция завершилась успешно, False — в противном случае.

set_ratchet_interval(*interval*)

Устанавливает минимальный интервал в секундах между поворотами ключа с храповым механизмом. По умолчанию используется значение `Destination.RATCHET_INTERVAL`.

Параметры::	**interval** – минимальный интервал в секундах. True,
Возвращаемые значения::	если операция завершилась успешно, False в противном случае.

create_keys()

Для пункта назначения типа `RNS.Destination.GROUP` создает новый симметричный ключ.

Генерирует::	`TypeError`, если вызвано для несовместимого типа назначения.

get_private_key()

Для назначения типа `RNS.Destination.GROUP` возвращает симметричный закрытый ключ.

Вызывает:	`TypeError`, если вызвано для несовместимого типа назначения.

load_private_key(*key*)

Для назначения типа `RNS.Destination.GROUP` загружает симметричный закрытый ключ.

Параметры::	**key** – объект *muna bytes,* содержащий симметричный ключ.
Вызывает::	`TypeError`, если вызвано для несовместимого типа назначения.

encrypt(*plaintext*)

Шифрует информацию для `RNS.Destination.SINGLE` или `RNS.Destination.GROUP`.

Параметры:: **plaintext** – объект *muna bytes,* содержащий открытый текст, который необходимо зашифровать.

Вызывает:: `ValueError`, если destination не содержит ключ, необходимый для шифрования.

decrypt(*ciphertext*)

Дешифрует информацию для `RNS.Destination.SINGLE` или `RNS.Destination.GROUP`.

Параметры:: **ciphertext** – *байты,* содержащие шифротекст, который необходимо расшифровать.

Вызывает:: `ValueError`, если в объекте destination отсутствует ключ, необходимый для расшифровки.

sign(*message*)

Подписывает информацию для адреса типа `RNS.Destination.SINGLE`.

Параметры:: **message** – *байты,* содержащие сообщение, которое необходимо подписать.

Возвращаемые значения:: Объект *muna bytes*, содержащий подпись сообщения, или *None,* если пункт назначения не смог подписать сообщение.

set_default_app_data(*app_data=None*)

Устанавливает app_data по умолчанию для пункта назначения. Если установлено, то
app_data будет включено в каждое объявление, отправляемое адресатом, если в методе *announce* не указано другое app_data.

Параметры:: **app_data** — объект *muna bytes*, содержащий app_data по умолчанию, или *вызываемая функция,* возвращающая объект *muna bytes,* содержащий app_data.

clear_default_app_data()

Очищает app_data по умолчанию, ранее установленные для пункта назначения.

Пакет

класс RNS.Packet(*destination*, *data*, *create_receipt=True*)

Класс Packet используется для создания экземпляров пакетов, которые можно отправлять по сети Reticulum. Пакеты будут автоматически зашифрованы, если они адресованы пункту назначения `RNS.Destination.SINGLE`, `RNS.Destination.GROUP` или RNS.Link.

Для адресатов `RNS.Destination.GROUP` Reticulum будет использовать предварительно согласованный ключ, настроенный для данного адреса. Все пакеты, отправляемые группе
Направления шифруются с использованием одного и того же ключа AES-256.

Для пунктов назначения `RNS.Destination.SINGLE` Reticulum будет использовать новый эфемеричный ключ AES-256 для каждого пакета.

Для адресатов RNS.Link Reticulum будет использовать эфемеричные ключи для каждого соединения и обеспечивает **передовую секретность**.

Параметры::

- **destination** – экземпляр RNS.Destination, на который будет отправлен пакет.
- **data** – полезные данные, которые будут включены в пакет в виде *байтов*.
- **create_receipt** – Указывает, следует ли создавать RNS.PacketReceipt при создании экземпляра пакета.

ENCRYPTED_MDU = *383*

Максимальный размер данных полезной нагрузки в одном зашифрованном

пакете PLAIN_MDU = *464*

Максимальный размер данных полезной нагрузки в одном незашифрованном пакете

send()

Отправляет пакет.

Возвращает:: Экземпляр RNS.PacketReceipt, если при создании пакета для параметра *create_receipt* было задано значение *True*; в противном случае возвращается *None*. Если пакет не удалось отправить, возвращается *False*.

resend()

Повторно отправляет пакет.

