

Сети Linux

Модели и приложения

Проект книги

Автор: Олег Цилюрик

Редакция **3.76**

10.01.2024г.



Оглавление

Введение (от автора).....	6
Что есть и чего нет в книге?.....	7
Соглашения и выделения, принятые в тексте.....	8
Код примеров и замеченные опечатки.....	8
Источники использованной информации.....	9
Сетевой стек, архитектура.....	10
Документы RFC.....	11
Принципы сетевого стека.....	11
Адреса MAC.....	14
Адреса IP.....	18
Сетевые интерфейсы.....	31
Порты транспортного уровня.....	49
Источники использованной информации.....	50
Протоколы и инструменты прикладного уровня.....	53
Сетевые инструменты.....	53
Сервисы сети и systemd.....	61
SSH.....	62
DHCP.....	72
Разрешение имён, служба DNS.....	75
Защищённость сети, файрвол.....	88
Суперсервер inetd.....	94
Прокси-сервера.....	104
Источники использованной информации.....	114
Программирование и инструменты.....	117
Общие принципы.....	117
Сетевые сокеты и операции.....	122
Управляющие операции.....	138
Классы обслуживания сервером.....	139
Независимые POSIX CLI сетевые тестеры.....	153
Расширенные операции ввода-вывода.....	155
Символьный сокет.....	163
Канальный уровень.....	164
Современные языки программирования.....	164
Источники использованной информации.....	179
Драйверы сетевых устройств в Linux (ядро).....	181
Введение в модули ядра.....	181
Структуры данных сетевого стека.....	190
Путь пакета сквозь стек протоколов.....	191
Драйверы: сетевой интерфейс.....	195
Протокол сетевого уровня.....	205
Протокол транспортного уровня.....	216
Источники использованной информации.....	218
За границами привычного Интернет.....	221
Обсуждение.....	221
Стек TOR.....	222
Mesh-сети.....	234
Альтернативные DNS.....	286
Доступ к хостам локальной сети.....	302
Источники использованной информации.....	306
Итоги и перспективы.....	309
Об авторе.....	310

Содержание

Введение (от автора).....	6
Что есть и чего нет в книге?.....	7
Соглашения и выделения, принятые в тексте.....	8
Код примеров и замеченные опечатки.....	8
Источники использованной информации.....	9
Сетевой стек, архитектура.....	10
Документы RFC.....	11
Принципы сетевого стека.....	11
Инкапсуляция данных.....	12
Сетевой порядок байт.....	13
Адреса MAC.....	14
Разрешение аппаратных адресов.....	16
Адреса IP.....	18
IPv4.....	18
Маски и подсети.....	18
Широковещательный и групповой обмен.....	19
Частные адреса.....	20
Частные IPv4 и NAT.....	21
IPv6.....	21
Префикс адреса.....	22
Сокращения записи IPv6.....	23
Локальные адреса.....	24
Синтаксис записи IPv6.....	25
Прогноз.....	27
Адресные переменные в программном коде.....	27
Разрешение адресов и имён.....	29
Разрешение имён в программном коде.....	30
Сетевые интерфейсы.....	31
Таблица маршрутизации.....	38
Управление роутингом.....	40
Алиасные IP адреса.....	42
Петлевой интерфейс.....	45
Переименование сетевого интерфейса.....	46
Альтернативные имена.....	48
Порты транспортного уровня.....	49
Источники использованной информации.....	50
Протоколы и инструменты прикладного уровня.....	53
Сетевые инструменты.....	53
Инструменты наблюдения и диагностики.....	54
Инструменты тестирования и управления.....	58
Сервисы сети и systemd.....	61
SSH.....	62
Передача файлов по SSH.....	65
SSH и mc.....	66
Графическая сессия в SSH.....	69
SSH в скриптах.....	71
DHCP.....	72
Разрешение имён, служба DNS.....	75
Локальный DNS резолвер bind.....	76
Кэширующий DNS dnsmasq.....	77
Кэширующий DNS средствами systemd.....	82
Оптимизация используемых серверов DNS.....	85
Защищённость сети, фаервол.....	88
Фаервол ufw.....	90
Суперсервер inetd.....	94
Сервер telnet.....	96
Сокетная активация в systemd.....	99

Прокси-сервера.....	104
Прокси сквозь SSH.....	110
Клиенты прокси.....	111
Кто и как использует прокси?.....	113
Источники использованной информации.....	114
Программирование и инструменты.....	117
Общие принципы.....	117
Клиент и сервер.....	117
Сети датаграммные и потоковые.....	117
Фазы соединения TCP.....	119
Адаптивные механизмы TCP.....	120
Сообщения прикладного уровня в TCP.....	121
Присоединённый UDP.....	122
Сетевые сокеты и операции.....	122
Обменные операции.....	127
Параметры сокета.....	130
Использование сокетного API.....	131
UDP клиент-сервер.....	132
TCP клиент-сервер.....	134
Взаимодействие запрос-ответ.....	137
Клиент-сервер в UNIX домене.....	138
Управляющие операции.....	138
Классы обслуживания сервером.....	139
Последовательный сервер.....	142
Параллельный сервер.....	142
Предварительное клонирование процесса.....	143
Создание потока по запросу.....	144
Пул потоков.....	145
Последовательный сервер с очередью обслуживания.....	146
Суперсервер и сокетная активация.....	147
Независимые POSIX CLI сетевые тестеры.....	153
Замечания о telnet.....	154
Расширенные операции ввода-вывода.....	155
Примеры реализации.....	156
Неблокируемый ввод-вывод.....	156
Замечания к примерам.....	156
Мультиплексирование ввода-вывода.....	158
Замечания к примерам.....	161
Ввод-вывод управляемый сигналом.....	161
Асинхронный ввод-вывод.....	162
Символьный сокет.....	163
Канальный уровень.....	164
Современные языки программирования.....	164
Python.....	165
Python и HTTP.....	168
Go.....	170
Rust.....	174
Эхо-сервер на shell.....	177
Источники использованной информации.....	179
Драйверы сетевых устройств в Linux (ядро).....	181
Введение в модули ядра.....	181
Сборка модуля.....	182
Точки входа и завершения.....	183
Вывод диагностики модуля.....	183
Загрузка модулей.....	184
Параметры загрузки модуля.....	185
Подсчёт ссылок использования.....	189
Структуры данных сетевого стека.....	190
Путь пакета сквозь стек протоколов.....	191
Приём: традиционный подход.....	191

Приём: высокоскоростной интерфейс.....	192
Передача пакетов.....	194
Драйверы: сетевой интерфейс.....	195
Статистики интерфейса.....	200
Виртуальный сетевой интерфейс.....	201
Протокол сетевого уровня.....	205
Ещё раз о виртуальном интерфейсе.....	210
Протокол транспортного уровня.....	216
Источники использованной информации.....	218
За границами привычного Интернет.....	221
Обсуждение.....	221
Стек TOR.....	222
TOR как прокси для любых служб.....	223
Торификация терминальной сессии.....	225
Ресурсы onion.....	228
Управляющий порт TOR.....	230
Утилита Nux.....	232
Mesh-сети.....	234
Сеть Yggdrasil.....	235
Установка.....	236
Настройка.....	245
Выбор внешних пиров для хоста.....	248
Майнинг IPv6 адресов.....	258
Yggdrasil в локальной сети.....	265
Короткие адреса.....	267
Работа в Yggdrasil даже без установки клиента.....	268
Скоростные показатели.....	271
А что в других ОС?.....	275
Ещё одна сеть: I2P.....	279
Запуск сети.....	279
Использование сети.....	283
Доступ к ресурсам извне.....	284
Альтернативные DNS.....	286
Регистрация домена.....	286
Блокчейн как регистратор домена.....	287
EmerCoin.....	288
OpenNIC.....	295
Alfis.....	296
Альтернативное разрешение имён в домене .ygg.....	302
Доступ к хостам локальной сети.....	302
Источники использованной информации.....	306
Итоги и перспективы.....	309
Об авторе.....	310

Введение (от автора)

*«Двадцать лет - это срок,
что длиннее и глубже могилы.
Для чего возвращаться туда,
где у всех помутится от страха?»
Александр Мирзаян, «Одиссея»*

Автор являлся непосредственным участником разработок, на протяжении свыше 40 лет, в проектах из области системотехники и программного обеспечения ... разного масштаба и разной целевой ориентации. Практически все из которых, в большей или меньшей мере, были связаны с сетевыми обменами. Текущий текст является обобщением практического опыта автора. Но, как и всякий практический опыт, обладает некоторой фрагментарностью — с акцентом на те аспекты, которые попадали в поле деятельности автора.

Первоначальный текст был подготовлен по приглашению, на заказ, как конспект учебного курса для программистов-разработчиков крупной международной софтверной фирмы Global Logic. В каковом качестве этот учебный курс и был прочитан один раз, в форме тренингов объёмом 30-40 академических часов, в 2012-2013 годах.

После чего, существенно дополняемый он использовался автором как «памятка для себя любимого», пусть и достаточно фрагментарная, ... конспект, справочник: отдельные вопросы сетевого программирования, которые, как мне казалось, нужно выделить. Кроме того, позволяющая и другим коллегам по профессии на начальном этапе работы с Linux как можно быстрее «въехать» в прямую программистскую деятельность затрагивающую сетевую область. В таком виде он (собственно, без ведома автора) и разошёлся достаточно широко по Интернет, примерно в 2014-2016 годах.

Настоящая ревизия (правильнее сказать: радикальная переделка всего текста дотла, оставив только его скелет) мотивирована следующими соображениями:

- Предыдущие 30 лет компьютерные сети развивались, и это не только в Linux, но и во всех других средах, базируясь исключительно (неявно) на протоколе IPv4. В несколько последних лет (с 2012 года, об этом будет ниже) произошло официальное введение в эксплуатацию в Интернет IPv6. И это радикально поменяло, по крайней мере в синтаксисе, привычное использование многочисленного сетевого инструментария.
- Расширение сферы использования IPv6 идёт, вопреки ожиданиям, гораздо медленнее прогнозируемому. Но это — будущее Интернет и, как следствие, сетевых технологий вообще. И в этом будущем IPv6 полностью вытеснит привычный IPv4. Поэтому именно с таким упором должен быть пересмотрен текст.
- В первоначальном варианте конспект планировался на аудиторию программистов, как учебный курс именно этого профессионального контингента. К настоящему времени создалось много разнообразных средств использования сетевых возможностей потребительского уровня, не требующих написания программного кода. Вплоть до формирования модной парадигмы и специализации DevOps — разработка крупных инфраструктурных проектов через администрирование («вместо дорогих и капризных программистов использовать более дешёвых администраторов»). Это потребовало сильно расширить охват в область пользовательского уровня.
- Изложение строится, по возможности, так, что те главы, где обсуждаются образцы программного кода, могут без особого ущерба для общей картины опущены всеми теми, кого не интересуют вопросы сетевой реализации в программном коде.

- Появляются, и ещё более будут появляться, системы передачи данных, использующие транспортные механизмы традиционного Интернет, но работающие **над** (сверх) его базовыми механизмами (это начиналось от прокси-серверов и VPN, и далее к ячеистым mesh-сетям и альтернативным DNS). Это нельзя оставить в стороне от рассмотрения, чему и отведена последняя часть книги.

Материалы данной книги (сам текст, сопутствующие ему примеры, файлы содержащие эти примеры), как и предмет её рассмотрения — задумывались и являются свободно распространяемыми. На них автором накладываются условия свободной лицензии (<http://legalfoto.ru/licenzii/>) **Creative Commons Attribution ShareAlike**: допускается копирование, коммерческое использование произведения, создание его производных при чётком указании источника, но при том единственном ограничении, что при использовании или переработке разрешается применять результат **только на условиях аналогичной лицензии**.

Многие из графических иллюстраций, рисунков в тексте заимствовано из книг У. Р. Стивенса (подробная библиография указывается после каждой части), на ранних этапах создавших наиболее полную и цельную картину сетевых технологий.

Что есть и чего нет в книге?

Конечно, дать исчерпывающую картину сетевого мира в Linux на таком ограниченном объёме (да и на любом обозримом) невозможно! Но задача ставилась не так: дать основу понятий архитектуры сети и наметить общую схему задач, решаемых в сетевом программировании, администрировании и использовании. В итоге, в тексте отчётливо сложились пять совершенно различных частей:

1. Архитектура и терминология сети.
2. Протоколы и программные инструменты работы с различными протоколами сети.
3. Сокетное программирование. Программирование приложений пользовательского адресного пространства.
4. Сетевые драйверы и протоколы. Программирование модулей ядра Linux — драйверов сетевых адаптеров и протоколов.
5. Новые, экспериментальные и экстравагантные проекты над Интернет. Тенденции и отдельные примеры.

Каждое из этих направлений требует, и вполне того заслуживает, для своего детального описания отдельной книги под тысячу страниц. Такие книги есть, и замечательные книги, некоторые из них приведены в рекомендуемой библиографии в конце каждой части, и именно к ним следует переходить для детального изучения каждого из этих предметов.

Задачей же данного текста ставилось дать цельную картину взаимодействия различных сетевых составляющих в Linux, назвать ключевые моменты, указать источники, целеуказание (стандарты, RFC, заголовочные файлы, ссылки в сети), где искать уточняющую информацию по таким ключевым моментам. Есть множество обстоятельнейших описаний по архитектуре и протоколам сети TCP/IP, по тому как сеть **устроена**, и по этим вопросам нужно именно к ним и обращаться. Меня это здесь не интересует, меня интересует то как это **использовать** для практиков. Поэтому первейшее внимание уделяется тем утилитам Linux, которые используются для работы с сетью, конкретному синтаксису их команд. И поэтому будет такое изобилие примеров выполнения этих команд.

Примечание (шёпотом): Детали функционирования сети TCP/IP, и, в конечном итоге, её потребительского порождения сети Интернет, описывается набором таких постоянно пополняемых документов как RFC (о чём будет рассказано вскоре). И они не зависят от программного окружения, операционной системы, внутри которой сеть функционирует — это с равным успехом может быть Linux, Free/Net BSD, Open Solaris, и разнообразные вариации Windows... Вынесение Linux в заголовок

книги — это, скорее, маркетинговый ход издателя, но все описываемые далее механизмы реализуемы во всех операционных системах, хотя и с внешней спецификой диктуемой используемой системой.

Соглашения и выделения, принятые в тексте

Для ясности чтения текста, он размечен шрифтами по функциональному назначению. Для выделения фрагментов текста по назначению используется разметка:

- Отдельные ключевые понятия и термины в тексте, на которые нужно обратить особое внимание, будут выделены **жирным шрифтом**.
- Тексты программных листингов, вывод в ответ на консольные команды пользователя размечен моноширинным шрифтом.
- Таким же моноширинным шрифтом (прямо в тексте) будут выделяться: имена команд, программ, файлов ... т.е. всех терминов, которые должны оставаться неизменяемыми, например: `/proc, mkdir, ./myprog, ...`
- Программным листингам предшествует имя файла (отдельной строкой), где находится этот код в архивах примеров, это имя файла выделяется **жирным курсивом с подчёркиванием**.
- Ввод пользователя в консольных командах (сами команды, или ответы пользователя в диалоге), кроме того, выделены **жирным моноширинным** шрифтом, чтобы отличать от ответного вывода системы, который набран просто моноширинным шрифтом.
- Текст, цитируемый из другого указанного источника, выделяется (для ограничения) *курсивным написанием*.

Код примеров и замеченные опечатки

Все протоколы выполнения команд и программные листинги, приводимые в качестве примеров, были опробованы и испытаны. Все примеры, обсуждаемые в тексте, предполагают воспроизведение и повторяемость результатов. Примеры программного кода сгруппированы по темам в архивы, поэтому всегда будет указываться имя архива (например, xxx) и имя файла (например, ууу.с); некоторые архивы могут содержать подкаталоги, тогда указывается и подкаталог для текущего примера. Большинство архивов (вида xxx) содержат одноимённые файлы вида xxx.hist — в них содержится скопированные с терминала результаты выполнения примера (протокол работы, журнал), показывающие как этот пример должен выполняться, в более сложных случаях здесь же могут содержаться команды, показывающие порядок компиляции и сборки примеров архива. Содержимое файлов *.hist накапливались в хронологии, годами работы над данным каталогом (даты там везде указываются), со всеми вариантами и ошибками, поэтому просмотр их разумно вести с конца.

Временами высказывалось недоумение: зачем такая, иногда излишняя, детализация примеров кодов и протоколов работы с ними? Мотивация здесь следующая: все примеры делались с тем, чтобы их простейшим образом можно было взять и воспроизвести, без выяснения недостающих деталей по документации и справочным руководствам. И взять такой работающий пример как отправную точку в развитии своего собственного проекта.

Конечно, и при самой тщательной выверке и вычитке, не исключены недосмотры и опечатки в таком объёмном тексте, могут проскочить мало внятные стилистические обороты и подобное. О замеченных таких дефектах я прошу сообщать по электронной почте o.tsiliuric@yandex.ru, и я был бы признателен за любые указанные недостатки рукописи, замеченные ошибки, или высказанные пожелания по её доработке.

Источники использованной информации

Я воздержался от создания более привычного и академического раздела в конце текста под названием «Библиография», исходя из нескольких соображений:

- Из за тематической разнородности частей текста общая библиография только вносила бы путаницу. Поэтому тематические указатели источников относящейся информации помещены отдельно, в конце **каждой части** книги.
- Помимо публикаций (книг, статей) здесь указываются электронные публикации в Интернет, по которым, обычно, доступно гораздо меньше информации (часто автор, или дата написания, или дата публикации ... оказываются неизвестными).
- Указанные ниже позиции никак не упорядочены, как это принято в настоящей библиографии. Это связано не столько с тем, что мне просто лень это делать, но ещё и с тем, что я просто не представляю как упорядочить смесь традиционных бумажных источников с электронными публикациями, когда всё это представлено единым списком.
- Приводятся источники не из соображения их фундаментальности, приводятся те источники которые оказались полезными при работе над рукописью и, главное, те которые позволяют уточнить в деталях то, что сказано в тексте.

Изданные «в бумаге» книги указаны с их полными выходными данными. Электронные публикации в Интернет даются со ссылками доступа к ним.

Сетевой стек, архитектура

Сетевой стек Linux предназначен для обслуживания самого различного коммуникационного оборудования (физический и канальный уровень: Ethernet, TokenRing, WiFi, последовательные линии передачи RS-232 и подобные, USB ...) и реализации над ним самых различных протоколов сетевого уровня (например, IPX/SPX от Novell). Но так сложилось со временем, что в подавляющем большинстве случаев в качестве сетевого уровня потребителя интересует протокол IP, а из физического оборудования наиболее частым случаем будет Ethernet (ну, и иногда WiFi). Далее мы будем рассматривать реализацию именно таких технологий, но не стоит упускать из виду, что точно такими же методами сетевой стек Linux обеспечивает и поддержку всех других используемых на практике протоколов (например, сетевой трафик можно направить через последовательный интерфейс RS-232 или USB).

Протокол IP является **маршрутизируемым**. В противовес этому, протоколы, используемые в локальной сети (LAN), например Ethernet, являются **немаршрутизируемыми** — это означает, что пакеты такого протокола не могут распространяться за пределы одного сегмента сети, к которому напрямую подключены сетевые интерфейсы. Также немаршрутизируемым, например, являлся первоначальный сетевой протокол Windows NetBEUI (NetBIOS Frame Protocol) разработки IBM, поддержка которого позже (с Windows 2003) была прекращена именно по причине немаршрутизируемости и заменена на NetBIOS over TCP/IP (NBT).

Таким образом, для того, чтобы сделать протокол LAN (протокол MAC уровня) маршрутизируемым в глобальной сети (WAN), его нужно «посадить сверху» на протокол сетевого уровня как наездника на коня. Функции такого сетевого протокола и выполняет IP. А поскольку в такой схеме 2 эти протокола имеют совершенно различный формат адреса (MAC адрес и IP адрес, соответственно), то возникает необходимость в протоколах взаимного разрешения таких адресов: ARP (Address Resolution Protocol) — разрешение IP в MAC, и RARP (Reverse Address Resolution Protocol) — разрешение MAC в IP.

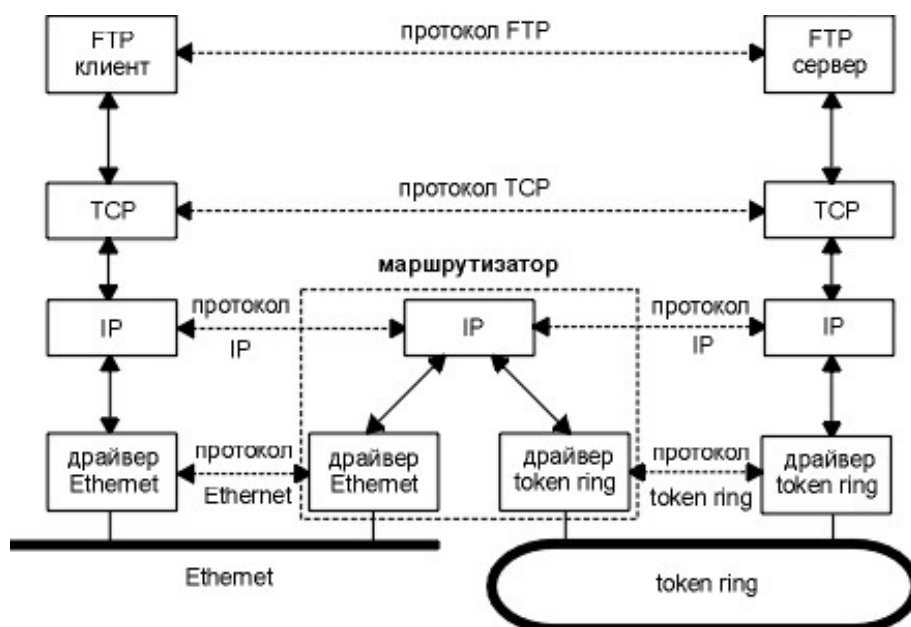


Рис. 1.1 Уровни сетевых протоколов

Документы RFC

Так сложилось исторически (с 1969 года и поныне), что все аспекты работы сети регламентируются и описываются не стандартами официальных международных комитетов и комиссий (как в других технических отраслях), а набором последовательно пронумерованных общедоступных описаний RFC (**R**equest **F**or **C**omments — заявка на отзывы, тема для обсуждения).

В настоящее время первичной публикацией документов RFC занимается IETF (Internet Engineering Task Force — Инженерный совет Интернета) под эгидой открытой организации ISOC (Internet Society — Общество Интернета). Правами на RFC обладает именно Общество Интернета.

Первые RFC относятся к 1969 году, на сегодня последовательные номера RFC вплотную приблизились к 10 000 (9506 на октябрь 2023 года). Некоторые RFC (уточняющие и исправляющие предыдущие) составляют несколько страниц. Другие, особенно описывающие детали сетевых протоколов, имеют объём в несколько сотен страниц каждый.

Каждый документ RFC имеет обозначение его категории, вот только некоторые из них: Proposed Standard (предложения для стандарта), Internet Standard (стандарт, высший статус), Experimental (экспериментальные предложения), Informational (чисто описательные материалы) ... Historic (устаревшие документы, в который статус переходят прочие, когда им в замену выходят новые RFC). Существует традиция выпуска первоапрельских выпусков RFC, например, RFC 1149 рассказывает о передаче пакетов IP с помощью почтовых голубей.

Документы RFC предоставляются для свободного скачивания и изучения. Ссылки на базы данных для доступа к RFC приведены в конце этой части, в информационных материалах.

Принципы сетевого стека

«Теория, мой друг, суха, но зеленеет жизни древо»
«Фауст» Иоганна Вольфганга фон Гёте
в переводе Бориса Пастернака

К началу 80-х годов международной организацией по стандартизации (ISO — **I**nternational **O**rganization for **S**tandardization) была разработана модель взаимодействия открытых систем (OSI — **O**pen **S**ystem **I**nterconnection). Средства взаимодействия в модели OSI делятся на семь уровней, каждый из которых призван решать свой круг задач:

1. Физический;
2. Канальный;
3. Сетевой;
4. Транспортный;
5. Сеансовый;
6. Представительный;
7. Прикладной.

Чаще всего именно эту модель привлекают к рассмотрению и её изучают студенты. Но нужно отчётливо представлять, что реальный TCP/IP стек Linux заметно отличается от модели OSI, как по границам разбиения уровней, так и по функциональной нагрузке каждого из слоёв. В сетевом стеке Linux (основной направленностью которого, всё-таки, при его широте охвата является протокол TCP/IP) выделяют 4 уровня (L, level):

1. Канальный: модуль ядра — драйвер устройства и сетевой интерфейс, ARP, RARP — называемый как L2;
2. Сетевой: IP, ICMP, IGMP — называемый как L3;
3. Транспортный: UDP, TCP, и более поздние SCTP и DCCP — называемый как L4;
4. Прикладной: HTTP, Telnet, FTP, e-mail (SMTP, POP3, IMAP4 ...) и все другие сервисы — эта огромная (по объёму и числу служб) часть реализуется уже не в ядре, а в адресном пространстве пользователя (его называют уровнем L7 ... но это только как дань уважения модели OSI);

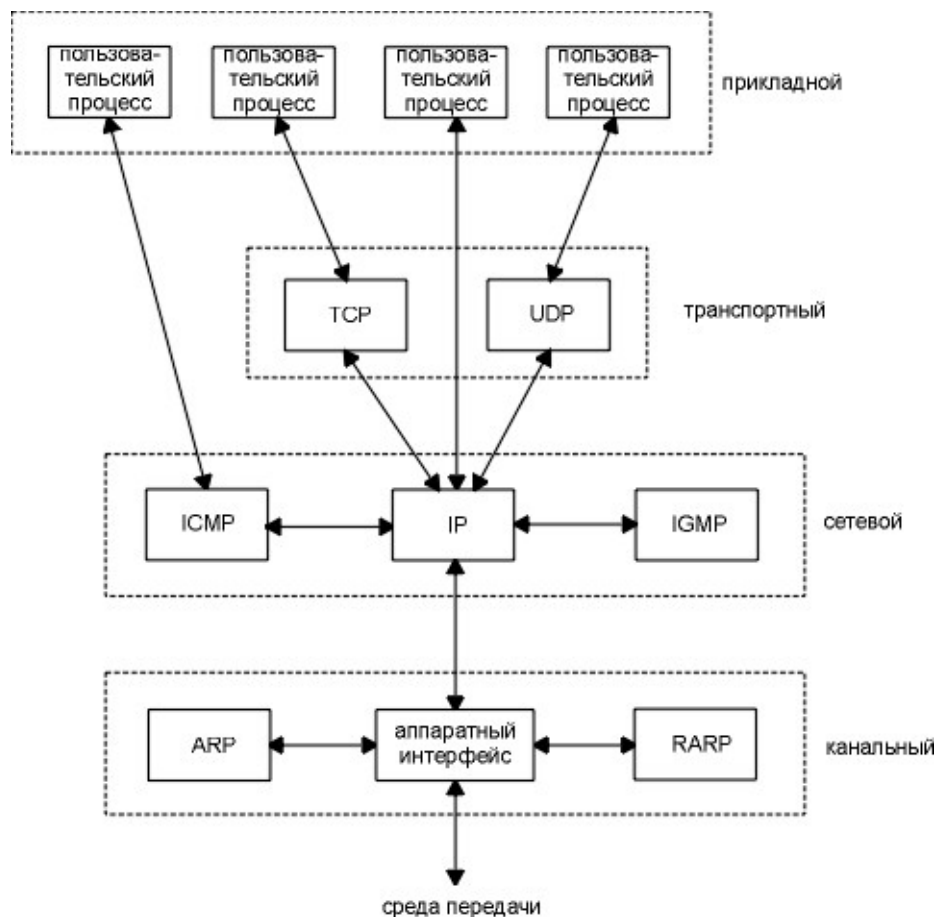


Рис. 1.2 Уровни сетевого стека Linux

Нижний, физический уровень OSI, не рассматривается в стеке протоколов Linux поскольку считается, что он реализуется аппаратно. Уровни канальный, сетевой и транспортный реализуются в коде ядра и модулей ядра (драйвера), в **привилегированном** режиме работы процессора (кольцо защиты 0 процессора x86). Инструменты прикладного уровня реализуются в коде **пространства пользователя**, в пользовательском режиме работы процессора (кольцо защиты 3 процессора x86).

Примечание: Сетевая передача производится в принципиально последовательных средах (бит за битом) распространения: проводные Ethernet или TokenRing, радиочастотные или инфракрасные среды в беспроводной передаче. Именно поэтому, на физическом и канальном уровне протоколов уместно и правильно рассматривать форматы и информацию в битовом представлении, которое, вообще то говоря, громоздко и непривычно с программистской точки зрения и протоколов вышележащих уровней. Поэтому даже относительно такой форматной информации физического представления мы зачастую будем условно пересчитывать битовые значения в байтовые, хотя никакого разграничения на байты в последовательно потоке передаваемых бит не происходит.

Инкапсуляция данных

Пакеты сетевых уровней TCP/IP вкладываются друг в друга, при этом заголовки пакетов каждого уровня несут исчерпывающую информацию своего уровня. Такую структура физически передаваемых пакетов называют **инкапсуляция** (вложение).

Такая же инкапсуляция на уровне структур данных, логическом — будет наблюдаться в структуре сокетных буферов в ядре Linux (и в модулях ядра, драйверах), что будет детально рассмотрено позже.

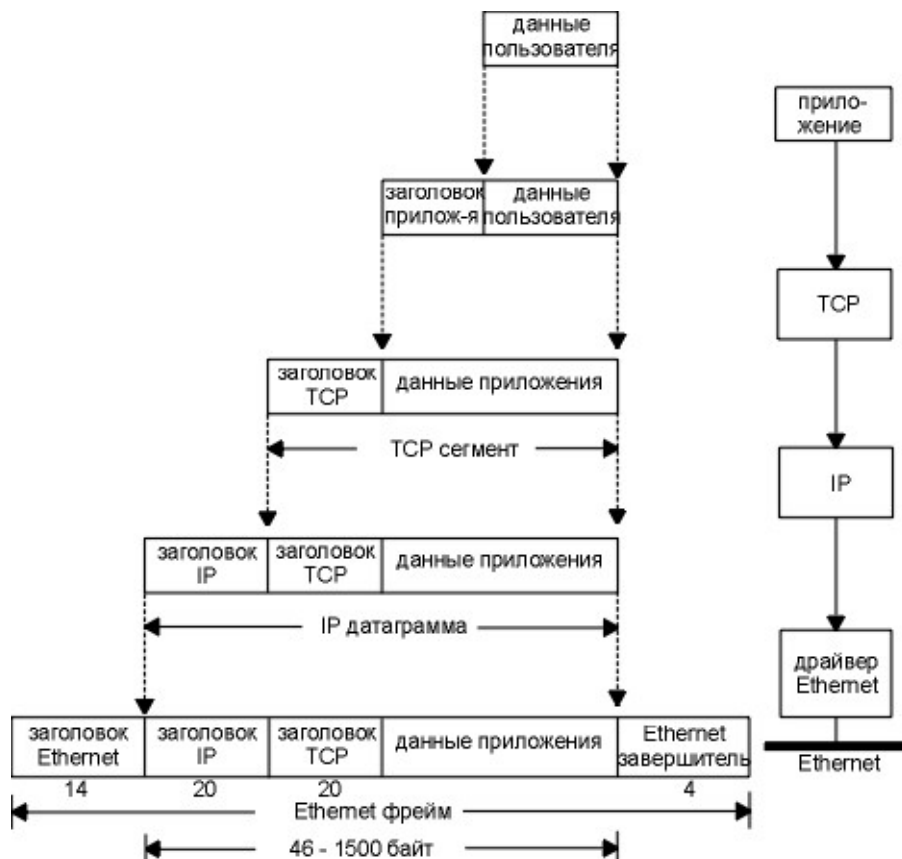


Рис. 1.3 Инкапсуляция данных

Сетевой порядок байт

Для 16-битового двоичного числа возможны 2 способа хранения его 2-х байт в памяти. Если первым (меньший адрес) идёт младший байт, то такой порядок называется прямым порядком байт (little-endian), а если первым идёт старший байт — обратным порядком байт (big-endian). Не существует единого стандарта, и на разных процессорных архитектурах используются разные порядки байт. При сетевом взаимодействии разных архитектур это вызовет проблемы. В самих сетевых протоколах используется сетевой порядок байт. Поэтому наша задача — всегда выполнить преобразование из порядка байт узла в сетевой и наоборот. Стандарт POSIX 1.g определяет для этого функции:

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Первые 2 функции возвращают значения, записанные в сетевом порядке байт, 2 последние — в порядке байт узлов.

Как мы увидим из кода, **любые** данные (константы) размером больше байта, передаваемые в сеть, перед передачей **должны** преобразовываться в сетевой порядок байт. На приёмной стороне, соответственно, должны восстанавливаться в порядок байт принимающего хоста. Это особенно хорошо видно на примере 16-битового значения **номера порта** транспортного уровня. Достаточно часто (почти всегда) мы не будем видеть подобных преобразований над **пользовательскими** данными, передаваемыми по сети. Но это только потому, и только до тех пор, пока эти пользовательские данные представляются текстовым

форматом, **поток**ом байт ASCII или UNICODE кодированием в его многобайтным представлением UTF-8. Но при обмене в UTF-16 или UTF-32 представлении для UNICODE (тип данных `wchar_t`) — у нас уже возникнут те же вопросы сетевого порядка байт. (Но это, UTF-16 или UTF-32 данные, в высшей степени не характерно для Linux, это чаще проблема в Windows, и она нас не интересует.)

Адреса MAC

Адреса MAC (Media Access Control) — это адреса непосредственно физического, аппаратного уровня. Это уникальный идентификатор, присваиваемый каждой единице сетевого оборудования (физическому адаптеру или логическому сетевому интерфейсу).

Большинство сетевых протоколов канального уровня (L1 в терминологии Linux) используют одно из 3-х пространств MAC-адресов, управляемых комитетом IEEE (или MAC-48, или EUI-48, или EUI-6 — первые 2 стандарта это 48-бит, последний — 64-бит). Адреса из пространства MAC-48 наиболее распространены — они используются в таких технологиях, как Ethernet, TokenRing, FDDI, WiMAX и других. Идентификаторы EUI-64 состоят из 64 бит и используются в FireWire, а также в IPv6 (в качестве младших 64 бит локального сетевого адреса назначаемого узлу).

Сетевой интерфейс — это, как мы вскоре увидим детально, **логическое** понятие. Сетевому интерфейсу может соответствовать реальный сетевой адаптер, а может и не соответствовать. Реальный сетевой адаптер — это уже **физическое** понятие, и он имеет адрес MAC уровня своего коммуникационного стандарта. Для Ethernet (IEEE 802.3) MAC адрес имеет длину 6 байт и записывается, например, так: 00:15:60:c4:ee:02.

Именно по MAC адресам обращаются друг к другу адаптеры Ethernet в локальной сети (LAN). В единой LAN все MAC адреса должны быть **уникальны**, нарушение этого условия приведёт к нарушению работы LAN.

Примечание: До определённого времени (в начальные, довольно давние времена) выдвигалось требование, чтобы все MAC адреса всех сетевых адаптеров, например Ethernet, вообще в мире были уникальными, централизованно распределялись определённым комитетом, и были зашиты в ROM адаптера. (И участки всего диапазона MAC адресов были разделены и приписаны к крупнейшим брендам-производителям аппаратного оборудования ... пример того что называют «недобросовестная конкуренция».) Но до настоящего времени эти соглашения не дожили, не прижились, изменены, и произвольный MAC адрес может быть программным путём записан через аппаратные регистры адаптера. (Тем не менее, у многих утилит уровня L1 вы сможете заметку привязать данные непосредственно к производителю — не стоит принимать это слишком всерьёз!)

В Linux есть инструмент для любой нужды (и появляются всё новые, именно в связи с тем что система открытая и свободная). Манипулировать с MAC-адресами вы можете, например, такой утилитой, которая есть в стандартных репозиториях дистрибутивов Linux (утилита стандарта GNU):

```
$ apt show macchanger
Package: macchanger
Version: 1.7.0-5.4
...
Description: Утилита для работы с MAC-адресами сетевых интерфейсов
GNU MAC Changer is an utility that makes the manipulation of MAC addresses
of network interfaces easier. ...
...
$ sudo apt install macchanger
...
$ macchanger --help
GNU MAC Changer
Usage: macchanger [options] device

-h, --help                Print this help
```

```

-V, --version          Print version and exit
-s, --show             Print the MAC address and exit
-e, --ending           Don't change the vendor bytes
-a, --another          Set random vendor MAC of the same kind
-A                    Set random vendor MAC of any kind
-p, --permanent       Reset to original, permanent hardware MAC
-r, --random           Set fully random MAC
-l, --list[=keyword]   Print known vendors
-b, --bia             Pretend to be a burned-in-address
-m, --mac=XX:XX:XX:XX:XX:XX
    --mac XX:XX:XX:XX:XX:XX Set the MAC XX:XX:XX:XX:XX:XX

```

Вот так выглядит индикация (диагностика) ... к примеру вот для такого аппаратного адаптера:

```

$ inxi -Nxxx
Network:
Device-1: Broadcom NetXtreme BCM5720 Gigabit Ethernet PCIe vendor: Dell
    driver: tg3 v: kernel pcie: speed: 5 GT/s lanes: 1 port: N/A
    bus-ID: 02:00.0 chip-ID: 14e4:165f class-ID: 0200
Device-2: Broadcom NetXtreme BCM5720 Gigabit Ethernet PCIe vendor: Dell
    driver: tg3 v: kernel pcie: speed: 5 GT/s lanes: 1 port: N/A
    bus-ID: 02:00.1 chip-ID: 14e4:165f class-ID: 0200
$ ip a s dev eno2
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 90:b1:1c:54:3a:47 brd ff:ff:ff:ff:ff:ff
    altname enp2s0f1
    inet 192.168.1.13/24 brd 192.168.1.255 scope global noprefixroute eno2
        valid_lft forever preferred_lft forever
    inet6 fe80::9bac:3791:1b79:7237/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
$ macchanger -s eno2
Current MAC: 90:b1:1c:54:3a:47 (Dell Inc.)
Permanent MAC: 90:b1:1c:54:3a:47 (Dell Inc.)
$ ipv6calc --showinfo -i -m 90:b1:1c:54:3a:47
no input type specified, try autodetection...found type: mac
no output type specified, try autodetection...found type: mac
EUI48=90:b1:1c:54:3a:47
EUI48_SCOPE=global
EUI48_TYPE=unicast
OUI="Dell Inc."
BUILTIN_DATABASE_INFO=IEEE:OUI/20170625
...

```

Это показано довольно подробно, потому что здесь уже выявляет несколько особенностей, которые нам потребуются в дальнейшем:

- Номер сетевого интерфейса (3 — утилита `ip`) никак не коррелирует с номером аппаратного устройства (2 — утилита `inxi`), или с индексом в имени сетевого интерфейса (`eno2`), это 3 разных значения, определяющиеся разными подсистемами Linux (номер сетевого интерфейса мы будем менять при его перенастройке, например, а индекс имени можем определять произвольно в коде драйвера ядра Linux ... и то и другое мы увидим вскоре).
- Некоторые утилиты будут пытаться привязать (Dell Inc.) аппаратное оборудование к его производителю по значениям полей MAC адреса (хотя и в этом случае реальной моделью производителя, что нужно для поиска соответствию драйверов, является: Broadcom NetXtreme BCM5720, который производил эти чипы для серверов компании Dell Inc.). Не стоит полагаться на такую информацию...