Возвращает:: Экземпляр RNS.PacketReceipt, если при создании пакета для параметра *create_receipt* было задано значение *True*; в противном случае возвращается *None*. Если пакет не удалось отправить, возвращается *False*.

get_rssi()

Возвращает:: *Показатель уровня сигнала* на физическом уровне, если доступен, в противном случае — `None`.

get_snr()

Возвращает:: *Коэффициент сигнал/шум* физического уровня, если доступен, в противном случае `None`.

get_q()

Возвращает:: *Качество канала* физического уровня, если доступно, в противном случае `None`.

Получение пакета

класс RNS.PacketReceipt

Класс PacketReceipt используется для получения уведомлений об экземплярах RNS.Packet, отправленных по сети. Экземпляры этого класса никогда не создаются вручную, а всегда возвращаются методом *send()* экземпляра RNS.Packet.

get_status()

Возвращает: Статус связанного экземпляра RNS.Packet. Может принимать одно из значений: `RNS.PacketReceipt.SENT`, `RNS.PacketReceipt.DELIVERED`, `RNS.PacketReceipt.FAILED` или `RNS.PacketReceipt.CULLED`.

get_rtt()

Возвращает:: Время прохождения в секундах

set_timeout(*timeout*)

Устанавливает таймаут в секундах

Параметры:: **timeout** – Тайм-аут в секундах.

set_delivery_callback(*callback*)

Задает функцию, которая вызывается, если подтверждена успешная доставка.

Параметры:: **callback** – *Вызываемая* функция с сигнатурой *callback(packet_receipt)*

set_timeout_callback(*callback*)

Задает функцию, которая вызывается, если доставка завершилась таймаутом.

Параметры:: **callback** – *Вызываемая* функция с сигнатурой *callback(packet_receipt)*

Link

class RNS.Link(*destination*, *established_callback=None*, *closed_callback=None*)

Этот класс используется для установления и управления соединениями с другими узлами. При создании экземпляра соединения Reticulum попытается установить проверенное и зашифрованное соединение с указанным адресатом.

Параметры::

- **destination** – экземпляр RNS.Destination, с которым будет установлено соединение.
- **established_callback** — опциональная функция или метод с сигнатурой *callback(link)*, вызываемый при установке соединения.
- **closed_callback** – опциональная функция или метод с сигнатурой *callback(link)*, вызываемый при закрытии соединения.

CURVE = *'Curve25519'*

Кривая, используемая для обмена ключами в алгоритме DH на основе эллиптических кривых

ESTABLISHMENT_TIMEOUT_PER_HOP = *6*

Таймаут на установку соединения в секундах на каждый прыжок до пункта назначения.

KEEPALIVE_TIMEOUT_FACTOR = *4*

Коэффициент таймаута RTT, используемый при расчете таймаута соединения.

STALE_GRACE = *5*

Период отсрочки в секундах, используемый при расчете таймаута соединения.

KEEPALIVE = *360*

Интервал по умолчанию для отправки пакетов keep-alive по установленным каналам связи в секундах.

STALE_TIME = *720*

Если в течение этого периода не поступает трафик или пакеты keep-alive, канал помечается как устаревший, и отправляется последний пакет keep-alive. Если после этого в течение `RTT * KEEPALIVE_TIMEOUT_FACTOR + STALE_GRACE` не поступает трафик или пакеты keep-alive, канал считается прерванным по таймауту и будет разорван.

identify(*identity*)

Идентифицирует инициатора соединения для удаленного узла. Это может произойти только после установления соединения и осуществляется по зашифрованному каналу. Идентификатор раскрывается только удаленному узлу, что позволяет сохранить анонимность инициатора. Этот метод может использоваться для аутентификации.

Параметры:: **identity** — экземпляр RNS.Identity, который будет использоваться для идентификации.

request(*path*, *data=None*, *response_callback=None*, *failed_callback=None*, *progress_callback=None*, *timeout=None*)

Отправляет запрос удаленному узлу.

Параметры::

- **path** – Путь запроса.
- **response_callback** — опциональная функция или метод с сигнатурой *response_callback(request_receipt)*, вызываемый при получении ответа. Дополнительную информацию см. в примере запроса.
- **failed_callback** – опциональная функция или метод с сигнатурой *failed_callback(request_receipt)*, вызываемый при сбое запроса. См. пример запроса для получения дополнительной информации.
- **progress_callback** – Необязательная функция или метод с сигнатурой *progress_callback(request_receipt)*, вызываемый при прогрессе в получении ответа. Прогресс доступен в виде числа с плавающей запятой от 0 до 1.0 через свойство *request_receipt.progress*.
- **timeout** – Необязательный таймаут в секундах для запроса. Если указано *None,* он будет рассчитан на основе RTT канала.