Но мы можем не только наблюдать значения адресов MAC, но и произвольно менять их (естественно это только от имени суперпользователя root):

```
$ sudo macchanger -m 00:1b:fc:82:ee:01 eno2
Current MAC: 90:b1:1c:54:3a:47 (Dell Inc.)
Permanent MAC: 90:b1:1c:54:3a:47 (Dell Inc.)
New MAC: 00:1b:fc:82:ee:01 (ASUSTek COMPUTER INC.)
$ ip6calc --showinfo -i -m 00:1b:fc:82:ee:01
...
OUI="ASUSTek COMPUTER INC."
BUILTIN_DATABASE_INFO=IEEE:OUI/20170625
...
```

MAC адаптера теперь поменялся! (И поменялся настолько радикально, что теперь утилиты видят производителем аппаратного адаптера не «Dell Inc.» а «ASUSTek COMPUTER INC.»). Но самое интересное (и ценное) здесь то, что MAC-адрес здесь поменялся по-горячему, "на ходу", без остановки Ethernet адаптера и его обмена, с тем же IP адресом:

```
$ ip a s dev eno2
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 00:1b:fc:82:ee:01 brd ff:ff:ff:ff:ff:ff permaddr 90:b1:1c:54:3a:47
    altnam enp2s0f1
    inet 192.168.1.13/24 brd 192.168.1.255 scope global noprefixroute eno2
        valid_lft forever preferred_lft forever
    inet6 fe80::9bac:3791:1b79:7237/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

И на другом хосте (на всех хостах!), на другом конце локальной сети, мы тут же, при таком горячем переключении, увидим отображение в кэш-таблице отображений IP адресов в MAC (за счёт рассылки широковещательных уведомлений протокола ARP) — последняя строка вывода:

```
$ arp -en
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.216            (incomplete)                                     enp3s0
192.168.1.5              (incomplete)                                     enp3s0
192.168.1.13            ether    00:1b:fc:82:ee:01    C                     enp3s0
192.168.1.3              ether    e4:c3:2a:46:ab:96    C                     enp3s0
192.168.1.14            ether    90:b1:1c:54:3a:46    C                     enp3s0
```

Разрешение аппаратных адресов

В масштабе сегмента локальной сети аппаратные адаптеры обмениваются пакетами только по MAC адресам. Протоколы канального уровня (L1) — не маршрутизируемые. Для выхода за пределы локальной сети пакеты MAC-уровня должны быть каким-то инкапсулированы в пакеты маршрутизируемого протокола. Чаще всего в этом качестве выступает протокол IP (когда-то это были такие популярные протоколы как IPC/SPX и другие ... но другие протоколы сетевого уровня мы не рассматриваем сейчас и здесь). Адреса уровня IP (вида 192.168.1.1) должны преобразовываться в адреса уровня MAC (вида 00:15:60:c4:ee:02) и, иногда, наоборот. Для преобразования MAC-адресов в адреса сетевого уровня и обратно применяются специальные протоколы (например, ARP и обратный ему RARP в сетях IPv4, и NDP в сетях IPv6).

Разрешение IP в MAC обеспечивается ядром Linux. Таблицы текущих разрешений IP в MAC хранятся в кэше (таблица ARP), некоторое время, которое известно как время жизни. Эта динамика связана с тем, что в локальной сети интерфейсы могут возникать и исчезать (компьютеры включаться и выключаться).

В традиционной LAN IPv4 для разрешений адресов IP в MAC предназначен отдельный протокол ARP (Address Resolution Protocol, RFC-826), а пары соответствий накапливаются хранятся в таблице ARP. Для работы с таблицей ARP-используется утилита:

```
$ arp -vn
```

Адрес	HW-тип	HW-адрес	Флаги	Маска	Интерфейс
192.168.1.11	ether	70:71:bc:a3:c5:c0	C		eno1
192.168.1.3	ether	e4:c3:2a:46:ab:96	C		eno1
192.168.1.12		(не заверше			eno1
192.168.1.17		(не заверше			eno1
192.168.1.10	ether	00:12:12:31:e9:d9	C		eno1
192.168.1.3	ether	e4:c3:2a:46:ab:96	C		eno2
169.254.169.254		(не заверше			eno1
192.168.1.241	ether	10:7b:44:47:a2:47	C		eno1
192.168.1.10	ether	00:12:12:31:e9:d9	C		eno2
192.168.1.138	ether	90:1b:0e:2b:fe:3a	C		eno1

Записей: 10 Пропущено: 0 Найдено: 10

Записи (строки) разрешений в таблице ARP валидны только на некоторое время жизни MAC-адреса в сети (после обнаружения их широковещанием). Этой утилитой вы можете не только наблюдать таблицу, но и добавлять новые записи (для переадресации и подмены, например), в том числе и перманентные, не исчезающие по времени, записи (опция - -he lp утилиты).

Но о соответствии MAC и IP адресов «знает» и утилита `ip` (уровня L2), из числа основных из тех, которыми мы будем пользоваться:

```
$ ip -4 neigh show
```

```
192.168.1.11 dev eno1 lladdr 70:71:bc:a3:c5:c0 STALE
192.168.1.3 dev eno1 lladdr e4:c3:2a:46:ab:96 REACHABLE
192.168.1.12 dev eno1 FAILED
192.168.1.17 dev eno1 FAILED
192.168.1.10 dev eno1 lladdr 00:12:12:31:e9:d9 STALE
192.168.1.3 dev eno2 lladdr e4:c3:2a:46:ab:96 STALE
169.254.169.254 dev eno1 FAILED
192.168.1.241 dev eno1 lladdr 10:7b:44:47:a2:47 STALE
192.168.1.10 dev eno2 lladdr 00:12:12:31:e9:d9 STALE
192.168.1.138 dev eno1 lladdr 90:1b:0e:2b:fe:3a STALE
```

Отдельного ARP протокола для IPv6 не существует. Для получения MAC адресов других компьютеров в IPv6 сетях используется протокол обнаружения соседей (Neighbor Discovery Protocol, NDP) - протокол из набора протоколов TCP/IP, используемый совместно с IPv6. NDP полностью обеспечивается средствами ICMPv6 (межсетевой протокол управляющих сообщений для межсетевого протокола версии 6). NDP протокол, в частности, использует пять различных типов пакетов ICMPv6.

Для просмотра и управления таблицей разрешений для IPv6 мы можем выполнять ту же команду, но вот с такой изменённой опцией (без опций, кстати, команда выводит данные сразу под оба протокола, но этого обычно слишком много):

```
$ ip -6 neigh show
```

```
fe80::762:c6bf:9eaa:93a9 dev eno1 lladdr 70:71:bc:a3:c5:c0 REACHABLE
fe80::522d:d0bd:b221:a526 dev eno2 lladdr 10:7b:44:47:a2:47 REACHABLE
fe80::921b:eff:fe2b:fe3a dev eno1 lladdr 90:1b:0e:2b:fe:3a STALE
fe80::522d:d0bd:b221:a526 dev eno1 lladdr 10:7b:44:47:a2:47 REACHABLE
```

Вот на этом, собственно, и всё, что нам достаточно знать про протоколы канального уровня (аппаратного, L1) для всего дальнейшего рассмотрения.

Адреса IP

Каждый **сетевой интерфейс** имеет свой IP адрес. Конечные точки коммуникаций в глобальной (маршрутизируемой) сети знают друг-друга **только** по их IP адресам. Все прочие сущности о которых мы будем говорить (имена хостов и интерфейсов, сетевые URL, маски и префиксы адреса ... да и всё другое) являются вторичными, предназначенными только для однозначного определения IP адресов.

Исторически протокол называемый IP проявился в 2-х последовательных ипостасях, версиях: IP версии 4 и IP версии 6. Первый из них (IPv4) определяется RFC начала 80-х годов, изменялся и уточнялся многими RFC последующих десятилетий, и является самым традиционным в использовании по сегодняшний день. Начало разработки второго (IPv6) положено серией RFC с 1996 года (начиная от RFC 1883). Но реальное внедрение его фактически начато только с середины 2012 года, и движется гораздо медленнее, чем предполагалось...

IPv4

Адреса IPv4 — это те адреса протокола TCP/IP, которыми мы традиционно привыкли пользоваться на протяжении 3-х десятилетий (с ранних 90-х). Это 4-байтная (32 бита) система IP адресов, называемая IP версии 4 или IPv4. 4 байта адреса IPv4 принято записывать как десятичные значения (0-255) разделённые символом точки ('.'), например: 192.168.1.5.

Такой размер IP адреса позволяет создать (всего) 4.3 миллиарда различающихся комбинаций (для ранних 90-х это было «чудовищно много», для 2020-х это «до смешного мало»¹).

Далее в этой главе будет рассматриваться только адресация IPv4.

Маски и подсети

Первоначально для разделения всего диапазона IP адресов на подсети было введено разделение на классы: выделялись большие диапазоны под категории сетей.

Класс	Диапазон	Назначение
A	0.0.0.0 - 127.255.255.255	Сверхбольшие подсети
B	128.0.0.0 - 191.255.255.255	Большие подсети
C	192.0.0.0 - 223.255.255.255	Малые подсети
D	224.0.0.0 - 239.255.255.255	Групповые операции
E	240.0.0.0 - 247.255.255.255	Резерв

Позже, с введением в обращение масок сети для разграничения подсетей (RFC 1219, бесклассовые сети, CIDR), классы утратили свой смысл (кроме групповых операций — multicast) и стало использоваться бесклассовое выделение подсетей. **Маска** в IPv4 является битовым образом, в котором ненулевые позиции определяют сеть (и подсеть), а нулевые позиции определяют те биты IP адреса, которые являются значащими при идентификации индивидуального адреса хоста в пределах подсети.

¹ 25 ноября 2019 года были распределены последние свободные IPv4-адреса в Европе, странах бывшего СССР и на Ближнем Востоке. Теперь получить IPv4-адрес можно будет, только если его освободит текущий владелец — например, закроется компания или какая-либо сеть освободит ненужный ей адресный ресурс.



Рис. 1.4 Маски и подсети

Таким образом маска полностью и однозначно определяет **предельное число** хостов, которое может работать в этой подсети. Младший адрес этого диапазона считается адресом самой подсети (например для записи в таблицы маршрутизации). Старший адрес этого же диапазона считается широковещательным (broadcast) адресом этой же подсети. Все остальные адреса этого диапазона (минус 2 крайних) являются индивидуальными адресами хостов этой подсети. Например, следующие значения маски позволяют организовать подсети следующих размерностей:

255.255.255.255 => ... 1111 1110 : 1 изолированный хост

255.255.255.248 => ... 1111 1000 : 8-2 = 6 хостов

255.255.255.240 => ... 1111 0000 : 16-2 = 14 хостов

255.255.255.224 => ... 1111 0000 : 32-2 = 30 хостов

255.255.255.248 => ... 1110 0000 : 64-2 = 62 хоста

Примечание: Первоначально предполагалось использовать только маски (префиксы) длиной 8, 16 и 24 бит (например, 192.168.1.5/24). Это в точности и соответствует классам сетей А, В и С, то есть не меняло принципиально классификацию. Но практика показала, что максимальный размер сетей C(/24), равный 254, для многих компаний и бизнес-структур мал, а B(/16) размером 65 534 — неоправданно велик. Таким образом практика разрушила предположения, и на сегодня используются маски **любой** длины.

IP адрес совместно с маской полностью определяют конфигурацию подсети и хоста. Для записи пары адреса и маски применяются 2 различные, но эквивалентные формы записи, например (это эквивалентно):

192.168.1.5 : 255.255.255.240 или 192.168.1.5/28

Здесь во 2-й форме записи 28 (называемое префиксом) — это число ведущих единиц в записи маски (число ведущих бит, относящихся к адресу сети в записи 32-бит адреса, это IPv4).

Примечание: Как сам термин «префикс», он довольно редко фигурирует в обсуждениях IPv4. Но в IPv6, где отсутствует понятие маски, он становится основополагающим!

Примечание: Первоначально, и это на практике работает и сейчас, допускались «рванные» маски, когда единичные биты могли чередоваться с нулевыми произвольным порядком, не составляя единого ведущего блока единичных бит. Это работает, но это крайне сложно для интерпретации и толкования поведения хостов сети. Поэтому позже было принято решение о использовании только «цельных» масок с ведущим блоком сплошных единичных бит. При таком ограничении, понятно, необходимость в длинной и сложного формата маске исчезает, и вместо этого может использоваться просто короткое число префикс — это же мы и видим в IPv6.

Широковещательный и групповой обмен

Вообще то, существуют три типа IP адресов: персональный (unicast), широковещательный (broadcast) и групповой (multicast). Широковещательные и групповые запросы применимы только к UDP и SCTP, подобные типы запросов позволяют приложению разослать одно сообщение нескольким получателям одновременно. TCP — протокол, ориентированный на соединение, с его помощью устанавливается соединение между двумя сетевыми интерфейсами (по указанному их IP адресу) с использованием одного адресата на каждом хосте (который идентифицируется по номеру порта).

Широковещательный адрес подсети (subnet-directed broadcast address) имеет идентификатор хоста, определяемый маской, установленный во все единицы (самый старший адрес диапазона адресов подсети, а самый младший адрес этого диапазона есть адресом самой подсети):

```
$ ifconfig enp2s14
enp2s14: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.5 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::215:60ff:fec4:ee02 prefixlen 64 scopeid 0x20<link>
    ether 00:15:60:c4:ee:02 txqueuelen 1000 (Ethernet)
    RX packets 280605 bytes 107024700 (102.0 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 326133 bytes 60602689 (57.7 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 16
```

Групповой адрес (multicast group address) состоит из четырех старших бит IP, установленных в 1110, и идентификатора группы (28 младших бит, определяющих конкретную группу). В десятичной записи групповые адреса находятся в диапазоне от 224.0.0.0 до 239.255.255.255. Групповая адресация позволяет направить датаграмму только выделенному подмножеству узлов, включённых в группу (но с этим вы будете встречаться редко).

Для работы с группой прежде нужно включить (или исключить) в группу каждый хост, который должен участвовать в этой групповой рассылке. Для этого существует отдельный протокол управления группами IGMP (Internet Group Management Protocol, RFC 1112), который используется хостами и маршрутизаторами, для того чтобы организовывать групповую рассылку сообщений. Он позволяет всем узлам физической сети знать, какие хосты в настоящее время объединены в группы и к каким группам они принадлежат. Эта информация необходима для групповых маршрутизаторов, именно так они узнают, какие групповые датаграммы необходимо перенаправлять и на какие интерфейсы.

Частные адреса

В RFC 1918 (1996г.) были специфицированы 3 блока адресов (первоначально по одному ка каждый из классов A, B, C) — которые не будут распределяться пользователям (организацией IANA) и которые никогда не маршрутизируются шлюзами IP. Это адреса из диапазонов:

10.0.0.0 — 10.255.255.255 (префикс 10/8)

172.16.0.0 — 172.31.255.255 (префикс 172.16/12)

192.168.0.0 — 192.168.255.255 (префикс 192.168/16)

Эти адреса получили название частные или приватные, и предназначаются они для организации локальных сетей. Адреса из **любого** из этих диапазонов можно выбирать для своей LAN (для организации Intranet).

Но их проблема в том, что такие хосты из LAN не смогут обращаться к внешним (за пределами LAN) ресурсам: шлюз по умолчанию из LAN не пропустит IP пакеты с таким исходящим адресом (а если даже и пропустит, то их порежет ближайший следующий маршрутизатор по трассе).

Для решения этой новой возникшей проблемы предложены несколько способов, основные из которых два:

1. На сетевом уровне — преобразование сетевых адресов (NAT — Network Address Translation). При этом хост IP шлюза **подменяет** в маршрутизируемых пакетах IP **адрес оправителя**, подставляя IP собственного интерфейса с глобальным IP. Получив на этот адрес ответ, шлюз подменяет в этом пакете **адрес получателя**, и маршрутизирует его в интерфейс LAN получателю. Самыми известными проектами реализации в разное время были последовательно: ipfw, ipfilter, iptables. На сегодня самым частым является:

```
$ which iptables
```

/usr/sbin/iptables

Но и предыдущие реализации (ipfw и ipfilter) находят до сих пор иногда применение в малых и встраиваемых архитектурах (например, в аппаратных роутерах).

2. На уровне протоколов прикладного уровня — прокси (проху, посредники). При этом программа прокси-сервера **ретранслирует** запрос хоста с частным IP **от своего имени** в интерфейс с глобальным IP. Получив ответ, прокси-сервер перенаправляет ответ получателю (через интерфейс LAN). Известнейшим примером такой техники является HTTPS-прокси и реализующий сервер squid.

Частные IPv4 и NAT

32-бит локальные адреса IPv4 в сети Интернет, раздаваемые организацией IANA, исчерпались бы количественно, наверное, ещё лет 10-15 назад, если бы не было принято расширение стандарта и не введены 3 диапазона показанных выше частных IPv4 адресов. После этого вы (или любая организация, компания, фирма...) в своей LAN можете произвольно, по собственному усмотрению, администрировать по IP хосты. Такое решение снижает общую потребность в индивидуальных IPv4 адресах (в просторечии называемых как «белые») в сотни и тысячи раз.

Но если эти частные адреса не маршрутизируются шлюзами IP в сеть WAN, то нужно искать способ вывода хостов LAN с такими адресами в глобальную сеть. И такой способ был предложен, и это механизм NAT (Network Address Translation — «преобразование сетевых адресов», ещё употребляются термины Masquerading, Network Masquerading и Native Address Translation). Способ NAT состоит в преобразовании IPv4 адресов транзитных, осуществляемом на шлюзе в WAN.

Очень коротко, «на пальцах»:

- заголовок IP пакета из локальной сети (на сетевом уровне, L2), с частным IPv4, преобразуется в заголовок пакета с «белым» адресом IP интерфейса шлюза, глядящим в WAN...
- и подменяется номер порта (чтобы различать ответные пакеты, адресованные разным локальным компьютерам)...
- пакет отправляется в WAN (forwarding между интерфейсами)
- заголовок приходящего в ответ (от сервера) на отправленный пакет из WAN модифицируется: «белый» адрес получателя подменяется на исходный частный IPv4 отправителя...
- и пересылается в LAN реальному получателю.

На сегодня (ещё с ядра 2.4) это механизм в Linux реализуется ядром (это к вопросу о эффективности), механизмом netfilter и соответствующей ему утилитой пространства пользователя iptables — для управления работой (правилами) этого ядерного механизма. (В Linux исторически, в разное время, прошли для этих целей разные механизмы, известные под названиями проектов ipchains и ipfwadm).

Примечание: Как альтернатива этому механизму сетевого уровня L2 (в ядре) для выхода в WAN локальных хостов из LAN может, в некоторых случаях, использоваться механизм транспортного уровня L3 (пользовательского пространства), работающий через прокси-сервер на шлюзе в WAN. Прокси-сервер может использовать транспортный протокол HTTP/HTTPS, или TCP (прокси SOCKS4/SOCKS5). О прокси-серверах мы ещё поговорим коротко позже.

IPv6

6 июня 2012 года крупнейшие провайдеры одновременно перешли к использованию **параллельно** с IPv4 16-ти байтовых (128 бит) IP адресов. Эту систему адресации, разрабатываемую ещё с 1992 года, называют IP версии 6 или IPv6. Адреса IPv6 записываются как последовательности 4-х шестнадцатеричных значений (0000-ffff — называемых хекстеты), разделённых ':', например: fe80::215:60ff:fec4:ee02.

Протокол IPv6 разработан как преемник протокола IPv4. Основные преимущества IPv6 над IPv4 заключаются в увеличенном адресном пространстве (IPv4 имел 32 бита, что равнялось 2^{32} адресам, а IPv6 имел 128 бит, что равнялось 2^{128} адресам).

Адрес протокола IPv6 состоит из 128 бит, то есть, он в 4 раза длиннее 32-битного IPv4 адреса. Подобно IPv4, в этом адресе можно выделить две части: сеть и хост. То есть, не все биты в адресе имеют одинаковое значение. Часть битов слева (сколько именно зависит от префикса) обозначают сеть, остальные биты справа – идентифицируют устройство внутри сети. Часть, ответственная за хранение информации о хосте называется идентификатор интерфейса (interface id).

Здесь сразу же зафиксируем важнейшее для пользователя отличие, которое несёт ему IPv6: здесь **нет отличия** между «глобальными» хостами подключёнными в Интернет, и «локальными» находящимися в LAN. Здесь любой хост равнозначно доступен извне, если он не закрыт файерволом каким-то образом.

В IPv6 определены 3 типа адресов, собственно, как и в IPv4 ... но здесь они «приписаны» к диапазонам адресов (при общем избытии адресов):

- unicast: Идентификатор одиночного интерфейса. Пакет, посланный по уникастному адресу, доставляется интерфейсу, указанному в адресе.
- anycast: Идентификатор набора интерфейсов (принадлежащих разным узлам). Пакет, посланный по уникастному адресу, доставляется одному из интерфейсов, указанному в адресе (ближайший, в соответствии с мерой, определенной протоколом маршрутизации).
- multicast: Идентификатор набора интерфейсов (обычно принадлежащих разным узлам). Пакет, посланный по мультикастинг-адресу, доставляется всем интерфейсам, заданным этим адресом.

Префикс адреса

В отличие от предыдущей версии протокола, в IPv6 не применяются маски подсети, так как они получились бы очень длинными, вместо этого используется префикс, который записывается так же через слэш после адреса. Например, префикс /64 означает, что из 128 бит, первые 64 – это сеть, а оставшаяся часть (в данном случае вторые 64) – это хост. Префикс описывает, сколько бит в адресе используется под хранение информации о сети.

Примечание: Опыт использования и трансформации представлений в мире IPv4 приводит к пониманию того, что при непрерывной маске (начиная от левого края и до...) нет необходимости в самом значении этой маски, достаточно просто сказать её длину.

Сам адрес записывают не в десятичном, а в шестнадцатеричном виде – так короче. Адрес разбивается на группы по 16 бит (хекстеты) и каждая такая группа представляется четырьмя шестнадцатеричными цифрами. Хекстеты отделяются друг от друга знаком двоеточия. Таким образом, полный адрес всегда состоит из 8 хекстетов ([8 хекстетов]*[16 бит в хекстете]=[128 бит] – общая длина адреса).

Стандарт IPv6 регламентируется существенно сложнее, подробнее и многообразнее чем IPv4. На основании префиксов выделяются классы использования. Вот некоторые диапазоны адресов, которые мы будем использовать дальше в обсуждениях:

Все адреса в диапазоне fe80::/10, называются link-local (о них речь впереди), Любой адрес из этого диапазона предназначен для автоконфигурирования ОС и пакеты, отправленные с такого адреса не пропустит ни один маршрутизатор, ограничивая их распространение в пределах сегмента локальной сети. Протокол IPv6 спроектирован так, что интерфейс обязан иметь link-local адрес, даже если у него есть другой IPv6 для других целей.

Для уникальных юникаст адресов в Интернет комитет IANA на настоящее время выделяет IPv6 адреса из класса 2000::/3.

Адреса ff00::/8 — это адреса мультикаст. По такому адресу мы можем получить отклик всех интерфейсов работающих в локальной сети данного интерфейса:

```
$ ping6 -c2 ff02::1%eno1
PING ff02::1%eno1(ff02::1%eno1) 56 data bytes
64 bytes from fe80::13f5:9fe2:6393:bf4a%eno1: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from fe80::9bac:3791:1b79:7237%eno1: icmp_seq=1 ttl=64 time=0.405 ms
64 bytes from fe80::522d:d0bd:b221:a526%eno1: icmp_seq=1 ttl=64 time=4.50 ms
64 bytes from fe80::921b:eff:fe2b:fe3a%eno1: icmp_seq=1 ttl=64 time=4.53 ms
64 bytes from fe80::b14:e016:f362:d141%eno1: icmp_seq=1 ttl=64 time=4.53 ms
64 bytes from fe80::762:c6bf:9eaa:93a9%eno1: icmp_seq=1 ttl=64 time=4.57 ms
64 bytes from fe80::212:12ff:fe31:e9d9%eno1: icmp_seq=1 ttl=64 time=4.59 ms
64 bytes from fe80::1%eno1: icmp_seq=1 ttl=64 time=4.60 ms
64 bytes from fe80::e6c3:2aff:fe46:ab96%eno1: icmp_seq=1 ttl=64 time=4.60 ms
64 bytes from fe80::2079:262e:4120:b9ff%eno1: icmp_seq=1 ttl=64 time=14.4 ms
64 bytes from fe80::2079:262e:4120:b9ff%eno1: icmp_seq=1 ttl=64 time=14.4 ms
64 bytes from fe80::2079:262e:4120:b9ff%eno1: icmp_seq=1 ttl=64 time=14.4 ms
64 bytes from fe80::13f5:9fe2:6393:bf4a%eno1: icmp_seq=2 ttl=64 time=0.086 ms

--- ff02::1%eno1 ping statistics ---
2 packets transmitted, 2 received, +11 duplicates, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.086/5.827/14.421/5.023 ms
```

С мультикаст IPv6 мы позже плотно познакомимся при рассмотрении пиринговой сети Yggdrasil, где он играет ключевую роль.

Сокращения записи IPv6

Пример адреса: 2001:0DB8:AA10:0001:0000:0000:0000:00FB. С таким длинным адресом работать достаточно неудобно, поэтому для IPv6 применяют сокращённую по специальным правилам запись.

Для того чтобы сократить данный адрес надо **последовательно** применить два правила.

Правило №1:

В каждом хекстете (группе из 4-х цифр) ведущие нули удаляются. Например, во втором хекстете 0DB0 заменяется на DB0. То есть ноль слева удаляется, ноль справа мы не трогаем. Если хекстет состоит из одних нулей, то он заменяется на один ноль. Таким образом адрес 2001:0DB0:0000:123A:0000:0000:0000:0030 преобразуется в 2001:DB0:0:123A:0:0:0:30. А, например, адрес loopback 0000:0000:0000:0000:0000:0000:0000:0001 таким правилом заменяется на 0:0:0:0:0:0:0:1.

Правило №2:

Это правило применяется **только после** первого. В адресе выбирается одна самая длинная группа, состоящая из полностью нулевых хекстетов, то есть самая длинная последовательность «:0:0:0:» и заменяется на два двоеточия «::». Эту замену можно произвести **только один раз** и только **с самой длинной** последовательностью (так как, если бы мы, например, сделали такую замену в двух местах адреса, то потом нельзя было бы восстановить, сколько именно хекстетов мы заменили в первом и во втором случае). Важный момент: нельзя заменять одну группу из :0: на ::, правило два применимо только если есть более одной нулевой группы.

Для примера возьмём адрес из предыдущей замены 2001:DB0:0:123A:0:0:0:30. Самая длинная последовательность из полностью пустых хекстетов — это «:0:0:0:», она начинается сразу после хекстета «123A». Есть ещё последовательность из одного пустого хекстета (между «DB0» и «123A»), но эта — длиннее, так что заменять будем её. Адрес станет совсем небольшим: 2001:DB0:0:123A::30. Это, конечно, длиннее IPv4 адреса, но гораздо короче исходного.

Обратное восстановление исходного адреса по сокращённой записи (должно производиться редко в представлении для человека, но выполняется каждый раз в сетевой подсистеме при обращении по адресу) достаточно тривиально, если мы уже умеем сокращать адреса.

Сначала надо посчитать, сколько хекстетов в адресе осталось. В нашем случае, в адресе 2001:DB0:0:123A::30 осталось 5 хекстетов. Мы знаем, что адрес должен состоять из **восьми** хекстетов — значит вместо «::» возвращаем три недостающих нулевых, получаем 2001:DB0:0:123A:0:0:0:30. Теперь в каждой группе, где меньше четырёх цифр дописываем слева такое количество нулей, чтобы в этой группе стало четыре цифры. В результате получим исходный адрес 2001:0DB0:0000:123A:0000:0000:0000:0030.

Несколько примеров сокращения адресов, сокращать будем по правилам в два этапа:

```
FF80:0000:0000:0000:0123:1234:ABCD:EF12 -> FF80:0:0:0:123:1234:ABCD:EF12 ->
FF80::123:1234:ABCD:EF12
FF02:0000:0000:0000:0000:0001:FF00:0300 -> FF02:0:0:0:0:1:FF00:300 -> FF02::1:FF00:300
2001:0DB8:0000:1111:0000:0000:0000:0200 -> 2001:DB8:0:1111:0:0:0:200 ->
2001:DB8:0:1111::200
0000:0000:0000:0000:0000:0000:0000:0001 -> 0:0:0:0:0:0:0:1 -> ::1
0000:0000:0000:0000:0000:0000:0000:0000 -> 0:0:0:0:0:0:0:0 -> ::
```

Предпоследний из показанных IPv7 адресов ::1 — это петлевой интерфейс, соответствующий в IPv4 привычному 127.0.0.1

Локальные адреса

IPv6 адреса различаются по области действия. Глобальный адрес IPv6 действует в интернете. Глобальные адреса должны быть уникальными в Интернете, поэтому адреса IPv6 распределяются организации IANA.

Локальные адреса IPv6 (unique local address), могут использоваться внутри организации без обращения в IANA, такие адреса не маршрутизируются в интернет, поэтому ничего страшного не произойдет, если несколько организаций будут использовать одни и те же локальные адреса. Локальные адреса IPv6 это аналоги частных или приватных адресов IPv4.

В IPv6 есть локальные адреса канала связи (link-local address), которые вообще не маршрутизируются, они назначаются автоматически и действуют **в пределах одного сегмента сети**, одного коммутатора или нескольких связанных между собой коммутаторов. Через маршрутизатор сообщения с такими IPv6 адресами не проходят.

Локальный адрес канала связи, действующий в рамках одного сегмента сети, начинается с цифр FE80. Если у вас для сетевого **интерфейса** (интерфейсы конфигурируются индивидуально) разрешён протокол IPv6 (проще всего это сделать через NetworkManager) то вы можете для интерфейса видеть что-то подобное следующему:

```
$ ip a s dev eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
qlen 1000
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
    altname enp0s25
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute eno1
        valid_lft forever preferred_lft forever
    inet6 fe80::762:c6bf:9eaa:93a9/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

А на любом другом хосте этой LAN можно выполнить:

```
$ ping -6 -c3 fe80::762:c6bf:9eaa:93a9%eno1
PING fe80::762:c6bf:9eaa:93a9%eno1(fe80::762:c6bf:9eaa:93a9%eno1) 56 data bytes
64 bytes from fe80::762:c6bf:9eaa:93a9%eno1: icmp_seq=1 ttl=64 time=8.54 ms
64 bytes from fe80::762:c6bf:9eaa:93a9%eno1: icmp_seq=2 ttl=64 time=3.64 ms
64 bytes from fe80::762:c6bf:9eaa:93a9%eno1: icmp_seq=3 ttl=64 time=3.68 ms

--- fe80::762:c6bf:9eaa:93a9%eno1 ping statistics ---
```



```
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 3.635/5.284/8.544/2.304 ms
```

Синтаксис записи IPv6

Характерно, что если вы выполните ту же команду `ping`, что и показанную выше, но в более привычном виде, то получите неожиданную ошибку:

```
$ ping -6 -c3 fe80::762:c6bf:9eaa:93a9
PING fe80::762:c6bf:9eaa:93a9(fe80::762:c6bf:9eaa:93a9) 56 data bytes

--- fe80::762:c6bf:9eaa:93a9 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2039ms
```

Это связано с тем, что даже в рамках одной локальной сети (LAN) соседние хосты могут иметь, в принципе, совпадающие **локальные** IPv6 для своих интерфейсов. Поэтому мы должны обязательно указывать (суффикс) с какого интерфейса мы намереваемся посылать ICMP пакеты (`ping` — это ICMP). В качестве суффикса может указываться не только **имя**, но и **номер** передающего интерфейса (номер — в том виде как его выводит команда `ip`: при остановках и запусках интерфейса, или его переконфигурации, номер может меняться, а последовательность нумерации интерфейсов будет нарушаться):

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
default qlen 1000
    link/ether 90:b1:1c:54:3a:46 brd ff:ff:ff:ff:ff:ff
    altname enp2s0f0
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
default qlen 1000
    link/ether 90:b1:1c:54:3a:47 brd ff:ff:ff:ff:ff:ff
    altname enp2s0f1
4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc fq_codel state UNKNOWN
mode DEFAULT group default qlen 500
    link/none

$ ping -6 -c3 fe80::762:c6bf:9eaa:93a9%3
PING fe80::762:c6bf:9eaa:93a9%3(fe80::762:c6bf:9eaa:93a9%eno2) 56 data bytes
64 bytes from fe80::762:c6bf:9eaa:93a9%eno2: icmp_seq=1 ttl=64 time=18.9 ms
64 bytes from fe80::762:c6bf:9eaa:93a9%eno2: icmp_seq=2 ttl=64 time=3.46 ms
64 bytes from fe80::762:c6bf:9eaa:93a9%eno2: icmp_seq=3 ttl=64 time=3.66 ms

--- fe80::762:c6bf:9eaa:93a9%3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 3.458/8.681/18.924/7.243 ms
```

Если вам смущает такая запись локальных IPv6 адресов в такой, несколько замысловатой, форме как показано здесь и/или чуть выше, то это может быть записано и в более традиционной форме:

```
$ ping -6 -c3 fe80::522d:d0bd:b221:a526 -Ieno2
ping: Warning: source address might be selected on device other than: eno2
PING fe80::522d:d0bd:b221:a526(fe80::522d:d0bd:b221:a526) from :: eno2: 56 data bytes
64 bytes from fe80::522d:d0bd:b221:a526%eno2: icmp_seq=1 ttl=64 time=3.46 ms
64 bytes from fe80::522d:d0bd:b221:a526%eno2: icmp_seq=2 ttl=64 time=3.44 ms
64 bytes from fe80::522d:d0bd:b221:a526%eno2: icmp_seq=3 ttl=64 time=3.44 ms

--- fe80::522d:d0bd:b221:a526 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 3.435/3.444/3.459/0.010 ms
```

Это всё показано и обсуждаем для того, что синтаксически адреса IPv6 должны записываться, иногда, в некоторой более сложной (уточнённой) форме, чем мы это привыкли делать для IPv4.

Например, IPv6 адреса в составе WEB URL (например, в адресной строке браузера) мы должны указывать помещая их в квадратные скобки [...], если в URL нужно указать TCP порт, то указывает его за пределами скобок. Вот как это выглядит (не обращайтесь, пока, внимания на «странную» доменную зону .lib — мы к этому вскоре вернёмся, нас пока интересуют только **синтаксические** формы записи адресов IPv6):

```
$ host ygg.lib
ygg.lib has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
$ curl [221:58c9:9a6:99be:f3d:c1ac:2b5b:9771] > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 52692    0 52692    0     0  44686      0 --:--:--  0:00:01 --:--:-- 44692
$ wget http://[221:58c9:9a6:99be:f3d:c1ac:2b5b:9771]:80/index.php -O - > /dev/null
--2023-04-21 17:11:23-- http://[221:58c9:9a6:99be:f3d:c1ac:2b5b:9771]/index.php
Подключение к [221:58c9:9a6:99be:f3d:c1ac:2b5b:9771]:80... соединение установлено.
HTTP-запрос отправлен. Ожидание ответа... 200 OK
Длина: нет данных [text/html]
Сохранение в: 'STDOUT'
-                                     [ <=> ] 51,54K --.-KB/s   за 0,1s
2023-04-21 17:11:24 (419 KB/s) - записан в stdout [52782]
```

(Я не показываю здесь графической картинке браузера, во избежание перегрузки места в тексте, но именно в таком синтаксисе мы должны показывать IPv6 в адресной строке браузера. А вот такой формой команды `wget` мы будем проверять работоспособность сетевых WEB-сервера.)

В других случаях, мы можем рассчитывать на привычный синтаксис записи IPv6 в консольных командах, здесь для `ping` используется не локальный IPv6 (удалённый в Интернет), когда сетевой стек может (из таблицы роутинга) однозначно определить трассу прохождения запроса, интерфейс отправки ICMP:

```
$ ping -6 -c3 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
PING 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771(221:58c9:9a6:99be:f3d:c1ac:2b5b:9771) 56 data
bytes
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=1 ttl=64 time=115 ms
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=2 ttl=64 time=117 ms
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=3 ttl=64 time=115 ms

--- 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 114.620/115.602/117.062/1.052 ms
```

Вот так мы можем определять открытость портов (файервол) и работоспособность WEB серверов на удалённых хостах Интернет, готовность их работать по HTTP (TCP порт 80) или HTTP (TCP порт 443):

```
$ telnet 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 80
Trying 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771...
Connected to 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771.
Escape character is '^]'.
^]
telnet> Connection closed.

$ telnet 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 443
Trying 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771...
Connected to 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771.
Escape character is '^]'.

```

```
^]  
telnet> Connection closed.
```

Наконец, вот так мы сканируем открытые порты удалённого хоста где-то далеко в Интернет (вспомним, что для IPv6, в отличие от IPv4, нет разницы в доступности хостов «локальные» они или «глобальные»):

```
$ nmap -6 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771  
Starting Nmap 7.80 ( https://nmap.org ) at 2023-04-21 14:59 EEST  
Nmap scan report for 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771  
Host is up (0.13s latency).  
Not shown: 997 filtered ports  
PORT      STATE SERVICE  
80/tcp    open  http  
443/tcp   open  https  
8080/tcp  closed http-proxy  
  
Nmap done: 1 IP address (1 host up) scanned in 23.21 seconds
```

Прогноз

Система IPv6 была официально введена в эксплуатацию 6 июня 2012 года (крупнейшими провайдерами одновременно). Вместо 2^{32} индивидуальных адресов IPv6 предоставляет 2^{128} адресов.

Проблема, казалось бы, решена... Однако переход с протокола IPv4 на IPv6 вызывает трудности, потому что эти протоколы несовместимы. И повсеместный переход пошёл гораздо медленнее, чем прогнозировалось на момент ввода в эксплуатацию. И изюминкой причины такого тяжелого перехода на 6-ю версию протокола является денежная стоимость! Многие провайдеры и компании просто не готовы вкладывать достаточное количество средств для перехода.

Но, как бы то ни было, IPv6 — это будущее Интернет! А вслед за ним и локальных сетей. Что приведёт со временем к полному вытеснению и отмиранию IPv4. Вопрос такого прогноза только в темпах и сроках.

Конечно, любой прогноз — дело неблагоприятное...

Адресные переменные в программном коде

Прототипы функций для работы с адресными переменными описаны `<arpa/inet.h>`.

Функция `inet_pton()` преобразовывает символьное изображение IP адреса в структуру адреса:

```
int inet_pton(int af, const char *src, void *dst);
```

Здесь:

- `af` — семейство адресов, может быть записано только одной из 2-х символьных констант: `AF_INET` или `AF_INET6`, что означает адрес IPv4 или IPv6, соответственно (`</usr/include/bits/socket.h>`);
- `src` — символьная строка, в которой записан исходный IP адрес. Для IPv4 адрес записывается в привычной точечной нотации: "d.d.d.d", где d — это десятичное число 0-255. Для IPv6 адрес может записываться в нескольких формах: h:h:h:h:h:h:h:h или h:h:h:h:h:h:d.d.d.d, где h — это до 4-х знаков шестнадцатеричные числа (0-ffff), кроме того, нулевые адресные группы в IPv6, как обычно, могут опускаться, например: ::FFFF:204.152.189.116.
- `dst` — результат преобразования, структура адреса, вид которой зависит от семейства адресов.

Для `AF_INET` это (определяется в `<linux/in.h>`):

```
struct in_addr {
    __be32 s_addr;
};
```

Для AF_INET6 это (определяется в <linux/in6.h>):

```
struct in6_addr {
    union {
        __u8          u6_addr8[16];
        ...
    } in6_u;
};
```

Вообще, в ядре Linux определено весьма много поддерживаемых семейств адресов (или семейств протоколов), что сильно усложняет поиск информации:

```
$ cat socket.h | grep 'AF_' | wc -l
44
```

Возвращаемое значение функции inet_pton() указывает на успешность операции преобразования:

- 1 — успешное преобразование;
- 0 — строка src не содержит строчное представление в специфицированном семействе адресов;
- 1 — в качестве параметра af указано недопустимое семейство адресов, при этом глобальная переменная errno устанавливается в EAFNOSUPPORT;

Функция inet_ntop() делает в точности наоборот: преобразовывает структуру сетевого адреса в символьное изображение IP адреса:

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

Здесь:

- af — семейство адресов AF_INET или AF_INET6;
- dst — строка результат преобразования, это же значение возвращается inet_ntop() при успешном выполнении;
- size — длина запрашиваемой строки результата, специфицируемая при вызове, это может быть символьная константа типа INET6_ADDRSTRLEN;

В заголовочном файле <arpa/inet.h> представлено ещё весьма много полезных функций манипуляции с IP адресами — они оставляются на самостоятельное изучение.

Чтобы не изобретать велосипед, в качестве примера работы с адресами (каталог архива address) воспользуемся кодом примера непосредственно из map страницы:

adr.c :

```
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    unsigned char buf[sizeof(struct in6_addr)];
    int domain, s;
    char str[INET6_ADDRSTRLEN];
    if (argc != 3) {
        fprintf(stderr, "Usage: %s {i4|i6|<num>} string\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    domain = (strcmp(argv[1], "i4") == 0) ? AF_INET :
             (strcmp(argv[1], "i6") == 0) ? AF_INET6 :
```

```

        atoi(argv[1]);
s = inet_pton(domain, argv[2], buf);
if (s <= 0) {
    if (s == 0)
        fprintf(stderr, "Not in presentation format");
    else
        perror("inet_pton");
    exit(EXIT_FAILURE);
}
if (inet_ntop(domain, buf, str, INET6_ADDRSTRLEN) == NULL) {
    perror("inet_ntop");
    exit(EXIT_FAILURE);
}
printf("%s\n", str);
exit(EXIT_SUCCESS);
}

```

Программа выполняет прямое, а затем обратное (для контроля) преобразование строчного изображения IP адреса. Выполнение примера иллюстрирует выше сказанное о форматах вызова функций:

```

$ ./adr i6 0:0:0:0:0:0:0:0
::
$ ./adr i6 1:0:0:0:0:0:0:8
1::8
$ ./adr i6 0:0:0:0:0:FFF:204.152.189.116
::ffff:204.152.189.116
$ ./adr i4 204.152.189.116
204.152.189.116

```

Разрешение адресов и имён

В принципе, сеть Интернет вполне могла бы работать **только** на основании адресации через числовые IP адреса (а как уже должно стать понятно и эти числовые записи в условных форматах — есть запись битовой последовательности длиной 32 или 64 бит). Однако человеку крайне сложно визуальное воспринимать такие длинные числовые последовательности. Поэтому адресуемым единицам в сети стали присваивать более-менее осмысленные имена, легко воспринимаемые человеком. Но тогда возникает задача поддержания устойчивых таблиц соответствий имён адресам, и задача быстрого разрешения имён в адреса и наоборот.

Для решения этих задач была создана (ещё на заре Интернет) система DNS (Domain Name System, система доменных имён) — компьютерная распределённая система для получения информации о доменах. Чаще всего она используется для получения IP адреса по имени хоста, получения информации о маршрутизации почты и некоторой другой адресной информации. Распределённая база данных DNS рассредоточена на иерархии выделенных DNS-серверов и по соответствующему (только для этих целей) протоколу. Из сказанного уже должно быть понятно, что протокол DNS — это один из старейших протоколов Интернет ... но он претерпел и множество изменений и имеет множество модификаций.

В POSIX/UNIX (утилитах) предусмотрен достаточный набор команд и функций для взаимного преобразования имён и адресов хостов и получения информации о хостах. Для таких адресных преобразований все эти инструменты должны привлекать информацию из файла `/etc/hosts`, или обращаться к механизму DNS.

Из командной строки подобное разрешение выглядит так (в 1-й команде для разрешения имени мы явно затребовали Интернет DNS-сервер 1.1.1.1):

```

$ nslookup qnx.org.ru 1.1.1.1
Server:      1.1.1.1
Address:     1.1.1.1#53

```

```
Non-authoritative answer:
Name:qnx.org.ru
Address: 89.223.70.101
```

```
$ host rus-linux.net
rus-linux.net has address 178.208.91.21
rus-linux.net has IPv6 address b2d0:5b15:400d:802::1004
rus-linux.net mail is handled by 10 mail.rus-linux.net.
```

Ещё вариант (здесь мы тоже явно указываем требуемый DNS-сервер 8.8.4.4):

```
$ dig @8.8.4.4 linux-ru.ru
; <<>> DiG 9.18.12-0ubuntu0.22.04.1-Ubuntu <<>> @8.8.4.4 linux-ru.ru
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12764
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;linux-ru.ru.                IN      A

;; ANSWER SECTION:
linux-ru.ru.                 600     IN      A      90.156.230.27

;; Query time: 96 msec
;; SERVER: 8.8.4.4#53(8.8.4.4) (UDP)
;; WHEN: Thu Apr 20 19:36:36 EEST 2023
;; MSG SIZE rcvd: 56
```

Мы ещё не раз и много будем говорить о механизме DNS-разрешения дальше, а пока взглянем на то как разрешение адресов и имён происходит в программном коде...

Разрешение имён в программном коде

Необходимость взаимного разрешения **имён** узлов (например yandex.ru) WAN в соответствующие им IP **адреса** (как 213.180.204.11) и наоборот возникает постоянно: показанные в этой фразе имя и адрес являются изображением одного и того же узла в разной системе адресации. Для осуществления этого преобразования в сети существует целая связанная сеть серверов DNS.

А из предназначенных для этих целей **функций**:

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

Функция возвращает информацию о хосте по его имени:

```
struct hostent {
    char *h_name;                /* Official name of host.          */
    char **h_aliases;            /* Alias list.                     */
    int h_addrtype;              /* Host address type AF_INET or AF_INET6 */
    int h_length;                /* Length of address 4 or 16        */
    char **h_addr_list;          /* List of addresses from name server. */
};
```

Ещё один образец — функция `gethostbyaddr()` ищущая информацию о хосте по IP адресу (обратная функциональность `gethostbyname()`)

```
#include <sys/socket.h>          /* for AF_INET */
```

```
struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
```

Функция `getaddrinfo()`, введенная более поздним стандартом POSIX 1.g, скрывает все зависимости в параметрах от типа протокола:

```
#include <netdb.h>
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Через указатель `res` функция возвращает указатель на связный **список** структур `addrinfo` (создаваемый динамически средствами `malloc()`):

```
struct addrinfo {
    int          ai_flags;
    int          ai_family; /* AF_xxx */
    int          ai_socktype; /* SOCK_DGRAM, SOCK_STREAM, ... */
    int          ai_protocol;
    socklen_t    ai_addrlen;
    struct sockaddr *ai_addr;
    char         *ai_canonname;
    struct addrinfo *ai_next; /* поле связи */
};
```

Поскольку список создаётся динамически, его позже нужно обязательно удалить вызовом:

```
void freeaddrinfo(struct addrinfo *res);
```

Пример использования `getaddrinfo()` показан в каталоге архива `address` на примере ретранслирующего сервера UDP (пример взят непосредственно из `man` функции `getaddrinfo()`):

- клиент:

```
$ ./gclie localhost 60000 привет
Received 13 bytes: привет
```

- сервер:

```
$ ./gserv 60000
Received 13 bytes from notebook.localdomain:58355
^C
```

Сетевые интерфейсы

В отличие от всех прочих **устройств** в системе, которым соответствуют имена (и номера) устройств в каталоге `/dev`, сетевые устройства создают сетевые **интерфейсы**, которые не отображаются как именованные устройства в `/dev`, но каждый из которых имеет набор своих характеристических параметров (MAC адрес, IP адрес, маска сети, префикс ...). Интерфейсы могут быть физическими (отображающими реальные аппаратные сетевые устройства, например, `eth0` — адаптер Ethernet), или логическими (отражающими некоторые моделируемые понятия, например, `tap0` — туннельный интерфейс). Одному аппаратному сетевому устройству может соответствовать одновременно **несколько** различных сетевых интерфейсов.