Экземпляр RNS.RequestReceipt, если запрос был отправлен, или *False,* если нет.

Возвращает::

track_phy_stats(*track*)

Вы можете включить статистику физического уровня для каждого соединения. Если эта функция включена, и соединение работает через интерфейс, поддерживающий отчетность по статистике физического уровня, вы сможете получить такие статистические данные, как *RSSI*, *SNR* и физическое *качество* соединения.

Параметры::

track – Укажите, следует ли отслеживать статистику физического уровня. Значение должно быть `True` или `False`.

get_rssi()

Возвращает:: *Показатель уровня сигнала* на физическом уровне, если доступен; в противном случае — «Нет». Чтобы этот метод возвращал значение, на канале связи должна быть включена статистика физического уровня.

get_snr()

Возвращает: *Отношение сигнал/шум* физического уровня, если доступно, в противном случае `None`. Статистика физического уровня
на канале связи, чтобы этот метод возвращал значение.

get_q()

Возвращает:: *Качество канала* физического уровня, если доступно, в противном случае `None`. Для того, чтобы этот метод возвращал значение, на канале связи должна быть включена статистика физического уровня.

get_establishment_rate()

Возвращает:: Скорость передачи данных, с которой произошла процедура установления соединения, в битах в секунду.

get_mtu()

Возвращает:: MTU установленного соединения.

get_mdu()

Возвращает:: Размер MDU пакета для установленного соединения.

get_expected_rate()

Возвращает:: Ожидаемую скорость передачи данных пакета по установленному каналу.

get_mode()

Возвращает:: Режим установленного соединения.

get_age()

Возвращает:: Время в секундах с момента установления этого соединения.

no_inbound_for()

Возвращает:: Время в секундах с момента поступления последнего входящего пакета по каналу. Сюда входят пакеты keepalive.

no_outbound_for()

Возвращает:: Время в секундах с момента последнего исходящего пакета на канале. Сюда входят пакеты keepalive.

no_data_for()

Возвращает:: Время в секундах с момента прохождения данных полезной нагрузки по каналу. Сюда не входят пакеты keepalive.

inactive_for()

Возвращает:: Время в секундах с момента последней активности на канале. Сюда входят пакеты keepalive.

get_remote_identity()

Возвращает:: Идентификатор удаленного узла, если он известен. Вызов этого метода не запрашивает у удаленного инициатора раскрытие его идентификатора. Возвращает `None`, если инициатор соединения еще не вызвал самостоятельно метод `identify(identity)`.

teardown()

Закрывает канал и очищает ключи шифрования. Новые ключи будут использоваться, если будет установлен новый канал к тому же адресату.

get_channel()

Получает канал для этого соединения.

Возвращает:: Объект канала

set_link_closed_callback(*callback*)

Регистрирует функцию, которая будет вызвана при разрыве соединения.

Параметры:: **callback** – Функция или метод с сигнатурой *callback(link)*, который будет вызван.

set_packet_callback(*callback*)

Регистрирует функцию, которая будет вызываться при получении пакета по данному каналу.

Параметры:: **callback** – Функция или метод с сигнатурой *callback(message, packet)*, который будет вызван.

set_resource_callback(*callback*)

Регистрирует функцию, которая будет вызываться при объявлении ресурса по данному каналу. Если функция возвращает *True,* ресурс будет

принят. Если она возвращает *False,* он будет игнорирован.

Параметры:: **callback** – Функция или метод с сигнатурой *callback(resource)*, который будет вызван. Обратите внимание, что на данный момент доступна только базовая информация о ресурсе, такая как *get_transfer_size()*, *get_data_size()*, *get_parts()* и *is_compressed()*.

set_resource_started_callback(*callback*)

Регистрирует функцию, которая будет вызвана, когда ресурс начнёт передачу по данному каналу.

Параметры:: **callback** — вызываемая функция или метод с сигнатурой *callback(resource)*.

set_resource_concluded_callback(*callback*)

Регистрирует функцию, которая будет вызвана, когда передача ресурса по данному каналу завершится.

Параметры:: **callback** – Функция или метод с сигнатурой *callback(resource)*, который будет вызван.

set_remote_identified_callback(*callback*)

Регистрирует функцию, которая будет вызываться при идентификации инициирующего узла по данному каналу.

Параметры:: call**back** – Функция или метод с сигнатурой *callback(link, identity)*, которую необходимо вызвать.

set_resource_strategy(*resource_strategy*)

Устанавливает стратегию ресурсов для

канала.

Параметры:: **resource_strategy** – одно из значений `RNS.Link.ACCEPT_NONE`, `RNS.Link.ACCEPT_ALL` или `RNS.Link.ACCEPT_APP`. Если установлено значение `RNS.Link.ACCEPT_APP`, будет вызван *resource_callback* для определения, следует ли принимать ресурс или нет. *TypeError,* если стратегия ресурсов не поддерживается.

Возможные исключения::

Получение подтверждения запроса

класс RNS.RequestReceipt

Экземпляр этого класса возвращается методом `request` экземпляров экземпляров `RNS.Link`. Его ни в коем случае нельзя создавать вручную. Он предоставляет методы для проверки статуса, времени ответа и данных ответа по завершении запроса.

get_request_id()

Возвращает:: Идентификатор запроса в виде *байтов*.

get_status()

Возвращает:: Текущий статус запроса: `RNS.RequestReceipt.FAILED`, `RNS.RequestReceipt.SENT`, `RNS.RequestReceipt.DELIVERED` или `RNS.RequestReceipt.READY`.

get_progress()

Возвращает: Прогресс получения ответа в виде числа *с плавающей запятой* от 0,0 до 1,0.

get_response()

Возвращает:: Ответ в виде *байтов*, если он готов, в противном случае — *None*.

get_response_time()

Возвращает:: Время ответа на запрос в секундах.

concluded()

Возвращает:: True, если связанный запрос завершился (успешно или с ошибкой), в противном случае False.

Ресурс

класс RNS.Resource(*data*, *link*, *advertise=True*, *auto_compress=True*, *callback=None*, *progress_callback=None*, *timeout=None*)

Класс Resource позволяет передавать произвольные объемы данных по каналу связи. Он автоматически обрабатывает последовательность, сжатие, координацию и вычисление контрольной суммы.

Параметры::

- **data** – Передаваемые данные. Может быть *байтами* или *дескриптором* открытого *файла*. Подробности см. в примере Filetransfer.
- **link** – экземпляр RNS.Link, по которому будут передаваться данные.
- **advertise** – Необязательный параметр. Указывает, следует ли автоматически объявлять ресурс. Может принимать значения *True* или *False*.
- **auto_compress** – Необязательный параметр. Указывает, следует ли автоматически сжимать ресурс. Может принимать значения *True* или *False*.
- **callback** — необязательный *вызываемый* объект с сигнатурой *callback(resource)*. Вызывается по завершении
  завершится передача ресурса.
- **progress_callback** — необязательная *вызываемая* функция с сигнатурой *callback(resource)*. Вызывается при каждом обновлении прогресса передачи ресурса

advertise()

Объявить ресурс. Если другой конец канала примет объявление о ресурсе, начнется передача.

cancel()

Отменяет передачу ресурса.

get_progress()

Возвращает: Текущий прогресс передачи ресурса в виде числа *с плавающей запятой* от 0,0 до 1,0.

get_transfer_size()

Возвращает:: Количество байтов, необходимое для передачи ресурса.

get_data_size()

Возвращает:: Общий размер данных ресурса.

get_parts()

Возвращает:: Количество частей, в которых будет передаваться ресурс.

get_segments()

Возвращает:: Количество сегментов, на которые разделен ресурс.

get_hash()

Возвращает:: Хеш ресурса.

is_compressed()

Возвращает:: Указывает, сжат ли ресурс.

Канал

класс RNS.Channel.Channel

Обеспечивает надежную доставку сообщений по каналу связи.

`Channel` отличается от `Request` и `Resource` по нескольким важным параметрам:

Непрерывность
Сообщения можно отправлять или получать, пока канал открыт.

Двунаправленность
Сообщения могут отправляться в любом направлении по каналу; ни одна из сторон не является клиентом или сервером.

Ограничение по размеру

Сообщения должны быть закодированы в один пакет.

`Channel` аналогичен `Packet`, за исключением того, что он обеспечивает надёжную доставку (автоматические повторные попытки), а также структуру для обмена несколькими типами сообщений по каналу связи.

Канал не создается напрямую, а получается из канала с помощью с помощью `get_channel()` .

register_message_type(*message_class: Type[MessageBase]*)

Зарегистрировать класс сообщения для приёма по каналу. Классы

сообщений должны наследоваться от `MessageBase`.