Сетевые интерфейсы **создаются** поддерживающими их драйверами — модулями ядра Linux. В общем случае, разработчик драйвера (модуля ядра) специфического сетевого устройства может выбрать имя для его интерфейса **произвольно** (определяется программным кодом драйвера).

Примечание: Продолжительные годы (десятилетия) существовала традиция (унаследованная из ранних UNIX) именовать сетевые интерфейсы по их принадлежности к той или иной физической **среде** (протоколу) передачи, например, интерфейс устройства WiFi должен был именоваться как `wlan0`. Но

в текущих реализациях Linux (ядра 3.x и далее) имена сетевых интерфейсов могут конструироваться автором кода **драйвера** не исходя из принадлежности к протоколам физического уровня (например, **eth0**, **eth1**, ... — для проводных Ethernet соединений), а совершенно произвольно. Временами предпринимаются попытки (Fedora 16, Fedora 17) ввести единообразное именование интерфейсов исходя, например, из расположения (адреса) данного сетевого адаптера на аппаратной шине PCI. Тогда имя интерфейса (для того же проводного Ethernet) может принять вид: **p7p1** или **p2p1**. Но и такие привязки не прижились широко.

Имена сетевых интерфейсов, как мы увидим вскоре, могут быть **произвольными** и определяются **модулем** ядра, реализующим интерфейс для устройств такого типа. В системе не может быть интерфейсов с совпадающими именами, поэтому поддерживающий модуль должен будет как-то модифицировать имена интерфейсов однотипных устройств, в соответствии с принятой схемой именований (например, добавляя суффикс: цифру, литеру, ...).

Посмотреть текущие существующие сетевые интерфейсы на узле можно, например, так:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp2s14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT
group default qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlp8s0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DORMANT group default
qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
```

Этим же интерфейсам соответствуют подкаталоги с соответствующими именами в **/proc**, каждый такой каталог содержит псевдофайлы-параметры (по диагностике или управлению) соответствующего интерфейса:

```
$ ls /proc/sys/net/ipv4/conf
all default enp2s14 lo wlp8s0
```

```
$ ls -w80 /proc/sys/net/ipv4/conf/enp2s14/
accept_local          disable_xfrm          proxy_arp_pvlan
accept_redirects      force_igmp_version    route_localnet
accept_source_route    forwarding             rp_filter
arp_accept            igmpv2_unsolicited_report_interval secure_redirects
arp_announce          igmpv3_unsolicited_report_interval send_redirects
arp_filter            log_martians          shared_media
arp_ignore            mc_forwarding         src_valid_mark
arp_notify            medium_id             tag
bootp_relay           promote_secondaries
disable_policy        proxy_arp
```

Так же, как для блочных устройств (дисков) в Linux, когда они ещё не пригодны для работы пока их не смонтируют, так и сетевые интерфейсы, сами по себе, ещё не пригодны для работы в сети, пока их не подготовят к использованию. Эта подготовка (конфигурирование) сетевых интерфейсов состоит в том, что сетевой интерфейс привязывается к своему индивидуальному IP адресу (одному или нескольким) и к подсети.

Важно: хотя определяем мы сетевые интерфейсы на уровне их имён, **вся** дальнейшая работа из программного кода с сетевым трафиком через интерфейсы происходит исключительно в терминах IP адресов. Корректное конфигурирование сетевых интерфейсов происходит на уровне **команд** системы, а уже дальнейшее их использование из программного кода — на уровне IP адресов.

Самым старым и известным инструментом диагностирования и управления сетевыми интерфейсами является утилита `ifconfig`. Вот как может выглядеть представляемая ею картина одновременно существующих интерфейсов:

\$ ifconfig

```
cipsec0 Link encap:Ethernet HWaddr 00:0B:FC:F8:01:8F
        inet addr:192.168.27.101 Mask:255.255.255.0
        inet6 addr: fe80::20b:fcff:fe8:18f/64 Scope:Link
        UP RUNNING NOARP MTU:1356 Metric:1
        RX packets:4 errors:0 dropped:3 overruns:0 frame:0
        TX packets:18 errors:0 dropped:5 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:538 (538.0 b) TX bytes:1670 (1.6 KiB)

em1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 192.168.1.20 netmask 255.255.255.0 broadcast 192.168.1.255
        inet6 fe80::a21d:48ff:fe4:935c prefixlen 64 scopeid 0x20<link>
        ether a0:1d:48:f4:93:5c txqueuelen 1000 (Ethernet)
        RX packets 1039 bytes 751246 (733.6 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 932 bytes 128724 (125.7 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
        device interrupt 17 memory 0xd4700000-d4720000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
        loop txqueuelen 0 (Local Loopback)
        RX packets 13 bytes 1360 (1.3 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 13 bytes 1360 (1.3 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ppp0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST> mtu 1500
        inet 77.52.137.120 netmask 255.255.255.255 destination 80.255.73.34
        ppp txqueuelen 3 (Point-to-Point Protocol)
        RX packets 57 bytes 3113 (3.0 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 6 bytes 111 (111.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlo1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 192.168.1.200 netmask 255.255.255.0 broadcast 192.168.1.255
        inet6 fe80::3623:87ff:fed6:850d prefixlen 64 scopeid 0x20<link>
        ether 34:23:87:d6:85:0d txqueuelen 1000 (Ethernet)
        RX packets 17 bytes 2022 (1.9 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 61 bytes 7534 (7.3 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
        device interrupt 19
```

Здесь представлены одновременно различные (по природе) сетевые интерфейсы:

- виртуальный интерфейс `cipsec0` (виртуальная частная сеть, VPN) созданный программными средствами (Cisco Systems VPN Client, от Cisco Systems), работающий через один из реальных физических каналов (что подтверждает сказанное выше о возможности нескольких сетевых интерфейсов над одним каналом).
- интерфейс физического проводного Ethernet адаптера `em1`.

- интерфейс физической беспроводной сети Wi-Fi wlan0 чипсета Broadcom Corporation BCM43228, и именно это имя wlan0 определяется использованием проприетарного драйвера от Broadcom.
- интерфейс ppp0 физического беспроводного 3G CDMA модема на USB шине.
- логический петлевой интерфейс lo, создающийся в любой системе, и поддерживающий любой из IP адресов локального диапазона 127.X.X.X:

```
$ ping 127.254.254.254
PING 127.254.254.254 (127.254.254.254) 56(84) bytes of data.
64 bytes from 127.254.254.254: icmp_seq=1 ttl=64 time=0.087 ms
64 bytes from 127.254.254.254: icmp_seq=2 ttl=64 time=0.071 ms
^C
--- 127.254.254.254 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.071/0.079/0.087/0.008 ms
```

Команда `ifconfig` имеет очень развитую функциональность, она позволяет выполнять не только диагностику, но и **управление** интерфейсами: запуск и останов интерфейса (операции `up` и `down`), присвоение IP адреса, маски, создание IP алиасов и многое другое. Для управления создаваемым сетевым интерфейсом (например, операции `up` или `down`), в отличие от диагностики, утилита `ifconfig` потребует прав `root`.

Более современным (более поздним), но и более развитым инструментом, является утилита `ip` (в некоторых дистрибутивах может потребоваться отдельная установка из репозитория пакета, известного под именем `iproute2`), вот результаты выполнения такой команды для несколько другой аппаратной конфигурации:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
4: vboxnet0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
5: pan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether ae:4c:18:a0:26:1b brd ff:ff:ff:ff:ff:ff
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff

$ ip addr show dev cipsec0
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff
    inet 192.168.27.101/24 brd 192.168.27.255 scope global cipsec0
    inet6 fe80::20b:fcff:fe8:18f/64 scope link
    valid_lft forever preferred_lft forever
```

Утилита `ip` имеет очень разветвлённый синтаксис, но, к счастью, и такую же разветвлённую, древовидную систему подсказок с **детализацией по ключевым словам**:

```
$ ip help
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
       ip [ -force ] -batch filename
where  OBJECT := { link | addr | addrlabel | route | rule | neigh | ntable |
                  tunnel | maddr | mroute | monitor | xfrm }
       OPTIONS := { -v[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
                   -f[amily] { inet | inet6 | ipx | dnet | link } |
                   -o[neline] | -t[imestamp] | -b[atch] [filename] }
```

```

$ ip addr help
Usage: ip addr {add|change|replace} IFADDR dev STRING [ LIFETIME ]
                                     [ CONFFLAG-LIST]

      ip addr del IFADDR dev STRING
      ip addr {show|flush} [ dev STRING ] [ scope SCOPE-ID ]
                                     [ to PREFIX ] [ FLAG-LIST ] [ label PATTERN ]

...

```

Примечание: Утилита `ip` допускает в записи команд сокращения ключевых слов до однозначного распознавания контекста, часто это всего одна буква. Вот такие «скороговорки»: `ip a s` — вместо записи `ip address show`, или `ip l` — вместо `ip link`. Это очень экономит время, удобно, и вы неоднократно будете видеть такие сокращения далее по тексту.

Широкое применение беспроводных сетевых технологий (в частности WiFi) породили целый круг новых специфических инструментов для их анализа и настройки их интерфейсов. Некоторые из них:

```

$ which iwconfig
/sbin/iwconfig

$ ls /sbin/iw*
/sbin/iw /sbin/iwconfig /sbin/iwevent /sbin/iwgetid /sbin/iwlist /sbin/iwpriv
/sbin/iwspy

$ iwconfig
lo          no wireless extensions.
em1         no wireless extensions.
wlo1        IEEE 802.11abg  ESSID:"ZTE"
           Mode:Managed  Frequency:2.412 GHz  Access Point: C8:64:C7:8A:50:16
           Retry short limit:7  RTS thr:off   Fragment thr:off
           Power Management:off

$ iw wlo1 info
Interface wlo1
    ifindex 3
    wdev 0x1
    addr 34:23:87:d6:85:0d
    ssid ZTE
    type managed
    wiphy 0

$ rfkill list
0: hci0: Bluetooth
    Soft blocked: no
    Hard blocked: no
1: phy0: Wireless LAN
    Soft blocked: no
    Hard blocked: no

$ iw phy0 info
wiphy phy0
    max # scan SSIDs: 1
    max scan IEs length: 0 bytes
    Coverage class: 0 (up to 0m)
    Supported Ciphers:
        * WEP40 (00-0f-ac:1)
        * WEP104 (00-0f-ac:5)
        * TKIP (00-0f-ac:2)

```

```

    * CCMP (00-0f-ac:4)
    * CMAC (00-0f-ac:6)
Available Antennas: TX 0 RX 0
Supported interface modes:
    * IBSS
    * managed
Band 1:
    Bitrates (non-HT):
        * 1.0 Mbps
        * 2.0 Mbps (short preamble supported)
        * 5.5 Mbps (short preamble supported)
        * 11.0 Mbps (short preamble supported)
        * 6.0 Mbps
        * 9.0 Mbps
        * 12.0 Mbps
        * 18.0 Mbps
        * 24.0 Mbps
        * 36.0 Mbps
        * 48.0 Mbps
        * 54.0 Mbps
    Frequencies:
        * 2412 MHz [1] (20.0 dBm)
        * 2417 MHz [2] (20.0 dBm)
    ...
        * 2467 MHz [12] (20.0 dBm)
        * 2472 MHz [13] (20.0 dBm)
        * 2484 MHz [14] (disabled)
Band 2:
    Bitrates (non-HT):
        * 6.0 Mbps
        * 9.0 Mbps
        * 12.0 Mbps
        * 18.0 Mbps
        * 24.0 Mbps
        * 36.0 Mbps
        * 48.0 Mbps
        * 54.0 Mbps
    Frequencies:
        * 5160 MHz [32] (20.0 dBm)
        * 5170 MHz [34] (20.0 dBm)
        * 5180 MHz [36] (20.0 dBm)
    ...
        * 5480 MHz [96] (disabled)
        * 5490 MHz [98] (disabled)
        * 5500 MHz [100] (20.0 dBm) (radar detection)
            DFS state: usable (for 5694 sec)
        * 5510 MHz [102] (20.0 dBm) (radar detection)
            DFS state: usable (for 5694 sec)
    ...
        * 6130 MHz [226] (disabled)
        * 6140 MHz [228] (disabled)
Supported commands:
    * set_interface
    * new_key
    * join_ibss
    * set_pmksa
    * del_pmksa
    * flush_pmksa
    * connect

```

```
* disconnect
software interface modes (can always be added):
interface combinations are not supported
Device supports scan flush.
```

Их использование специфично, но может быть в достаточной мере изучено и понято, используя возможности справочной системы (`man` и `--help`).

За последние годы, практически в любой графический (GUI) менеджер рабочего стола (DE — Desktop Environment) включен такой апплет управления окружения, как NetworkManager (NM):

```
$ ls /sbin/N*
/sbin/NetworkManager
```

Такой инструмент присутствует в любом графическом менеджере (GNOME, KDE, Xfce, LXDE, Mate, ...), хотя конкретные его реализации под разные DE могут в деталях отличаться. Этого инструмента, зачастую, вполне достаточно для создания, конфигурирования и настройки параметров как реальных, так и виртуальных (OpenVPN) сетевых интерфейсов:

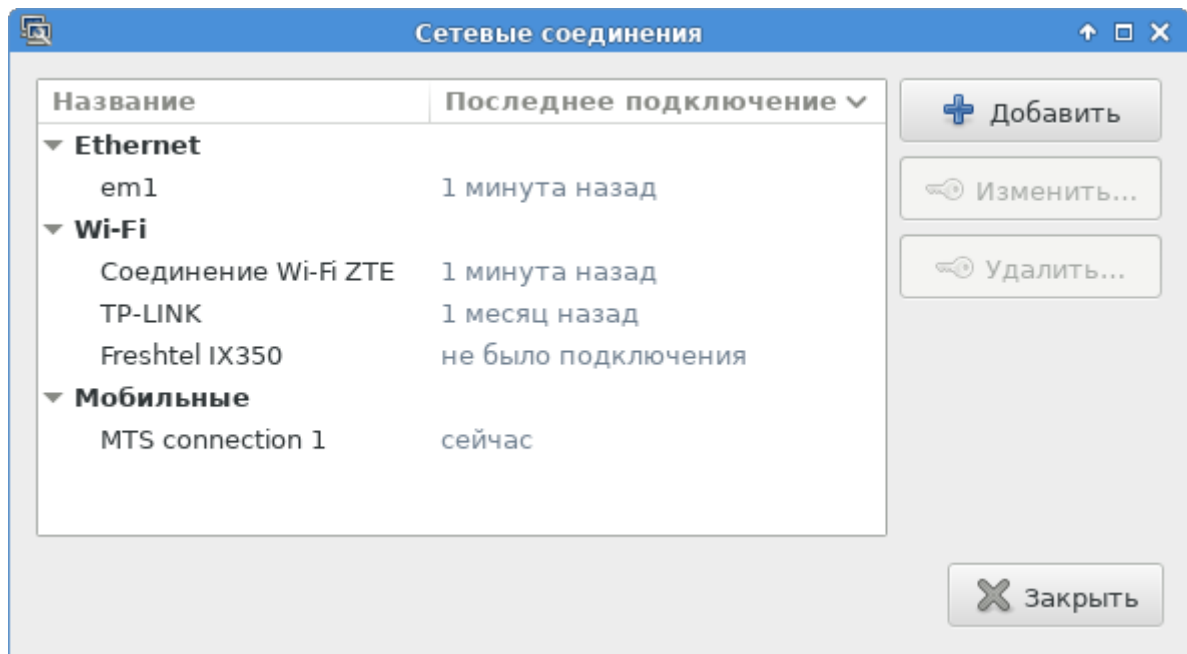


Рис. 1.5 Сетевые интерфейсы в графическом интерфейсе NetworkManager

Кроме графического (GUI) интерфейса NetworkManager имеет ещё и развитый консольный (CLI) интерфейс `nmcli` (command-line tool for controlling NetworkManager):

```
$ nmcli --help
Использование: nmcli [ПАРАМЕТРЫ] ОБЪЕКТ { КОМАНДА | help }

ПАРАМЕТРЫ
-a, --ask                запрос отсутствующих параметров
-c, --colors auto|yes|no использовать ли цветной вывод
-e, --escape yes|no      опускать разделители столбцов в значениях
-f, --fields <field,...>|all|common указать выводимые поля
-g, --get-values <field,...>|all|common краткая форма для -m tabular -t -f
-h, --help              показать данную справку
-m, --mode tabular|multiline режим вывода
-o, --overview          режим обзора
-p, --pretty            красивый вывод
```

-s, --show-secrets	разрешить показ паролей
-t, --terse	краткий вывод
-v, --version	показать версию программы
-w, --wait <seconds>	настроить таймаут завершения операций

ОБЪЕКТ

g[eneral]	общий статус и операции NetworkManager
n[etworking]	общее управление сетями
r[adio]	переключатели NetworkManager
c[onnection]	подключения NetworkManager
d[evice]	устройства, которыми управляет NetworkManager
a[gent]	агент секретов или агент polkit для NetworkManager
m[onitor]	отслеживание изменений в NetworkManager

Командный интерфейс NetworkManager очень обширный, желающие его использовать должны изучить его возможности по:

```
$ man 1 nmcli
```

```
...
```

NetworkManager — это целая огромная подсистема управления сетевыми интерфейсами, прирастающая год от года возможностями для конфигурирования новых сетевых служб. Это альтернативный (или дополняющий) сетевым утилитам способ **быстрого** управления сетевой подсистемой Linux. Эту дополнительную возможность нужно «держать в уме» (особенно для краткосрочных изменений). Но мы, в дальнейшем изложении, для большей наглядности не будем его использовать, а станем оперировать исключительно утилитами.

Таблица маршрутизации

Настолько же важной информацией как параметры интерфейсов (IP адреса, маски и др.) является таблица маршрутизации (ядра операционной системы). При нарушенной структуре таблицы маршрутизации работоспособность сетевого хоста невозможна (что часто упускают из виду), и нужно только корректно восстановить записи (строки) этой таблицы.

Если все рассматриваемые ранее параметры: IP, маска, префикс... — это атрибуты сетевого интерфейса, то таблица роутинга — это атрибут сетевого хоста в целом, это таблица ядра операционной системы.

Таблица маршрутизации хоста может быть диагностирована, например в таком виде:

```
$ route -n
```

```
Kernel IP routing table
```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.1.1	0.0.0.0	UG	1024	0	0	em1
80.255.73.34	0.0.0.0	255.255.255.255	UH	0	0	0	ppp0
192.168.1.0	0.0.0.0	255.255.255.0	U	0	0	0	em1
192.168.1.0	0.0.0.0	255.255.255.0	U	0	0	0	wlo1

Этой же командой (route), с соответствующими параметрами, производится редактирование таблицы (добавление, удаление строк).

Один из сетевых интерфейсов всегда отмечается в таблице маршрутизации как шлюз по умолчанию. Все пакеты в сетевом стеке (порождённые приложениями этого хоста, или пришедшие извне по другим внешним интерфейсам), чей адрес получателя не относится ни к одной строке таблицы маршрутизации — отправляется в **шлюз по умолчанию**. Маршрут по умолчанию обозначается записью 0.0.0.0 в численном изображении (опция -n), или записью default в символьном изображении таблицы.

Обычно, при управлении сетевыми интерфейсами (командами ifconfig, ip, или GUI апплетом NetworkManager) в таблице маршрутизации корректно добавляются или удаляются соответствующие строки. Однако иногда это соответствие разрушается, что

полностью нарушает работу сети. Тогда возникает необходимость редактировать таблицу вручную (добавлять или удалять строки). Делается это той же командой `route` с разнообразным набором опций и параметров. Простейший пример использования утилиты `route` для изменений в таблице роутинга будет показан ниже, в примере переименований сетевого интерфейса.

Для адресов IPv6 и IPv6 в ядре сохраняются 2 разных таблицы роутинга одновременно (показано для сравнения с одного и того же хоста):

```
$ route -n
```

Таблица маршрутизации ядра протокола IP

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.1.3	0.0.0.0	UG	100	0	0	eno1
0.0.0.0	192.168.1.3	0.0.0.0	UG	600	0	0	wlx008736005357
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0	eno1
192.168.1.0	0.0.0.0	255.255.255.0	U	100	0	0	eno1
192.168.1.0	0.0.0.0	255.255.255.0	U	600	0	0	wlx008736005357

```
$ route -n -6
```

Таблица маршрутизация ядра IPv6

Destination	Next Hop	Flag	Met	Ref	Use	If
::1/128	::	U	256	1	0	lo
fe80::/64	::	U	1024	3	0	eno1
fe80::/64	::	U	1024	1	0	wlx008736005357
::/0	::	!n	-1	1	0	lo
::1/128	::	Un	0	6	0	lo
fe80::762:c6bf:9eaa:93a9/128	::	Un	0	5	0	eno1
fe80::46a4:9424:cb31:2bd6/128	::	Un	0	2	0	wlx008736005357
ff00::/8	::	U	256	5	0	eno1
ff00::/8	::	U	256	5	0	wlx008736005357
::/0	::	!n	-1	1	0	lo

Если вы как-то изменяете правила роутинга для интерфейса в IPv6, то они никак не изменяют и не отображаются в роутинге для IPv6.

Ещё больше глубину и гибкость управления роутингом предоставляет уже упоминавшаяся команда `ip`, в новой нотации это выглядит так:

```
$ ip --help route
```

Usage: ip [OPTIONS] OBJECT { COMMAND | help }

ip [-force] -batch filename

where OBJECT := { address | addrlabel | fou | help | ila | ioam | l2tp | link |
macsec | maddress | monitor | mptcp | mroute | mrule |
neighbor | neighbour | netconf | netns | nexthop | ntable |
ntbl | route | rule | sr | tap | tcpmetrics |
token | tunnel | tuntap | vrf | xfrm }

OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
-h[uman-readable] | -iec | -j[son] | -p[retty] |
-f[amily] { inet | inet6 | mpls | bridge | link } |
-4 | -6 | -M | -B | -0 |
-l[oops] { maximum-addr-flush-attempts } | -br[ief] |
-o[neline] | -t[imestamp] | -ts[hort] | -b[atch] [filename] |
-rc[vbuf] [size] | -n[etns] name | -N[umeric] | -a[ll] |
-c[olor]}

Диагностика для того же хоста что показан выше:

```
$ ip route show
```

```
default via 192.168.1.3 dev eno1 proto static metric 100
default via 192.168.1.3 dev wlx008736005357 proto dhcp metric 600
169.254.0.0/16 dev eno1 scope link metric 1000
192.168.1.0/24 dev eno1 proto kernel scope link src 192.168.1.11 metric 100
```

```
192.168.1.0/24 dev wlx008736005357 proto kernel scope link src 192.168.1.235 metric 600
```

```
$ ip -6 route show
::1 dev lo proto kernel metric 256 pref medium
fe80::/64 dev eno1 proto kernel metric 1024 pref medium
fe80::/64 dev wlx008736005357 proto kernel metric 1024 pref medium
```

Управление роутингом

Пример управление роутинга IPv6 с помощью утилиты `ip` (всё тот же хост) ...

```
$ ip a s dev eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
qlen 1000
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
    altname enp0s25
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute eno1
        valid_lft forever preferred_lft forever
    inet6 fe80::762:c6bf:9eaa:93a9/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

Добавляю (физическому Ethernet) интерфейсу `eno1`, который не имеет никакого IPv6 адреса кроме локального (`fe80::762:c6bf:9eaa:93a9`), новый глобальный IPv6:

```
$ sudo ip address add 31e:af75:7a27:75::5/64 dev eno1
$ ip a s dev eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
qlen 1000
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
    altname enp0s25
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute eno1
        valid_lft forever preferred_lft forever
    inet6 31e:af75:7a27:75::5/64 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::762:c6bf:9eaa:93a9/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

В таблице роутинга IPv6 **автоматически** добавилась одна новая строка:

```
$ ip -6 route show
::1 dev lo proto kernel metric 256 pref medium
31e:af75:7a27:75::/64 dev eno1 proto kernel metric 256 pref medium
fe80::/64 dev eno1 proto kernel metric 1024 pref medium
fe80::/64 dev wlx008736005357 proto kernel metric 1024 pref medium
```

Но это ещё не всё что хотелось бы получить. Добавлю вручную через этот интерфейс/адрес ещё одну строку роутинга в реальную удалённую сеть `0200::/7` (это mesh-сеть Yggdrasil, но это пока не важно — об этом подробно будет позже):

```
$ sudo ip -6 route add 0200::/7 via 31e:af75:7a27:75::1
$ ip -6 route show
::1 dev lo proto kernel metric 256 pref medium
31e:af75:7a27:75::/64 dev eno1 proto kernel metric 256 pref medium
200::/7 via 31e:af75:7a27:75::1 dev eno1 metric 1024 pref medium
fe80::/64 dev eno1 proto kernel metric 1024 pref medium
fe80::/64 dev wlx008736005357 proto kernel metric 1024 pref medium
```

Всё! Теперь этот локальный хост LAN легко ping-ится по IPv6 к реальному удалённому хосту, находящемуся за тысячи километров в Казахстане:

```
$ ping -c3 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
PING 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771(221:58c9:9a6:99be:f3d:c1ac:2b5b:9771) 56 data
bytes
```



```

64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=1 ttl=63 time=862 ms
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=2 ttl=63 time=118 ms
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=3 ttl=63 time=118 ms

--- 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 118.003/365.914/861.613/350.512 ms

```

Аналогичное управление роутингом хоста можно производить (add, del, ...) с равным успехом и утилитой «старого стиля» route:

\$ route --help

Использование: route [-nNvee] [-FC] [<AF>] Отобразить таблицу маршрутизации ядра
route [-v] [-FC] {add|del|flush} ... Изменить таблицу маршрутизации для AF.

route {-h|--help} [<AF>] Детальное описание использование указанной AF.
route {-V|--version} Отобразить версию/автора и выйти.

-v, --verbose более детальный вывод
-n, --numeric не преобразовывать адреса в имена
-e, --extend отображать другую/больше информации
-F, -fib отобразить информацию форвардинга базы (по умолчанию)
-C, --cache отобразить кэш маршрутизации вместо FIB

<AF>=Use -4, -6, '-A <af>' or '--<af>'; default: inet

Список возможных адресных семейств (которые поддерживают маршрутизацию):

inet (DARPA Internet) inet6 (IPv6) ax25 (AMPR AX.25)
netrom (AMPR NET/ROM) ipx (Novell IPX) ddp (Appletalk DDP)
x25 (CCITT X.25)

Вот как выглядит «в старой нотации» таблица роутинга IPv6 после только-что проделанных мной манипуляций:

\$ route -6

Таблица маршрутизация ядра IPv6

Destination	Next Hop	Flag	Met	Ref	Use	If
ip6-localhost/128	:::	U	256	2	0	lo
31e:af75:7a27:75::/64	:::	U	256	2	0	eno1
200::/7	raspberrypi	UG	1024	2	0	eno1
fe80::/64	:::	U	1024	3	0	eno1
fe80::/64	:::	U	1024	1	0	
wlx008736005357						
:::/0	:::	!n	-1	1	0	lo
ip6-localhost/128	:::	Un	0	7	0	lo
nvidia/128	:::	Un	0	3	0	eno1
nvidia/128	:::	Un	0	5	0	eno1
nvidia/128	:::	Un	0	3	0	wlx008736005357
ip6-mcastprefix/8	:::	U	256	5	0	eno1
ip6-mcastprefix/8	:::	U	256	5	0	wlx008736005357
:::/0	:::	!n	-1	1	0	lo

Попутно тут же убеждаемся, что никакие манипуляции (достаточно сложные выше проведенные) с таблицей роутинга для IPv6 **никак** не затрагивают таблицу роутинга для IPv4 для тех же сетевых интерфейсов:

\$ route

Таблица маршрутизации ядра протокола IP

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
default	GPON	0.0.0.0	UG	100	0	0	eno1
default	GPON	0.0.0.0	UG	600	0	0	wlx008736005357

link-local	0.0.0.0	255.255.0.0	U	1000	0	0	eno1
192.168.1.0	0.0.0.0	255.255.255.0	U	100	0	0	eno1
192.168.1.0	0.0.0.0	255.255.255.0	U	600	0	0	wlx008736005357

Алиасные IP адреса

Простейшим примером множественности **логических** сетевых интерфейсов, разделяющих общий **физический** адаптер, могут служить **сетевые алиасы**, когда существующему интерфейсу дополнительно присваиваются адрес и маска, делающие его представленным ещё в другой подсети.

Рассмотрим пример ... положим, в LAN подсеть 192.168.1.0/24, хост с локальным именем:

```
$ hostname
nvidia
$ ip -4 a s dev eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    altname enp0s25
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute eno1
        valid_lft forever preferred_lft forever
```

```
$ route -n
```

Таблица маршрутизации ядра протокола IP

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.1.3	0.0.0.0	UG	100	0	0	eno1
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0	eno1
192.168.1.0	0.0.0.0	255.255.255.0	U	100	0	0	eno1

Тривиальный случай... Добавим этому физическому (проводному Ethernet) интерфейсу eno1 ещё один алиасный адрес IPv4 из совершенно другой подсети 10.0.0.2/24 :

```
$ sudo ifconfig eno1:1 10.0.0.2 up
$ ifconfig
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.11 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::762:c6bf:9eaa:93a9 prefixlen 64 scopeid 0x20<link>
    ether 70:71:bc:a3:c5:c0 txqueuelen 1000 (Ethernet)
    RX packets 4890 bytes 496512 (496.5 KB)
    RX errors 0 dropped 1 overruns 0 frame 0
    TX packets 2637 bytes 261068 (261.0 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 20 memory 0xfe400000-fe420000

eno1:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.0.2 netmask 255.0.0.0 broadcast 10.255.255.255
    ether 70:71:bc:a3:c5:c0 txqueuelen 1000 (Ethernet)
    device interrupt 20 memory 0xfe400000-fe420000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Локальная петля (Loopback))
    RX packets 772 bytes 68251 (68.2 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 772 bytes 68251 (68.2 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Таблица маршрутизации этого хоста автоматически (не требует нашего участия) поменяется — в ней добавилась строка (маршрут) для новой подсети:

```
$ route -n
```

Таблица маршрутизации ядра протокола IP

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.1.3	0.0.0.0	UG	100	0	0	eno1
10.0.0.0	0.0.0.0	255.0.0.0	U	0	0	0	eno1
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0	eno1
192.168.1.0	0.0.0.0	255.255.255.0	U	100	0	0	eno1

На хосте у нас появился новый, фиктивный интерфейс eno1:1 с алиасным IPv4, который (адрес), естественно, не видится в исходной LAN, у него по маске другая подсеть. С любого хоста LAN он недоступен:

```
$ ping -c3 10.0.0.2
```

```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
```

```
--- 10.0.0.2 ping statistics ---
```

```
3 packets transmitted, 0 received, 100% packet loss, time 2025ms
```

Теперь выберем другой, произвольно взятый, хост этой LAN:

```
$ hostname
```

```
nvme
```

```
$ ip -4 a s dev enp3s0
```

```
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
```

```
inet 192.168.1.138/24 brd 192.168.1.255 scope global dynamic noprefixroute enp3s0  
valid_lft 165109sec preferred_lft 165109sec
```

```
$ /sbin/route -n
```

Kernel IP routing table

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.1.3	0.0.0.0	UG	100	0	0	enp3s0
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0	enp3s0
192.168.1.0	0.0.0.0	255.255.255.0	U	100	0	0	enp3s0

(Здесь у меня работает другой дистрибутив Linux, у которого каталог /sbin не прописан в переменную окружения \$PATH, что потребует указывать полный путь при вызове некоторых сетевых команд. Такая возможность довольно частая, и не должна вводить вас в недоумение.)

Прделаю с этим хостом примерно то же самое, что и с предыдущим:

```
$ sudo ifconfig enp3s0:1 10.0.0.5 up
```

```
$ echo $?
```

```
0
```

(Иногда целесообразно сразу же проверить код успешности завершения только-что выполненной команды.)

Интерфейс поменял вид:

```
$ /sbin/ifconfig enp3s0
```

```
enp3s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
inet 192.168.1.138 netmask 255.255.255.0 broadcast 192.168.1.255  
inet6 fe80::921b:eff:fe2b:fe3a prefixlen 64 scopeid 0x20<link>  
ether 90:1b:0e:2b:fe:3a txqueuelen 1000 (Ethernet)  
RX packets 442744 bytes 152787250 (145.7 MiB)  
RX errors 0 dropped 51 overruns 0 frame 0  
TX packets 133419 bytes 13095746 (12.4 MiB)  
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
$ /sbin/ifconfig enp3s0:1
```

```
enp3s0:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
```

```
inet 10.0.0.5 netmask 255.0.0.0 broadcast 10.255.255.255
ether 90:1b:0e:2b:fe:3a txqueuelen 1000 (Ethernet)
```

Примечание: Характерно, что как в этом, так и в предыдущем случае (там я не стал акцентироваться на этом для экономии места) команда `ip` (в отличие от `ifconfig`) не знает о новом (фиктивном, расщеплённом) интерфейсе `enp3s0:1`, но видит новый алиасный IPv4 для интерфейса, как в его «новом», так и «старом» именовании:

```
$ ip -4 a s dev enp3s0
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    inet 192.168.1.138/24 brd 192.168.1.255 scope global dynamic noprefixroute enp3s0
        valid_lft 161823sec preferred_lft 161823sec
    inet 10.0.0.5/8 brd 10.255.255.255 scope global enp3s0:1
        valid_lft forever preferred_lft forever

$ ip -4 a s dev enp3s0:1
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    inet 192.168.1.138/24 brd 192.168.1.255 scope global dynamic noprefixroute enp3s0
        valid_lft 161816sec preferred_lft 161816sec
    inet 10.0.0.5/8 brd 10.255.255.255 scope global enp3s0:1
        valid_lft forever preferred_lft forever
```

Таблица роутинга, как и для предыдущего хоста, поменялась:

```
$ /sbin/route -n
Kernel IP routing table
Destination      Gateway          Genmask          Flags Metric Ref    Use Iface
0.0.0.0          192.168.1.3     0.0.0.0          UG    100    0      0 enp3s0
10.0.0.0         0.0.0.0         255.0.0.0        U     0      0      0 enp3s0
169.254.0.0     0.0.0.0         255.255.0.0      U     1000   0      0 enp3s0
192.168.1.0     0.0.0.0         255.255.255.0    U     100    0      0 enp3s0
```

Маршруты к двум разным подсетям (192.168.1.0/24 и 10.0.0.0/8) представлены в таблице маршрутизации разными записями (строками). И теперь мы имеем прозрачность трафика в новой подсети 10.0.0.0/8 (то, собственно, для чего всё и делалось):

```
$ ping -c3 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.554 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.250 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.368 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2032ms
rtt min/avg/max/mdev = 0.250/0.390/0.554/0.125 ms
```

Естественно, для `ping` (прозрачность) тех же «препарированных» хостов в исходной подсети 192.168.1.0/24 прекрасно сохранился!

Но ещё интереснее эксперимент со сканированием хостов подсети (я его показываю с первого из рассмотренных хостов, поскольку у меня только там есть установленная утилита `ntar`):

```
$ ping -c3 10.0.0.5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=0.328 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=0.205 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=0.259 ms

--- 10.0.0.5 ping statistics ---
```

```
3 packets transmitted, 3 received, 0% packet loss, time 2045ms
rtt min/avg/max/mdev = 0.205/0.264/0.328/0.050 ms
```

```
$ nmap -sP 10.0.0.0/24
```

```
Starting Nmap 7.80 ( https://nmap.org ) at 2023-04-22 05:18 EEST
Nmap scan report for nvidia (10.0.0.2)
Host is up (0.00019s latency).
Nmap scan report for 10.0.0.5
Host is up (0.00027s latency).
Nmap done: 256 IP addresses (2 hosts up) scanned in 3.00 seconds
```

Сканер подсети 10.0.0.0, работающей **над** физической LAN с добрым десятком хостов подсети 192.168.1.0, видит только 2 хоста (10.0.0.2 и 10.0.0.5). Фактически, мы создали виртуальную сеть над реальной физической.

Техника создания и работы с сетевыми алиасами является в высшей степени полезной при отработке сетевых проектов и написании модулей ядра, обсуждаемых далее.

Примечание: Я показал работу с алиасными адресами пользуясь утилитами «старого стиля» (ifconfig и route из Linux пакета net-tools). То же самое можно наверняка проделывать и с утилитой «нового стиля» (ip из Linux пакета iproute2). Я этого не делал и не проверял ... и оставляю это в качестве задания для самостоятельной проработки читателям.

Петлевой интерфейс

На любом компьютерном хосте (даже если он вообще не использует сеть, или даже просто никак не подключен к сети) присутствует петлевой интерфейс (loopback, интерфейс lo)

Группа адресов 127/8 выделены под петлевой интерфейс:

```
$ ifconfig lo | head -n2
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
```

Иногда указывается (в публикациях), что адресом петлевого интерфейса является 127.0.0.1 — это заблуждение: адресами петлевого интерфейса являются **все** IP адреса подсети 127.0.0.0:255.0.0.0 класса A:

```
$ ping 127.255.255.254
PING 127.255.255.254 (127.255.255.254) 56(84) bytes of data.
64 bytes from 127.255.255.254: icmp_seq=1 ttl=64 time=0.061 ms
64 bytes from 127.255.255.254: icmp_seq=2 ttl=64 time=0.065 ms
^C
--- 127.255.255.254 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.061/0.063/0.065/0.002 ms
```

Для семейства IPv6 адрес петлевого интерфейса (loopback) выглядит в сокращённой записи особенно элегантно ::1. Даже если вы не пользуетесь IPv6, но работаете на одной из современных операционных систем, у вас наверняка установлен этот протокол. Это легко проверить, пропинговав loopback.

```
$ ping -c3 ::1
PING ::1(::1) 56 data bytes
64 bytes from ::1: icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from ::1: icmp_seq=2 ttl=64 time=0.036 ms
64 bytes from ::1: icmp_seq=3 ttl=64 time=0.036 ms

--- ::1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2024ms
rtt min/avg/max/mdev = 0.025/0.032/0.036/0.005 ms
```

Этот интерфейс позволяет клиенту и серверу на одном и том же компьютере общаться друг с другом используя стек TCP/IP. Можно предположить, что транспортный уровень распознает, что указанный ему удаленный адрес — это петлевой адрес и каким-либо образом сокращает процесс обработки датаграмм. Однако это не так. Осуществляется **полная** обработка данных на транспортном и сетевом уровнях, после чего IP датаграмма направляется по петле назад, двигаясь вверх достигает пользовательского приложения, ожидающего приёма с этого интерфейса. Но IP датаграмма, посылаемая в петлевой интерфейс, никогда не попадает в физическую среду передачи (кабель). Может показаться неэффективным то что транспортный и сетевой уровни обрабатывают данные, которые посылаются по петле. Но на такую реализацию есть два основания:

- проще реализовать обработку единообразно, когда петлевой интерфейс для сетевого уровня выглядит просто как еще один интерфейс канального уровня (мы будем реализовывать такие интерфейсы при рассмотрении сетевых драйверов далее);
- полновесная и единообразная обработка трафика по петлевому интерфейсу делает его незаменимым инструментом отладки, тестирования и диагностики для сетевых инструментов и проектов в локальном окружении.

Петлевой интерфейс активно используют различные подсистемы Linux и установленных сторонних программ. Неаккуратное использование и разрушение петлевого интерфейса может поэтому привести к дефектам работы в самых неожиданных местах. Никогда не удаляйте петлевой интерфейс, даже если он кажется вам совершенно лишним: этим вы полностью разрушите операционную систему Linux, и добьётесь только её новой полной инсталляции!

Переименование сетевого интерфейса

Имя сетевого интерфейса, как было уже сказано, задаётся модулем, создающим этот интерфейс (мы увидим это на примерах написания модулей позже). Но имя сетевого интерфейса (созданного модулем, или присутствующего ранее в системе) не есть совершенно константное значение, которое должно оставаться неизменным во всё время работы интерфейса в системе. Мы можем интерфейс динамически переименовать (в произвольное имя)! И начнём мы это делать с традиционного Ethernet проводного интерфейса:

```
$ ip -4 a s dev eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
qlen 1000
    altname enp0s25
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute eno1
        valid_lft forever preferred_lft forever
```

И зафиксируем нормальное состояние в котором находится таблица маршрутизации (IPv4 в данном случае):

```
$ route -n
Таблица маршрутизации ядра протокола IP
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0      192.168.1.3  0.0.0.0      UG    100    0      0 eno1
169.254.0.0  0.0.0.0      255.255.0.0   U     1000    0      0 eno1
192.168.1.0  0.0.0.0      255.255.255.0 U     100    0      0 eno1
```

Переименовываем интерфейс командой:

```
$ sudo ip link set dev eno1 name eth0
RTNETLINK answers: Device or resource busy
```

Закономерный итог: нельзя манипулировать интерфейсом который активен (поднят)! Поэтому правильный путь будет несколько сложнее:

```
$ sudo ifconfig eno1 down
$ sudo ip link set dev eno1 name eth0
```

```

$ ip l
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT
group default qlen 1000
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
    altname enp0s25
    altname eno1
3: wlx008736005357: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DEFAULT group
default qlen 1000
    link/ether 00:87:36:00:53:57 brd ff:ff:ff:ff:ff:ff

```

Подымаем интерфейс и восстанавливаем его IP (я экспериментировал с интерфейсом со статически присвоенным IP, при DHCP, возможно, адрес можно не указывать и всё будет проще):

```

$ sudo ifconfig eth0 192.168.1.11 up
$ ip -4 a s dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
qlen 1000
    altname enp0s25
    altname eno1
    inet 192.168.1.11/24 brd 192.168.1.255 scope global eth0
        valid_lft forever preferred_lft forever

```

Адрес восстановлен, но этого мало: команда `ip`, манипулируя с интерфейсом, зачастую портит таблицу маршрутизации:

```

$ route -n
Таблица маршрутизации ядра протокола IP
Destination Gateway Genmask Flags Metric Ref Use Iface
169.254.0.0    0.0.0.0          255.255.0.0      U        1000    0        0 eth0
192.168.1.0    0.0.0.0          255.255.255.0    U         0        0        0 eth0

```

Восстанавливаем маршрут по умолчанию на роутер из LAN во внешнюю WAN:

```

$ sudo ip route add default via 192.168.1.3
$ route -n
Таблица маршрутизации ядра протокола IP
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0        192.168.1.3      0.0.0.0          UG         0        0        0 eth0
169.254.0.0    0.0.0.0          255.255.0.0      U        1000    0        0 eth0
192.168.1.0    0.0.0.0          255.255.255.0    U         0        0        0 eth0

```

Вот теперь можно проверять прозрачность трафика для нового интерфейса (указывается явно в команде) и для локальной сети, и для Интернет:

```

$ ping 192.168.1.3 -c3 -Ieth0
PING 192.168.1.3 (192.168.1.3) from 192.168.1.11 eth0: 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.679 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=0.670 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=0.606 ms

--- 192.168.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2032ms
rtt min/avg/max/mdev = 0.606/0.651/0.679/0.032 ms

$ ping 1.1.1.1 -c3 -Ieth0
PING 1.1.1.1 (1.1.1.1) from 192.168.1.11 eth0: 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=57 time=9.50 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=57 time=8.79 ms

```

```
64 bytes from 1.1.1.1: icmp_seq=3 ttl=57 time=9.07 ms

--- 1.1.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 8.785/9.116/9.499/0.293 ms
```

Далее с этим новым именем интерфейса (eth0) работают все сетевые утилиты и протоколы.

Эта возможность оказывается чрезвычайно **мощным** инструментом при тестировании, при экспериментах, или при конфигурировании программных пакетов, включающих модули ядра сетевых устройств, которые создают свои новые сетевые интерфейсы.

Примечание: Существует и описан в публикациях (и не один) способ (ищите) изменить имя интерфейса перманентно (постоянно от загрузки) конфигурационными параметрами операционной системы. Но меня не интересует такой способ, а интересует возможность изменять имена динамически, многократно (при необходимости) и без перезагрузки.

В порядке итога сформулируем те правила, которые нужно соблюдать при **динамическом** переименовании сетевого интерфейса:

- Изменить имя можно только для остановленного (down) интерфейса, в противном случае интерфейс будет «занят» для операции;
- После переименования интерфейс нужно сделать активным (up) с присвоением (восстановление) ему IP адреса;
- Последним действием нужно в таблице роутинга восстановить маршрут по умолчанию на шлюз во внешнюю сеть, Интернет (если у вас интерфейс ходил вовне, что зачастую так и бывает);
- для восстановления маршрута используем команду, как это было показано выше, вида:

```
# ip route add default via 192.168.1.3
```

Или, в «старой нотации» (net - tools), для тех же значений (сравните):

```
# route add default gw 192.168.1.3 eth0
```

В обязательном порядке проверяйте реальную прозрачность трафика (команды ping, traceroute, ... с явным указанием имени интерфейса в команде) по завершению переименований.

Альтернативные имена

Есть ещё одна возможность из подобной области, появившаяся в пакете iproute2 относительно недавно, утверждается что с версии v5.4.0 (это примерно год 2019) — это альтернативные имена интерфейсов:

```
$ ip -v
ip utility, iproute2-5.15.0, libbpf 0.5.0
```

```
$ sudo ip link property add dev eth0 altname eno2
```

```
$ ip l show dev eth0
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT
group default qlen 1000
```

```
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
```

```
    altname enp0s25
```

```
    altname eno1
```

```
    altname eno2
```

```
$ ip -4 a s dev eth0
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
qlen 1000
```



```

altname enp0s25
altname eno1
altname eno2
inet 192.168.1.11/24 brd 192.168.1.255 scope global eth0
    valid_lft forever preferred_lft forever

```

Обратите внимание: я добавляю новое альтернативное имя интерфейсу (а позже удалю другой имя) не останавливая (down) его, делается «на ходу»! При этом, естественно, что изменения в альтернативных именах никак не влияют на таблицу роутинга, а поэтому и не нарушают работу интерфейса.

Симметрично, альтернативное имя может быть удалено (ликвидировано), даже если это первичное имя интерфейса, данное ему при загрузке системы:

```

$ sudo ip link property del dev eno2 altname eno1
$ ip l sh dev eno2
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT
group default qlen 1000
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
    altname enp0s25
    altname eno2

```

Важно чтобы удаляемое имя интерфейса не фигурировало как целевой интерфейс в записях таблицы маршрутизации:

```

$ route -n
Таблица маршрутизации ядра протокола IP
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0        192.168.1.3    0.0.0.0        UG    0        0        0 eth0
169.254.0.0    0.0.0.0        255.255.0.0    U    1000     0        0 eth0
192.168.1.0    0.0.0.0        255.255.255.0  U     0        0        0 eth0

```

Эта относительно новая возможность решает многие проблемы с именованием интерфейсов.

Порты транспортного уровня

Адрес назначения для сетевого интерфейса характеризуется его IP адресом. Но на хосте, которому принадлежит этот интерфейс со своим IP адресом, может работать одновременно много самых различных сетевых приложений. Для разграничения принадлежности пакетов между процессами (серверами, сервисами, службами) и прикладными протоколами вводится ещё один уровень адресации сверх того: порты транспортного уровня (на уровне протоколов TCP, UDP, SCTP, ...).

Каждому протоколу более **высоких уровней** (SSH, FTP, HTTP, ...) соответствует свой стандартный порт. Порт выражается как 16-битовое целочисленное значение, количество портов ограничено с учётом 16-битной адресации ($2^{16}=65536$, начало — 0). Порты TCP и порты UDP — это совершенно разные сущности, а их возможное численное **совпадение** для отдельных служб делается только для удобства. Все порты разделены на три диапазона — **общезвестные** (или **системные**, 0—1023), **зарегистрированные** (или **пользовательские**, 1024—49151) и **динамические** (или **частные**, 49152—65535).

Работа с системными портами (0—1023) всегда требует прав `root`. Зарегистрированные порты — это порты, которые комиссия по регистрации IANA официально зарегистрировала за определёнными протоколами (рекомендуемые). Динамические и/или приватные порты — от 49152 до 65535. Эти порты динамические, в том смысле, что они могут быть использованы любым процессом и с любой целью. Часто, программа, работающая на зарегистрированном порту (от 1024 до 49151) порождает другие дочерние процессы, которые затем используют свои динамические порты. Самая свежая информация о регистрации номеров портов может быть найдена здесь: <http://www.iana.org/numbers.htm#P>.

Соответствия протоколов их **численным значениям портов** UDP или TCP смотрим в файле описания **сетевых служб** /etc/services :

```
$ cat /etc/services
...
ftp          21/tcp
ftp          21/udp          fsp fspd
ssh          22/tcp          # SSH Remote Login Protocol
ssh          22/udp          # SSH Remote Login Protocol
telnet       23/tcp
telnet       23/udp
...

$ grep -v ^$ /etc/services | grep -v ^# | wc -l2
318
```

Вот такое число сервисов предопределено для различных протоколов комиссией IANA. Но это не значит что любой протокол обязан **использовать** этот свой порт — это только значения по умолчанию, рекомендуемые. Мы вполне можем, например, проверить работает ли на каком-то WEB сервер (стандартный порт 80) протокола HTTP (мы не собираемся вести диалог, и тут же прерываем коннект):

```
$ telnet linux-ru.ru 80
Trying 90.156.230.27...
Connected to linux-ru.ru.
Escape character is '^]'.
^]
telnet> Connection closed.
```

А если нас интересует прозондировать защищённый WEB сервер протокола HTTPS (стандартный порт 443), то поступаем так:

```
$ telnet linux-ru.ru 443
Trying 90.156.230.27...
Connected to linux-ru.ru.
Escape character is '^]'.
^]
telnet> Connection closed.
```

Но и сами WEB сервера используют порты (80, 443) только как рекомендуемые для них в /etc/services — весьма часто вы сможете наблюдать WEB сервера на портах 8080, 3128 и других.

Главное правило состоит в том, что на одном сервере не могут работать 2 сервиса одного транспортного протокола (UDP с TCP) с **совпадающими номерами** порта! Не могут один и тот же порт прослушивать одновременно 2 серверных приложения.

Но прежде, чем обсуждать возможности и особенности работы с любым из сетевых протоколов, нужно убедиться, что использование этого протокола не запрещено в настройках файервола вашей системы. После этого (уточнив характеристики интересующего нас протокола) разрешаем его к использованию средствами конфигурирования файервола (утилиты iptables или GUI оболочки для её управления).

Источники использованной информации

[1] Таблица ссылок на документы RFC по основным протоколам сети —

<http://ru.wikipedia.org/wiki/RFC>

² Вот такую форму команд, с двумя grep -v ... , мы вынуждены и многократно будем использовать для просмотра конфигурационных файлов, которые содержат, обычно, множество комментированных и пустых строк, для того чтобы их исключить из вывода.

[2] База данных RFC (любой RFC может быть здесь найден и свободно скачан) —

<https://www.rfc-editor.org/>

[3] Request for Comments — <https://web.archive.org/web/20090202124230/http://www.ietf.org/rfc.html>

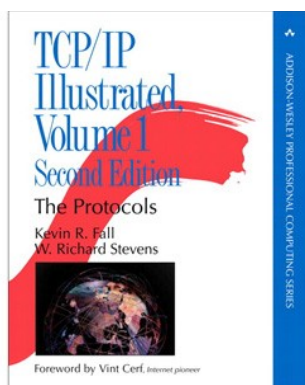
[4] У. Р. Стивенс, Протоколы TCP/IP. В подлиннике, ВНВ, СПб, 2003, ISBN: 5-94157-300-6, 672 стр.



У. Р. Стивенс, Протоколы TCP/IP. В подлиннике, Невский Диалект, СПб, 2004, ISBN: 5-7940-0093-7, 672 страниц

Оригинал: TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, 1994, ISBN 0-201-63346-9

[5] TCP/IP Illustrated, Volume 1: The Protocols, 2nd Edition W. Richard Stevens, Kevin R. Fall, May 05, 2012



[6] Семенов Ю.А. (ГНЦ ИТЭФ), Telecommunication technologies - телекоммуникационные технологии (v2.1) — https://www.opennet.ru/docs/RUS/inet_book/

[7] NAT (Network Address Translation) для новичков — <https://habr.com/ru/articles/583172/>

[8] IPv6 теория и практика: введение в IPv6 — <https://habr.com/ru/post/210100/>

[9] Адреса протокола IPv6 — локальный и глобальный тип —

<https://zvondozvon.ru/tehnologii/kompyuternye-seti/adresa-ipv6>

[10] IANA, Internet Protocol Version 6 Address Space —

<https://www.iana.org/assignments/ipv6-address-space/ipv6-address-space.xhtml>

[11] IPv6 в Linux.

Часть 1 — <https://h4cks4w.blogspot.com/2015/04/ipv6-linux-1.html>

Часть 2 — <https://h4cks4w.blogspot.com/2015/04/ipv6-linux-2.html>

[12] Семенов Ю.А. (ГНЦ ИТЭФ), Адресация IPv6 —

https://www.opennet.ru/docs/RUS/inet_book/4/44/ip6_4411.html

[13] Онлайн IPv6 Subnet Calculator — <https://www.vultr.com/resources/subnet-calculator-ipv6/>

[14] Дж. Курроуз, Росс К., Компьютерные сети. Нисходящий подход, М.:Эксмо, январь 2017, 3-е издание., ISBN: 978-5-699-94358-6, 912 страниц



Протоколы и инструменты прикладного уровня

Число утилит предназначенных для работы с сетевым стеком огромно. Оно превосходит любую другую раздел работы компьютера. Значительную часть доступных утилит мы уже видели по тексту ранее. Здесь же мы проведём только некоторую их систематизацию.

Часть обсуждаемых утилит устанавливается в Linux по умолчанию при инсталляции системы (и то какая это часть — зависит от конкретного дистрибутива Linux). Другая часть может быть установлена из стандартного репозитория дистрибутива типовым способом (apt, dnf и другие). Я, за очень редкими исключениями, не буду называть никаких утилит и проектов, отсутствующих в стандартных репозиториях (и вам советую того же и не обольщаться сторонними источниками). В тех редких случаях, когда речь будет идти о сборке из программных исходных кодов, это будет оговариваться явно.

Сетевые инструменты

*«Купила самокат. Оказалось, он так называется не потому что он сам катится, а потому что я должна сама его катить. Продам самокат.»
Из Интернет*

Важнейшими характеристиками сетевой подсистемы хоста являются конфигурация сетевых интерфейсов и таблица маршрутизации ядра, которая полностью и однозначно определяет направления распространения трафика между интерфейсами, например:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
group default qlen 1000
    link/ether a0:1d:48:f4:93:5c brd ff:ff:ff:ff:ff:ff
3: wlo1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT
group default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
4: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN
mode DEFAULT group default
    link/ppp
```

```
$ route -n
```

Kernel IP routing table

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.1.1	0.0.0.0	UG	1024	0	0	em1
80.255.73.34	0.0.0.0	255.255.255.255	UH	0	0	0	ppp0
192.168.1.0	0.0.0.0	255.255.255.0	U	0	0	0	em1
192.168.1.0	0.0.0.0	255.255.255.0	U	0	0	0	wlo1

Таблицы маршрутизации для IPv4 и IP v6 различные

Несоответствие таблицы роутинга состояниям сетевых интерфейсов (что весьма часто случается при экспериментах и отладке сетевых модулей ядра) — наиболее частая причина отличия поведения сети от ожидаемого (картина восстанавливается соответствующими командами route, добавляющими или удаляющими направления в таблицу). Самое краткое и **исчерпывающее** описание работы TCP/IP сети (лучшее из известных автору) дал У. Р. Стивенс:

1. IP-пакеты (создающиеся на хосте или приходящие на него снаружи), если они не предназначены данному хосту, ретранслируются в соответствии с одной из строк таблицы роутинга на основе IP адреса **получателя**.
2. Если ни одна строка таблицы не соответствует адресу получателя (подсеть или хост), то пакеты ретранслируются в интерфейс, который обозначен как интерфейс по умолчанию, который **всегда** присутствует в таблице роутинга (интерфейс с Destination равным 0.0.0.0 в примере показанном выше).
3. Пакет, пришедший с некоторого интерфейса, **никогда** не ретранслируется в этот же интерфейс.

По этому алгоритму всегда можно разобрать картину происходящего в системе с любой самой сложной конфигурацией интерфейсов.

Основные инструменты управления сетевыми **интерфейсами** (ip, ifconfig, route, NetworkManager, nmcli ...) были обсуждены ранее. Они применимы как для реальных, так и для виртуальных сетевых интерфейсов, например, разнообразных VPN (Virtual Private Network):

```
$ ifconfig
...
cipsec0  Link encap:Ethernet  HWaddr 00:0B:FC:F8:01:8F
         inet addr:192.168.27.101  Mask:255.255.255.0
         inet6 addr: fe80::20b:fcff:fef8:18f/64 Scope:Link
         UP RUNNING NOARP  MTU:1356  Metric:1
         RX packets:4 errors:0 dropped:3 overruns:0 frame:0
         TX packets:18 errors:0 dropped:5 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:538 (538.0 b)  TX bytes:1670 (1.6 KiB)
...
```

Здесь показан виртуальный интерфейс (виртуальная частная сеть, VPN) созданный программными средствами (Cisco Systems VPN Client) от Cisco Systems (cipsec0), работающий через один из физических каналов хоста. Показательно, что если тот же VPN-канал создать «родными» Linux средствами OpenVPN (с помощью NetworkManager) к тому же самому удалённому серверу-хосту, то мы получим совершенно другой (и даже, если нужно, ещё один **дополнительно**, параллельно) сетевой интерфейс — различия в интерфейсах обусловлены **модулями ядра**, которые их создавали:

```
$ ifconfig
...
tun0     Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
         inet addr:192.168.27.112  P-t-P:192.168.27.112  Mask:255.255.255.0
         UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1412  Metric:1
         RX packets:13 errors:0 dropped:0 overruns:0 frame:0
         TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:500
         RX bytes:1905 (1.8 KiB)  TX bytes:883 (883.0 b)
```

Кроме базовых инструментов сети (наблюдения, диагностики, тестирования и управления), для отработки сетевых проектов предоставляется (и создаются всё новые) очень широкий набор инструментальных средств³.

Инструменты наблюдения и диагностики

Поскольку представление сетевых интерфейсов достаточно мудрёно и принципиально отличается от символьных или блочных устройств (в /dev), то при отработке модулей ядра поддержки сетевых средств (да и других сетевых компонент пространства пользователя)

³ Число самих сетевых команд Linux само по себе соизмеримо с общим числом всех остальных «не сетевых» команд, связанных с управлением операционной системой.

используется совершенно особое множество **команд-утилит**. Их мы используем для контроля, диагностики и управления сетевыми интерфейсами. Набор сетевых утилит, используемых в сетевой разработке — воистину огромен! Ниже мы только назовём некоторые из них, без которых такая работа просто невозможна...

Простейшими инструментами **диагностики** в нашей отработке примеров будет посылка «пульсов» — тестирующих ICMP пакетов **ping** (проверка достижимости хоста) и **traceroute** (трасса и задержки прохождения промежуточных хостов на трассе маршрута):

```
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=51 time=57.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=51 time=58.5 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 57.374/57.959/58.544/0.585 ms

$ host ya.ru
ya.ru has address 77.88.55.242
ya.ru has address 5.255.255.242
ya.ru has IPv6 address 2a02:6b8::2:242
ya.ru mail is handled by 10 mx.yandex.ru.
```

Трасса к **ya.ru** (показаны только несколько начальных переходов):

```
$ traceroute 5.255.255.242
traceroute to 5.255.255.242 (5.255.255.242), 30 hops max, 60 byte packets
 1 GPON (192.168.1.3) 3.462 ms 3.361 ms 3.368 ms
 2 10.0.53.1 (10.0.53.1) 4.625 ms 4.633 ms 4.577 ms
 3 172.16.1.254 (172.16.1.254) 4.626 ms 4.601 ms 4.593 ms
 4 194-242-96-153.pitline.net (194.242.96.153) 4.573 ms 6.107 ms 4.513 ms
 5 gw0.g50.kv.pitline.net (185.43.248.106) 12.061 ms 12.067 ms 12.029 ms
...
```

Всё время, как уже было отмечено, появляются всё новые и новые инструменты, иногда эквивалентные существующим, но с новыми нюансами функционирования (сравните старый **ifconfig** с новым **ip**). Как пример: пакеты **iputils-arping** и **iputils-tracpath** (потребуют стандартной установки посредством **apt**):

```
$ arping -c3 192.168.1.138
ARPING 192.168.1.138 from 192.168.1.14 eno1
Unicast reply from 192.168.1.138 [90:1B:0E:2B:FE:3A] 4.730ms
Unicast reply from 192.168.1.138 [90:1B:0E:2B:FE:3A] 3.909ms
Unicast reply from 192.168.1.138 [90:1B:0E:2B:FE:3A] 3.929ms
Sent 3 probes (1 broadcast(s))
Received 3 response(s)

$ tracpath 5.255.255.242
1?: [LOCALHOST] pmtu 1500
1: GPON 3.540ms
1: GPON 3.339ms
2: 10.0.53.1 5.508ms
3: 172.16.1.254 5.274ms
4: 194-242-96-153.pitline.net 5.310ms
5: gw0.g50.kv.pitline.net 11.312ms
...
```

Здесь **arping** тестирует достижимость используя протокол ARP (вместо ICMP), а **tracpath** — детализирует трассу но с использованием протокола UDP вместо ICMP.

Ещё одно представление сетевых интерфейсов и отображение статистики использования их в динамике:

```
$ netstat -i
Kernel Interface table
Iface      MTU Met      RX-OK RX-ERR RX-DRP RX-OVR      TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0       1500  0          0      0      0      0        0      0      0      0  0 BMU
lo         16436  0       5508      0      0      0       5508      0      0      0  0 LRU
wlan0      1500  0     154771      0      0      0     165079      0      0      0  0 BMRU
```

Установленные TCP соединения:

```
$ netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address          State
tcp      0      0 notebook.localdomain:56223 2ip.ru:http              TIME_WAIT
tcp      0      0 notebook.localdomain:45804 178-82-198-81.dynamic:31172 ESTABLISHED
tcp      0      0 notebook.localdomain:48314 c-76-19-81-120.hsd1.ct:9701 ESTABLISHED
tcp      0      0 notebook.localdomain:56228 2ip.ru:http              TIME_WAIT
tcp      0      0 notebook.localdomain:56220 2ip.ru:http              TIME_WAIT
tcp      0      0 notebook.localdomain:41762 mail.ukrpost.ua:imap     ESTABLISHED
tcp      0      0 notebook.localdomain:46302 bw-in-f16.1e100.net:imaps ESTABLISHED
tcp      0      0 notebook.localdomain:56222 2ip.ru:http              TIME_WAIT
tcp      0      0 notebook.localdomain:ssh   192.168.1.20:57939       ESTABLISHED
tcp      0      0 notebook.localdomain:56204 2ip.ru:http              TIME_WAIT
tcp      0      0 notebook.localdomain:48861 mail1.ks.pochta.ru:imap  ESTABLISHED
```

(Многие из опции просмотра `netstat` потребуют прав `root`.)

Диагностика и управление таблицей разрешения MAC адресов в адреса IP (этот механизм определён только для IPv4):

```
$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.20             ether    f4:6d:04:60:78:6f   C                    wlan0
192.168.1.1              ether    94:0c:6d:a5:c1:1f   C                    wlan0
```

Утилиты `nslookup`, `host`, `dig` используются для работы с DNS серверами разрешения Интернет имён и IP адресов. Такое многообразие альтернативных утилит определяется, наверное, фундаментальностью этого процесса в ходе работы всемирной сети. Примеры запросов к DNS на прямое и обратное разрешение имени:

```
$ nslookup fedora.com
Server:      192.168.1.1
Address:     192.168.1.1#53

Non-authoritative answer:
Name: fedora.com
Address: 174.137.125.92

$ nslookup 174.137.125.92
Server:      192.168.1.1
Address:     192.168.1.1#53

Non-authoritative answer:
92.125.137.174.in-addr.arpaname = mdnh-siteboxparking.phl.marchex.com.

Authoritative answers can be found from:
125.137.174.in-addr.arpa    nameserver = c.ns.marchex.com.
```



```
125.137.174.in-addr.arpa    nameserver = d.ns.marchex.com.
125.137.174.in-addr.arpa    nameserver = a.ns.marchex.com.
125.137.174.in-addr.arpa    nameserver = b.ns.marchex.com.
```

Вторым (необязательным) параметром команды `nslookup` может быть IP адрес DNS-сервера, через который требуется выполнить разрешение имён (при его отсутствии будет использована последовательность DNS, конфигурированных в системе по умолчанию).

```
$ host fedora.com
fedora.com has address 86.105.245.69

$ host 86.105.245.69
69.245.105.86.in-addr.arpa domain name pointer 86-105-245-69.haip.transip.net.

$ dig fedora.com
; <<>> DiG 9.18.12-0ubuntu0.22.04.1-Ubuntu <<>> fedora.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31162
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;fedora.com.                IN      A

;; ANSWER SECTION:
fedora.com.                42      IN      A      86.105.245.69

;; Query time: 0 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Wed Apr 26 14:14:25 EEST 2023
;; MSG SIZE rcvd: 55
```

Для **анализа трафика** разрабатываемого сетевого интерфейса вам безусловно потребуются что-то из числа известных утилит **сетевых снифферов**, таких как `tcpdump` (<http://www.tcpdump.org/>), или её GUI эквивалент `Wireshark` (<http://www.wireshark.org/>). Посмотрим как будет выглядеть результат `tcpdump` для вот такого сетевого интерфейса (p7p1):

```
$ ip addr show dev p7p1
3: p7p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:9e:02:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global p7p1
    inet6 fe80::a00:27ff:fe9e:202/64 scope link
    valid_lft forever preferred_lft forever
```

Здесь мы можем рассмотреть полученный в `tcpdump` дамп сетевого трафика (показано только начало) при выполнении операции `ping` на этот интерфейс с внешнего хоста LAN:

```
$ sudo tcpdump -i p7p1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on p7p1, link-type EN10MB (Ethernet), capture size 65535 bytes
08:57:53.070217 ARP, Request who-has 192.168.56.101 tell 192.168.56.1, length 46
08:57:53.070271 ARP, Reply 192.168.56.101 is-at 08:00:27:9e:02:02 (oui Unknown), length 28
08:57:53.070330 IP 192.168.56.1 > 192.168.56.101: ICMP echo request, id 2478, seq 1,
length 64
08:57:53.070373 IP 192.168.56.101 > 192.168.56.1: ICMP echo reply, id 2478, seq 1, length
64
08:57:54.071415 IP 192.168.56.1 > 192.168.56.101: ICMP echo request, id 2478, seq 2,
length 64
```

08:57:54.071464 IP 192.168.56.101 > 192.168.56.1: ICMP echo reply, id 2478, seq 2, length 64

...

Видно работу ARP механизма разрешения IP адресов в локальной сети (начало протокола), и приём и передачу IP пакетов (тип протокола ICMP).

С помощью программы tcpdump, формируя сложные **условия фильтра** отбора пакетов для протокола (по сетевым интерфейсам, протоколам, адресам источника и получателя...) можно локализовать проблемы и изучить практически любой сетевой обмен.

Альтернативой (функционально) tcpdump, но уже с графическим интерфейсом (GUI) является программа Wireshark.

Утилита Wireshark, в отличие от tcpdump, содержит, кроме того, большое количество парсеров для сетевых пакетов (например можно прослушать не закодированные видео/аудио поток, передающийся в пакетах протокола RTP при SIP соединениях в IP-телефонии), а также позволяет добавлять собственные парсеры для своих пакетов. Это очень полезно, например, если содержимое пакета закодировано (encrypted), что очень распространено в коммерческих проектах.

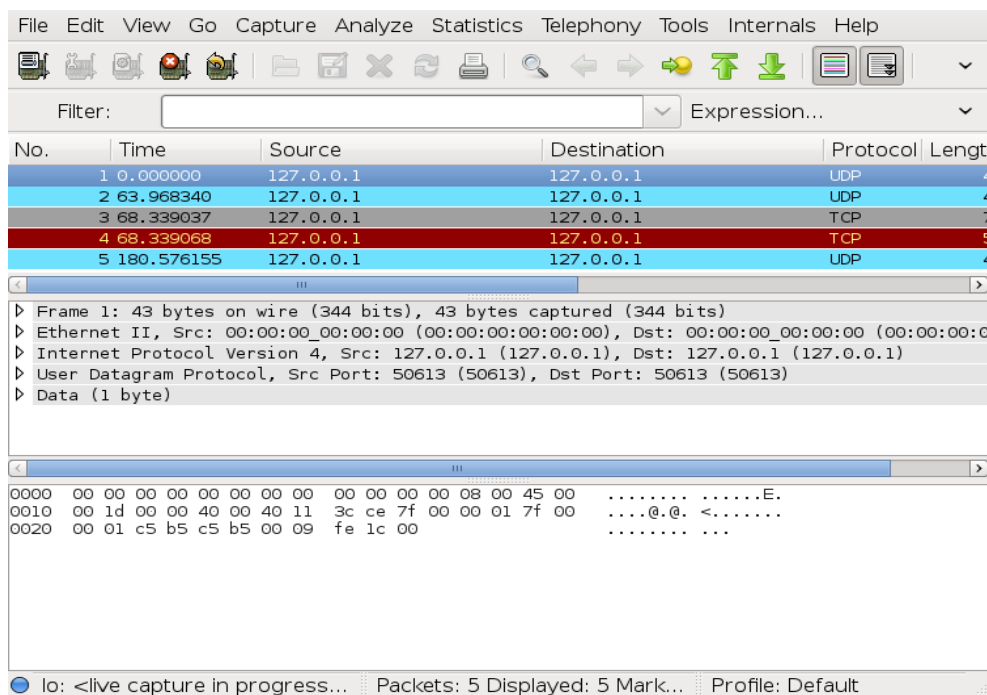


Рис. 2.1 Протокол сеанса Wireshark

Инструменты тестирования и управления

В системе Linux (POSIX-совместимой, UNIX-like) наилучшими инструментами тестирования являются сами стандартные утилиты POSIX/GNU, такие как echo, cat, cp и другие (в этом проявляется симметричность и ортогональность систем UNIX). Для тестирования и отладки сетевых модулей ядра также хорошо бы предварительно определиться с набором тестовых инструментов (утилит), которые также были бы относительно стандартизованы (или широко используемые), и которые позволяли бы проводить тестирование быстро, достоверно и с наименьшими затратами.

Один из удачных вариантов тестеров могут быть утилиты передачи файлов по протоколу SSH — sftp и scp. Обе утилиты копируют указанный (по URL) **сетевой файл**. Разница состоит в том, что sftp требует указания только источника и копирует его в текущий каталог, а для

scp указывается и источник и приёмник (и каждый из них может быть сетевым URL, таким образом допускается выполнение копирования и из 3-го, стороннего узла):

```
$ sftp olej@192.168.1.9:/home/olej/YYY
olej@192.168.1.137's password:
Connected to 192.168.1.9.
Fetching /home/olej/YYY to YYY
/home/olej/YYY                               100% 98MB 10.9MB/s 00:09

$ scp olej@192.168.1.137:/boot/initramfs-3.6.11-5.fc17.i686.img img1
olej@192.168.1.137's password:
initramfs-3.6.11-5.fc17.i686.img              100% 18MB 17.6MB/s 00:01
```

Один из лучших инструментов сетевого программиста для тестирования и отладки — это утилита Netcat (имя исполнимого файла nc). Подобно утилите cat она позволяет отправлять (или получать) байтовый поток, но не в поток ввода-вывода, а в сетевой сокет. У этой утилиты есть множество возможностей (описанных в man), в частности она позволяет как передавать данные в сеть (клиент), так и ожидать принимаемые эти данные из сети (сервер, с опцией -l). Простейший пример использования nc может быть показан, если мы в одном терминале запустим экземпляр nc в режиме прослушивания сокета (**сервер**, TCP порт 1234), того, что будет вводиться с терминала другим экземпляром nc, работающим в режиме **клиента**:

```
$ nc -l 1234
входная строка
11111111111111
22222222222
333333
^C
```

А вот терминал с которого та же программа nc работала как клиент, копирующий в сетевой сокет ввод с клавиатуры (то, что мы и получаем на стороне сервера):

```
$ nc 127.0.0.1 1234
входная строка
11111111111111
22222222222
333333
```

Пример упрощён и схематичен, но многочисленными опциями запуска nc можно варьировать: адреса сетевых узлов, протоколы (TCP по умолчанию и UDP при опции -u), порты и многое другое. Это делает утилиту универсальным **отладочным** инструментом практически неограниченных возможностей. Во многих случаях, при отладочных работах, nc может заменить протокол и утилиту telnet рассматриваемые далее.

P.S. Обратите внимание (это важно и может сэкономить много напрасно потерянного времени), что в современных дистрибутивах может быть (в пакетной системе, в виде пакетов) **две** совершенно разные реализации, названные: традиционный Netcat и Netcat BSD. Более того, по умолчанию может устанавливаться именно Netcat BSD, а традиционный Netcat потребует ручной установки. Реализации **несовместимы** ни по командному интерфейсу, ни по сетевому **протоколу**! Поскольку nc предназначен для связывания разных сетевых хостов — следите чтобы на разных концах работали сходные реализации nc (это можно выяснить запустив утилиту с опцией -h).

Две реализации допускают **одновременную** установку в систему, а быстрое переключение текущей используемой версии обеспечивает система управления альтернативами Linux:

```
$ galternatives
...
```

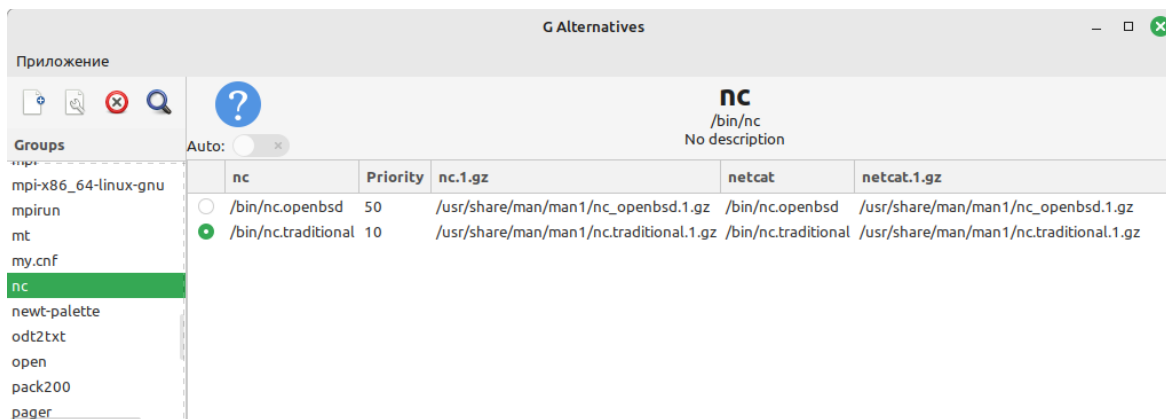


Рис. 2.2 Выбор (динамический) реализации nc

Или то же самое, но используя консольную (CLI) утилиту управления альтернативами Linux:

```
$ update-alternatives --display nc
```

nc - ручной режим

лучшая версия ссылки — /bin/nc.openbsd

ссылка сейчас указывает на /bin/nc.traditional

ссылка nc — /bin/nc

подчинённая nc.1.gz — /usr/share/man/man1/nc.1.gz

подчинённая netcat — /bin/netcat

подчинённая netcat.1.gz — /usr/share/man/man1/netcat.1.gz

/bin/nc.openbsd — приоритет 50

подчинённый nc.1.gz: /usr/share/man/man1/nc.openbsd.1.gz

подчинённый netcat: /bin/nc.openbsd

подчинённый netcat.1.gz: /usr/share/man/man1/nc.openbsd.1.gz

/bin/nc.traditional — приоритет 10

подчинённый nc.1.gz: /usr/share/man/man1/nc.traditional.1.gz

подчинённый netcat: /bin/nc.traditional

подчинённый netcat.1.gz: /usr/share/man/man1/nc.traditional.1.gz

Как легко отсюда видеть, сами оригинальные программы-утилиты устанавливаются в систему под именами nc.traditional и nc.openbsd:

```
$ nc.traditional -h
```

[v1.10-47]

connect to somewhere: nc [-options] hostname port[s] [ports] ...

listen for inbound: nc -l -p port [-options] [hostname] [port]

...

```
$ nc.openbsd -h
```

OpenBSD netcat (Debian patchlevel 1.218-4ubuntu1)

usage: nc [-46CDdFhklNnrStUuvZz] [-I length] [-i interval] [-M ttl]

[-m minttl] [-O length] [-P proxy_username] [-p source_port]

[-q seconds] [-s sourceaddr] [-T keyword] [-V rtable] [-W recvlimit]

[-w timeout] [-X proxy_protocol] [-x proxy_address[:port]]

[destination] [port]

...

Но конечно же, основным инструментом сетевого управления для нас остаются утилита ip (пакет Linux iproute2), или её более давние эквиваленты ipconfig и route (пакет Linux net-tools, это инструментарий присутствующий во всех более ранних вариантах UNIX, в некоторых дистрибутивах Linux может считаться устаревшим, и требующим ручной установки средствами менеджера apt).

Средствами утилиты `ip` мы можем, например (в отличие от использования для диагностики, использование для управления потребует прав `root`, естественно):

- Принудительно создавать и удалять сетевые интерфейсы (`ip-link(8)`):

```
# ip link add xxx type bridge
# ip link del dev xxx
```
- Давать интерфейсам дополнительные (алиасные) имена:

```
# ip link property add dev eth0 altname eno2
# ip link property del dev eno2 altname eno1
```
- Присваивать уже существующим интерфейсам дополнительные IP (алиасные) адреса (`ip-address(8)`):

```
# sudo ip address add 10.0.1.5 dev eno1
```
- Добавлять (у удалять, соответственно) записи в таблицу роутинга ядра:

```
# ip route add default via 192.168.1.3
# ip -4 route add 0.0.0.0 via 192.168.1.3
# ip -4 route add 10.0.0.0 via 192.168.1.11
```

Практически все те же действия можно осуществлять (когда удобнее или привычнее) и парой команд в «старой» нотации: `ipconfig` и `route`.

Выше показан только ограниченный набор возможностей управления сетевой подсистемой, но мы ещё увидим множество примеров подобных команд в дальнейшем рассмотрении.

Сервисы сети и `systemd`

Практически все далее рассматриваемые сетевые сервисы (и ещё множество не рассматриваемых) построены на клиент-сервисной архитектуре: клиенты запрашивают обслуживания у сервера по сетевому протоколу. Структура практически всех свободно развиваемых сетевых проектов (Linux — операционная система свободная и с открытыми исходными кодами всех её продуктов!), из-за их сетевой специфичности, совершенно одинакова, и состоит из 3-х компонент основных: 1). протокол xxx, 2). клиент xxx, 3). сервер xxx. Клиент, как правило, готов для использования, хотя иногда и требует предварительной конфигурации. Запуск серверов, в современных дистрибутивах (в подавляющем большинстве из них) обеспечивается средствами `systemd`.

Схема установки и запуска в эксплуатацию всех сетевых проектов однотипна и единообразна, и поэтому, чтобы не повторять её далее каждый раз в деталях и относительно каждого нового сервиса, вспомним единую схему обращения с сетевыми дистрибутивными пакетами, общую для подавляющего большинства сетевых проектов:

1. Установка выбранного проекта (условно назовём его xxx) из сетевого репозитория своего дистрибутива:

```
$ apt install xxx
```


Или для RPM дистрибутивов:

```
$ dnf install xxx
```
2. Конфигурируем сервер устанавливаемого протокола xxx. Местоположение конфигурационных файлов, обычно, уточняется во время установки, смотрим для начала пути и файлы `/etc/default/xxx`, `/etc/xxx.conf`, `/etc/xxx.d/xxx.conf`
3. Запускаем сервер средствами `systemd`:

```
# sudo systemctl start xxx
```
4. Проверяем статус запущенного сервера на предмет отсутствия ошибок:

```
$ systemctl status xxx
```


...

Повторяем снова начиная с п.2 до тех пор, пока этот шаг не проходим без ошибок.

5. Если планируем долгосрочно использовать сервер протокола XXX, устанавливаем для него автоматический запуск при загрузке операционной системы:

```
# sudo systemctl enable xxx
```

SSH

Протокол SSH (Secure SHell) пришёл на смену протоколу telnet и, как понятно и из самого названия, первоначальное и основное его предназначение было — организация защищённых шифрованием в канале терминальных сессий (в защищённости и состоит главное отличие от telnet). Но в дальнейшем SSH так «пришёлся ко двору», что на него было возложено множество других возможностей (только малую часть которых мы сейчас и посмотрим).

Для Linux существует несколько реализаций для протокола SSH. Но самая широко используемая из них — это openssh. Клиент openssh-client установлен по умолчанию в любом дистрибутиве Linux — это основной и любимый инструмент любого сетевого администратора. Сервер же из того же проеукта openssh-server (и по желанию openssh-sftp-server) нужно установить дополнительно (что я рекомендовал бы делать первым действием после инсталляции любого дистрибутива Linux):

```
$ aptitude search openssh- | grep ^i
i openssh-client - клиент протокола SSH, для защищённого удалённого доступа
i openssh-server - Сервер SSH для безопасного доступа с удалённых компьютеров
i openssh-sftp-server - secure shell (SSH) sftp server module, for SFTP access from
remote machines
```

После установки сервер запускается через systemd стандартным способом:

```
$ sudo systemctl start sshd
$ systemctl status sshd
• ssh.service - OpenBSD Secure Shell server
   Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2023-04-23 07:07:20 EEST; 6h ago
     Docs: man:sshd(8)
           man:sshd_config(5)
   Process: 1313 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
  Main PID: 1331 (sshd)
    Tasks: 1 (limit: 115786)
   Memory: 3.2M
      CPU: 25ms
   CGroup: /system.slice/ssh.service
           └─1331 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"
```

```
anp 23 07:07:20 R420 systemd[1]: Starting OpenBSD Secure Shell server...
anp 23 07:07:20 R420 sshd[1331]: Server listening on 0.0.0.0 port 22.
anp 23 07:07:20 R420 sshd[1331]: Server listening on :: port 22.
anp 23 07:07:20 R420 systemd[1]: Started OpenBSD Secure Shell server.
```

```
$ pgrep sshd
1331
19252
19262
```

Что означают последние 2 строчки вывода (1-я это понятно, это PID стартовавшего сервера sshd, совпадающий и с значением в статусе systemd показанным выше). А 2 последние строчки: а). соответствуют каждая одной подключенной клиентской сессии, б). указывают на то что SSH-сервер это fork-ающий сервер, в). и каждая из этих строк это PID дочернего процесса для обслуживания своей терминальной сессии.

И я сразу же после установки сервера SSH разрешил бы через `systemd` его автоматический запуск после перезагрузки системы:

```
$ sudo systemctl enable sshd
```

Синтаксис клиента SSH⁴:

```
$ ssh olej@192.168.1.13
olej@192.168.1.13's password:
olej@R420:~$ hostname
R420
olej@R420:~$ exit
выход
Connection to 192.168.1.13 closed.
```

Или так:

```
$ ssh -l olej 192.168.1.13
olej@192.168.1.13's password:
...
```

Показано это для того чтобы подчеркнуть: клиент SSH **всегда запросит** пароль пользователя (это пароль для этого имени пользователя, зарегистрированного на удалённом хосте, а не где-то в недрах самой системы SSH). (Это если на вашем удалённом хосте не разрешён беспарольный SSH доступ, но разрешать такой доступ — это последнее дело из соображений безопасности.) Нет никакой синтаксической формы запроса SSH-коннекта в которой явно указывался бы пароль одновременно с именем (через знак двоеточия как это допускают многие сетевые протоколы и команды, например, `ftp`). Это сделано для предотвращения любой видимости пароля при наборе его на терминале.

SSH допускает использование адресов как IPv4 так и IPv6 (у клиента SSH даже есть уточняющие опции `-4` и `-6`, но он распознаёт их и по правильной записи IP адреса). Вот, например, сеанс SSH в локальной сети по локальному (`fe80::/64`) адресу IPv6 (как уже упоминалось ранее, для локального адреса должен быть обязательно указано имя интерфейса или его номер, в данном случае номер):

```
$ ssh olej@fe80::522d:d0bd:b221:a526%2
olej@fe80::522d:d0bd:b221:a526%eno1's password:
Linux nvme 5.10.0-21-amd64 #1 SMP Debian 5.10.162-1 (2023-01-21) x86_64
```

```
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
```

```
Last login: Fri May 5 13:03:47 2023 from 192.168.1.14
```

```
olej@nvme:~$ exit
```

```
выход
```

```
Connection to fe80::522d:d0bd:b221:a526%eno1 closed.
```

В конфигурации SSH сервера тот или иной протокол может быть разрешён (по умолчанию обычно разрешены оба) или запрещён:

```
$ grep AddressFamily /etc/ssh/sshd_config
# AddressFamily inet      # только IPv4
# AddressFamily inet6     # только IPv6
# AddressFamily any       # по умолчанию, используются оба
```

⁴ Я везде в примерах показываю в примерах своё регистрационное имя пользователя `olej` (чтобы не городить каких-то условных конструкций типа `<user>` не похожих на синтаксис реальных команд). Вы, естественно, должны подставить вместо него своё регистрационное имя пользователя в своей системе.

При подключении к удалённому серверу в Интернет по IPv6 (в отличие от показанного подключения в локальной сети), вас скорее всего, ожидает такая вот неожиданность (только с заметной паузой по времени):

```
$ ssh 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
ssh: connect to host 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 port 22: Connection timed out
```

Такой эффект связан не с конфигурацией сервера SSH, а с тем, что фаервол, практически всегда запущенный на серверах доступных в Интернет, позволяет IPv4 на 22 порт TCP, но запрещает IPv6 доступ. Если мы хотим иметь доступ и IPv6, то, зайдя на сервер по SSH для IPv4, выполним там:

```
# sudo ufw allow in on tun0 to any port 22
Rule added
Rule added (v6)
# ufw status numbered | grep "22 (v6)"
[14] 22 (v6) on tun0          ALLOW IN    Anywhere (v6)
```

И вот теперь:

```
$ ssh 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
The authenticity of host '221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
(221:58c9:9a6:99be:f3d:c1ac:2b5b:9771)' can't be established.
ED25519 key fingerprint is SHA256:tUEdylSRCHZzp1T6ds7GmMI6L0EBXy95rAXPMoXEic.
This host key is known by the following other names/addresses:
  ~/.ssh/known_hosts:57: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '221:58c9:9a6:99be:f3d:c1ac:2b5b:9771' (ED25519) to the list of
known hosts.
olej@221:58c9:9a6:99be:f3d:c1ac:2b5b:9771's password:
Linux 277938.local 4.19.0-23-amd64 #1 SMP Debian 4.19.269-1 (2022-12-20) x86_64
...
```

И попросим сервер показать открытые там на нём сессии:

```
$ who
olej      tty1      2023-03-29 11:16
olej      pts/0      2023-05-06 12:43 (193.28.177.65)
olej      pts/2      2023-05-06 12:43 (193.28.177.65)
olej      pts/3      2023-05-06 13:01 (21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f)
```

Иногда нас может интересовать в клиенте не возможность установить долговременную терминальную сессию, а разово запустить удалённо программу (или несколько программ), с отображением результата на наш локальный терминал. Тогда формат команды немного изменится — последним необязательным параметром мы задаём команду, которая должна выполняться, и далее, возможно, параметры и опции самой этой команды:

```
$ ssh olej@192.168.1.13 df -h
olej@192.168.1.13's password:
Файл.система  Размер  Использовано  Дост  Использовано%  Смонтировано в
tmpfs          9,5G      2,1M      9,5G          1% /run
/dev/sda5      109G      45G      60G          43% /
tmpfs          48G       20K      48G          1% /dev/shm
tmpfs          5,0M      4,0K      5,0M          1% /run/lock
/dev/nvme0n1p1 229G      58G     160G          27% /home
/dev/sdb2      229G      83G     135G          39% /home/olej/Загрузки
/dev/sda1      511M      3,3M     508M          1% /boot/efi
/dev/sdc2      910G      19G     846G          3% /mnt/sdc2
/dev/sdc3      576G     140G     408G          26% /mnt/sdc3
tmpfs          9,5G     132K      9,5G          1% /run/user/1000
```


После выполнения удалённо команды в таком формате сессия SSH завершается.