Параметры:: **message_class** – класс для регистрации

add_message_handler(*callback: MessageCallbackType*)

Добавляет обработчик входящих сообщений. Обработчик имеет следующую сигнатуру:

```
(message: MessageBase) -> bool
```

Обработчики обрабатываются в том порядке, в котором они были добавлены. Если какой-либо обработчик возвращает True, обработка сообщения останавливается; обработчики, следующие за возвращающим обработчиком, вызываться не будут.

Параметры:: **callback** – Функция для вызова

remove_message_handler(*callback: MessageCallbackType*)

Удаление обработчика, добавленного с помощью `add_message_handler`.

Параметры:: **callback** – обработчик, который нужно удалить

is_ready_to_send() → bool

Проверяет, готов ли `канал` к отправке.

Возвращает:: True, если готов

send(*message: MessageBase*) → Envelope

Отправить сообщение. Если происходит попытка отправки сообщения, а `канал` не готов, генерируется исключение.

Параметры:: **сообщение** — экземпляр класса `MessageBase` подкласса

свойство mdu

Максимальная единица данных: количество байтов, доступных для использования сообщением при одной отправке. Это значение корректируется на основе `Link` MDU с учётом информации заголовка сообщения.

Возвращает:: количество доступных байтов

MessageBase

класс RNS.MessageBase

Базовый тип для любых сообщений, отправляемых или получаемых по каналу. Подклассы должны определять два абстрактных метода, а также переменную класса `MSGTYPE`.

MSGTYPE = *None*

Определяет уникальный идентификатор для класса сообщения.

- Должен быть уникальным среди всех классов, зарегистрированных в `канале`
- Должен быть меньше `0xf000`. Значения, большие или равные `0xf000` зарезервированы.

abstractmethod pack() → bytes

Создать и вернуть двоичное представление сообщения

Возвращает:: двоичное представление сообщения

абстрактный метод unpack(*raw: bytes*)

Заполнение сообщения из двоичного представления

Параметры:: **raw** – двоичное представление

Буфер

класс RNS.Buffer

Статические функции для создания буферизованных потоков, которые отправляют и принимают данные через `Channel` .

Эти функции используют `BufferedReader`, `BufferedWriter` и `BufferedRWPair` для добавления буферизации к `RawChannelReader` и `RawChannelWriter`.

static create_reader(*stream_id: int*, *channel: Channel*, *ready_callback: Callable[[int], None] | None = None*) → BufferedReader

Создает буферизованный считыватель, который считывает двоичные данные, передаваемые по каналу `Channel`, с опциональным обратным вызовом при появлении новых данных.

Сигнатура обратного вызова: `(ready_bytes: int) -> None`

Для получения дополнительной информации о функциях этого объекта, относящихся к считыванию, см. документацию Python по `BufferedReader`

Параметры::
- **stream_id** – локальный идентификатор потока, из которого будет осуществляться прием
- channel – канал для приема
- **ready_callback** – функция, вызываемая при появлении новых данных

Возвращает:: объект BufferedReader

static create_writer(*stream_id: int*, *channel: Channel*) → BufferedWriter Создает буферизованный записывающий объект, который записывает двоичные данные через `канал`.

Для получения дополнительной информации о функциях записи этого объекта см. документацию Python по `BufferedWriter`

Параметры::
- **stream_id** – идентификатор удаленного потока, в который будет отправляться
- **channel** – канал, по которому будет осуществляться отправка

Возвращаемые значения:: Объект BufferedWriter

static create_bidirectional_buffer(*receive_stream_id: int*, *send_stream_id: int*, *channel: Channel*, *ready_callback: Callable[[int], None] | None = None*) → BufferedRWPair

Создает буферизованную пару читателя/записывающего устройства, которая читает и записывает двоичные данные через `канал`, с опциональным обратным вызовом при появлении новых данных.

Сигнатура обратного вызова: `(ready_bytes: int) -> None`

Дополнительную информацию о функциях этого объекта, относящихся к конкретному считывающему устройству, см. в документации Python по `BufferedRWPair`

Параметры::
- **receive_stream_id** — идентификатор локального потока, на который будет осуществляться прием
- **send_stream_id** — идентификатор удаленного потока, на который будет отправляться
- **channel** – канал для отправки и получения данных
- **ready_callback** – функция, вызываемая при появлении новых данных

Возвращает:: объект BufferedRWPair

RawChannelReader

class RNS.RawChannelReader(*stream_id: int*, *channel: Channel*)

Реализация RawIOBase, которая принимает данные двоичного потока, отправляемые по каналу `Channel`.