Или, если нам нужно выполнить последовательность нескольких (сколь угодно много) команд, то объединяете из в одну общую строку:

```
$ ssh olej@192.168.1.13 "uptime && free -m"
olej@192.168.1.13's password:
18:59:04 up 11:51, 1 user, load average: 3,74, 2,27, 1,71
              total        used        free      shared  buff/cache   available
Память:      96607         7194       85954         389       3458       88249
Подкачка:     2047           0         2047
```

Примечание: Напомню, что в цепочке (последовательности) команд могут использоваться в качестве разделителя, как варианты, либо а). условный && — выполнять только при успешном завершении предыдущей команды, либо б). безусловный ; — выполнять при любом исходе предыдущей команды.

Возможности системы SSH на сегодня просто необозримы: туннели для других сетевых протоколов, проксирование с одного ресурса на другой и многое другое... Мы коснёмся только некоторых из возможностей, которые очень полезны на практике, но, иногда, недостаточно широко известны.

Передача файлов по SSH

К содружеству утилит SSH позже были добавлены утилиты копирования удалённых файлов через протокол SSH: команды `sftp` и `scp`.

Команда копирует `sftp` удалённый файл на файловую систему локального компьютера на котором мы работаем (или из локального хоста на удалённый). Это немного напоминает протокол FTP, но передача идёт в зашифрованном канале. Для чистоты эксперимента копирование делаю в вновь созданный пустой каталог:

```
$ mkdir tmp && cd tmp
$ ls -l
итого 0
```

Копируем файл по сети ... с сервера находящегося за тысячи километров (конкретно здесь хостинг в Казахстане):

```
$ sftp olej@linux-ru.ru:prefix
olej@linux-ru.ru's password:
Connected to linux-ru.ru.
Fetching /home/olej/prefix to prefix
prefix                                     100%   36    0.2KB/s   00:00
```

Полученный результат:

```
$ ls -l
итого 4
-rwxr-xr-x 1 olej olej 36 apr 23 15:01 prefix
```

Обращаем внимание на несколько необычный синтаксис указания локализации копируемого файла на удалённом хосте: 1). имя сервера отделяется от путевого имени файла двоеточием, 2). путевое имя файла, после двоеточия, отсчитывается не абсолютно (от корня удалённой файловой системы), а от домашнего каталога того пользователя, который регистрируется в команде (перед символом @).

Команда `scp` ещё интереснее: она копирует файл между двумя локациями, каждая из которых может быть удалённой (но может и локальной). В следующем примере мы копируем файл всё с того же весьма удалённого ресурса (за тысячи километров), но не на свой локальный компьютер, а на соседний хост в нашей локальной сети:

```
$ scp olej@linux-ru.ru:prefix olej@192.168.1.11:
```

```
olej@linux-ru.ru's password:
```

На хосте получателя наблюдаем (и тут же контролируем время создания, чтобы убедиться что это только-что полученный файл):

```
$ ls -l prefix
-rwxr-xr-x 1 olej olej 36 apr 23 15:05 prefix
$ date
Вс 23 апр 2023 15:06:05 EEST
```

Команды эти хоть и просты, но обладают возможностями гораздо большими чем те, что мы уже успели посмотреть:

```
$ sftp --help
unknown option -- -
usage: sftp [-46AaCfNpqrv] [-B buffer_size] [-b batchfile] [-c cipher]
          [-D sftp_server_path] [-F ssh_config] [-i identity_file]
          [-J destination] [-l limit] [-o ssh_option] [-P port]
          [-R num_requests] [-S program] [-s subsystem | sftp_server]
          destination

$ scp --help
scp: unknown option -- -
usage: scp [-346ABCOpqRrsTv] [-c cipher] [-D sftp_server_path] [-F ssh_config]
          [-i identity_file] [-J destination] [-l limit]
          [-o ssh_option] [-P port] [-S program] source ... target
```

Из самых очевидных вещей: 1). команды работают как с IPv4 так и с IPv6, 2). если команды выполнять с опцией `-r`, то копируется не отдельный файл, а весь каталог рекурсивно со всем его содержимым.

Но синтаксически просто перенести форму записи команды из IPv4 в IPv6 не получится:

```
$ sftp -6 olej@221:58c9:9a6:99be:f3d:c1ac:2b5b:9771:backup-10.05.2023_19-14.sql
ssh: Could not resolve hostname 221: Address family for hostname not supported
Connection closed.
Connection closed
```

Тут придётся воспользоваться той формой записи IPv6 о которой уже говорилось (адрес заключаем в скобки), например:

```
$ sftp -6 olej@[221:58c9:9a6:99be:f3d:c1ac:2b5b:9771]:backup-10.05.2023_19-14.sql
olej@221:58c9:9a6:99be:f3d:c1ac:2b5b:9771's password:
Connected to 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771.
Fetching /home/olej/backup-10.05.2023_19-14.sql to backup-10.05.2023_19-14.sql
backup-10.05.2023_19-14.sql                                100% 57MB 2.2MB/s
00:26
```

Естественно, что для этих команд на хосте источнике (а для `scp` и на хосте получателя тоже) должен работать сервер SSH.

SSH и mc

Это не достаточно широко известно, про то, что в таком популярном 2-панельном файловом менеджере `mc` (Midnight Commander) операция для панелей (левой и правой), которая названа в меню как «Shell-соединение» — это и есть присоединение этой панели к сетевому хосту по зашифрованному протоколу SSH. Это самый простой и визуальный способ работы с файлами на удалённом сетевом ресурсе⁵.

⁵ И этот способ позволяет создать подключение к файловой системе в операционных системах отличных от Linux: FreeBSD, Solaris OS, Minix 3 ... и даже Windows с некоторыми оговорками (из-за различий в кодировках UNICODE, имена файлов на национальных языках будут искажаться).

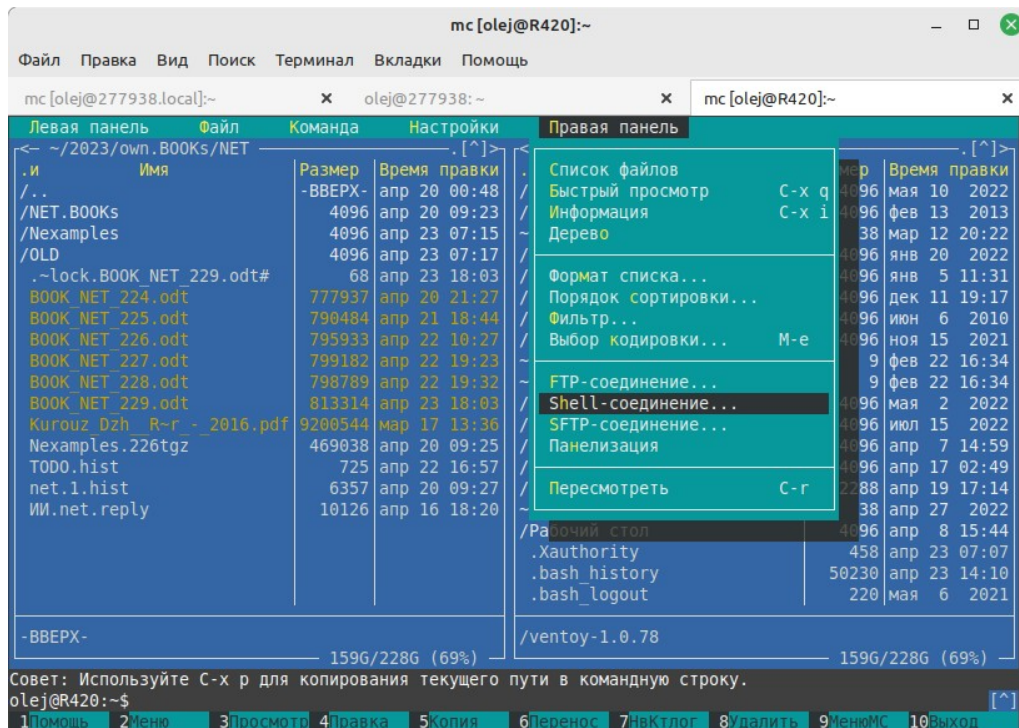


Рис. 2.3 SSH подключение в mc

Тогда, для наглядности, мы можем в правой панели, например, указать соседний хост в локальной сети:

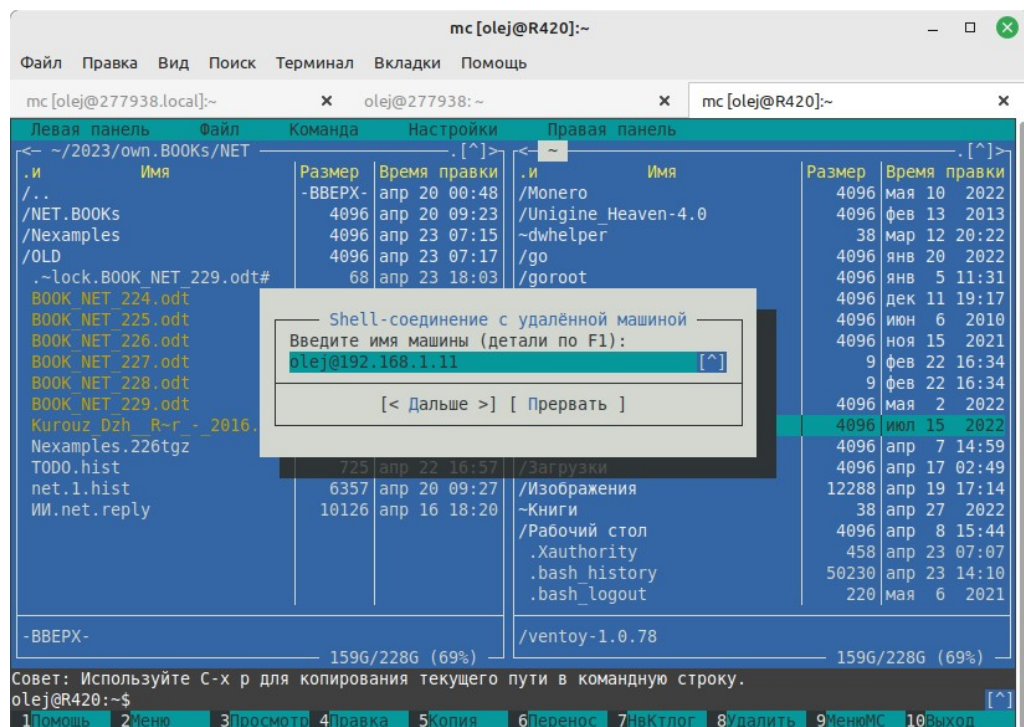


Рис. 2.4 Подключение по локальной сети

А в правой панели, аналогично, можем определить для подключения, например, удалённый (на несколько тысяч километров) сервер в Интернет:

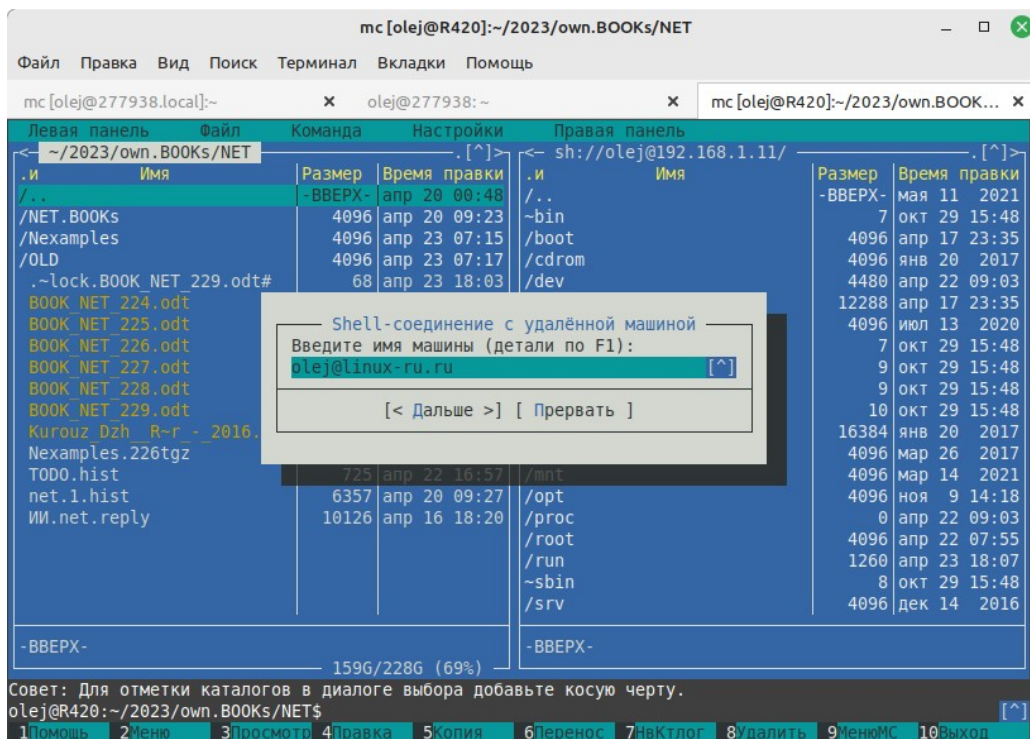


Рис. 2.5 Подключение к дальнему хосту в Интернет

Естественно, и в том и в другом подключении у нас попросят пароли авторизации для указанного имени пользователя. Вот итоговая картина панелей файлового менеджера после осуществления подключений:

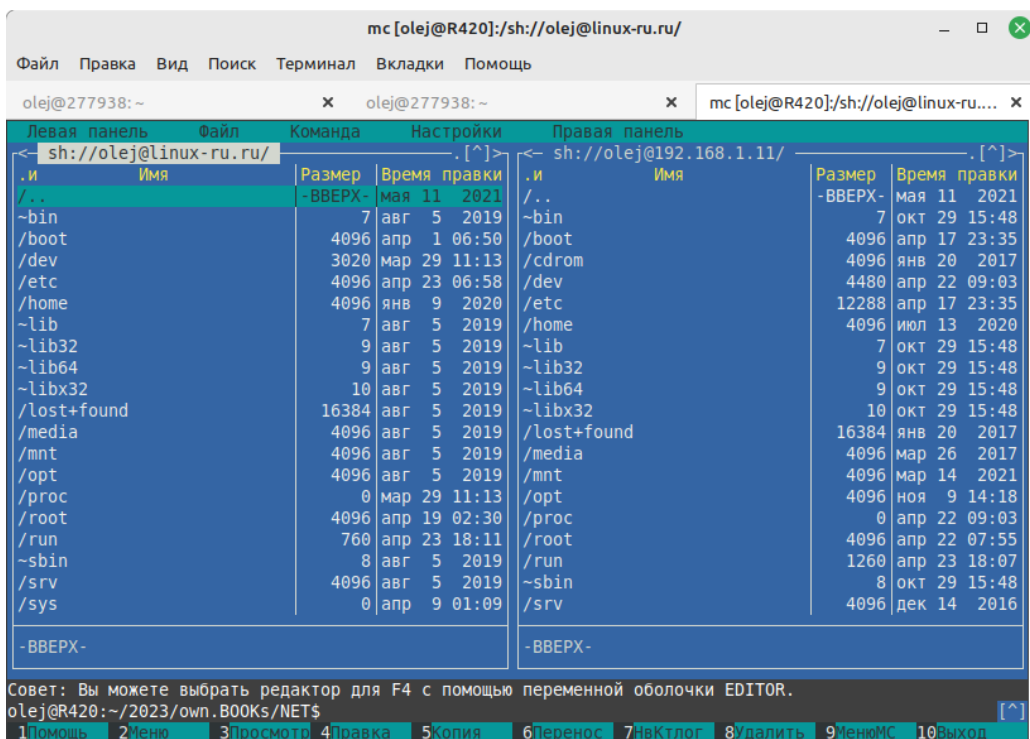


Рис. 2.6 Два подключенных соединения SSH

Теперь мы можем (простыми и привычными действиями, так как делаем это в локальной файловой системе) с визуальным контролем:

- по клавише Tab перемещаться между левой-правой локации;
- свободно перемещаться по файловой иерархии каждой из локаций, пользуясь для этого стрелкам клавиатуры (и Enter);
- копировать (по F5) или перемещать (по F6) файлы между локациями (не взирая на то что их разделяет, возможно, тысячи километров);
- просматривать (по F3) или даже редактировать (по F4) содержимое файлов, находящихся от нас за тысячи километров...

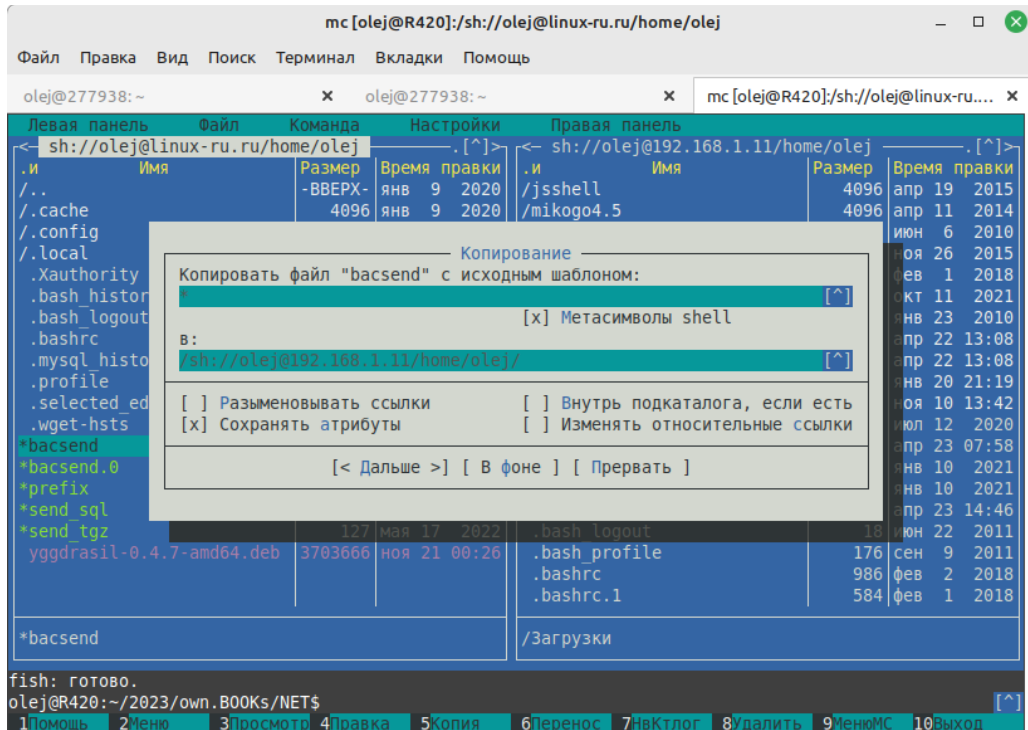


Рис. 2.7 Выполнение операции копирования между двумя удалёнными хостами

Графическая сессия в SSH

Выше было уже вскользь упомянуто о том, что протокол SSH может использоваться как туннель для многих других сетевых протоколов внутри себя. Это очень обширная отдельная тема, но нас здесь интересует частность: SSH может туннелировать, в том числе, и графический протокол X11 (на котором построена вся графика в UNIX/Linux). А это значит что мы можем на (сколь угодно) удалённом сервере запускать любые привычные графические GUI приложения с тем, чтобы их графический экран отображался (и взаимодействовал) на нашем локальном мониторе (локального хоста).

Для того чтобы SSH сессия запускалась с туннелированием протокола X11, команду SSH клиента нужно запускать с опциями `-X`, или даже `-Y`, что предусматривает, во втором случае, **отсутствие** шифрования в SSH протоколе. Второй вариант (`-Y`) делает соединение ещё быстрее, и может применяться в локальных сетях, когда у вас есть доверие к трассе прохождения IP пакетов.

Простейшее GUI приложение выполняющееся удалённо:

```
$ ssh -X olej@192.168.1.13 xclock
olej@192.168.1.13's password:
```

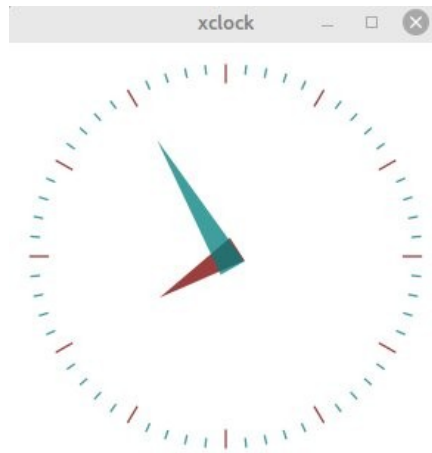


Рис. 2.8 Простейшее удалённое графическое приложение

Более сложное (весьма сложное) GUI приложение, выполняемое удалённо — это может быть, например, менеджер виртуальных машин VirtualBox, когда виртуальные машины будут выполняться на одном хосте LAN, а отображение (десктопов виртуальных машин) направлять на другом:

```
$ ssh -Y olej@192.168.1.13 VirtualBox
olej@192.168.1.13's password:
...
```

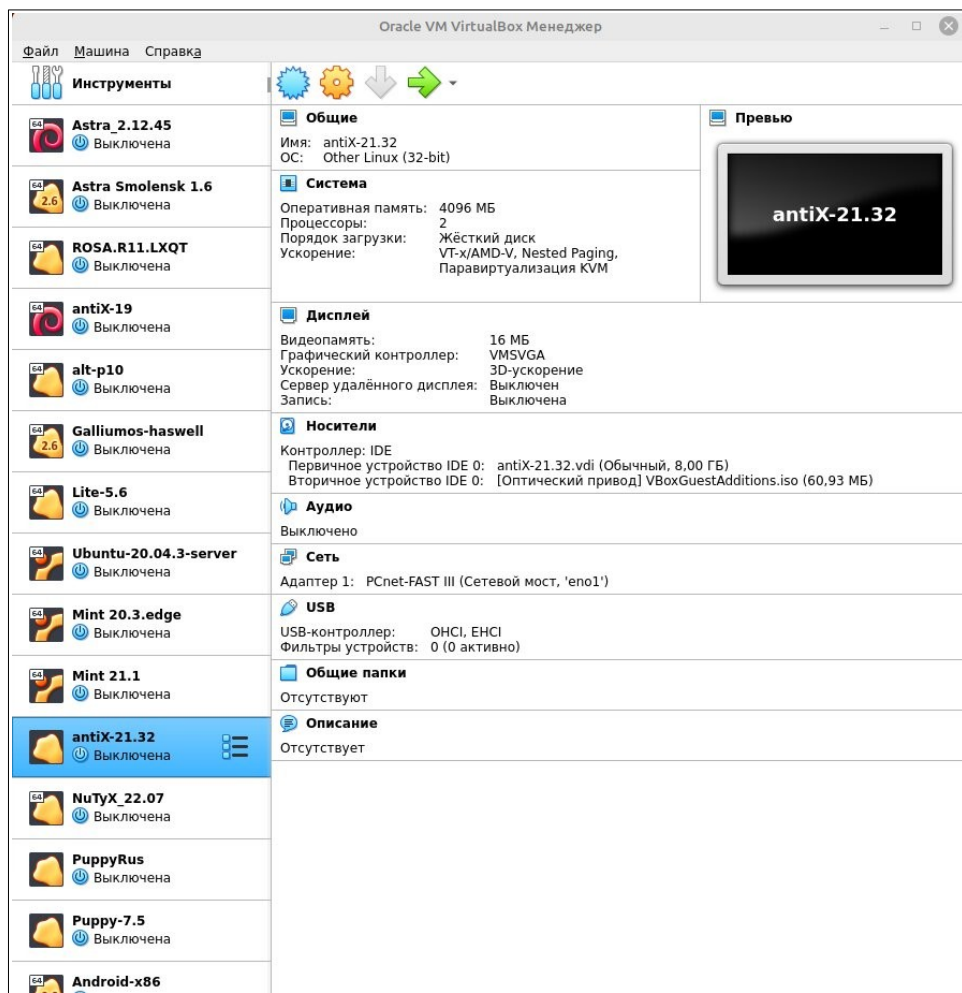



Рис. 2.9 Так виртуальные машины выполняются на удалённом компьютере

SSH в скриптах

Как последнюю из особенностей пользования SSH хотелось бы затронуть возможность использования команд клиента SSH в исполнимых скриптах. Как уже упоминалось ранее, у команды `ssh`, при всём богатстве выбора, нет синтаксической формы, которая позволяла бы записать символьную строку пароля непосредственно в команду — это для того, чтобы такую команду можно было выполнять в составе скрипта, не требующего диалогового вмешательства пользователя. Это бывает здорово нужно при работе в собственной доверенной LAN, или с собственными виртуальными машинами ... и не только.

Для таких целей в Linux (в стандартных репозиториях пакетных систем) имеется такое небольшое приложение, которое нужно отдельно установить:

```
$ sudo apt install sshpass
```

```
Чтение списков пакетов... Готово
```

```
Построение дерева зависимостей
```

```
Чтение информации о состоянии... Готово
```

```
Следующие НОВЫЕ пакеты будут установлены:
```

```
  sshpass
```

```
Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 6 пакетов не обновлено.
```

```
Необходимо скачать 11,3 kB архивов.
```

```
После данной операции объём занятого дискового пространства возрастёт на 31,7 kB.
```

```
Пол:1 http://deb.debian.org/debian buster/main amd64 sshpass amd64 1.06-1 [11,3 kB]
```

```
Получено 11,3 kB за 1с (11,2 kB/s)
```

```
Выбор ранее не выбранного пакета sshpass.  
(Чтение базы данных ... на данный момент установлено 379076 файлов и каталогов.)  
Подготовка к распаковке .../sshpass_1.06-1_amd64.deb ...  
Распаковывается sshpass (1.06-1) ...  
Настраивается пакет sshpass (1.06-1) ...  
Обрабатываются триггеры для man-db (2.8.5-1) ...
```

После этого вы любые команды клиента SSH можете передавать приложению `sshpass` на выполнение. Проще чем объяснять это показать на примере, вот как может выглядеть терминальная сессия:

```
$ sshpass -p 'xyz123' ssh -l vasja -Y 192.168.1.107  
You have new mail.  
Last login: Fri Jul 19 16:23:52 2019 from 192.168.1.103  
$ uname -a  
Linux astra 4.19.0-1-generic #astra1 SMP Wed Mar 20 12:59:21 UTC 2019 x86_64 GNU/Linux  
$ whoami  
vasja  
...
```

Здесь (как должно быть понятно):

- `vasja` — это логин пользователя на хосте 192.168.1.107
- `xyz123` — это пароль к этому логину

Естественно, что в такой же манере для `sshpass` можно передавать **все** синтаксические формы `ssh` рассматриваемые ранее.

DHCP

Все показанные выше примеры относительно использования IP адресов были показаны на примерах фиксированных значений адреса, типа: 192.168.1.11 — такие адреса присваивались интерфейсам вручную. Для локальной сети в которой до десятка хостов это вполне приемлемо. Но что делать с LAN и как её администрировать если в ней сотня хостов? Для этого предложен протокол DHCP (**D**ynamic **H**ost **C**onfiguration **P**rotocol, протокол динамической конфигурации узлов) и реализующие его службы (сервер, клиенты, управление).

Протокол DHCP — это клиент-серверный протокол, который автоматически предоставляет хосту, в момент его загрузки, для него IP-адрес (из определённого пула адресов) и другие связанные сведения о конфигурации, такие как маска подсети и шлюз по умолчанию. Идея DHCP крайне простая:

- Серверу в конфигурациях прописывается один или несколько диапазон IP адресов (пул адресов);
- Для этого диапазона устанавливается единый набор сопутствующей информации (маска, шлюз по умолчанию, DNS сервер подсети);
- На запрос очередного загружающегося клиента сервер DNS выбирает ему свободный (не занятый ещё) IP адрес из пула (если там, конечно, ещё остался незанятый адрес)...
- .. и возвращает его клиенту вместе с комплектом сопутствующей информации для этого диапазона;
- После выделения IP по запросу сервер хранит информацию о нём как о занятом (зарезервированном) ещё некоторое время, называемое временем аренды (связывая эту аренду с MAC адресом клиента которому был возвращён этот IP).

RFC 2131 и 2132 определяют протокол DHCP в качестве стандарта Internet Engineering Task Force (IETF), основанного на более раннем протоколе начальной загрузки (BOOTP)⁶, протокола, через который DHCP и предоставляет много нужных сведений.

DHCP клиент в Linux присутствует изначально в любой инсталляции системы, другой вопрос, что он может быть включен для отдельного сетевого интерфейса, или выключен и тогда никак себя не проявляет.

DHCP сервер менее известен, но это только потому, что он на сегодня встраивается в любой сетевой роутер, даже в простейшие и дешёвые SOHO, домашнего использования, класса. Таким образом, все мы и каждый день пользуемся услугами DHCP сервера просто об этом часто не зная, и его не замечая. Но каждый (почти) роутер на сегодня — это Linux, и на нём работает типовой DHCP сервер.

Но иногда требуется по необходимости и возникает такое желание установить DHCP сервер отдельно на автономном хосте локальной сети (при этом отключив, как правило, встроенный DHCP сервер роутера — 2 и более DHCP сервера в одной LAN это, как правило, трудно диагностируемые проблемы!).

Один из самых популярных в Linux DHCP серверов `isc-dhcp-server` (ранее известный как `dhcp3-server`):

```
$ aptitude search isc-dhcp-server
p  isc-dhcp-server                - ISC DHCP server for automatic IP address
assignment
p  isc-dhcp-server-ldap           - DHCP server that uses LDAP as its backend
```

Стандартный репозиторий — стандартная установка:

```
$ sudo apt install isc-dhcp-server
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Будут установлены следующие дополнительные пакеты:
  libirs-export161 libiscfg-export163 policycoreutils selinux-utils
...
```

После такой установки сервер уже присутствует, но он пока остановлен, что вполне естественно — перед 1-м запуском его нужно конфигурировать:

```
$ systemctl status isc-dhcp-server
● isc-dhcp-server.service - LSB: DHCP server
   Loaded: loaded (/etc/init.d/isc-dhcp-server; generated)
   Active: inactive (dead)
     Docs: man:systemd-sysv-generator(8)
```

После установки есть смысл определить какие сетевые интерфейсы будет прослушивать DHCP сервер и, соответственно, куда он будет отвечать:

```
$ ip l sh
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
DEFAULT group default qlen 1000
    link/ether 90:1b:0e:2b:fe:3a brd ff:ff:ff:ff:ff:ff

# tail -n3 /etc/default/isc-dhcp-server
#       Separate multiple interfaces with spaces, e.g. "eth0 eth1".
INTERFACESv4="enp3s0"
#INTERFACESv6="enp3s0"
```

⁶ BOOTP первоначально создавался и широко используется для начальной загрузки через сеть всяких разнообразных встраиваемых (embedded) и автономных устройств в не обслуживаемом режиме.

Как видите из заготовки этого файла (созданного при установке `isc-dhcp-server`) а). сервер может обслуживать как DHCP4 так и DHCP6, б). можно выборочно указать интерфейсы на хосте со многими интерфейсами и в). по каждому протоколу можно указать по несколько (при необходимости) интерфейсов.

А сами конфигурации сервера находятся (если этот путь не переопределен в файле `/etc/default/isc-dhcp-server`) в `/etc/dhcp/`:

```
# ls -l /etc/dhcp/dhcpd*.conf
-rw-r--r-- 1 root root 3331 окт  4 2022 /etc/dhcp/dhcpd6.conf
-rw-r--r-- 1 root root 3496 окт  4 2022 /etc/dhcp/dhcpd.conf
```

Конфигурационные файлы отлично прокомментированы (как это чаще всего и бывает в сетевых инструментах) и там из значащих полей нужно оставить и/или вписать, например для IPv4, что-то типа:

```
$ grep -v ^# /etc/dhcp/dhcpd.conf | grep -v ^$
default-lease-time 600;
max-lease-time 7200;
subnet 192.168.1.0 netmask 255.255.255.0 {
    option routers 192.168.1.3;
    option subnet-mask 255.255.255.0;
    option domain-name-servers 192.168.1.3,8.8.4.4,1.1.1.1;
    range 192.168.1.25 192.168.1.35;
    range 192.168.1.240 192.168.1.250;
}
```

Здесь некоторые параметры записаны с предшествующим `option`, а другие без. Вот то что записано с `option` — это и будет передаваться клиенту в ответ на его запрос.

Здесь: `default-lease-time` и `max-lease-time` — время аренды выделенного IP в секундах, соответственно по умолчанию и максимальный лимит если клиент запросил явно; параметры подсети, шлюз по умолчанию для неё и список DNS серверов сети; `range` — диапазон IP из которого DHCP сервер, по своему усмотрению, выделяет в аренду IP на запрос клиента. Я специально показал 2 `range` участка чтобы было понятно, что диапазон аренды может быть «рваный» и состоять из нескольких участков.

Таких секций `subnet` может быть одна или несколько (например, для DHCP4 и DHCP6, или для нескольких разных подсетей).

Поскольку на практике конфигурирование DHCP сервера — дело трудоёмкое, хлопотное и требующее большой тщательности, проект предусматривает возможность автономной (без запуска) проверки подготовленного файла конфигурации на отсутствие ошибок синтаксиса.

```
# which dhcpd
/usr/sbin/dhcpd
```

Правим файл конфигурации до тех пор пока не получим что-то подобное (сообщения с отсутствующими ошибками):

```
# dhcpd -t -cf /etc/dhcp/dhcpd.conf
Internet Systems Consortium DHCP Server 4.4.1
Copyright 2004-2018 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/
Config file: /etc/dhcp/dhcpd.conf
Database file: /var/lib/dhcp/dhcpd.leases
PID file: /var/run/dhcpd.pid
```

Вот теперь всё готово чтобы запустить приготовленный сервер (**предупреждение:** если вы не конфигурировали нормально файл `/etc/dhcp/dhcpd6.conf`, а мы по тексту этого не делали

— то прокомментируйте интерфейс IPv6 в /etc/default/isc-dhcp-server, без этого DHCP сервер не стартует, он завершится фатально на старте DHCP6!)

```
$ sudo systemctl start isc-dhcp-server
$ sudo systemctl status isc-dhcp-server
● isc-dhcp-server.service - LSB: DHCP server
   Loaded: loaded (/etc/init.d/isc-dhcp-server; generated)
   Active: active (running) since Sun 2023-04-23 13:33:10 EEST; 17s ago
     Docs: man:systemd-sysv-generator(8)
   Process: 14925 ExecStart=/etc/init.d/isc-dhcp-server start (code=exited,
status=0/SUCCESS)
     Tasks: 4 (limit: 14232)
    Memory: 4.4M
         CPU: 42ms
    CGroup: /system.slice/isc-dhcp-server.service
            └─14941 /usr/sbin/dhcpd -4 -q -cf /etc/dhcp/dhcpd.conf enp3s0

anp 23 13:33:08 esprimop420 systemd[1]: Starting LSB: DHCP server...
anp 23 13:33:08 esprimop420 isc-dhcp-server[14925]: Launching IPv4 server only.
anp 23 13:33:08 esprimop420 dhcpd[14941]: Wrote 0 leases to leases file.
anp 23 13:33:08 esprimop420 dhcpd[14941]: Server starting service.
anp 23 13:33:10 esprimop420 isc-dhcp-server[14925]: Starting ISC DHCPv4 server: dhcpd.
anp 23 13:33:10 esprimop420 systemd[1]: Started LSB: DHCP server.

$ ps -A | grep dhcpd
14941 ?          00:00:00 dhcpd
```

Всё! У вас работает собственный DHCP сервер!

Разрешение имён, служба DNS

Система доменных имен (DNS, domain name system) представляет собой распределенную систему хранения и обработки информации о доменных зонах. Она необходима, в первую очередь, для соотнесения (прямого и обратного) IP адресов устройств в сети и более удобных для человеческого восприятия удобочитаемых символьных имен.

Система доменных имён разработана в еще 80-х годах прошлого века и продолжает обеспечивать удобство работы с адресным пространством Интернета до сих пор. И естественно поэтому для неё существует множество тонких модификаций.

Организационно и топологически система DNS представляет собой иерархическую систему непрерывно работающих DNS-серверов. Во главе иерархии располагаются 13 DNS-серверов отвечающий за корневые зоны, с учётом дублирующих серверов, на самом деле, как утверждается, их 123. Они управляются различными операторами и организациями, что исключает сбои по подчинённости.

По логике работа системы DNS (на качественном уровне, «на пальцах») выглядит так:

- Для вашей LAN или для вашего хоста в LAN назначено (указанием их собственных IP) несколько (1, 2, 3, 4 ... обычно не больше) DNS-серверов сети. Это назначение долгосрочное и может происходить самым разным образом: предоставляться вашим сетевым провайдером, назначаться вручную, при указании в NetworkManager и другие механизмы.
- Когда вы набираете URL в адресной строке браузера он через сетевой стек (сам браузер ничего не умеет — это общее свойство стека для всех сетевых приложений), он посылает запрос DNS-серверу сети в специальном сетевом DNS-протоколе. DNS-сервер сети, в свою очередь, либо отвечает сам (если этот URL ему известен), либо пересылает запрос по иерархии одному из доменных серверов более высокого уровня (или, в конце цепочки корневому серверу).

- Затем запрос начинает свое путешествие – корневой сервер пересылает его серверу первого уровня (поддерживающего, например, доменную зону .ru). Тот – серверу второго уровня в запрошенном URL (отделяемых точкой), и так далее, пока не найдется сервер, который точно знает запрошенное имя и адрес, либо выясняется, что такого имени не существует вообще.
- После получения ответа запрос начинает движение обратно ровно по тому же пути, только в обратной последовательности.
- Практически все сервера в узлах этой иерархии являются кэширующими, локально запоминают часто повторяющиеся имена в потоке запросов, поэтому вся названная выше цепочка оказывается многократно короче — популярные Интернет имена разрешаются на первых уровнях иерархии из локального кэша очередного сервера.

Сам DNS-протокол может базироваться либо на UDP, либо на TCP транспорте (на ранних этапах это был только UDP), и использует по умолчанию порт 53 (и в UDP и в TCP — раньше мы уже говорили что порты UDP и TCP это очень разные вещи и могут быть численно совпадать).

Вот на качественном уровне описание DNS того уровня детализации, которой достаточно для конкретизации темы использования DNS.

Из сказанного главным является следующие требования к DNS:

- Работа его должна быть бесперебойная 24 x 365 (всё время работы хоста);
- Обслуживание запросов должно быть как можно быстрее — именно оно определяет, во многом, «темп» работы хоста с Интернет;

Локальный DNS резолвер bind

Уже несколько десятилетий классикой DNS резолвера является bind9. Вы можете найти его в репозитории используемого вами дистрибутива.

```
$ apt search bind9
p bind9 - служба DNS
p bind9:i386 - служба DNS
p bind9-dev - Static Libraries and Headers used by BIND 9
p bind9-dev:i386 - Static Libraries and Headers used by BIND 9
i bind9-dnsutils - Clients provided with BIND 9
p bind9-dnsutils:i386 - Clients provided with BIND 9
p bind9-doc - Documentation for BIND 9
p bind9-dyndb-ldap - LDAP back-end plug-in for BIND
i bind9-host - утилита для выполнения запросов DNS
p bind9-host:i386 - утилита для выполнения запросов DNS
i bind9-libs - Shared Libraries used by BIND 9
p bind9-libs:i386 - Shared Libraries used by BIND 9
p bind9-utils - Utilities for BIND 9
p bind9-utils:i386 - Utilities for BIND 9
p bind9utils - Transitional package for bind9-utils
v bind9utils:i386 -
```

Что-то из его **клиентов**, почти наверняка, установлено у вас (по умолчанию) прямо при инсталляции Linux, и они используют список используемых DNS-серверов сети (о которых упоминалось ранее) из файла:

```
$ cat /etc/resolv.conf
# Generated by NetworkManager
nameserver 1.1.1.1
nameserver 8.8.8.8
nameserver 192.168.1.3
```

Это список DNS-серверов 1-го уровня к которому за разрешениями имён обращается с запросами сетевая подсистема вашего хоста. (Здесь, для большей общности, первые 2 IP указывают на DNS-сервера далеко в глобальной сети, а 3-й — на местном роутере, служащим шлюзом из локальной сети в глобальную. Таким образом, территориальность размещения, «далеко/близко», для работы резолвера DNS не имеет значения — что ему указали то он и использует.)

Идея локального резолвера состоит в том, что:

- Установить кэширующий сервер `bind9` на один из хостов LAN, скажем `192.168.1.53`;
- Указать ему список DNS-серверов верхнего уровня подобный показанному;
- А для всех остальных хостов LAN указывать в качестве DNS-сервера сети `192.168.1.53`;

Идея состоит в том, что для каждого пользователя массированная часть его запросов идёт к очень небольшому набору URL, и все они повторно будут разрешаться не выходя за пределы своей же локальной сети.

Я не буду показывать здесь установку и запуск `bind9` потому что:

- За долгие годы это многократно и подробно описано в публикациях;
- Сам `bind9` — это крупная громоздкая система, которую локально имеет смысл устанавливать только для LAN очень крупных организаций, насчитывающей многие десятки или даже сотни хостов.
- Мне важно зафиксировать сам принцип кэширующего DNS, который остаётся неизменным независимо от способа реализации — приблизить резолвер как можно ближе к месту создания запросов и тем радикально уменьшить время ожидания разрешения запроса.

Кэширующий DNS `dnsmasq`

Один из популярных кэширующих DNS для небольших сетей `Dnsmasq` — нетребовательный, простой в настройке DNS транслятор и DHCP сервер. Разработан для предоставления служб DNS и, дополнительно, DHCP для небольших сетей. Может обслуживать имена локальных машин не находящихся в глобальной DNS. DHCP сервер интегрирован с DNS сервером и позволяет машинам с адресами, полученными по DHCP, публиковать в DNS имена, заданные на самом хосте или в глобальном файле настроек. `Dnsmasq` поддерживает статическое и динамическое выделение адресов по DHCP, а также протокол BOOTP/TFTP для удалённой загрузки бездисковых машин.

Проект `Dnsmasq` за годы стал настолько популярным в Linux, что во многих дистрибутивах он установлен пакетной системой даже по умолчанию:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 11 (bullseye)
Release:        11
Codename:       bullseye
```

```
$ aptitude search dnsmasq
```

p	dnsmasq	- Small caching DNS proxy and DHCP/TFTP server
i A	dnsmasq-base	- Small caching DNS proxy and DHCP/TFTP server
p	dnsmasq-base-lua	- Small caching DNS proxy and DHCP/TFTP server
p	dnsmasq-utils	- Utilities for manipulating DHCP leases

По крайней мере в его базовых компонентах ...

```

$ aptitude show dnsmasq-base
Пакет: dnsmasq-base
Версия: 2.85-1
Состояние: установлен
Установлен автоматически: да
Приоритет: необязательный
Раздел: net
Сопровождающий: Simon Kelley <simon@thekelleys.org.uk>
Архитектура: amd64
Размер в распакованном виде: 963 k
Зависит: adduser, libc6 (>= 2.28), libdbus-1-3 (>= 1.9.14), libgmp10, libhogweed6 (>= 2.4-3), libidn2-0 (>= 2.0.0), libnetfilter-contrack3 (>= 1.0.1), libnettle8 (>= 2.4-3)
Рекомендует: dns-root-data
Конфликтует: dnsmasq-base-lua
Ломает: dnsmasq (< 2.63-1~)
Заменяет: dnsmasq (< 2.63-1~), dnsmasq-base
Предоставляется: dnsmasq-base-lua (2.85-1)
Описание: Small caching DNS proxy and DHCP/TFTP server
  This package contains the dnsmasq executable and documentation, but not the
  infrastructure required to run it as a system daemon. For that,
  install the dnsmasq package.
Домашняя страница: http://www.thekelleys.org.uk/dnsmasq/doc.html
Метки: admin::boot, admin::configuring, implemented-in::c, interface::daemon,
network::server, protocol::dhcp, protocol::dns, protocol::tftp,
  role::program, scope::utility, use::proxying

```

Для использования его как серверной службы нужно его доустановить до полного состава:

```

$ sudo apt install dnsmasq
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Предлагаемые пакеты:
  resolvconf
Следующие НОВЫЕ пакеты будут установлены:
  dnsmasq
Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 1
пакетов не обновлено.
Необходимо скачать 32,0 kB архивов.
После данной операции объём занятого дискового пространства возрастёт на 120 kB.
Пол:1 http://deb.debian.org/debian bullseye/main amd64 dnsmasq all 2.85-1 [32,0 kB]
Получено 32,0 kB за 0с (226 kB/s)
Выбор ранее не выбранного пакета dnsmasq.
(Чтение базы данных ... на данный момент установлено 351157 файлов и каталогов.)
Подготовка к распаковке .../dnsmasq_2.85-1_all.deb ...
Распаковывается dnsmasq (2.85-1) ...
Настраивается пакет dnsmasq (2.85-1) ...
Created symlink /etc/systemd/system/multi-user.target.wants/dnsmasq.service →
/lib/systemd/system/dnsmasq.service.
invoke-rc.d: policy-rc.d denied execution of start.

$ /sbin/dnsmasq -v
Dnsmasq version 2.85 Copyright (c) 2000-2021 Simon Kelley
Compile time options: IPv6 GNU-getopt DBus no-UBus i18n IDN2 DHCP DHCPv6 no-Lua TFTP
conntrack ipset auth cryptohash DNSSEC loop-detect inotify dumpfile

This software comes with ABSOLUTELY NO WARRANTY.
Dnsmasq is free software, and you are welcome to redistribute it
under the terms of the GNU General Public License, version 2 or 3.

```

```
$ systemctl status dnsmasq
• dnsmasq.service - dnsmasq - A lightweight DHCP and caching DNS server
  Loaded: loaded (/lib/systemd/system/dnsmasq.service; enabled; vendor preset: enabled)
  Active: inactive (dead)
```

Дальше, как и для всех сетевых сервисов о чём и как уже говорилось ранее, предстоит конфигурировать и запустить сервис. Но, в данном случае (сервер DNS) предварительно следовало бы проверить какое разрешение имён DNS используется на хосте на данный момент, и убедиться не работает ли здесь уже какая-то система кэширования DNS:

```
$ cat /etc/resolv.conf
# Generated by NetworkManager
nameserver 1.1.1.1
nameserver 8.8.8.8
nameserver 192.168.1.3

$ nslookup linux-ru.ru | grep -i server
Server:          1.1.1.1
```

При установке dnsmasq создаётся структура его конфигурационных файлов. Заготовка файла /etc/dnsmasq.conf очень обстоятельная, и исчерпывающе в комментариях описывает конфигурирование, но все строки этого большого файла комментируются:

```
$ grep -v ^$ /etc/dnsmasq.conf | grep -v ^#
$ ls -l /etc/dnsmasq.conf
-rw-r--r-- 1 root root 27381 anp  4  2021 /etc/dnsmasq.conf
$ ls -l /etc/dnsmasq.d
итого 4
-rw-r--r-- 1 root root 211 anp  4  2021 README
```

Я бы в общем конфигурационном файле убрал знак комментария только с одной строки, ссылающейся на отдельные файлы конфигураций в /etc/dnsmasq.d:

```
$ grep -v ^$ /etc/dnsmasq.conf | grep -v ^#
conf-dir=/etc/dnsmasq.d/, *.conf
```

А для DNS и DHCP сделал бы отдельные конфигурационные файлы /etc/dnsmasq.d/dhcp.conf и /etc/dnsmasq.d/dns.conf в каталог /etc/dnsmasq.d, скопировав в них, за основу, этот обстоятельный /etc/dnsmasq.conf, а дальше убрав комментарии только с нужных мне строк (это только иллюстрация для простых случаев, но ни в коем случае не рекомендация для подражания):

```
$ ls /etc/dnsmasq.d/*.conf
/etc/dnsmasq.d/dhcp.conf  /etc/dnsmasq.d/dns.conf

$ grep -v ^$ /etc/dnsmasq.d/dhcp.conf | grep -v ^#
dhcp-range=192.168.1.30,192.168.1.45,255.255.255.0,12h # объявляем диапазон
                                                         # адресов для аренды
dhcp-option=option:router,192.168.1.3                  # основной шлюз для этой под сети
log-dhcp                                              # записывать отладочную информацию

$ grep -v ^$ /etc/dnsmasq.d/dns.conf | grep -v ^#
listen-address=127.0.0.1  # принимаем запросы на локальном адресе
interface=enp3s0          # слушать только эти интерфейсы (из LAN)
domain-needed             # никогда не пересылать адреса без доменной части
bogus-priv                # никогда не пересылать адреса
                           # из немаршрутизируемого пространства
stop-dns-rebind           # отклонять ответы от вышестоящих DNS серверов
                           # с IP адресами локальной сети (блокировать DNS атаки)
rebind-localhost-ok       # отключить проверки для 127.0.0.0/8
```

```

strict-order          # пересылать запросы, с первого и по порядку
no-resolv              # не использовать /etc/resolv.conf
no-hosts              # не требуется читать /etc/hosts
domain=localdomain
local-ttl=7200        # настройки времени жизни кэша в секундах 7200=2h (два часа)
neg-ttl=14400
max-ttl=86400
server=192.168.1.3    # LAN router to WAN
server=8.8.4.4#53     # google OpenDNS
server=4.2.2.6#53     # Verizon DNS

```

Теперь можно запускать сконфигурированный сервер:

```

# systemctl start dnsmasq
# systemctl --no-pager --full status dnsmasq
• dnsmasq.service - dnsmasq - A lightweight DHCP and caching DNS server
  Loaded: loaded (/lib/systemd/system/dnsmasq.service; enabled; vendor preset: enabled)
  Active: active (running) since Tue 2023-04-25 15:17:26 EEST; 3min 59s ago
  Process: 19486 ExecStartPre=/etc/init.d/dnsmasq checkconfig (code=exited,
status=0/SUCCESS)
  Process: 19494 ExecStart=/etc/init.d/dnsmasq systemd-exec (code=exited,
status=0/SUCCESS)
  Process: 19505 ExecStartPost=/etc/init.d/dnsmasq systemd-start-resolvconf
(code=exited, status=0/SUCCESS)
  Main PID: 19504 (dnsmasq)
    Tasks: 1 (limit: 14232)
    Memory: 588.0K
    CPU: 44ms
    CGroup: /system.slice/dnsmasq.service
            └─19504 /usr/sbin/dnsmasq -x /run/dnsmasq/dnsmasq.pid -u dnsmasq -7
/etc/dnsmasq.d,.dpkg-dist,.dpkg-old,.dpkg-new --local-service --trust-
anchor=.,20326,8,2,e06d44b80b8f1d39a95c0b0d7c65d08458e880409bbc683457104237c7f8ec8d

  anp 25 15:17:26 esprimop420 systemd[1]: Starting dnsmasq - A lightweight DHCP and caching
DNS server...
  anp 25 15:17:26 esprimop420 dnsmasq[19504]: started, version 2.85 cachesize 150
  anp 25 15:17:26 esprimop420 dnsmasq[19504]: compile time options: IPv6 GNU-getopt DBus no-
UBus i18n IDN2 DHCP DHCPv6 no-Lua TFTP conntrack ipset auth cryptohash DNSSEC loop-detect
inotify dumpfile
  anp 25 15:17:26 esprimop420 dnsmasq-dhcp[19504]: DHCP, IP range 192.168.1.30 --
192.168.1.45, lease time 12h
  anp 25 15:17:26 esprimop420 dnsmasq[19504]: using nameserver 4.2.2.6#53
  anp 25 15:17:26 esprimop420 dnsmasq[19504]: using nameserver 8.8.4.4#53
  anp 25 15:17:26 esprimop420 dnsmasq[19504]: using nameserver 192.168.1.3#53
  anp 25 15:17:26 esprimop420 dnsmasq[19504]: cleared cache
  anp 25 15:17:26 esprimop420 systemd[1]: Started dnsmasq - A lightweight DHCP and caching
DNS server.

# cat /var/log/syslog | grep dnsmasq-dhcp
Apr 25 15:17:26 esprimop420 dnsmasq-dhcp[19504]: DHCP, IP range 192.168.1.30 --
192.168.1.45, lease time 12h

```

Вот теперь мы можем поэкспериментировать с DNS сервером, и оценить что даёт нам факт кэширования...

Проверяем: разрешение через сетевой DNS (это Google DNS, из числа популярных и лучших), для 2-х произвольно взятых имён⁷:

```
$ host -v qnx.org.ru | grep Received
```

⁷ Команда host отправляет последовательно 3 запроса и получает 3 ответа для получения DNS записей, соответственно, типов: A (IPv4), AAAA (IPv6), MX (запись для электронной почты, указывающая, какими серверами она обрабатывается). Поэтому для оценки общего времени ответа можно использовать **сумму** этих 3-х значений.


```
Received 44 bytes from 1.1.1.1#53 in 348 ms
Received 80 bytes from 1.1.1.1#53 in 60 ms
Received 57 bytes from 1.1.1.1#53 in 48 ms
```

```
$ host -v linux-ru.ru | grep Received
```

```
Received 45 bytes from 1.1.1.1#53 in 60 ms
Received 81 bytes from 1.1.1.1#53 in 64 ms
Received 77 bytes from 1.1.1.1#53 in 56 ms
```

А теперь делаем то же самое, но обращаясь за разрешением к только-что запущенному локальному кэширующему DNS. Делаем последовательно 2 запроса, парами (по 1-му локальный DNS обратится в сеть по иерархии, 2-й станет разрешаться из внутреннего кэша):

```
$ host -v linux-ru.ru 127.0.0.1 | grep Received
```

```
Received 45 bytes from 127.0.0.1#53 in 108 ms
Received 81 bytes from 127.0.0.1#53 in 128 ms
Received 77 bytes from 127.0.0.1#53 in 76 ms
```

```
$ host -v linux-ru.ru 127.0.0.1 | grep Received
```

```
Received 45 bytes from 127.0.0.1#53 in 0 ms
Received 29 bytes from 127.0.0.1#53 in 0 ms
Received 77 bytes from 127.0.0.1#53 in 56 ms
```

```
$ host -v qnx.org.ru 127.0.0.1 | grep Received
```

```
Received 44 bytes from 127.0.0.1#53 in 144 ms
Received 80 bytes from 127.0.0.1#53 in 76 ms
Received 57 bytes from 127.0.0.1#53 in 228 ms
```

```
$ host -v qnx.org.ru 127.0.0.1 | grep Received
```

```
Received 44 bytes from 127.0.0.1#53 in 0 ms
Received 28 bytes from 127.0.0.1#53 in 0 ms
Received 57 bytes from 127.0.0.1#53 in 44 ms
```

Видно насколько значительно ускоряется запрос разрешения имени (при повторном запросе), которое уже находится в кэше сервера. Остаётся вопрос: почему работает (успешно как видим) кэширующий DNS, но запросы по умолчанию (без явного указания сервера) идут по-прежнему к удалённым сетевым DNS? Потому что всё ещё:

```
$ cat /etc/resolv.conf
```

```
# Generated by NetworkManager
nameserver 1.1.1.1
nameserver 8.8.8.8
nameserver 192.168.1.3
```

Меняем:

```
$ cat /etc/resolv.conf
```

```
# Generated by NetworkManager
nameserver 127.0.0.1
```

И теперь:

```
$ host -v linux-ru.ru | grep Received
```

```
Received 45 bytes from 127.0.0.1#53 in 0 ms
Received 29 bytes from 127.0.0.1#53 in 0 ms
Received 77 bytes from 127.0.0.1#53 in 60 ms
```

```
$ host -v qnx.org.ru | grep Received
```

```
Received 44 bytes from 127.0.0.1#53 in 0 ms
Received 28 bytes from 127.0.0.1#53 in 0 ms
Received 57 bytes from 127.0.0.1#53 in 40 ms
```

Наконец, пусть любой другой хост этой LAN запросит разрешение имени у этого вновь созданного DNS сервера:

```
$ host -v linux-ru.ru 192.168.1.138 | grep Received
Received 45 bytes from 192.168.1.138#53 in 116 ms
Received 81 bytes from 192.168.1.138#53 in 84 ms
Received 77 bytes from 192.168.1.138#53 in 1156 ms
$ host -v linux-ru.ru 192.168.1.138 | grep Received
Received 45 bytes from 192.168.1.138#53 in 4 ms
Received 29 bytes from 192.168.1.138#53 in 4 ms
Received 77 bytes from 192.168.1.138#53 in 72 ms
```

Показанные результаты позволяют убедиться в том, что использование локального кэширующего сервера DNS позволяет снизить задержку разрешения имени часто используемых URL в несколько десятков раз (в разных ситуациях автор наблюдал уменьшение времени ответов до 60 раз).

Кэширующий DNS средствами systemd

Первый выпуск проекта Dnsmasq относится к 2001 (последняя редакция 2.89 — это 04.02.2023). Успех проекта на протяжении 22 лет, наверное, и побудил разработчиков systemd (Lennart Poettering с соавторами) ввести функциональность кэширующего DNS непосредственно в состав системы управления сервисами Linux.

Запуск systemd встроенной системы кэшированного DNS я покажу, для разнообразия и убедительности, на микро-архитектуре ARM одноплатного (SoC — system on a chip) компьютера Raspberry Pi:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Raspbian
Description: Raspbian GNU/Linux 11 (bullseye)
Release: 11
Codename: bullseye
$ uname -a
Linux raspberrypi 6.1.21-v7+ #1642 SMP Mon Apr 3 17:20:52 BST 2023 armv7l GNU/Linux
```

Резолвер systemd носит название systemd-resolved.service, но в отличие от множества других сетевых сервисов-серверов (рассмотренных выше и тех которые будут ещё рассмотрены позже) **не требует инсталляции!** Он присутствует (установлен) в системе изначально, по умолчанию, вместе с самой подсистемой systemd, но не активирован по умолчанию:

```
$ systemctl status systemd-resolved.service --no-pager --full
• systemd-resolved.service - Network Name Resolution
   Loaded: loaded (/lib/systemd/system/systemd-resolved.service; disabled; vendor
   preset: enabled)
   Active: inactive (dead)
     Docs: man:systemd-resolved.service(8)
           man:org.freedesktop.resolve1(5)
   ...
$ resolvectl status
Failed to get global data: Unit dbus-org.freedesktop.resolve1.service not found.
```

А разрешение имён в не модифицированной системе идёт традиционным образом, через список сетевых серверов DNS:

```
$ nslookup linux-ru.ru
Server:      1.1.1.1
Address:     1.1.1.1#53
```

```
Non-authoritative answer:
Name:   linux-ru.ru
Address: 90.156.230.27
```

```
$ cat /etc/resolv.conf
# Generated by resolvconf
nameserver 1.1.1.1
nameserver 8.8.8.8
```

Нам остаётся некоторым образом заполнить конфигурацию и запускать сервис. Конфигурация резолвера находится в `/etc/systemd/resolved.conf`, прекрасно документирована (в комментариях) и, в простейшем виде, туда достаточно заполнить только одну секцию:

```
$ grep -v ^# /etc/systemd/resolved.conf | grep -v ^$
[Resolve]
DNS=1.0.0.1 8.8.4.4 9.9.9.9
```

Всё! Можно запускать:

```
# systemctl start systemd-resolved.service
# systemctl status systemd-resolved.service --no-pager --full
• systemd-resolved.service - Network Name Resolution
   Loaded: loaded (/lib/systemd/system/systemd-resolved.service; disabled; vendor
   preset: enabled)
   Active: active (running) since Tue 2023-04-18 15:00:59 EEST; 2s ago
     Docs: man:systemd-resolved.service(8)
           man:org.freedesktop.resolve1(5)
           https://www.freedesktop.org/wiki/Software/systemd/writing-network-
configuration-managers
           https://www.freedesktop.org/wiki/Software/systemd/writing-resolver-clients
   Main PID: 22564 (systemd-resolve)
   Status: "Processing requests..."
    Tasks: 1 (limit: 1595)
      CPU: 702ms
   CGroup: /system.slice/systemd-resolved.service
           └─22564 /lib/systemd/systemd-resolved
```

```
anp 18 15:00:58 raspberrypi systemd[1]: Starting Network Name Resolution...
anp 18 15:00:58 raspberrypi systemd-resolved[22564]: Positive Trust Anchors:
anp 18 15:00:58 raspberrypi systemd-resolved[22564]: . IN DS 20326 8 2
e06d44b80b8f1d39a95c0b0d7c65d08458e880409bbc683457104237c7f8ec8d
anp 18 15:00:58 raspberrypi systemd-resolved[22564]: Negative trust anchors: 10.in-
addr.arpa 16.172.in-addr.arpa 17.172.in-addr.arpa 18.172.in-addr.arpa 19.172.in-addr.arpa
20.172.in-addr.arpa 21.172.in-addr.arpa 22.172.in-addr.arpa 23.172.in-addr.arpa 24.172.in-
addr.arpa 25.172.in-addr.arpa 26.172.in-addr.arpa 27.172.in-addr.arpa 28.172.in-addr.arpa
29.172.in-addr.arpa 30.172.in-addr.arpa 31.172.in-addr.arpa 168.192.in-addr.arpa
d.f.ip6.arpa corp home internal intranet lan local private test
anp 18 15:00:59 raspberrypi systemd-resolved[22564]: Using system hostname 'raspberrypi'.
anp 18 15:00:59 raspberrypi systemd[1]: Started Network Name Resolution.
```

В итоге получаем:

```
$ ps -A | grep resolve
22564 ?          00:00:00 systemd-resolve

$ resolvectl status
Global

   Protocols: +LLMNR +mDNS -DNSoverTLS DNSSEC=no/unsupported
   resolv.conf mode: foreign
   Current DNS Server: 1.0.0.1
   DNS Servers: 1.0.0.1 8.8.4.4 9.9.9.9
```

```
Link 2 (eth0)
Current Scopes: LLMNR/IPv4 LLMNR/IPv6
Protocols: -DefaultRoute +LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported

Link 3 (tun0)
Current Scopes: LLMNR/IPv6
Protocols: -DefaultRoute +LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported
```

И проверяем его работоспособность:

```
$ host -6 ya.ru 127.0.0.53
Using domain server:
Name: 127.0.0.53
Address: ::ffff:127.0.0.53#53
Aliases:

ya.ru has address 77.88.55.242
ya.ru has address 5.255.255.242
ya.ru has IPv6 address 2a02:6b8::2:242
ya.ru mail is handled by 10 mx.yandex.ru.
```

(Обращаем внимание, что резолвер `systemd` использует не вообще какой-то адрес петлевого интерфейса `localhost`, а выделенный для этого IP `127.0.0.53`)

Но дефолтное разрешение, пока ещё, будет происходить через список адресов DNS серверов:

```
$ cat /etc/resolv.conf
# Generated by resolvconf
nameserver 1.1.1.1
nameserver 8.8.8.8
```

Один из вариантов состоит в том, чтобы заменить содержимое `/etc/resolv.conf` строкой:

```
nameserver 127.0.0.53
```

Но документация `systemd` рекомендует другой способ: сделать вместо `/etc/resolv.conf` ссылку на подготовленный файл `/run/systemd/resolve/stub-resolv.conf` (возможно сохранив старый `/etc/resolv.conf` под новым именем, типа `/etc/resolv.conf.old`, на всякий случай, на будущее):

```
# sudo ln -svi /run/systemd/resolve/stub-resolv.conf /etc/resolv.conf
ln: заменить '/etc/resolv.conf'? y
'/etc/resolv.conf' -> '/run/systemd/resolve/stub-resolv.conf'

$ grep -v ^# /etc/resolv.conf | grep -v ^$
nameserver 127.0.0.53
options eth0 trust-ad
search .
```

Проверяем:

```
$ host ya.ru
ya.ru has address 77.88.55.242
ya.ru has address 5.255.255.242
ya.ru has IPv6 address 2a02:6b8::2:242
ya.ru mail is handled by 10 mx.yandex.ru.
```

Оптимизация используемых серверов DNS

Как уже было замечено ранее, главная (и, пожалуй, единственная) задача пользователя DNS (мы не говорим о тех кто предоставляет услуги DNS) состоит в том, чтобы максимально ускорить получение ответов на запросы DNS. Ведь по большинству случаев пользователь (или пользователи LAN одной организации) занимаются сёрфингом по весьма ограниченной группе доменов и субдоменов, связанных с их целевой деятельностью. А скорость отклика DNS и создаёт, в значительной мере, субъективное ощущение «быстрого Интернет».

Но, будь то простая сетевая схема разрешения через список серверов в `/etc/resolv.conf`, или любая изошрённая схема кэширования, запросы в иерархической системе запросов DNS, в конечном счёте, идут к тем нескольким серверам DNS, которые определены как сетевые сервера 1-го уровня для хоста. Адреса которых, в конечном итоге, мы вручную прописываем, например, в процессе конфигурирования кэширования DNS.

Очень часто в качестве серверов 1-го уровня ваш провайдер Интернет «подбрасывает» свои собственные сервера DNS. И это может быть совсем не плохое решение. С другой стороны вы можете самостоятельно выбрать адреса оптимальных серверов 1-го уровня, оптимальность которых определяется, например, вашим территориальным расположением и доступом к этим серверам. В любом случае стоит произвести анализ и выбор набора оптимальных серверов для своей ситуации. Здесь общие рекомендации не помогут!

Тестирование и сравнение скорости DNS (по порядку величины) можно производить и типовыми утилитами (`host`, `dig`, ...), но нужно учитывать что время отклика DNS — величина статистическая. Для оценки скорости DNS сериями запросов, потому как это задача насущная, было сделано достаточно много приложений ... но они, по большинству, для Windows/MacOS, а это нас не интересует. Один из пригодных проектов для тестирования может быть получен из архивов Google (<https://code.google.com/archive/p/namebench/downloads>), архив проекта:

```
$ ls -l namebench-1.3.1-source.tgz
-rw-rw-r-- 1 olej olej 1118505 anp 26 12:36 namebench-1.3.1-source.tgz
```

После разархивирования:

```
$ ls -l namebench-1.3.1
итого 272
-rw-r--r-- 1 olej olej 197383 июн 6 2010 ChangeLog.txt
drwxr-xr-x 4 olej olej 4096 июн 6 2010 cocoa
drwxr-xr-x 2 olej olej 4096 июн 6 2010 config
-rw-r--r-- 1 olej olej 11358 июн 6 2010 COPYING
drwxr-xr-x 2 olej olej 4096 июн 6 2010 data
-rw-r--r-- 1 olej olej 8601 июн 6 2010 JSON.txt
drwxr-xr-x 2 olej olej 4096 июн 6 2010 libnamebench
-rwxr-xr-x 1 olej olej 2477 июн 6 2010 namebench.py
drwxr-xr-x 8 olej olej 4096 июн 6 2010 nb_third_party
-rw-r--r-- 1 olej olej 12152 июн 6 2010 README.txt
-rw-r--r-- 1 olej olej 5268 июн 6 2010 setup.py
drwxr-xr-x 2 olej olej 4096 июн 6 2010 templates
drwxr-xr-x 2 olej olej 4096 июн 6 2010 tools
```

Программа написана на Python, поэтому не требует никакой сборки, а поэтому сразу готова к запуску. Программа может запускаться в консольном режиме и в графическом. Но, чтобы не разбираться с недокументированным консольным режимом и использовать GUI нужно в системе иметь установленной (или установить) языковую систему Tcl/Tk, 2 небольших пакета:

```
$ aptitude search tcl | grep ' tcl ' | grep ^i
i tcl - Tool Command Language (default version) - shell
```

```
$ aptitude search python-tk | grep ^i
```

Программа достаточно старая (последняя версия 1.3.1 — середина 2010 года), но это не умаляет её достоинств и применимости. Единственная особенность, что написана она под Python2, что при запуске это нужно теперь указывать явно:

```
$ python2 namebench.py
Starting Tk interface for namebench...
...
```

(Почему для тестирования были выбраны именно эти конкретно 3 сервера DNS (показанные на рисунке)? Потому что это сервера OpenNIC поддерживающие альтернативные DNS, о чём речь впереди, в последней части книги. Вы можете указать произвольное число тестируемых серверов, более того, включить в анализ крупнейших глобальных и региональных провайдеров DNS ... но нужно учитывать, что время тестирования значительное, при большом числе серверов может составлять несколько часов.)

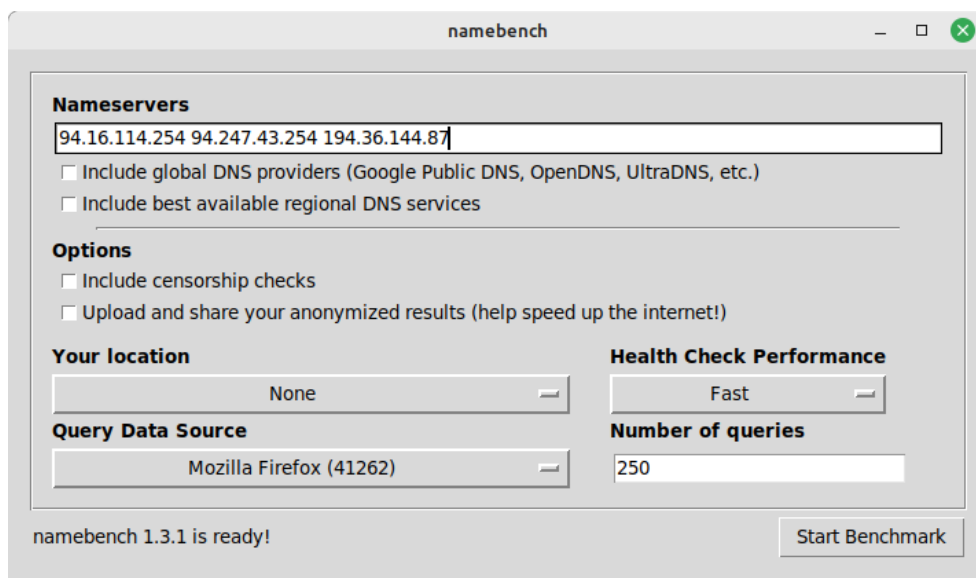


Рис. 2.10 Список DNS серверов для тестирования

После завершения тестирования (достаточно продолжительного — в это время можно чем-то заняться) во вкладку открытого браузера отображается ранжированный список серверов по предпочтительности, и графики их временных откликов:

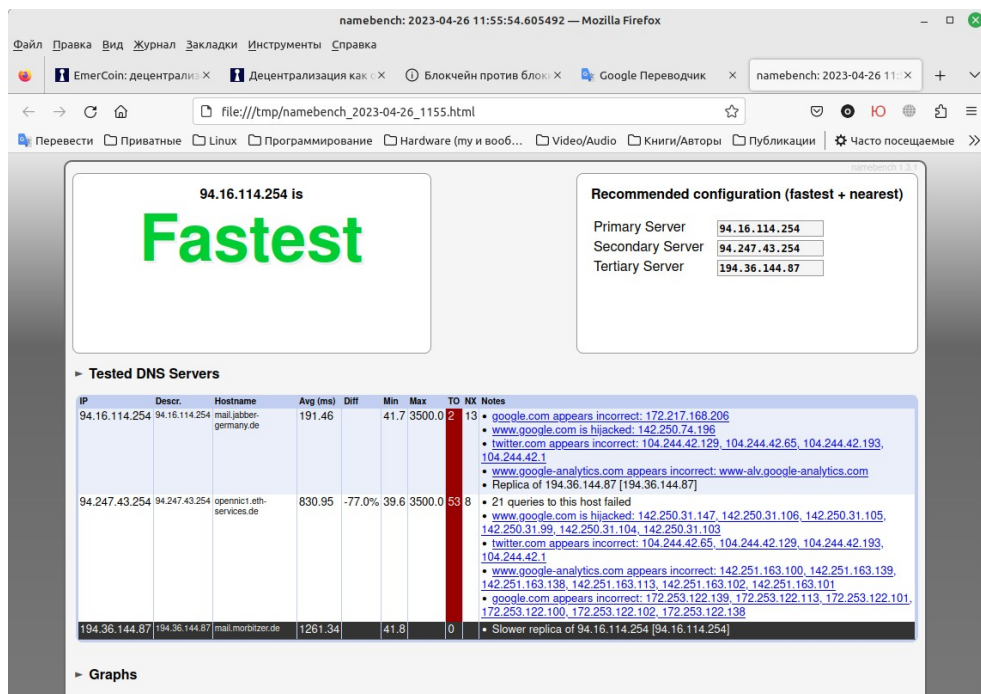


Рис. 2.11 Результат тестирования

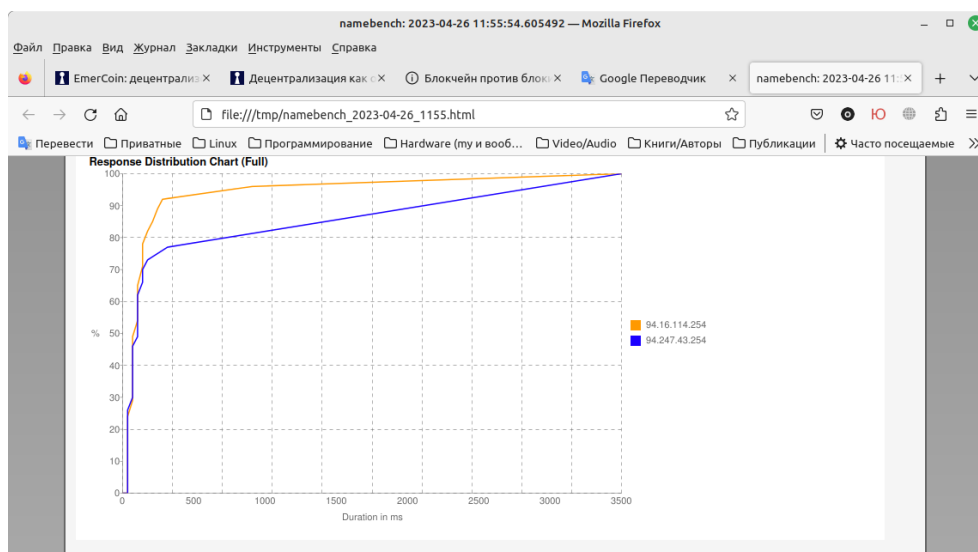


Рис. 2.12 Результат тестирования

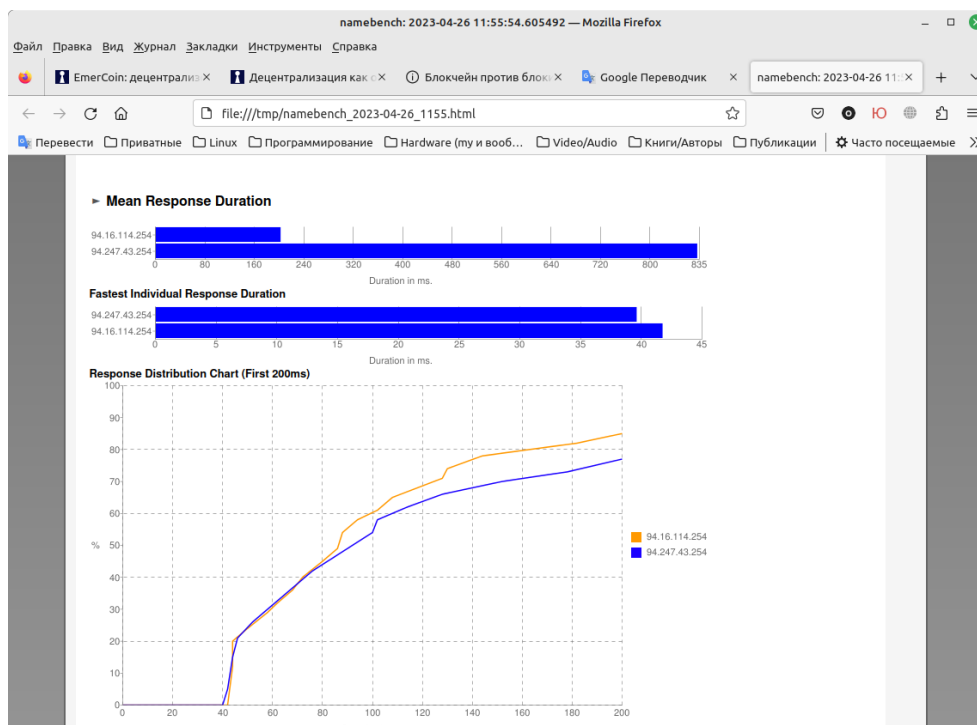


Рис. 2.13 Результат тестирования

Защищённость сети, фаервол

Здесь мы говорим о защищённости сети в смысле доступа к серверам или хостам локальной наружи, из глобальной сети. Для локальных сетей IPv4 (традиционных и привычных на протяжении нескольких десятилетий) проблема смягчается использованием NAT: хосты локальной сети имеют частные IPv4 не маршрутизируемые адреса, и недоступны по этим адресам извне.

Но с переходом к IPv6 исчезает разделение адресов на «серые» и «белые», и локальные хосты могут быть доступны по IPv6 из любой точки мира. И, тем более, остро всегда стояла задача максимально ограничить доступ к отдельно стоящим серверам Интернет (кроме тех протоколов, для обслуживания которых и выделен сервер).

Управление доступностью адресов и портов на сегодня обеспечивается подсистемой ядра Linux `netfilter`, и его правил, для управления которыми используются утилита `iptables` (в разные времена в Linux для таких целей использовались подобные подсистемы под разными названиями: `ipchains`, `ipfw`, `ipfilter` — само обилие реализаций говорит о важности проблемы!). Утилита `iptables` устанавливается в системе автоматически, в составе инсталляции, даже если вы её не используете:

```
$ aptitude search iptables | grep ^i
i A iptables - утилита для фильтрации сетевых пакетов и NAT
```

Правила `netfilter` распределены по 4-м таблицам (`filter`, `nat`, `mangle`, `raw`), в целях фаервола применяются манипуляции с таблицей `filter`. Каждая таблица состоит из цепочек (`chain`) правил, каждая таблица имеет свой предопределённый набор цепочек. Но сверх предопределённых цепочек могут создаваться и дополнительные. Для интересующей нас сейчас таблицы `filter` предопределённые цепочки — это `INPUT` (входящие пакеты), `OUTPUT` (исходящие) и `FORWARD` (транзитные).

Правила в цепочках применяются последовательно ко всем сетевым пакетам и могут запрещать или разрешать прохождение пакетов фильтруя их по разным признакам: IP адресата, порт и другим признакам.

Важно помнить, что правила `netfilter` выполняются в ядре Linux, и даже если мы удалим вообще из системы пакет `iptables` (или любой другой пакет управления `netfilter`, такие например, как `firewall` в RedHat или это `ufw` в Ubuntu), то правила фильтрации будут всё-равно выполняться, а для того чтобы правило или правила отменить нужно а). правило удалить, или б). всю цепочку правил очистить, или в). таблицу целиком очистить.

Правила цепочки и правила в цепочки можно добавлять (удалять, перемещать, управлять, диагностировать...) непосредственно самой утилитой `iptables`:

```
$ iptables --help
iptables v1.8.7

Usage: iptables -[ACD] chain rule-specification [options]
       iptables -I chain [rulenum] rule-specification [options]
       iptables -R chain rulenum rule-specification [options]
       iptables -D chain rulenum [options]
       iptables -[LS] [chain [rulenum]] [options]
       iptables -[FZ] [chain] [options]
       iptables -[NX] chain
       iptables -E old-chain-name new-chain-name
       iptables -P chain target [options]
       iptables -h (print this help information)

...
```

Понятно, что в силу своей важности для работоспособности сети вообще, все операции с правилами `netfilter`, даже просто просмотр правил, могут выполняться только с правами `root`.

Добавление правил посредством утилиты `iptables` — это подробное, тщательное (низкоуровневое), весьма трудоёмкое и требующее кропотливого (и достаточно сложного) тестирования функционирования. Изначально предопределённый цепочки правил пусты (не накладывается запретов):

```
$ sudo iptables -L -v -n
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target    prot opt in     out     source                   destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target    prot opt in     out     source                   destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target    prot opt in     out     source                   destination
```

Вот как, только в качестве нескольких примеров внешнего вида, могут выглядеть операции `iptables` над правилами `netfilter`:

- Просмотр правил трансляции адресов NAT:
`# iptables -t nat -L --line-numbers`
- Разрешить все исходящие пакеты (таблицу `filter` можно не указывать, она дефавтная):
`# iptables -t filter -I OUTPUT 1 -j ACCEPT`
- Запретить все входящие пакеты из подсети 10.10.10.0/24:
`# iptables -t filter -A INPUT -s 10.10.10.0/24 -j DROP`
- Разрешить SSH:
`# iptables -A INPUT -p tcp --dport 22 -j ACCEPT`
- Разрешить диапазон входящих портов TCP:
`# iptables -A INPUT -p tcp --dport 3000:4000 -j ACCEPT`

- Разрешаем ICMP (для выполнения команды ping):

```
# iptables -A INPUT -p icmp -j ACCEPT
```

Из показанного понятно, что команды `iptables` очень детализированы и подробны. Правила действуют до перезагрузки, должны сохраняться (предусмотрено несколько способов) и восстанавливаться при загрузке. Мы не будем детально углубляться в синтаксис и написание правил `iptables` потому как они многократно и полно документированы и описаны.

Отдельные дистрибутивы Linux вводят дополнительно свои, более высокоуровневые механизмы **управления** правилами (которые, сами правила, так и продолжают называться правилами `iptables`, и могут просматриваться утилитой `iptables`). Из таких подсистем, как вариант, в Fedora, RedHat, CentOS — это `firewall`, а в Debian, Ubuntu, Mint, LMDE — это `ufw`.

Файервол ufw

UFW (Uncomplicated FireWall — несложный брандмауэр) — удобный интерфейс для управления политиками безопасности межсетевого экрана. Он выполняет, собственно, то что и `iptables`: формирует и загружает правила для `netfilter` в ядре.

```
$ aptitude search ufw
```

```
p  gufw                    - графический интерфейс пользователя для ufw
p  ufw                    - program for managing a Netfilter firewall
```

```
$ sudo apt install ufw
```

```
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Следующие НОВЫЕ пакеты будут установлены:
  ufw
...
```

При установке пакета создаётся сервис:

```
$ systemctl status ufw
```

```
• ufw.service - Uncomplicated firewall
   Loaded: loaded (/lib/systemd/system/ufw.service; enabled; vendor preset: enabled)
   Active: inactive (dead)
   Docs: man:ufw(8)
```

Именно он будет запускаться при загрузке системы и загружать составленные нами правила для `netfilter` (после загрузки правил этот сервис останавливается, в работающей системе он не нужен):

```
$ systemctl is-enabled ufw
enabled
```

Файервол может обслуживать как протокол IPv4 так и IPv4:

```
# grep IPV6 /etc/default/ufw
IPV6=yes
```

А для управления самими правилами используется утилита `ufw` (должна выполняться, естественно, от `root`), она же используется для активизации (поднятия) файервола и его отключения:

```
# ufw status
Status: inactive
# ufw enable
Command may disrupt existing ssh connections. Proceed with operation (y|n)? y
Firewall is active and enabled on system startup
```

```
# ufw status verbose
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip
```

Это политики (изначальные) ufw по умолчанию: все выходные соединения разрешены, все входные соединения и форвардинг запрещены.

Посмотрим как работает файервол на примере протокола SSH, выполняем с соседнего хоста LAN (я специально использую адреса IPv6):

```
$ ssh -l olej fe80::921b:eff:fe2b:fe3a%eno1
ssh: connect to host fe80::921b:eff:fe2b:fe3a%eno1 port 22: Connection timed out
```

Теперь остановим файервол:

```
# ufw disable
Firewall stopped and disabled on system startup
```

И повторим сеанс SSH, теперь:

```
$ ssh -l olej fe80::921b:eff:fe2b:fe3a%eno1
olej@fe80::921b:eff:fe2b:fe3a%eno1's password:
Linux esprimop420 5.10.0-21-amd64 #1 SMP Debian 5.10.162-1 (2023-01-21) x86_64
Last login: Thu Apr 27 11:16:02 2023 from fe80::13f5:9fe2:6393:bf4a%enp3s0
olej@esprimop420:~$ exit
выход
Connection to fe80::921b:eff:fe2b:fe3a%eno1 closed.
```

Теперь о том как определяются правила, например, SSH:

```
# ufw allow OpenSSH
Rules updated
Rules updated (v6)
# ufw status numbered
Status: active
```

	To	Action	From
	--	-----	----
[1]	OpenSSH	ALLOW IN	Anywhere
[2]	OpenSSH (v6)	ALLOW IN	Anywhere (v6)

Добавлять можно указанием как числовых портов (TCP и/или UDP), так по именам стандартных сетевых служб, большой список которых предопределён в приложении:

```
# ufw app list
Available applications:
AIM
Bonjour
CIFS
CUPS
DNS
Deluge
IMAP
IMAPS
IPP
KTorrent
Kerberos Admin
Kerberos Full
Kerberos KDC
Kerberos Password
```

```

LDAP
LDAPS
LPD
MSN
MSN SSL
Mail submission
NFS
OpenSSH
POP3
POP3S
PeopleNearby
SMTP
SSH
Socks
Telnet
Transmission
Transparent Proxy
VNC
WWW
WWW Cache
WWW Full
WWW Secure
XMPP
Yahoo
qBittorrent
svnserve

```

Добавим ещё HTTP/HTTPS доступ:

```

# ufw allow in from any to any port 80,443,8080:8090 comment 'web app' proto tcp
Rule added
Rule added (v6)
# ufw status numbered
Status: active

```

To	Action	From	
--	-----	----	
[1] OpenSSH	ALLOW IN	Anywhere	
[2] 80,443,8080:8090/tcp	ALLOW IN	Anywhere	# web app
[3] OpenSSH (v6)	ALLOW IN	Anywhere (v6)	
[4] 80,443,8080:8090/tcp (v6)	ALLOW IN	Anywhere (v6)	# web app

А вот так удаляем выбранное правило:

```

# ufw delete 4
Deleting:
  allow 80,443,8080:8090/tcp comment 'web app'
Proceed with operation (y|n)? y
Rule deleted (v6)
# ufw status
Status: active

```

To	Action	From	
--	-----	----	
OpenSSH	ALLOW	Anywhere	
80,443,8080:8090/tcp	ALLOW	Anywhere	# web app
OpenSSH (v6)	ALLOW	Anywhere (v6)	

Синтаксис хорошо описан (man ufw) и допускает очень тонкое управление правилами. Запуск команды с опцией --dry-run не производит фактических изменений правил netfilter, но моделирует изменения и выводит на терминал набор создаваемых командой правил в терминологии iptables.