Этот класс, как правило, не нужно создавать напрямую. Используйте `RNS.Buffer.create_reader()` , `RNS.Buffer.create_writer()` и `RNS.Buffer.create_bidirectional_buffer()` для создания буферизованных потоков с опциональными обратными вызовами.

Дополнительную информацию об API этого объекта см. в документации Python по `RawIOBase`.

__init__(*stream_id: int*, *channel: Channel*)

Создать считыватель необработанного канала.

Параметры::

- **stream_id** — идентификатор локального потока для приема на
- **channel** – объект `канала`, с которого будет осуществляться прием

add_ready_callback(*cb: Callable[[int], None]*)

Добавляет функцию, которая будет вызываться при появлении новых данных. Функция должна иметь сигнатуру `(ready_bytes: int) -> None`

Параметры:: **cb** – вызываемая функция

remove_ready_callback(*cb: Callable[[int], None]*)

Удаляет функцию, добавленную с помощью `RNS.RawChannelReader.add_ready_callback()`

Параметры:: **cb** – функция для удаления

RawChannelWriter

класс RNS.RawChannelWriter(*stream_id: int*, *channel: Channel*)

Реализация RawIOBase, которая принимает данные двоичного потока, отправляемые по каналу.

Этот класс, как правило, не нужно создавать напрямую. Используйте Функции `RNS.Buffer.create_reader()`, `RNS.Buffer.create_writer()` и `RNS.Buffer.create_bidirectional_buffer()` для создания буферизованных потоков с опциональными обратными вызовами.

Дополнительную информацию об API этого объекта см. в документации Python по `RawIOBase`.

__init__(*stream_id: int*, *channel: Channel*)

Создать записывающий модуль необработанного канала.

Параметры::

- **stream_id** – идентификатор удаленного потока, в который будет отправляться
- **channel** – объект `канала`, по которому будет осуществляться отправка

Транспорт

класс RNS.Transport

С помощью статических методов этого класса можно взаимодействовать с системой транспорта Reticulum.

PATHFINDER_M = *128*

Максимальное количество прыжков, на которое Reticulum будет транспортировать пакет.

static register_announce_handler(*handler*) Регистрирует

обработчик объявлений.

Параметры:: **handler** — должен быть объектом, имеющим атрибут *aspect_filter* и выполнимую функцию *received_announce(destination_hash, announced_identity, app_data)* или *received_announce(destination_hash, announced_identity, app_data, announce_packet_hash)* или *received_announce(destination_hash, announced_identity, app_data, announce_packet_hash, is_path_response)*. Опционально может иметь атрибут *receive_path_responses*, установленный в `True`, чтобы получать все ответы по пути в дополнение к текущим объявлениям. См. пример объявления для получения дополнительной информации.

static deregister_announce_handler(*handler*) Отменяет

регистрацию обработчика объявлений.

Параметры:: **handler** – обработчик объявлений, который необходимо отменить.

static has_path(*destination_hash*)

Параметры:: **destination_hash** – хеш назначения в виде *байтов*. *True,*
Возвращает:: если путь к назначению известен, в противном случае *False*.

static hops_to(*destination_hash*)

Параметры:: **destination_hash** – хеш назначения в виде *байтов*.
Возвращает:: Количество прыжков до указанного назначения или `RNS.Transport.PATHFINDER_M`, если количество прыжков неизвестно.

static next_hop(*destination_hash*)

Параметры:: **destination_hash** – хеш назначения в *байтах*. Хеш
Возвращает:: назначения в *байтах* для следующего прыжка до указанного назначения или *None*, если следующий прыжок неизвестен.

static next_hop_interface(*destination_hash*)

Параметры:: **destination_hash** — хеш адреса назначения в виде
Возвращает:: *байтов*. Интерфейс для следующего прыжка к указанному адресу назначения или *None*, если интерфейс неизвестен.

static await_path(*destination_hash*, *timeout=None*, *on_interface=None*)

Запрашивает у сети путь к пункту назначения и блокируется до тех пор, пока путь не станет доступен или не истечет время ожидания.