Наконец, отметим что ufw не модифицирует предопределённые цепочки (INPUT, OUTPUT, FORWARD) таблицы filter, а добавляет свои, новые цепочки в таблицу, и довольно много:

```
# iptables -t filter -L | grep ^"Chain ufw-" | wc -l
32
```

Для удобства конфигурирования, если вы предпочитаете такую форму, можно установить и графический (GUI) инструмент:

```
# apt install gufw
```

Чтение списков пакетов... Готово

Построение дерева зависимостей... Готово

Чтение информации о состоянии... Готово

Следующие НОВЫЕ пакеты будут установлены:

gufw

Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 0 пакетов не обновлено.

Необходимо скачать 876 kB архивов.

После данной операции объём занятого дискового пространства возрастёт на 3.538 kB.

Пол:1 http://deb.debian.org/debian bullseye/main amd64 gufw all 20.04.1-1 [876 kB]

Получено 876 kB за 0с (2.559 kB/s)

Выбор ранее не выбранного пакета gufw.

(Чтение базы данных ... на данный момент установлено 351300 файлов и каталогов.)

Подготовка к распаковке .../gufw_20.04.1-1_all.deb ...

Распаковывается gufw (20.04.1-1) ...

Настраивается пакет gufw (20.04.1-1) ...

Обрабатываются триггеры для mailcap (3.69) ...

Обрабатываются триггеры для desktop-file-utils (0.26-1) ...

Обрабатываются триггеры для hicolor-icon-theme (0.17-2) ...

Обрабатываются триггеры для man-db (2.9.4-2) ...

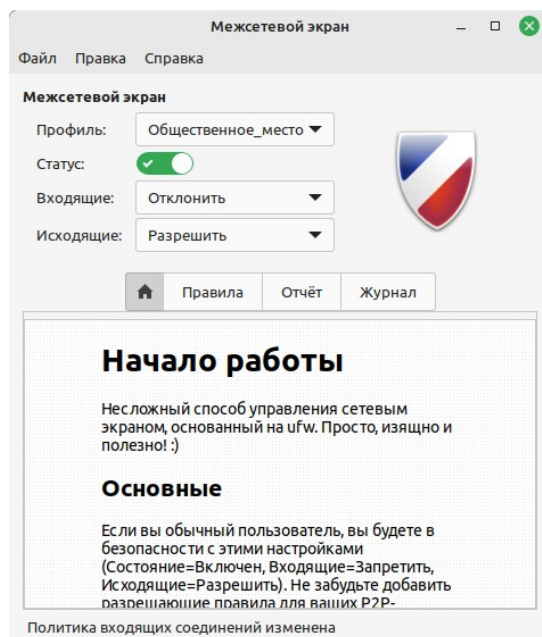


Рис. 2.14 Графический интерфейс

В результате последовательно формируется набор правил, что, вообще то говоря, эквивалентно тому, как мы генерировали эти же правила консольной утилитой `ufw`:

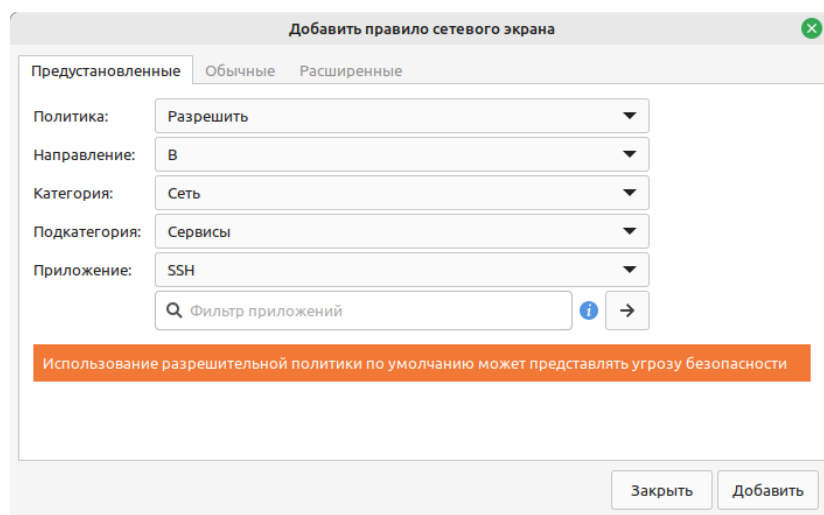


Рис. 2.15 Правила файервола

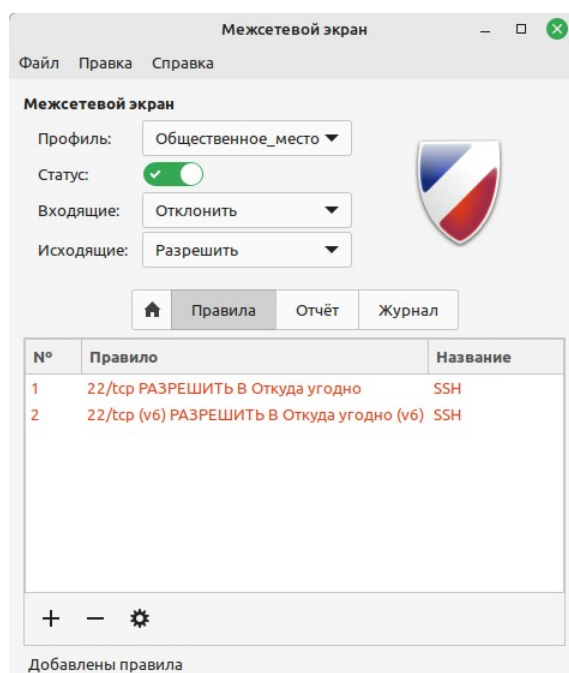


Рис. 2.16 Настроенный файервол

Суперсервер `inetd`

Ещё с самых ранних UNIX (практически с 80-х годов) сложилась практика для предоставления ряда сетевых сервисов с помощью суперсервера. Идея состоит в том, что, вместо запуска многих серверов разных сервисов и на разных портах, одна программа-демон ожидает одновременно запросы на соединения с множеством адресов портов. Когда клиент подключается к прослушивающему сервису, программа-демон запускает соответствующий сервер как дочерний процесс, и продолжает прослушивание запросов дальше. При таком

подходе серверам не нужно работать постоянно и потреблять ресурсы, они могут запускаться по требованию.

Такая программа-демон получила название суперсервер (сервер запускающий другие сервера). В качестве суперсерверов используется более давний проект `inetd` (Internet Service Daemon) или его более позднее развитие `xinetd` (Extended Internet Service Daemon) ... для наших целей сейчас это не имеет значения, и я покажу использование на примере `inetd` (дистрибутив LMDE5):

```
$ cat /etc/debian_version
11.2
$ aptitude search inetd | grep ^i
i A openbsd-inetd - OpenBSD Internet Superserver
i update-inetd - программа обновления файла настройки inetd
$ aptitude search xinetd | grep -v xinetd$
p xinetd - replacement for inetd with many enhancements
p xinetd:i386 - replacement for inetd with many enhancements
```

То какие порты прослушивать (быть в готовности) и какие действия на них выполнять определяется в конфигурационном файле суперсервера: для `inetd` это `/etc/inetd.conf`, а для `xinetd`, соответственно, `/etc/xinetd.conf` — форматы конфигурационных файлов `inetd.conf` и `xinetd.conf` радикально отличаются, хотя, по сути, служат одной цели. В `inetd.conf` каждой службе соответствует одна строка записи, и формат такой строки (взято непосредственно из комментария `inetd.conf`):

```
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
```

Здесь:

- `<service_name>` - имя сервиса;
- `<sock_type>` - тип сокета, это `stream` для TCP или `dgram` для UDP;
- `<proto>` - протокол ... это может быть, например: `tcp`, `tcp4`, `tcp6`;
- `<flags>` - это `wait` или `nowait`;
- `<user>` - имя пользователя от лица которого запускается сервер;
- `<server_path>` - путевое имя запускаемого сервера;
- `<args>` - параметры которые нужно (возможно) передать серверу при запуске:

Имена имя сервисов (`<service_name>`) — это то, что мы уже видели раньше в большом файле `/etc/services` — имена сервисов известные системе, например:

```
$ cat /etc/services | grep telnet
telnet      23/tcp
telnet8     992/tcp          # Telnet over SSL
tfido       60177/tcp         # fidonet EMSI over telnet
```

Раньше, достаточно много лет назад, в поле `<server_path>` (или в публикациях тех лет) вы прочитали бы: `/usr/sbin/in.proftpd` или `/usr/sbin/in.telnetd`, и это были прямые путевые имена программ серверов (или ссылки на них). Но сейчас в качестве сервера записывают `/usr/sbin/tcpd` — это демон `tcpd` (система TCP-Wrappers)⁸. Его смысл в аутентификации доступа к сервисам через файлы `/etc/hosts.allow` и `/etc/hosts.deny`:

```
$ ls -l /etc/hosts.*
-rw-r--r-- 1 root root 411 map 11 2022 /etc/hosts.allow
-rw-r--r-- 1 root root 711 map 11 2022 /etc/hosts.deny
```

А вот последний параметр `<args>` и будет полным путевым именем сервера службы, он передаётся демону `tcpd` в качестве параметра «кого запускать».

⁸ Хотя во внутренней надёжной сети вполне работоспособен и старый способ с прямым указанием путевых имён серверов.

Теперь у нас готово всё чтобы проверить это в деле...

Сервер telnet

В качестве образца для экспериментов мы выберем сервис telnet. Это, пожалуй, старейший протокол удалённого доступа не только в UNIX, но вообще в IT, возникший ещё в эпоху связи по сериальным линиям скорости 9600 бит/сек. Со временем telnet, как не шифрованный, не защищённый протокол, был заменен на SSH. Кто-то может удивиться: «Почему telnet?», или даже: «Фу, telnet!». Но:

- telnet вполне применим и уместен при административных операциях внутри локальной сети;
- telnet замечательный тестер для проверки открытых портов в сети и коннекта к ним (например, все TCP порты WEB-серверов, что мы уже неоднократно делали по тексту);
- telnet, и это самое главное, предельно прост и проверка сокетной активации на нём ничем не замутнена и максимально понятна;

Установим интересующие нас пакеты, клиент и сервер протокола telnet (хотя клиент, скорее всего, уже установлен в системе совсем для других нужд):

```
$ sudo apt install telnet telnetd
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Будут установлены следующие дополнительные пакеты:
  openbsd-inetd tcpd
Следующие НОВЫЕ пакеты будут установлены:
  openbsd-inetd tcpd telnet telnetd
Обновлено 0 пакетов, установлено 4 новых пакетов, для удаления отмечено 0 пакетов, и 0
пакетов не обновлено.
Необходимо скачать 177 kB архивов.
После данной операции объём занятого дискового пространства возрастёт на 497 kB.
Хотите продолжить? [Д/н] у
Пол:1 http://deb.debian.org/debian bullseye/main amd64 tcpd amd64 7.6.q-31 [23,8 kB]
Пол:2 http://deb.debian.org/debian bullseye/main amd64 openbsd-inetd amd64 0.20160825-5
[36,8 kB]
Пол:3 http://deb.debian.org/debian bullseye/main amd64 telnet amd64 0.17-42 [71,1 kB]
Пол:4 http://deb.debian.org/debian bullseye/main amd64 telnetd amd64 0.17-42 [45,4 kB]
Получено 177 kB за 0с (747 kB/s)
Выбор ранее не выбранного пакета tcpd.
(Чтение базы данных ... на данный момент установлено 351726 файлов и каталогов.)
Подготовка к распаковке .../tcpd_7.6.q-31_amd64.deb ...
Распаковывается tcpd (7.6.q-31) ...
Выбор ранее не выбранного пакета openbsd-inetd.
Подготовка к распаковке .../openbsd-inetd_0.20160825-5_amd64.deb ...
Распаковывается openbsd-inetd (0.20160825-5) ...
Выбор ранее не выбранного пакета telnet.
Подготовка к распаковке .../telnet_0.17-42_amd64.deb ...
Распаковывается telnet (0.17-42) ...
Выбор ранее не выбранного пакета telnetd.
Подготовка к распаковке .../telnetd_0.17-42_amd64.deb ...
Распаковывается telnetd (0.17-42) ...
Настраивается пакет telnet (0.17-42) ...
update-alternatives: используется /usr/bin/telnet.netkit для предоставления
/usr/bin/telnet (telnet) в автоматическом режиме
Настраивается пакет tcpd (7.6.q-31) ...
Настраивается пакет openbsd-inetd (0.20160825-5) ...
invoke-rc.d: policy-rc.d denied execution of start.
Created symlink /etc/systemd/system/multi-user.target.wants/inetd.service →
/lib/systemd/system/inetd.service.
```



```

/usr/sbin/policy-rc.d returned 101, not running 'start inetd.service'
Настраивается пакет telnetd (0.17-42) ...
Добавление пользователя telnetd в группу utmp
invoke-rc.d: policy-rc.d denied execution of start.
Обрабатываются триггеры для man-db (2.9.4-2) ...

```

Очень характерно, что в этой инсталляции устанавливаются, по зависимостям, и суперсервер `openbsd-inetd` и TCP-wrapper `tcpd` (это значит, что в этой системе мы ещё не использовали ранее сервисов активируемых через суперсервер). Кроме того, автоматически создаётся пользователь `telnetd`.

```

$ cat /etc/passwd | grep telnet
telnetd:x:120:130::/nonexistent:/usr/sbin/nologin

```

Сразу же создадим нужные нам записи в конфигурационный файл:

```

$ grep -v ^# /etc/inetd.conf | grep -v ^$
telnet      stream  tcp4    nowait  telnetd /usr/sbin/tcpd  /usr/sbin/in.telnetd
telnet      stream  tcp6    nowait  telnetd /usr/sbin/tcpd  /usr/sbin/in.telnetd

```

Старт суперсервера:

```

# systemctl start inetd
# systemctl status inetd
• inetd.service - Internet superserver
   Loaded: loaded (/lib/systemd/system/inetd.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2023-04-27 19:40:57 EEST; 5s ago
     Docs: man:inetd(8)
  Main PID: 6382 (inetd)
    Tasks: 1 (limit: 14232)
   Memory: 704.0K
      CPU: 3ms
   CGroup: /system.slice/inetd.service
           └─6382 /usr/sbin/inetd

```

```

anp 27 19:40:57 esprimp420 systemd[1]: Starting Internet superserver...
anp 27 19:40:57 esprimp420 systemd[1]: Started Internet superserver.

```

Наш сетевой интерфейс, контролируемый со стороны суперсервера `inetd`:

```

$ ip a s dev enp3s0
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    link/ether 90:1b:0e:2b:fe:3a brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.138/24 brd 192.168.1.255 scope global dynamic noprefixroute enp3s0
        valid_lft 96177sec preferred_lft 96177sec
    inet6 fe80::921b:eff:fe2b:fe3a/64 scope link
        valid_lft forever preferred_lft forever

```

Вот взгляд со стороны серверного хоста на прослушиваемые (ожидающие) порты `telnet`:

```

# netstat -lptu | grep telnet
tcp        0      0 0.0.0.0:telnet        0.0.0.0:*           LISTEN     6382/inetd
tcp6       0      0 [::]:telnet         [::]:*              LISTEN     6382/inetd

```

Теперь взглянем на этот интерфейс со стороны любого другого хоста этой локальной сети:

```

$ nmap 192.168.1.138
Starting Nmap 7.80 ( https://nmap.org ) at 2023-04-27 18:45 EEST
Nmap scan report for 192.168.1.138
Host is up (0.011s latency).
Not shown: 997 closed ports

```

```
PORT    STATE SERVICE
22/tcp  open  ssh
23/tcp  open  telnet
53/tcp  open  domain
```

Nmap done: 1 IP address (1 host up) scanned in 0.25 seconds

Или вот так:

```
$ nmap -6 fe80::921b:eff:fe2b:fe3a%eno1
Starting Nmap 7.80 ( https://nmap.org ) at 2023-04-27 18:49 EEST
Nmap scan report for fe80::921b:eff:fe2b:fe3a
Host is up (0.0064s latency).
Not shown: 997 closed ports
PORT    STATE SERVICE
22/tcp  open  ssh
23/tcp  open  telnet
53/tcp  open  domain
```

Nmap done: 1 IP address (1 host up) scanned in 1.51 seconds

И, наконец, с этого же хоста LAN — долгожданная удалённая терминальная сессия по `telnet`, из-за которой мы всё это и городили. И проведём мы её умышленно в протоколе IPv6 (назло недоброжелателям, утверждающим что `inetd` не работает с протоколами IPv6):

```
$ telnet -6 fe80::921b:eff:fe2b:fe3a%eno1
Trying fe80::921b:eff:fe2b:fe3a%eno1...
Connected to fe80::921b:eff:fe2b:fe3a%eno1.
Escape character is '^]'.
Debian GNU/Linux 11
esprimop420 login: olej
Password:
Linux esprimop420 5.10.0-21-amd64 #1 SMP Debian 5.10.162-1 (2023-01-21) x86_64
```

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in `/usr/share/doc/*/copyright`.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

Last login: Thu Apr 27 11:27:20 EEST 2023 from fe80::13f5:9fe2:6393:bf4a%enp3s0 on pts/2

```
olej@esprimop420:~$ who
olej      tty7          2023-04-26 22:25 (:0)
olej      pts/0            2023-04-27 08:33 (192.168.1.13)
olej      pts/1            2023-04-27 10:31 (192.168.1.13)
olej      pts/3            2023-04-27 20:01 (fe80::13f5:9fe2:6393:bf4a%enp3s0)
```

```
olej@esprimop420:~$ exit
```

выход

Connection closed by foreign host.

P.S. Я умышленно выполнил на удалённом хосте команду `who`: здесь 1-я строка — это графическая сессия, окружение рабочего стола Cinnamon с которым загрузился компьютер, следующие 2 строки — это 2 SSH терминальные сессии, в которых я правлю конфиги, запускаю программы и копирую их результаты, а вот 4-я строка — это и есть `telnet` сессия в протоколе IPv6.

Сокетная активация в systemd

Точно так же, как и с кэширующим сервером DNS, разработчики systemd (Lennart Poettering с сотоварищи) просто не могли пройти мимо 40-летней истории успеха суперсерверов и запуска сервисов по сетевым запросам. В systemd этот способ запуска серверов получил название сокета-активации и его описание занимает в документации целую большую главу.

В каталоге сервисов systemd есть несколько сценариев запуска сокетной активации. В принципе, мы могли бы рассматривать и использовать сценарии для SSH:

```
$ pwd
/usr/lib/systemd/system
$ ls ssh*
ssh.service  ssh@.service  ssh.socket
```

(Здесь 1-й сценарий `ssh.service` мы уже встречали раньше — это традиционный сценарий запуска SSH сервера, а вот 2 следующих: `ssh@.service` и `ssh.socket` — именно относятся к сокетной активации.)

Но для чистоты эксперимента мы создадим схему сокетной активации совсем другого сервера, которого не по умолчанию под управлением systemd, а именно — сервера протокола FTP (попутно вспомним ещё один из популярных протоколов Интернет, которому незаслуженно не уделили внимания раньше).

О признанной популярности протокола FTP говорит хотя бы просто перечень пакетов в репозиториях дистрибутивов:

```
$ aptitude search ftpd | grep -v tftp
p ftpd - FTP-сервер
p ftpd-ssl - FTP-сервер с поддержкой SSL
p fusiondirectory-plugin-pureftpd - pureftpd plugin for FusionDirectory
p fusiondirectory-plugin-pureftpd-schema - LDAP schema for FusionDirectory pureftpd
plugin
p gosa-plugin-pureftpd - pureftpd plugin for G0sa²
p gosa-plugin-pureftpd-schema - LDAP schema for G0sa² pureftpd plugin
p inetutils-ftp - Сервер FTP
p inetutils-ftp:i386 - Сервер FTP
p mysqlmail-pure-ftp-logger - real-time logging system in MySQL - Pure-FTPd traffic-
logger
p nordugrid-arc-gridftpd - ARC GridFTP server
p owftpd - FTP daemon providing access to 1-Wire networks
v proftpd -
v proftpd-abi-1.3.7c -
p proftpd-basic - Transitional dummy package for ProFTPD
p proftpd-core - Versatile, virtual-hosting FTP daemon - binaries
p proftpd-dev - Versatile, virtual-hosting FTP daemon - development files
p proftpd-doc - Versatile, virtual-hosting FTP daemon - documentation
p proftpd-mod-autohost - ProFTPD module mod_autohost
p proftpd-mod-case - ProFTPD module mod_case
p proftpd-mod-clamav - ProFTPD module mod_clamav
p proftpd-mod-counter - ProFTPD module mod_counter
p proftpd-mod-crypto - Versatile, virtual-hosting FTP daemon - TLS/SSL/SFTP modules
v proftpd-mod-dnsbl -
p proftpd-mod-fsync - ProFTPD module mod_fsync
p proftpd-mod-geoip - Versatile, virtual-hosting FTP daemon - GeoIP module
p proftpd-mod-geoip2 - ProFTPD module mod_geoip2
p proftpd-mod-ldap - Versatile, virtual-hosting FTP daemon - LDAP module
p proftpd-mod-msg - ProFTPD module mod_msg
p proftpd-mod-mysql - Versatile, virtual-hosting FTP daemon - MySQL module
p proftpd-mod-odbc - Versatile, virtual-hosting FTP daemon - ODBC module
p proftpd-mod-pgsql - Versatile, virtual-hosting FTP daemon - PostgreSQL module
```

```

p proftpd-mod-proxy - ProFTPD module mod_proxy
p proftpd-mod-snmpp - Versatile, virtual-hosting FTP daemon - SNMP module
p proftpd-mod-sqlite - Versatile, virtual-hosting FTP daemon - SQLite3 module
p proftpd-mod-statsd - ProFTPD module mod_statsd
p proftpd-mod-tar - ProFTPD module mod_tar
p proftpd-mod-vroot - ProFTPD module mod_vroot
p proftpd-mod-wrap - Versatile, virtual-hosting FTP daemon - tcpwrapper module
p pure-ftpd - защищённый и эффективный FTP сервер
p pure-ftpd-common - Pure-FTPd FTP server (Common Files)
p pure-ftpd-ldap - защищённый и эффективный FTP-сервер с идентификацией пользователей через LDAP
p pure-ftpd-mysql - Secure and efficient FTP server with MySQL user authentication
p pure-ftpd-postgresql - Secure and efficient FTP server with PostgreSQL user authentication
p python-pyftplib-doc - documentation for Python FTP server library
p python3-pyftplib - Python FTP server library (Python 3)
p twoftpd - простой и защищённый FTP-сервер (программы)
p twoftpd-run - a simple secure efficient FTP server
p vsftpd - легковесный, эффективный FTP-сервер, написанный с упором на безопасность
p vsftpd-dbg - lightweight, efficient FTP server written for security (debug)

```

Как видим, представлено на выбор множество разных FTP серверов, можете выбрать для экспериментов любой, на свой вкус (мне хорошо известны в разные годы работы: proftpd, vsftpd, pure-ftpd, поэтому, во избежание сюрпризов, выбираем что-то из них):

```

$ sudo apt install pure-ftpd
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Следующие пакеты устанавливались автоматически и больше не требуются:
  libhiredis0.14 libmemcached11 libmemcachedutil2 proftpd-doc
Для их удаления используйте «sudo apt autoremove».
Будут установлены следующие дополнительные пакеты:
  openbsd-inetd pure-ftpd-common tcpd
Следующие пакеты будут УДАЛЕНЫ:
  proftpd-core
Следующие НОВЫЕ пакеты будут установлены:
  openbsd-inetd pure-ftpd pure-ftpd-common tcpd
...

```

Для организации сокетной активации этого сервера создадим 2 файла в каталоге сценариев systemd (всё это делаем с правами root):

```

# pwd
/usr/lib/systemd/system
# touch pure-ftpd.socket
# touch pure-ftpd@.service

```

И заполняем их следующим содержимым (сервис proftpd сам устанавливает при инсталляции свои сценарии сокетной активации, поэтому можете их взять за основу):

```

$ cat /usr/lib/systemd/system/pure-ftpd.socket
[Unit]
Description=pure-ftp FTP Server Activation Socket
Conflicts=pure-ftp.service

[Socket]
ListenStream=21
Accept=true

```

```
[Install]
WantedBy=sockets.target

$ cat /usr/lib/systemd/system/pure-ftpd@.service
[Unit]
Description=pure-ftpd FTP Server
After=network-online.target

[Service]
ExecStart=-/usr/sbin/pure-ftpd
StandardInput=socket
```

Собственно, на этом всё! Но для таких экспериментов, или тестирования результатов своей работы, нужно тщательно **убедиться**, что никакой другой сервер FTP статически не конфигурирован в системе:

```
# systemctl status pure-ftpd
• pure-ftpd.service
  Loaded: loaded (/etc/init.d/pure-ftpd; generated)
  Active: active (exited) since Fri 2023-04-28 15:36:57 EEST; 1h 32min ago
  Docs: man:systemd-sysv-generator(8)
  Process: 19009 ExecStart=/etc/init.d/pure-ftpd start (code=exited, status=0/SUCCESS)
  CPU: 59ms

anp 28 15:36:57 R420 systemd[1]: Starting pure-ftpd.service...
anp 28 15:36:57 R420 pure-ftpd[19009]: Starting ftp server:
anp 28 15:36:57 R420 pure-ftpd[19019]: Running: /usr/sbin/pure-ftpd -l pam -E -J HIGH -u
1000 -O clf:/var/log/pure-ftpd/transfer.log -B
anp 28 15:36:57 R420 systemd[1]: Started pure-ftpd.service.
anp 28 15:36:57 R420 pure-ftpd[19020]: (?@?) [ERROR] Unable to start a standalone server:
[Address already in use]
```

Это не должно быть сюрпризом: **все** проекты FTP стартуют сразу при инсталляции пакетов, в отличие от многих сервисов которые мы разбирали раньше!

```
# systemctl stop pure-ftpd
# systemctl status pure-ftpd
○ pure-ftpd.service
  Loaded: loaded (/etc/init.d/pure-ftpd; generated)
  Active: inactive (dead) since Fri 2023-04-28 17:10:20 EEST; 2s ago
  Docs: man:systemd-sysv-generator(8)
  Process: 19009 ExecStart=/etc/init.d/pure-ftpd start (code=exited, status=0/SUCCESS)
  Process: 22165 ExecStop=/etc/init.d/pure-ftpd stop (code=exited, status=0/SUCCESS)
  CPU: 26ms

...
```

P.S. Не ищите сценарии запуска и установки в привычном каталоге `/usr/lib/systemd/system` — серверы FTP помещают свои скрипты управления «в старом стиле» в каталог `/etc/init.d` (команды `systemd`, к счастью, умеют так же хорошо находить и работать с сервисами конфигурированными «в старом стиле»).

```
$ ls -l /etc/init.d/ | grep ftp
-rwxr-xr-x 1 root root 5322 сен 19  2021 proftpd
-rwxr-xr-x 1 root root 3172 янв 31  2022 pure-ftpd
```

Кроме непосредственно самих сервисов FTP нужно остановить и `inetd` (`xinetd`) чтобы избежать маскирующего запуска FTP средствами суперсервера:

```
# systemctl status inetd
• inetd.service - Internet superserver
  Loaded: loaded (/lib/systemd/system/inetd.service; enabled; vendor preset: enabled)
```

```

    Active: active (running) since Fri 2023-04-28 15:36:56 EEST; 1h 38min ago
    Docs: man:inetd(8)
Main PID: 18954 (inetd)
    Tasks: 1 (limit: 115786)
    Memory: 548.0K
    CPU: 401ms
    CGroup: /system.slice/inetd.service
            └─18954 /usr/sbin/inetd

anp 28 15:36:56 R420 systemd[1]: Starting Internet superserver...
anp 28 15:36:56 R420 systemd[1]: Started Internet superserver.
# systemctl stop inetd
# systemctl status inetd
○ inetd.service - Internet superserver
   Loaded: loaded (/lib/systemd/system/inetd.service; enabled; vendor preset: enabled)
   Active: inactive (dead) since Fri 2023-04-28 17:15:25 EEST; 1s ago
     Docs: man:inetd(8)
  Process: 18954 ExecStart=/usr/sbin/inetd (code=exited, status=0/SUCCESS)
 Main PID: 18954 (code=exited, status=0/SUCCESS)
    CPU: 402ms
...

```

Окончательная проверка на чистоту эксперимента перед запуском созданного сервиса — здесь ничего не должно быть (в ответ):

```

# systemctl --full | grep ftp
# ps -A | grep ftp

```

Теперь запускаем свой сконфигурированный для FTP сервис (сокетной активации):

```

# systemctl start pure-ftpd.socket
# systemctl status pure-ftpd.socket
● pure-ftpd.socket - pure-ftp FTP Server Activation Socket
   Loaded: loaded (/lib/systemd/system/pure-ftpd.socket; disabled; vendor preset:
enabled)
   Active: active (listening) since Fri 2023-04-28 17:14:27 EEST; 5s ago
     Listen: [::]:21 (Stream)
  Accepted: 0; Connected: 0;
    Tasks: 0 (limit: 115786)
    Memory: 8.0K
    CPU: 845us
    CGroup: /system.slice/pure-ftpd.socket

```

```

anp 28 17:14:27 R420 systemd[1]: Listening on pure-ftp FTP Server Activation Socket.

```

Всё! Сервер FTP в готовности и ожидании. Что мы и проверим с любого иного хоста LAN:

```

$ ftp 192.168.1.13
Connected to 192.168.1.13.
220----- Welcome to Pure-FTPd [privsep] [TLS] -----
220-You are user number 1 of 50 allowed.
220-Local time is now 17:19. Server port: 21.
220-IPv6 connections are also welcome on this server.
220 You will be disconnected after 15 minutes of inactivity.
Name (192.168.1.13:olej): olej
331 User olej OK. Password required
Password:
230 OK. Current directory is /home/olej
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> pwd

```

```

257 "/home/olej" is your current location
ftp> system
215 UNIX Type: L8
ftp> exit
221-Goodbye. You uploaded 0 and downloaded 0 kbytes.
221 Logout.

```

Здесь показана полная FTP-сессия в режиме диалога, от её начала до конца, как она происходила на клиентском компьютере. А на сервере во время этой сессии посмотрим:

```

$ systemctl --full | grep ftp
pure-ftpd@0-192.168.1.13:21-192.168.1.241:48136.service loaded active running pure-
ftpd FTP Server (192.168.1.241:48136)
system-pure\x2dftpd.slice loaded active active
Slice /system/pure-ftpd
pure-ftpd.socket loaded active listening pure-
ftp FTP Server Activation Socket

```

И наконец ... «как вишенка на торте», я зайду по SSH на сервер `linux.ru.ru`, базирующийся за тысячи километров в Казахстане ... как легко видеть, это арендуемый виртуальный сервер крупного регионального провайдера:

```

$ sudo inxi -Mxxx
Machine:      Type: Kvm System: QEMU product: Standard PC (i440FX + PIIX, 1996) v: pc-i440fx-
bionic serial: N/A Chassis: type: 1
              v: pc-i440fx-bionic serial: N/A
              Mobo: N/A model: N/A serial: N/A BIOS: SeaBIOS v: 1.13.0-1ubuntu1.1 date:
04/01/2014
$ ip -6 a s dev tun0
3: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 state UNKNOWN qlen 500
    inet6 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::d86f:9577:d828:cb4d/64 scope link stable-privacy
        valid_lft forever preferred_lft forever

```

Это вот ping ICMPv6 к хосту **локальной сети** на которой я веду отработку FTP (через тысячи километров, местного провайдера Интернет, роутер-шлюз выхода из локальной сети в Интернет — это к вопросу грядущей «прозрачности» IPv6):

```

$ ping -c3 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f
PING 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f(21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f) 56
data bytes
64 bytes from 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f: icmp_seq=1 ttl=64 time=426 ms
64 bytes from 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f: icmp_seq=2 ttl=64 time=118 ms
64 bytes from 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f: icmp_seq=3 ttl=64 time=118 ms

--- 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 117.950/220.699/426.072/145.220 ms

```

А вот клиентская сессия FTP в IPv6 из этого удалённого хоста к только что созданному серверу FTP с сокетной активацией:

```

$ ftp 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f
Connected to 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f.
220----- Welcome to Pure-FTPd [privsep] [TLS] -----
220-You are user number 1 of 50 allowed.
220-Local time is now 18:30. Server port: 21.
220 You will be disconnected after 15 minutes of inactivity.
Name (21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f:olej): olej
331 User olej OK. Password required
Password:

```

```
230 OK. Current directory is /home/olej
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> exit
221-Goodbye. You uploaded 0 and downloaded 0 kbytes.
221 Logout.
```

Прокси-сервера

Дословно термин *proxy* — доверенное лицо. Принцип состоит в том, что клиент осуществляет запросы к целевому ресурсу не непосредственно, а как косвенные запросы через некоторый промежуточный прокси-сервер. Делается это с различными целями:

- Клиент может физически не иметь прямого доступа в Интернет, как это имеет место массово в локальных сетях (корпоративных, ведомственных), компьютеры которых имеют частные IPv4 адреса, не маршрутизируемые в сети. В такой архитектуре запросы направляются через прокси-сервер, являющийся для клиентов шлюзом в Интернет.
- Проксирование во внешней сети может использоваться для преодоления территориальных ограничений накладываемых организационно. Особенно возрастает такое использование в последнее время с разрушением цельности Интернет в результате военных действий, когда одни страны ограничивают трафик в или из других стран. В таких случаях прокси-сервер, расположенный в нейтральной зоне позволяет восстановить целостность.
- Прокси-сервер позволяет защищать компьютер клиента от некоторых сетевых атак снаружи.
- Прокси-сервер в некоторой (минимальной) степени помогает сохранять анонимность клиента (за счёт подмены IP).
- Очень часто прокси-сервер используют разнообразные мошенники и недоброжелатели для скрытия своего IP адреса (и, как следствие, средствами геолокации выяснения местоположения). Поэтому обнаружения факта проксирования, используемого клиентом, уже должно вызывать повышенную настороженность.



Рис. 2.17 Логика прокси

Проксирование (передача полномочий) может производиться на разных уровнях протокола TCP/IP, в связи с этим и различают группы широко используемых прокси-серверов:

- Проксирование на уровне протоколов прикладных уровней. Здесь массово используемыми являются HTTP/HTTPS прокси-сервера (HTTPS требует аутентификацию клиента в отличие от HTTP) — это самая известная и самая древняя категория. Понятно, что с такими прокси-серверами может работать только ограниченная группа клиенты, работающих в протоколах HTTP/HTTPS, но это и самая массовая группа: браузеры, передача файлов, утилиты `wget`, `curl`, некоторые формы DNS протоколов... Самый известный проект этой группы `squid`, из малых реализаций — например, `polipo`, рекомендуемый к использованию в проекте TOR, но последние релизы `polipo` относятся к 2014 году.
- Проксирование на уровне протоколов транспортных (L4) протоколов TCP и, иногда UDP. Здесь самая известная категория — это SOCKS4 и SOCKS5 прокси-сервера (различие между ними в том, что SOCKS5 допускает настройками возможность аутентификации и поддерживает UDP протокол). Из числа самых известных проектов можно назвать проекты Dante и 3proxy.
- Сервер 3proxy умеет, в зависимости от конфигурации, осуществлять проксирование как SOCKS, так и HTTP/HTTPS.
- По существу, повсеместно используемая в LAN IPv4 трансляция адресов NAT, осуществляемая на сетевом (L3) уровне протокола IP механизмом ядра `netfilter` (о нём говорили раньше), что работает на шлюзах LAN в Интернет по умолчанию — также является прокси (использование посредника). Но его в этом качестве прокси не называют.

Создание собственного прокси-сервера (SOCKS5), скажем на небольшом арендуемом VDS (виртуальном сервере провайдера), посмотрим на примере реализации Dante:

```
$ aptitude search Dante
```

```
p  dante-client
p  dante-server
```

```
- SOCKS wrapper for users behind a firewall
- SOCKS (v4 and v5) proxy daemon (danted)
```

В этой паре dante-client это средство «научить» любое приложение работать через SOCKS5 (из числа тех кто не умеют это делать), и ещё как некоторое средство в ограниченных функциях файервола. Мы к нему ещё вернёмся, в пару, слов позже...

А сейчас нас интересует именно сервер:

```
$ sudo apt install dante-server
Чтение списков пакетов... Готово
Построение дерева зависимостей
Чтение информации о состоянии... Готово
Следующие НОВЫЕ пакеты будут установлены:
  dante-server
Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 6
пакетов не обновлено.
Необходимо скачать 375 kB архивов.
После данной операции объём занятого дискового пространства возрастёт на 1.013 kB.
Пол:1 http://mirror.timeweb.ru/debian buster/main amd64 dante-server amd64 1.4.2+dfsg-6
[375 kB]
Получено 375 kB за 1с (519 kB/s)
Выбор ранее не выбранного пакета dante-server.
(Чтение базы данных ... на данный момент установлено 47759 файлов и каталогов.)
Подготовка к распаковке .../dante-server_1.4.2+dfsg-6_amd64.deb ...
Распаковывается dante-server (1.4.2+dfsg-6) ...
Настраивается пакет dante-server (1.4.2+dfsg-6) ...
Created symlink /etc/systemd/system/multi-user.target.wants/danted.service →
/lib/systemd/system/danted.service.
Обрабатываются триггеры для man-db (2.8.5-2) ...
Mar 19 20:00:01 R420 kernel: [38065.020377] EXT4-fs (sdc1): mounted filesystem with
ordered data mode. Opts: (null)
. Quota mode: none.
Обрабатываются триггеры для systemd (241-7~deb10u8) ...

$ sudo danted -v
Dante v1.4.2. Copyright (c) 1997 - 2014 Inferno Nettverk A/S, Norway
```

После установки danted попытается средствами systemd стартовать, но это ему не удастся и завершится ошибкой, поскольку его нужно конфигурировать для нормальной работы. Его конфигурационный файл:

```
$ ls -l /etc/danted.conf
-rw-r--r-- 1 root root 8134 янв  5  2019 /etc/danted.conf
```

В первом приближении, без авторизации, это может выглядеть так (файл конфигурации тщательнейшим образом прокомментирован):

```
$ grep -v '^$|^#' danted.conf
logoutput: stderr
internal: eth0 port = 1080
external: eth0
socksmethod: none
user.privileged: proxy
user.unprivileged: nobody
user.libwrap: nobody
client pass {
    from: 0/0 to: 0/0
    log: connect disconnect error ioop
}
socks pass {
    from: 0/0 to: 0/0
    log: connect disconnect error ioop
}
```

Здесь (из самого важного):

- `internal` и `external` — это входящий и исходящий интерфейсы, могут быть заданы как именами, там и указанием их IP адресов;
- `port` — это прослушиваемый (проксируемый) порт SOCKS4/5;
- `socksmethod` — собственно, и определяет способ авторизации, `none` — без авторизации;

Теперь мы можем производить запуск, в общем обычным для серверов путём, но мы должны добиться безошибочный старт (более-менее правильный файл конфигурации):

```
# systemctl start danted
# systemctl status --no-pager --full danted
• danted.service - SOCKS (v4 and v5) proxy daemon (danted)
   Loaded: loaded (/lib/systemd/system/danted.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2023-03-19 22:11:32 MSK; 3s ago
     Docs: man:danted(8)
           man:danted.conf(5)
   Process: 9542 ExecStartPre=/bin/sh -c uid=`sed -n -e "s/[[[:space:]]//g" -e "s/#.*//" -e "/^user\\.privileged/{s/[^:]*://p;q;}" /etc/danted.conf`; if [ -n "$uid" ]; then touch /var/run/danted.pid; chown $uid /var/run/danted.pid; fi (code=exited, status=0/SUCCESS)
 Main PID: 9546 (danted)
    Tasks: 20 (limit: 1149)
   Memory: 20.4M
   CGroup: /system.slice/danted.service
           └─9546 /usr/sbin/danted
             └─9547 danted: monitor
               └─9548 danted: negotia
                 └─9549 danted: request
                   └─9550 danted: request
                     └─9551 danted: request
                       └─9552 danted: request
                         └─9553 danted: request
                           └─9554 danted: request
                             └─9555 danted: request
                               └─9556 danted: request
                                 └─9557 danted: request
                                   └─9558 danted: request
                                     └─9559 danted: request
                                       └─9560 danted: request
                                         └─9561 danted: request
                                           └─9562 danted: request
                                             └─9563 danted: request
                                               └─9564 danted: request
                                                 └─9565 danted: io-chil
```

```
map 19 22:11:32 277938.local systemd[1]: Starting SOCKS (v4 and v5) proxy daemon (danted)...
```

```
map 19 22:11:32 277938.local systemd[1]: Started SOCKS (v4 and v5) proxy daemon (danted).
```

```
map 19 22:11:33 277938.local danted[9546]: Mar 19 22:11:33 (1679253093.602826)
danted[9546]: info: Dante/server[1/1] v1.4.2 running
```

Но для того, чтобы убедиться в работоспособности сервера, мы должны прежде разрешить на файерволе (если он работает на этом хосте) разрешить заказанный входной порт, или приостановить (временно) работу файервола:

```
# ufw allow 1080
Rule added
```

И теперь с хоста, возможно весьма удалённого, возможно в локальной сети с NAT и ограниченным числом разрешённых портов внаружу, можем проверить работоспособность SOCKS прокси:

```
$ curl -x socks5://90.156.230.27:1080 ifconfig.co
90.156.230.27
```

Здесь, как должно быть понятно, 90.156.230.27 — это и есть IP удалённого сервера с «белым» адресом, для которого я и проделываю настройку. И контрольный запрос с того же места, но без задействования прокси:

```
$ curl ifconfig.co
193.28.177.124
```

Здесь в качестве ответа я получаю совсем другой IP 193.28.177.124 — это динамический адрес, который присваивает, за счёт динамической трансляции NAT, мой **провайдер** Интернет всем компьютерам в моей локальной сети.