Параметры::
- dest**ination_hash** – хеш назначения в виде *байтов*
- **timeout** – Необязательный таймаут в секундах.
- **on_interface** – если указан, запрос на маршрут будет отправлен только по этому интерфейсу. При обычном использовании Reticulum обрабатывает это автоматически, и этот параметр не следует использовать.

Возвращает:: *True,* если путь к месту назначения найден, в противном случае *False*.

static request_path(*destination_hash*, *on_interface=None*, *tag=None*, *recursive=False*)

Запрашивает путь к пункту назначения у сети. Если другой доступный узел в сети знает путь, он объявит его.

Параметры::

- **destination_hash** – хеш пункта назначения в виде *байта*
- **on_interface** — если указан, запрос на путь будет отправлен только по этому интерфейсу. При обычном использовании Reticulum обрабатывает это автоматически, и этот параметр не следует использовать.

Индекс

_ | A | B | C | D | E | F | G | H | I | K | L | M | N | P | R | S | T | U | V

_

__init__() (метод RNS.RawChannelReader)
(метод RNS.RawChannelWriter)

A

accepts_links() (метод RNS.Destination)
add_message_handler() (метод RNS.Channel.Channel)
add_ready_callback() (метод RNS.RawChannelReader)
advertise() (метод RNS.Resource)
announce() (метод RNS.Destination)
ANNOUNCE_CAP (Атрибут RNS.Reticulum)
app_and_aspects_from_name() (статический метод RNS.Destination)
await_path() (статический метод RNS.Transport)

B

blackhole_sources() (статический метод RNS.Reticulum)
Buffer (класс в RNS)

C

cancel() (метод RNS.Resource)
Channel (класс в RNS.Channel)
clear_default_app_data() (метод RNS.Destination)
create_keys() (метод RNS.Destination)
create_reader() (статический метод RNS.Buffer)

concluded() (метод RNS.RequestReceipt)
create_bidirectional_buffer() (статический метод RNS.Buffer)
create_writer() (статический метод RNS.Buffer)
current_ratchet_id() (статический метод RNS.Identity)
CURVE (атрибут RNS.Identity)
(атрибут RNS.Link)

D

decrypt() (метод RNS.Destination)
(метод RNS.Identity)
deregister_announce_handler() (статический метод RNS.Transport)
deregister_request_handler() (метод RNS.Destination)
Destination (класс в RNS)
discovered_interfaces() (статический метод RNS.Reticulum)

E

enable_ratchets() (метод RNS.Destination)
encrypt() (метод RNS.Destination)
(метод RNS.Identity)
ENCRYPTED_MDU (атрибут RNS.Packet)
enforce_ratchets() (метод RNS.Destination)
ESTABLISHMENT_TIMEOUT_PER_HOP (атрибут RNS.Link)
expand_name() (статический метод RNS.Destination)

F

from_bytes() (статический метод RNS.Identity)
from_file() (статический метод RNS.Identity)
full_hash() (статический метод RNS.Identity)

G

get_age() (метод RNS.Link)
get_channel() (метод RNS.Link)
get_data_size() (метод RNS.Resource)
get_establishment_rate() (метод RNS.Link)
get_expected_rate() (метод RNS.Link)
get_hash() (метод RNS.Resource)
get_instance() (статический метод RNS.Reticulum)
get_mdu() (метод RNS.Link)
get_mode() (метод RNS.Link)
get_mtu() (метод RNS.Link)
get_parts() (метод RNS.Resource)
get_private_key() (метод RNS.Destination)
(метод RNS.Identity)
get_progress() (метод RNS.RequestReceipt)
(метод RNS.Resource)
get_public_key() (метод RNS.Identity)
get_q() (метод RNS.Link) (метод RNS.Packet)
get_random_hash() (статический метод RNS.Identity)
get_remote_identity() (метод RNS.Link)
get_request_id() (метод RNS.RequestReceipt)
get_response() (метод RNS.RequestReceipt)
get_response_time() (метод RNS.RequestReceipt)
get_rssi() (метод RNS.Link)
(метод RNS.Packet)
get_rtt() (метод RNS.PacketReceipt)
get_segments() (метод RNS.Resource)
get_snr() (метод RNS.Link) (метод RNS.Packet)
get_status() (метод RNS.PacketReceipt)
(метод RNS.RequestReceipt)
get_transfer_size() (Метод RNS.Resource)

H

has_path() (статический метод RNS.Transport)
hash() (статический метод RNS.Destination)
hash_from_name_and_identity() (статический метод RNS.Destination)
hops_to() (статический метод RNS.Transport)

I

identify() (метод RNS.Link)
Identity (класс в RNS)
inactive_for() (метод RNS.Link)
interface_discovery_sources() (статический метод RNS.Reticulum)
is_compressed() (метод RNS.Resource)
is_ready_to_send() (метод RNS.Channel.Channel)

K

KEEPALIVE (атрибут RNS.Link)
KEEPALIVE_TIMEOUT_FACTOR (атрибут RNS.Link)
KEYSIZE (атрибут RNS.Identity)

L

Link (класс в RNS)
LINK_MTU_DISCOVERY (атрибут RNS.Reticulum)
link_mtu_discovery() (статический метод RNS.Reticulum)
load_private_key() (метод RNS.Destination)
load_private_key() (метод RNS.Identity)
load_public_key() (метод RNS.Identity)

M

mdu (свойство RNS.Channel.Channel)
MessageBase (класс в RNS)
MINIMUM_BITRATE (атрибут RNS.Reticulum)
MSGTYPE (атрибут RNS.MessageBase)
MTU (атрибут RNS.Reticulum)

N

next_hop () (статический метод RNS.Transport)
next_hop_interface() (статический метод RNS.Transport)
no_data_for() (метод RNS.Link)
no_inbound_for() (метод RNS.Link)

no_outbound_for() (метод RNS.Link)

P

pack() (метод RNS.MessageBase)
Packet (класс в RNS)
PacketReceipt (класс в RNS)
PATHFINDER_M (атрибут RNS.Transport)
PLAIN_MDU (атрибут RNS.Packet)
publish_blackhole_enabled() (статический метод RNS.Reticulum)

R

RATCHET_COUNT (атрибут RNS.Destination)
RATCHET_EXPIRY (атрибут RNS.Identity)
RATCHET_INTERVAL (атрибут RNS.Destination)
RATCHETSIZE (атрибут RNS.Identity
) RawChannelReader (класс в RNS)
RawChannelWriter (класс в RNS)
recall() (статический метод RNS.Identity)
recall_app_data() (статический метод RNS.Identity)
register_announce_handler() (статический метод RNS.Transport)
register_message_type() (метод RNS.Channel.Channel)
register_request_handler() (метод RNS.Destination)
remote_management_enabled() (статический метод RNS.Reticulum)
remove_message_handler() (метод RNS.Channel.Channel)
remove_ready_callback() (метод RNS.RawChannelReader)
request() (метод RNS.Link)
request_path() (статический метод RNS.Transport)
RequestReceipt (класс в RNS)
required_discovery_value() (статический метод RNS.Reticulum)
resend() (метод RNS.Packet)
Resource (класс в RNS) Reticulum (класс в RNS)

S

send() (метод RNS.Channel.Channel)
(метод RNS.Packet)
set_default_app_data() (метод RNS.Destination)
set_delivery_callback() (метод RNS.PacketReceipt)
set_link_closed_callback() (метод RNS.Link)
set_link_established_callback() (метод RNS.Destination)
set_packet_callback() (метод RNS.Destination)
(метод RNS.Link)
set_proof_requested_callback() (метод RNS.Destination) set_proof_strategy() (метод RNS.Destination)
set_ratchet_interval() (метод RNS.Destination)
set_remote_identified_callback() (метод RNS.Link)
set_resource_callback() (метод RNS.Link)
set_resource_concluded_callback() (метод RNS.Link)
set_resource_started_callback() (метод RNS.Link) set_resource_strategy() (метод RNS.Link)
set_retained_ratchets() (метод RNS.Destination) set_timeout() (метод RNS.PacketReceipt)
set_timeout_callback() (метод RNS.PacketReceipt)
should_use_implicit_proof() (статический метод RNS.Reticulum)
sign() (метод RNS.Destination)
(метод RNS.Identity)
STALE_GRACE (атрибут RNS.Link)
STALE_TIME (атрибут RNS.Link)

T

teardown() (метод RNS.Link)
to_file() (метод RNS.Identity)
track_phy_stats() (метод RNS.Link)
Transport (класс в RNS) transport_enabled() (статический метод RNS.Reticulum)
truncated_hash() (статический метод RNS.Identity)
TRUNCATED_HASHLENGTH (атрибут RNS.Identity)

U

unpack() (метод RNS.MessageBase)

V

validate() (метод RNS.Identity)

Объяснение Reticulum для людей