Сервер работоспособен ... Но оставить его в такой конфигурации нельзя. Опыт проводимых отладочных работ показал (файлы системных журналов), что только за пару часов отладки сервера на его использование набежали до сотни незваных мерзавцев, непрерывно барражирующих в Интернет, и сканирующих везде не закрытые порты и прокси-сервера. И если завтра через ваш прокси-сервер взломают и разденут на миллион долларов Bank of America, то засвечен будет IP вашего прокси-сервера и претензии будут предъявляться именно вам. Поэтому нужно сразу конфигурировать аутентификацию пользователей на прокси-сервере ... и, возможно, сменить типовой порт 1080 на другой (на практике приходится видеть в этом качестве: 5555, 1085 ...). Кроме того, хотелось бы добавить и обслуживание адресов IPv6. В конечном итоге (после целого ряда изменений и перезапуска danted) файл конфигурации может принять вид подобный следующему:

```
$ grep -v "^$|^#" /etc/danted.conf
logoutoutput: stderr
internal: 90.156.230.27 port = 1085
internal: 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 port = 1085
external: 90.156.230.27
external: 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
socksmethod: username
user.privileged: root
user.unprivileged: nobody
user.libwrap: nobody
client pass {
    from: 0/0 to: 0/0
    log: error ioop
}
socks pass {
    from: 0/0 to: 0/0
    log: error ioop
    group: proxy
}
```

Для сетевых интерфейсов сервера:

```
$ ip a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
```

```

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    link/ether 2e:49:10:b4:b2:02 brd ff:ff:ff:ff:ff:ff
    inet 90.156.230.27/24 brd 90.156.230.255 scope global dynamic eth0
        valid_lft 63386sec preferred_lft 63386sec
    inet6 fe80::2c49:10ff:feb4:b202/64 scope link
        valid_lft forever preferred_lft forever
3: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc pfifo_fast state
UNKNOWN group default qlen 500
    link/none
    inet6 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::d86f:9577:d828:cb4d/64 scope link stable-privacy
        valid_lft forever preferred_lft forever

```

В файле конфигурации мы теперь поменяли тип аутентификации на `username`, а сетевые интерфейсы (и входные и выходные) определяем не именами, а их адресами (и IPv4 и IPv6). Но теперь нам нужно:

- создать (если она не создавалась при инсталляции пакета) группу пользователей `proxy`:
- создать (одного или нескольких) имён пользователей для пользования прокси, без возможности логирования в терминальной сессии:

```
# useradd --shell /usr/sbin/nologin proxy_user_02
```
- добавить пользователям пароли использования проксирования:

```
# passwd proxy_user_02
```

Новый пароль :
Повторите ввод нового пароля :
passwd: пароль успешно обновлён
- добавить таких пользователей в группу `proxy`:

```
# sudo usermod -a -G proxy proxy_user_02
```

В итоге, только пользователи с этими именами (здесь их 2) могут авторизоваться на этом прокси-сервере:

```

$ cat /etc/group | grep proxy
proxy:x:13:proxy_user_01,proxy_user_02
$ cat /etc/passwd | grep proxy
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
proxy_user_01:x:1003:1003::/home/proxy_user_01:/usr/sbin/nologin
proxy_user_02:x:1004:1004::/home/proxy_user_02:/usr/sbin/nologin

```

Вот теперь можно испытывать и далее использовать SOCKS5 прокси ... и увидеть запаздывание которое вносит проксирование (относительно прямого подключения к тому же URL):

```

$ time curl -x socks5://proxy_user_02:xxxxxx@90.156.230.27:1085 check-host.net/ip
90.156.230.27
real 0m0,940s
user 0m0,008s
sys 0m0,009s
$ time curl --no-proxy '*' check-host.net/ip
193.28.177.119
real 0m0,125s
user 0m0,004s
sys 0m0,010s

```

А вот так выглядит обращение к HTTP серверу по его IPv6 адресу:

```
$ curl -x socks5://proxy_user_02:xxxxxx@90.156.230.27:1085 \
[221:58c9:9a6:99be:f3d:c1ac:2b5b:9771] > /dev/null
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %             0         0      0  42897       0  --:--:--  0:00:01 --:--:-- 42902
```

Наконец, последнее действие, полезное в использовании прокси — это определить на своих клиентских компьютерах системными переменными параметры используемых прокси по умолчанию, чтобы в каждой команде не набирать эти непростые параметры. Для того чтобы эти значения восстанавливались после перезагрузки системы (клиента) пропишем их в файл `/etc/environment`:

```
$ cat /etc/environment
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
export all_proxy=socks5://proxy_user_02:xxxxxx@90.156.230.27:1085/
export ALL_PROXY=socks5://proxy_user_02:xxxxxx@90.156.230.27:1085/
export no_proxy="localhost,127.0.0.0/8,192.168.1.0/24,::1"
```

Тогда программы, которые умеют использовать прокси (но далеко не все это умеют) будут использовать определения в переменных окружения:

```
$ env | grep proxy
no_proxy=localhost,127.0.0.0/8,192.168.1.0/24,::1
ALL_PROXY=socks5://proxy_user_02:xxxxxx@90.156.230.27:1085/
all_proxy=socks5://proxy_user_02:xxxxxx@90.156.230.27:1085/
```

В том что прокси теперь подхватывается по умолчанию можно убедиться по времени отклика команды `curl` без явного указания ему использовать прокси:

```
$ time curl check-host.net/ip
90.156.230.27
real 0m0,940s
user 0m0,008s
sys 0m0,009s
```

Прокси сквозь SSH

Как уже упоминалось, протокол SSH может туннелировать множество других сетевых протоколов. Таким способом среди множества других возможностей представляется возможность организовать SOCKS-проксирование сквозь такой туннель. Организуем его вот таким SSH подключением к любому серверу в Интернет, к которому у вас есть доступ:

```
$ ssh -N -D 5555 90.156.230.27
olej@90.156.230.27's password:
...
```

Обратите внимание! Мы хотим создать прокси через удалённый сервер через произвольный порт 5555, который, естественно, не открыт как входной на файерволе `ufw` сервера. В этом случае трафик прокси будет туннелироваться через порт 22 протокола SSH.

Если вы на эту команду `ssh` создания прокси получите в ответ: `Permission denied, please try again`. — то с **другого** терминала **этого** же хоста от имени **этого** пользователя вы можете выполнять:

```
$ curl -s -x socks4://127.0.0.1:5555 ifconfig.co
90.156.230.27
$ curl -s -x socks5://127.0.0.1:5555 ifconfig.co
90.156.230.27
```

P.S. Оговорка об ошибках команды установления туннеля здесь не случайная и не лишняя! В ней имя пользователя **явно** нельзя указывать, неявно предполагается текущее имя логина на

запрашиваемом клиенте. А пароль, затем, запрашивается на сервере SSH для этого же имени. Для подтверждения того как это происходит можете выполнить команду в терминале от root:

```
$ sudo ssh -N -D 5555 90.156.230.27
[sudo] пароль для olej:
root@90.156.230.27's password:
```

Здесь запрашивается 2 разных пароля: 1-й для sudo на локальном клиенте, второй — на удалённом сервере. Если вы будете упорствовать в вводе «не того» пароля, то после нескольких попыток (обычно 2-3) сервер оборвёт все текущие и новые SSH подключения чтобы воспрепятствовать дальнейшим попыткам. Этот «молчание» будет продолжаться несколько минут, значение определяется настройками параметров логирования на сервере.

Клиенты прокси

Многие сетевые клиенты сами умеют работать с проксированием. Некоторым программам использование прокси нужно указать явно: опциями командной строки, как, например, curl, другим — в настроечных параметрах, как браузер FireFox. Другие программы вообще не имеют возможностей индивидуальной настройки (например все браузеры производные от Chromium), а используют переменные окружения из числа тех что были показаны выше: all_proxy, http_proxy, socks_proxy.

Но встречаются программы, которые принципиально не умеют использовать прокси (как пример программа управления версиями cvs). А иногда и для программ названных абзацем выше (то есть так или иначе умеют использовать прокси) из-за конкретики задачи неудобно использовать настроечные параметры. Для таких случаев предусмотрены пакет (проект) dante-client и его программа (команда) socksify:

```
# aptitude install dante-client
Следующие НОВЫЕ пакеты будут установлены:
  dante-client libsocksd0{a} libfile-which-perl{a}
0 пакетов обновлено, 3 установлено новых, 0 пакетов отмечено для удаления, и 2 пакетов не
обновлено.
Необходимо получить 207 кБ архивов. После распаковки 578 кБ будет занято.
Хотите продолжить? [Y/n/?] y
Получить: 1 http://mirror.mirohost.net/ubuntu jammy/universe amd64 libsocksd0 amd64
1.4.2+dfsg-7build4 [179 kB]
Получить: 2 http://mirror.mirohost.net/ubuntu jammy/main amd64 libfile-which-perl all
1.23-1 [13,8 kB]
Получить: 3 http://mirror.mirohost.net/ubuntu jammy/universe amd64 dante-client all
1.4.2+dfsg-7build4 [14,3 kB]
Получено 207 кБ в 1с (390 кБ/с)
Выбор ранее не выбранного пакета libsocksd0:amd64.
(Чтение базы данных ... на данный момент установлено 551432 файла и каталога.)
Подготовка к распаковке .../libsocksd0_1.4.2+dfsg-7build4_amd64.deb ...
Распаковывается libsocksd0:amd64 (1.4.2+dfsg-7build4) ...
Выбор ранее не выбранного пакета libfile-which-perl.
Подготовка к распаковке .../libfile-which-perl_1.23-1_all.deb ...
Распаковывается libfile-which-perl (1.23-1) ...
Выбор ранее не выбранного пакета dante-client.
Подготовка к распаковке .../dante-client_1.4.2+dfsg-7build4_all.deb ...
Распаковывается dante-client (1.4.2+dfsg-7build4) ...
Настраивается пакет libfile-which-perl (1.23-1) ...
Настраивается пакет libsocksd0:amd64 (1.4.2+dfsg-7build4) ...
Настраивается пакет dante-client (1.4.2+dfsg-7build4) ...
Обрабатываются триггеры для man-db (2.10.2-1) ...
Обрабатываются триггеры для libc-bin (2.35-0ubuntu3.1) ...
$ which socksify
/usr/bin/socksify
```

Программа `socksify` умеет работать с прокси как HTTP так и SOCKS. Но для её успешного использования нам предстоит настроить конфигурацию `/etc/dante.conf`.

Для этого, прежде, создадим временный туннель SOCKS средствами SSH без авторизации, как было показано ранее, который иногда называют как «VPN для бедных»:

```
$ ssh -N -D 5555 90.156.230.27
olej@90.156.230.27's password:
```

Почему «выскочило» такое имя пользователя мы уже выясняли, но ещё подробно вернёмся к этому вопросу. А для использования такого прокси-туннеля пропишем в конфигурацию `dante` (всё остальное в этом достаточно большом файле — это комментированные строки примеров применения для различных случаев):

```
$ grep -v "#\|^$" /etc/dante.conf
route {
    from: 0.0.0.0/0 to: 0.0.0.0/0 via: 127.0.0.1 port = 5555
    protocol: tcp udp
    proxyprotocol: socks_v5
    method: none
}
```

И испытываем полученный SOCKS-прокси на работоспособность:

```
$ curl -s -x socks4://127.0.0.1:5555 ifconfig.co
90.156.230.27
$ curl --noproxy '*' ifconfig.co
193.28.177.119
$ socksify curl --noproxy '*' ifconfig.co
90.156.230.27
```

Дальше перейдём к использованию стационарного сервера `danted` (SOCKS прокси), который мы оставили работающим постоянно на удалённом сервере `90.156.230.27` 2-мя главами ранее... Он требует, нашими стараниями, авторизации пользователя именем и паролем. Для этого перепишем конфигурационный файл клиента так:

```
$ grep -v "#\|^$" /etc/dante.conf
route {
    from: 0.0.0.0/0 to: 0.0.0.0/0 via: 90.156.230.27 port = 1085
    protocol: tcp udp
    proxyprotocol: socks_v4 socks_v5
    method: username
}
```

Здесь, попутно, заметим, что программа `dante` это не сервер, и не потребует рестарта или посылки сигнала (`kill ...`) перечтения конфигурации — он перечитает конфигурацию сразу же при выполнении клиентского запроса.

Первая попытка так использовать прокси-сервер «с налёта» завершится неудачей:

```
$ socksify curl --noproxy '*' check-host.net/ip
olej@90.156.230.27.1080 socks password:
olej@90.156.230.27.1080 socks password:
curl: (7) Failed to connect to check-host.net port 80 after 33901 ms: В соединении отказано
```

Это связано с тем, что в качестве имени авторизации на SOCKS-прокси отправляется текущее имя пользователя, в чьём сеансе отправляется запрос. А на прокси-сервере сервера мы специально создали имена новых пользователей, которым разрешено авторизоваться для использования прокси, и поместили их там в группу пользователей `proxu` (см. 2-мя главами

выше). Но на клиентском компьютере у меня (и на многих других компьютерах) нет **таких** пользователей, даже если я попытаюсь проделать команду от такого имени, что-то типа:

```
$ su proxy_user_02 -c "socksify curl --no-proxy '*' check-host.net/ip"
su: user proxy_user_02 does not exist or the user entry does not contain all the required fields
```

Синтаксически верно, но с точки зрения операционной системы бессмысленно!

Разнообразные перепробованные варианты заставить socksify выполняться от имени произвольного пользователя, как и поиск по Интернет такой возможности, не увенчались успехом. Команда socksify не допускает опций ожидаемой подсказки (--help или -h), но помощь пришла из man страницы в виде списка предопределённых переменных (этой и многих других) окружения программы:

```
$ man socksify
...
SOCKS_USERNAME
        Use the value of SOCKS_USERNAME as the username when doing username authentication.
```

Теперь (и успешно!) команду использования прокси через socksify выполняем так (2-я форма показана для сравнения):

```
$ export SOCKS_USERNAME=proxy_user_02; socksify curl --no-proxy '*' ifconfig.co
proxy_user_02@90.156.230.27:1080 socks password:
90.156.230.27
$ curl -x socks5://proxy_user_02:xxxxxx@90.156.230.27 ifconfig.co
90.156.230.27
```

Кто и как использует прокси?

С ростом возможностей использования разнообразных прокси (и косвенных путей запросов клиентов) можно легко запутаться кто из программ умеет использовать и кто использует проксирование, ... в каком направлении и каком протоколе? Для этого установите такую простую утилиту:

```
$ sudo apt install sockstat
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Следующие НОВЫЕ пакеты будут установлены:
  sockstat
Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 0
пакетов не обновлено.
Необходимо скачать 12,5 kB архивов.
После данной операции объём занятого дискового пространства возрастёт на 37,9 kB.
Пол:1 http://ftp.bme.hu/debian bullseye/main amd64 sockstat amd64 0.4.1-1 [12,5 kB]
Получено 12,5 kB за 0с (45,5 kB/s)
Выбор ранее не выбранного пакета sockstat.
(Чтение базы данных ... на данный момент установлен 367121 файл и каталог.)
Подготовка к распаковке .../sockstat_0.4.1-1_amd64.deb ...
Распаковывается sockstat (0.4.1-1) ...
Настраивается пакет sockstat (0.4.1-1) ...
Обрабатываются триггеры для man-db (2.9.4-2) ...

$ sockstat -h
usage: sockstat [-46clhu] [-p ports] [-U uid|user] [-G gid|group] [-P pid|process]
               [-R protocol]
      protocol = 'tcp' or 'udp' or 'raw' or 'unix'
```

Вот возможный вид использования прокси-серверов в весьма нагруженной приложениями (специально) системе:

\$ sockstat -4

USER	PROCESS	PID	PROTO	SOURCE ADDRESS	FOREIGN ADDRESS	STATE
olej	firefox-bin	4480	tcp4	192.168.1.14:60232	198.16.66.101:436	ESTABLISHED
olej	firefox-bin	4480	tcp4	192.168.1.14:44514	198.16.66.101:436	ESTABLISHED
olej	firefox-bin	4480	tcp4	192.168.1.14:37994	198.16.66.101:436	ESTABLISHED
...						
olej	firefox-bin	4480	tcp4	192.168.1.14:54268	198.16.66.101:436	ESTABLISHED
olej	Telegram	5479	tcp4	127.0.0.1:54978	127.0.0.1:9050	ESTABLISHED
olej	thunderbird	6127	tcp4	127.0.0.1:36602	127.0.0.1:9050	ESTABLISHED
olej	thunderbird	6127	tcp4	127.0.0.1:44294	127.0.0.1:9050	ESTABLISHED
olej	thunderbird	6127	tcp4	127.0.0.1:59310	127.0.0.1:9050	ESTABLISHED
olej	thunderbird	6127	tcp4	127.0.0.1:55596	127.0.0.1:9050	ESTABLISHED
olej	thunderbird	6127	tcp4	127.0.0.1:58848	127.0.0.1:9050	ESTABLISHED
olej	Viber	27304	tcp4	127.0.0.1:30666	*:*	LISTEN
olej	Viber	27304	tcp4	127.0.0.1:45112	*:*	LISTEN
olej	Viber	27304	tcp4	192.168.1.14:37880	44.192.201.149:4244	ESTABLISHED
olej	ssh	36721	tcp4	192.168.1.14:47460	192.168.1.138:22	ESTABLISHED
olej	ssh	36837	tcp4	192.168.1.14:43592	192.168.1.241:22	ESTABLISHED
olej	chrome	39461	udp4	224.0.0.251:5353	*:*	CLOSED
olej	chrome	39461	udp4	224.0.0.251:5353	*:*	CLOSED
olej	chrome	39510	tcp4	127.0.0.1:55572	127.0.0.1:9050	ESTABLISHED
olej	opera	40562	udp4	224.0.0.251:5353	*:*	CLOSED
olej	opera	40562	udp4	224.0.0.251:5353	*:*	CLOSED
olej	opera	40598	tcp4	127.0.0.1:50608	127.0.0.1:9050	ESTABLISHED
olej	opera	40598	udp4	224.0.0.251:5353	*:*	CLOSED
olej	opera	40598	udp4	224.0.0.251:5353	*:*	CLOSED
olej	opera	40598	udp4	*:40987	*:*	CLOSED
olej	chromium	42821	udp4	224.0.0.251:5353	*:*	CLOSED
olej	chromium	42821	udp4	224.0.0.251:5353	*:*	CLOSED
olej	chromium	42857	tcp4	127.0.0.1:54148	127.0.0.1:9050	ESTABLISHED
olej	chromium	42857	tcp4	127.0.0.1:40240	127.0.0.1:9050	ESTABLISHED
olej	chromium	42857	tcp4	127.0.0.1:44706	127.0.0.1:9050	ESTABLISHED
olej	chromium	42857	tcp4	127.0.0.1:60058	127.0.0.1:9050	ESTABLISHED
olej	chromium	42857	tcp4	127.0.0.1:44728	127.0.0.1:9050	ESTABLISHED
olej	chromium	42857	tcp4	127.0.0.1:60138	127.0.0.1:9050	ESTABLISHED

Изучить, даже такие обширные «простыни», в высшей степени полезно, здесь мы всё видим: FireFox (множество его вкладок) работает через браузерное дополнение Browsec через VPN в Нидерландах (198.16.66.101:436), Thunderbird, Telegram, Google Chrome, Opera и Chromium — через локальный SOCKS-прокси системы TOR (о которой будет дальше), но это ничем принципиально не отличает этот прокси от рассмотренного ранее SOCKS5.

Источники использованной информации

- [1] systemd для администраторов — http://www2.kangran.su/~nnz/pub/s4a/s4a_latest.pdf
- [2] Памятка пользователям ssh — <https://habr.com/ru/articles/122445/>
- [3] Магия SSH — <https://habr.com/ru/articles/331348/>
- [4] Настройка DHCP-сервера в Linux — <https://itproffi.ru/nastrojka-dhcp-servera-v-linux/>
- [5] Установка и базовая настройка DHCP сервера на Ubuntu — <https://www.dmosk.ru/miniinstruktions.php?mini=dhcp-ubuntu>
- [6] Как это работает: Пара слов о DNS — <https://habr.com/ru/companies/1cloud/articles/309018/>
- [7] Кэширующий DNS сервер на BIND —

<https://линуксблог.рф/keshiruyushhij-dns-server-na-bind>

[8] Настройка DNS на Ubuntu Server 18.04 LTS — <https://tokmakov.msk.ru/blog/item/522>

[9] Настройка systemd-resolved для локального кэширования DNS в Debian 10 —

<https://zevilz.dev/posts/496/>

[10] Настройка netfilter с помощью iptables —

<https://www.dmosk.ru/instruktsions.php?object=iptables-settings>

[11] Настройка iptables для чайников — <https://losst.pro/nastrojka-iptables-dlya-chajnikov>

[12] Конфигурирование сервера. Суперсерверы inetd и xinetd —

http://www.redov.ru/kompyutery_i_internet/linux_server_svoimi_rukami/p8.php#metkadoc9

[13] Интернет-демон (xinetd/inetd) — <https://wm-help.net/lib/b/book/1696396857/364>

[14] Поднимаем SOCKS5 прокси в шесть шагов — https://fckrkn.github.io/dante_proxy/

[15] Создание и настройка SOCKS5 прокси сервера в Ubuntu —

<https://la2ha.ru/dev-seo-diy/unix/socks5-proxy-server-ubuntu>

[16] Установка и настройка Dante SOCKS5 Proxy сервера на Ubuntu 18.04 LTS —

<https://blog.xenot.ru/ustanovka-i-nastrojka-dante-socks5-proxy-servera-na-ubuntu-18-04-lts-godnaya-instruksiya-po-obhodu-blokirovki-messendzhera-telegram-i-dr-sajtov-roskomnadzorom.fuck>

[17] Dante, учим любой софт работать через socks proxy — <https://habr.com/ru/articles/82727/>

[18] SOCKS через SSH — не только для ICQ — <https://habr.com/ru/articles/49801/>

[19] Руководство по использованию ifconfig в сравнении с командой ip на Linux, October 17, 2023 —

<https://telegra.ph/Rukovodstvo-po-ispolzovaniyu-ifconfig-v-sravnenii-s-komandoj-ip-na-Linux-10-17>

Следующие 2 части являются разбором того как сетевой трафик обеспечивается с позиции зрения программного кода: сначала в пользовательских приложениях, а затем и в коде ядра Linux. Только таким образом можно с исчерпывающей полнотой проследить полный путь информации от того момента когда вы, например, набираете текстовую строку на терминале, и до того момента, когда эта строка попадёт на обработку программе сервера, стоящего за много тысяч километров от вашего терминала.

Естественно, что эти части предполагают некоторое, хотя бы начальное знакомство с языком программирования C, на котором написана основная часть системного программного обеспечения операционной системы Linux. Но если вы не располагаете такими знаниями, или просто не расположены вникать в программный код — то эти 2 части можно совершенно безболезненно для всего остального изложения.

Программирование и инструменты

Первый вопрос, который естественным образом возникает в отношении последующих частей относительно программных кодов: почему для рассмотрения использован язык низкого уровня C и, иногда, и то только для краткости изложения, его клон C++? Ответ здесь прост:

- Linux, как и любой UNIX, написан на C, это базовый язык всего этого класса операционных систем;
- Только в терминах такого базового языка можно целиком проследить путь, по которому сокетные данные пользовательского пространства превращаются в структуры сокетных буферов ядра Linux, отправляются в физическую среду передачи, и на ответном конце снова, проходя форму сокетных буферов ядра, через данные сокета обращаются в конечные данные пользовательских приложений;
- Языки высокого уровня, особенно новые языки, такие как Python, Go, Rust, Kotlin, предоставляют языковые интерфейсы (библиотеки, пакеты, модули, крейты...), на высоком уровне позволяющие продуктивно реализовывать сетевые механизмы. И это само по себе отдельные обширные темы, заслуживающие и отдельных книг для их изложения... Но «под капотом» всех этих и любых механизмов высокого уровня всё равно лежат базовые POSIX механизмы языка C.
- Поэтому **реализовывать** сетевые проекты можно, и даже нужно, на инструментах высокого уровня. Но для **понимания** того что при этом происходит нужно использовать уровень POSIX нотаций.

Общие принципы

Сетевые проекты могут иметь самую разнообразную архитектуру и использовать любые протоколы пользовательского уровня. Но при этом все они подчиняются некоторому набору общих принципов, неизменных для всех сетевых проектов.

Клиент и сервер

В сетевом взаимодействии **всегда** присутствуют две **несимметричные** стороны взаимодействия: клиент и сервер.

- Сервер находится непрерывно в **пассивном** ожидании запросов от клиентов. Получив запрос, сервер активируется, осуществляет требуемую обработку запроса (иногда, зачастую, эта обработка состоит в возврате клиенту некоторого ответа, но это совершенно не обязательно).

- Клиент **активно** (по своей инициативе) запрашивает обслуживание у сервера.

Из-за этого сетевые приложения принципиально несимметричные, то и используют разные API для клиентской и серверной стороны. Поэтому сетевые проекты именуют клиент-серверными. Даже в случаях симметричного по функциям взаимодействия сторон (peer-to-peer сети) каждая из сторон, периодически меняясь ролями, выступает то как клиент, то как сервер.

Сети датаграммные и потоковые

На уровне **физической** среды передачи (уровни L1 и L2 в терминологии Linux), сетевой обмен **всегда** организуется пакетами ограниченной длины (MTU параметр — максимальная длина пакета для данного **сетевого интерфейса**).

Но на уровне транспортных механизмов сети (уровень L3) делятся на 2 большие категории по логике своего взаимодействия: датаграммные и потоковые.

- Датаграммные транспортные механизмы обеспечивают обмен дейтаграммами, логическими пакетами (имеющими начало и конец). Длина датаграммы (сообщения) может совпадать (так чаще всего и бывает), но может и не совпадать с длиной пакета, передаваемого в физическую среду — тогда происходит сегментация датаграммы на несколько пакетов сетевого, канального или физического уровня. Или наоборот, пакет физической среды с короткой датаграммой уходит в «укороченном» виде, длиной заметно ниже MTU. Самым известным из датаграммных транспортных протоколов в сети IP является UDP. При передаче последовательности 5-ти сообщений длинами, соответственно, 10, 10, 10, 10, 10 байт, на приёмном конце будет **обязательно** считано последовательными чтениями в цикле 5 сообщений длинами 10, 10, 10, 10, 10 байт.

- Потокковые транспортные механизмы логически представляют обмен на уровне сокета как непрерывный поток байт (труба, в один конец которого втекает, а из другого вытекает некоторый поток). При обмене посредством потоккового механизма не может быть никаких «пакетов»: поток отдельных байт передаётся в поток, и поток отдельных байт считывается из потока. Самым известным из потокковых транспортных протоколов в сети IP является TCP. При последовательной передаче в потокковый сокет порций (сообщений), соответственно, 10, 10, 10, 10, 10 байт, на приёмном конце в цикле чтений может быть считано, с равным успехом, любое произвольное число байт в каждом чтении, например, 5, 20, 15, 5, 5, более того, этот объём информации (50 байт) может быть считан за отличающееся число считываний, например 4 а не 5: 15, 15, 15, 5 или 6 считываний: 8, 8, 8, 8, 8, 10 (тут срабатывают различные дополнительные механизмы оптимизации, такие как отсроченная отправка, алгоритм Нэйгла и другие).

Предположение о наличии каких-либо «пакетов» на приёмном конце при TCP обмене является самой частой грубейшей ошибкой начинающих программистов.

Ещё одним заблуждением является представление, что UDP и TCP протоколы являются двумя альтернативами в IP сети **соизмеримой сложности** функционирования. И исходящее отсюда желание построить на UDP (свой) протокол надёжной доставки, например, отправкой подтверждений приёма. На самом деле протокол TCP **на порядок** сложнее в функционировании, включающий в логику своей работы много дополнительных механизмов, таких как: адаптивное согласование окон приёма и передачи, медленные старт, алгоритм Нэйгла, отсроченные подтверждения, адаптивное управление размерами окон приёма и передачи и другие. Другими словами, реализовать на UDP механизм надёжной доставки — это значит в собственном коде вручную реализовать протокол TCP ... только «очень плохой протокол TCP».

Из предыдущего заблуждения вытекает (неоднократно приходилось слышать) ещё одна красивая народная легенда о том, что, якобы, для одних и тех же действий реализация на UDP будет заметно быстрее чем на TCP. Это полная ерунда!

Каждый из транспортных механизмов предназначен для своих целей. В дальнейшем тексте везде, чтоб не создавать многословности, там где мы хотим говорить о датаграммной передаче, мы будем называть это UDP. А там, где речь идёт о потокковой передаче — TCP.

Датаграммный протокол UDP ориентирован на быструю передачу информации, ничем не гарантирующую доставку. Вплоть до того, что приёмная сторона (её сетевой стек) имеет право просто успешно принимать UDP сообщения, но, если она перегружена, то по собственной инициативе просто сбрасывать в мусор уже принятые пакеты, никак не уведомляя о том ни приёмную, ни передающую сторону.

Потокковый протокол TCP ориентирован на **надёжную доставку** информации. В случае потери пакетов или их искажения на тракте передающая сторона делает многократные попытки повторной передачи.

Протокол UDP определён в RFC 768. Протоколу TCP посвящены RFC 790, 791, 793, 1025, 1323. Это базовые RFC, в последующие годы они дополнялись и расширялись.

Примечание в порядке предупреждения: не обольщайтесь кажущейся простотой и **понятностью** протокола UDP для реализации **надёжного** взаимодействия в создаваемом

вами проекте. Надстраивание над UDP средств контроля и резервирования — это ловушка, в которую попадают многие разработчики, впервые начинающие сетевое проектирование. При этом вы только повторяете путь развития TCP, который занял не одно десятилетие.

Фазы соединения TCP

В формате каждого пакета транспортного уровня TCP предусмотрено несколько битовых флагов, определяющие предназначение пакета. Поэтому в публикациях пакеты часто называют по имени установленных в них флагов. Некоторые из этих флагов могут быть установлены одновременно (но не в любых комбинациях). Вот эти флаги:

URG - указатель срочности (urgent pointer) данных (другие названия: внеполосовые данные, приоритетные данные).

ACK - Номер подтверждения необходимо принять в рассмотрение (acknowledgment).

PSH - Получатель должен передать эти данные приложению как можно скорее.

RST - Сбросить соединение.

YN - Синхронизирующий номер последовательности для установления соединения.

FIN - Отправитель заканчивает посылку данных.

TCP это протокол, ориентированный на соединение. Перед тем как какая-либо сторона может послать данные другой, между ними должно быть **установлено** соединение. Чтобы установить TCP соединение пересылаются 3 сегмента:

1. Запрашивающая сторона (клиент) отправляет SYN (флаг) сегмент, указывая **номер порта** сервера, к которому клиент хочет подсоединиться, и **исходный номер последовательности** клиента (ISN).
2. Сервер отвечает своим сегментом SYN, содержащим **исходный номер последовательности** сервера. Сервер также **подтверждает** приход SYN от клиента с использованием ACK флаг (указывая номер ISN клиента плюс один).
3. Клиент должен подтвердить приход SYN от сервера с использованием ACK флага (ISN сервера плюс один).

Этих трех сегментов достаточно для установления соединения. Часто это называется трехразовым рукопожатием (three-way handshake).

При пересылке данных передающая сторона пересылает **сегмент данных**, а приёмная сторона должна **подтвердить** его получение сегментом ACK с инкрементированным номером последовательности. Если подтверждение не приходит в установленный тайм-аут, производятся попытки повторной передачи. Первый тайм-аут устанавливается обычно в 1,5 секунды после первой передачи. После этого величина тайм-аута удваивается для каждой передачи, причем верхний предел составляет 64 секунды. После 12 неудачных повторов (с увеличивающимся интервалом) TCP осуществляет сброс и возвращает ошибку канала.

После полного завершения обмена соединение нужно **разорвать**. Соединения обычно устанавливаются клиентом, то есть первый SYN движется от клиента к серверу. Однако любая сторона может активно закрыть соединение (послать первый FIN). Часто, однако, именно клиент определяет, когда соединение должно быть разорвано, так как процесс клиента в основном управляется пользователем.

Когда сервер получает FIN от клиента, он отправляет назад ACK с принятым номером последовательности плюс один (сегмент 5). На FIN тратится один номер последовательности, так же как на SYN. В этот момент TCP сервер также доставляет приложению признак конца файла (end-of-file) чтобы выключить сервер. Затем сервер закрывает свое соединение, что заставляет его TCP послать FIN (сегмент 6), который клиент должен подтвердить (ACK), увеличив на единицу номер принятой последовательности (сегмент 7).

Так как TCP соединение полнодуплексное (данные могут передвигаться в каждом направлении независимо от другого направления), каждое направление должно быть закрыто независимо от другого. Можно сказать, что та сторона, которая первой закрывает соединение (отправляет первый FIN), осуществляет активное закрытие, а другая сторона (которая

приняла этот FIN) осуществляет пассивное закрытие. Правило заключается в том, что каждая сторона должна послать FIN, когда передача данных завершена. Когда TCP принимает FIN, он должен уведомить приложение, что удаленная сторона разрывает соединение и прекращает передачу данных в этом направлении.

Получение FIN означает только, что в этом направлении прекращается движение потока данных. TCP, получивший FIN, может все еще посылать данные в своём направлении. Несмотря на то, что приложение все еще может посылать данные при наполовину закрытом TCP соединении, на практике только некоторые (совсем немного) TCP приложения используют это.

Для того чтобы установить соединение, необходимо 3 сегмента, а для того чтобы разорвать — нужно 4. Это объясняется тем, что TCP соединение может быть в наполовину закрытом состоянии.

Адаптивные механизмы TCP

Протокол TCP годами проходил многократную модернизацию (разными RFC) и предполагает на сегодня в своём функционировании ряд **адаптивных механизмов**. Главным образом эти механизмы направлены а). на повышение пропускной способности протокола и б). обеспечение объявленной надёжной передачи TCP меньшими затратами. Вот только некоторые (их больше) из таких адаптивных механизмов, которые можно **отключить**, или **изменить** численные характеристики их функционирования есть:

1. Задержанные подтверждения (delayed ACK). Обычно TCP не отправляет ACK сразу по приему данных. Вместо этого он осуществляет задержку подтверждений в надежде на то, что в том же направлении им будут отправлены данные (ответ), таким образом ACK может быть отправлено вместе с данными (присоединено). Большинство реализаций используют задержку равную 200 миллисекунд — таким образом, TCP задерживает ACK на время до 200 миллисекунд, чтобы посмотреть, не направляются ли данные в том же направлении, что и ACK.
2. Алгоритм Нэйгла (Nagle). При передаче коротких сегментов (скажем, длины L) от клиента к серверу генерируются пакеты размером (обычно) 40+L байт: 20 байт - IP заголовок, 20 байт - TCP заголовок и L байт данных. Короткие сегменты (тиниграммы, от tiny - крошечный) обычно не проблема для LAN, так как большинство LAN не излишне перегружены, однако они могут привести к серьёзной перегрузке WAN. Простое и элегантное решение было предложено в RFC 896, которое сейчас называется алгоритмом Нэйгла. Из алгоритма следует, что в TCP соединении может присутствовать только один исходящий маленький сегмент, который еще не был подтвержден. Следующие маленькие сегменты могут быть посланы только после того, как было получено подтверждение уже ранее отправленного. Вместо того чтобы отправляться последовательно, маленькие порции данных накапливаются и отправляются одним TCP сегментом, когда прибывает подтверждение на первый пакет. Красота этого алгоритма заключается в том, что он сам настраивает временные характеристики: чем быстрее придет подтверждение, тем быстрее будут отправлены данные. В медленных глобальных сетях, где необходимо уменьшить количество маленьких пакетов, отправляется меньше сегментов. При последовательных передачах от клиента 1, 1, 2, 1, 2, 2, 3, 1 и 3 байт, на сервере длины возвращаемые операцией чтения вполне, таким образом, могут быть: 1 и 14.
3. Контроль потока данных TCP осуществляется на каждом конце с использованием размера окна (window size). Это количество байт, начинающееся с указанного в поле номера подтверждения, которое приложение собирается принять. Это 16-битовое поле ограничивает размер окна в 65535 байт. Принимающая сторона, если она не успевает обрабатывать передаваемые данные, может запросить изменение размера окна.
4. Медленный старт. Отправитель начинает свою работу, отправив несколько сегментов в сеть. Размер сегментов может достигать размера окна, объявленного получателем. При этом всё будет в порядке, если два хоста находятся в одной и той же локальной

сети, однако если между отправителем и получателем присутствуют маршрутизаторы или медленные каналы, могут возникнуть проблемы. Некоторые промежуточные маршрутизаторы должны будут поместить пакеты в очередь, которая может переполниться. Поэтому от ТСП требуется, чтобы он поддерживал алгоритм, который называется медленным стартом. Он заключается в том, что отправитель начинает работу, отправив один небольшой сегмент и ожидая АСК на этот сегмент. Когда АСК получен, могут быть отправлены 2 сегмента. Когда каждый из этих двух сегментов подтвержден, окно переполнения увеличивается до 4. Таким образом, осуществляется экспоненциальное увеличение.

Про адаптивные механизмы ТСП нужно знать хотя бы о их существовании и их перечень, с тем, что их можно либо отключить при необходимости (алгоритм Нэйгла), либо изменить параметры их функционирования (окна приёма-передачи).

Сообщения прикладного уровня в ТСП

А что делать, если посредством ТСП протокола необходимо передавать определённые порции данных, разграниченные сообщения прикладного уровня? Тогда эти сообщения в **потоке** ТСП нужно **форматировать** на уровне **содержания** самих сообщений. На то существует несколько основных способов:

1. Концевые ограничители. Для потока информации формулируют некоторые логические разграничители сообщений в потоке (признак EOF). Для символьного потока, например, это может быть символ '\0' так же для формата ASCIIZ строк языка C (хотя это и не лучшее решение). Один из самых распространённых способов — использование в **текстовом** потоке строк одной пустой строки в качестве разграничителя, 2 идущих подряд символов перевода строки '\n' считаются завершением сообщения. Это использовано во многих протоколах прикладного уровня: HTTP в WEB (запрос GET), SIP в IP-телефонии и другие случаи. Этого способа **недостаток** состоит в том, что он неприменим к произвольным **бинарным** потокам данных, в которых значащими являются любые (все возможные) значения байта данных. Но и это решается (некоторыми дополнительными затратами) путём **экранирования** отдельных символов (байт). Большим **достоинством** такого способа является то, что он **восстанавливает синхронизацию** (границу сообщения) при временном нарушении структуры потока в результате искажений в канале, помех, потери связи, ...

2. Сообщения самоопределённой длины. Это означает, например, что первыми N (1,2,4,...) байтами сообщения передаётся его **длина** в байтах, а затем следует непосредственно само тело сообщения. Длина обычно передаётся в бинарном виде фиксированной длины, но интересным вариантом может быть передача длины и в символьном формате с разделителем. Вариантом использования этого способа есть реализация запроса POST протокола HTTP в WEB, способ достаточно широко используется в проектах промышленной автоматики. Для этого способа **достоинством** будет возможность передачи любого рода **бинарной** информации. А недостатком — невозможность (или сложность) восстановления синхронизации (границы сообщения) при нарушении структуры потока (потере границы сообщений).

3. Наконец, в некоторых целевых системах прикладного свойства может быть заложен обмен сообщениями вообще заранее **фиксированной** длины L: если передаваемое сообщение не умещается в эту длину — оно сегментируется на 2 или несколько последовательных сообщений, если сообщение короче L — оно дополняется до этой длины «пустым» (фиктивным) содержимым. Такая схема часто применяется в протоколах и проектах промышленной автоматики. Не следует впадать в заблуждение, что такая схема (а она к тому подталкивает!) упрощает приём сообщений: передача сообщений длины L в **поток** несколько не гарантирует что на приёмный конец будут приходить «пакеты» длины L. Приходить могут фрагменты любой длины (как меньше, так, иногда, и больше L), как уже отмечалось выше, и принимать их нужно в **кольцевой буфер** достаточного объёма для последующего разбора.

Иногда, в проектах требующих экстремальной надёжности, используют и комбинацию этих двух методов. Это может быть связано, например, с требованием восстановления

синхронизации при нарушении структуры потока. Например: длина сообщения, за которой следует тело сообщения, которое, возможно, завершается повтором значения длины (дублирование для страховки) и всё это ограничивается разделителем сообщения.

Весьма часто в системах не очень высокой нагрузки (например интерактивных) **клиент** открывает соединение TCP только на передачу **одного единственного** сообщения прикладного уровня, после чего сразу же закрывает сокет. При необходимости передачи следующих сообщений, **клиент** откроет новые соединения, опять же для передачи каждого единичного сообщения. Так работает, например, прикладной протокол HTTP. При этом естественным образом решается проблема разграничения сообщений прикладного уровня.

Присоединённый UDP

Протокол UDP, как уже подчёркивалось, никак не гарантирует надёжность доставки датаграмм. Получатель (например сервер) датаграмм, его сетевой стек, если он перегружен имеет право вообще сбросить (уничтожить) приходящую датаграмму, никак на неё не реагируя, и не извещая даже об ошибке посылкой ICMP пакета.

Клиент протокола UDP, который мы увидим далее, может отправлять датаграммы серверу даже на не существующий IP или порт (когда на хосте не выполняется подходящий сервер UDP). В последнем случае хост получателя ответит сообщением ICMP об ошибке «недоступный порт». Но отправитель (клиент) не получит это уведомление как код завершения отправки, это уведомление ICMP является **асинхронным**. Операция записи в сокет `sendto()` сообщает только о возможных **синхронных** ошибках в момент отправки сообщений.

Иногда используются (например в сети DNS) такой малоизвестный вариант, как **присоединённый UDP**, когда для сокета UDP (в коде клиента) вызывается функция `connect()`. Но при этом не происходит ничего похожего на соединение TCP: ядро просто записывает адрес и порт удалённого корреспондента в сокет. После этого клиент уже не может использовать вызовы `sendto()` и `recvfrom()`, а пользуется вызовами `send()`, `write()`, `recv()`, `read()`. Естественно, такой клиент не может рассылать широковещательные сообщения.

Такой клиент не сможет отправлять сообщения на ошибочный адрес IP (сокет связан с адресом). Но самое главное, что такой сокет будет возвращать асинхронные сообщения ICMP об ошибках от удалённого хоста.

Сетевые сокеты и операции

В конкурсе на лучшую компьютерную идею всех времен и народов сокеты, без сомнения, могли бы рассчитывать на призовое место.
Андрей Боровский, «Программирование для Linux»

Вся мощь и многообразие сетевых возможностей обеспечивается концепцией сетевого сокета, введенного операционной системой BSD и некоторым количеством вызовов API вокруг этого понятия (показываемые прототипы функций полностью **скопированы** из заголовочных файлов определений, вплоть до оригинальных имён в записи параметров).

Создание сокета производится вызовом:

```
#include <sys/socket.h>
/* Create a new socket of type TYPE in domain DOMAIN, using
   protocol PROTOCOL. If PROTOCOL is zero, one is chosen automatically.
   Returns a file descriptor for the new socket, or -1 for errors. */
extern int socket(int __domain, int __type, int __protocol) __THROW;
```

Здесь:

- `__domain` — семейство протоколов, которое может быть:

```
#include <bits/socket.h>
/* Address families. */
#define AF_UNSPEC      PF_UNSPEC
#define AF_LOCAL      PF_LOCAL
```

```

#define AF_UNIX      PF_UNIX
#define AF_FILE      PF_FILE
#define AF_INET      PF_INET
#define AF_AX25      PF_AX25
#define AF_IPX       PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM    PF_NETROM
#define AF_BRIDGE    PF_BRIDGE
#define AF_ATMPVC    PF_ATMPVC
#define AF_X25       PF_X25
#define AF_INET6     PF_INET6
#define AF_ROUTE     PF_ROUTE
#define AF_DECnet    PF_DECnet
#define AF_NETBEUI   PF_NETBEUI
#define AF_SECURITY  PF_SECURITY
#define AF_KEY       PF_KEY
#define AF_NETLINK   PF_NETLINK
#define AF_ROUTE     PF_ROUTE
#define AF_PACKET    PF_PACKET
#define AF_ASH       PF_ASH
#define AF_ECONET    PF_ECONET
#define AF_ATMSVC    PF_ATMSVC
#define AF_RDS       PF_RDS
#define AF_SNA       PF_SNA
#define AF_IRDA      PF_IRDA
#define AF_PPPOX     PF_PPPOX
#define AF_WANPIPE   PF_WANPIPE
#define AF_LLC       PF_LLC
#define AF_CAN       PF_CAN
#define AF_TIPC      PF_TIPC
#define AF_BLUETOOTH PF_BLUETOOTH
#define AF_IUCV      PF_IUCV
#define AF_RXRPC     PF_RXRPC
#define AF_ISDN      PF_ISDN
#define AF_PHONET    PF_PHONET
#define AF_IEEE802154 PF_IEEE802154
#define AF_CAIF      PF_CAIF
#define AF_ALG       PF_ALG
#define AF_NFC       PF_NFC
#define AF_VSOCK     PF_VSOCK
#define AF_MAX       PF_MAX

```

- `__type` — тип протокола (не все типы допускаются во всех семействах):

```

#include <bits/socket_type.h>
/* Types of sockets. */
enum __socket_type
{
    SOCK_STREAM = 1,          /* Sequenced, reliable, connection-based
                             byte streams. */
    SOCK_DGRAM = 2,          /* Connectionless, unreliable datagrams
                             of fixed maximum length. */
    SOCK_RAW = 3,            /* Raw protocol interface. */
    SOCK_RDM = 4,            /* Reliably-delivered messages. */
    SOCK_SEQPACKET = 5,      /* Sequenced, reliable, connection-based,
                             datagrams of fixed maximum length. */
    SOCK_DCCP = 6,           /* Datagram Congestion Control Protocol. */
    SOCK_PACKET = 10,        /* Linux specific way of getting packets
                             at the dev level. For writing rarp and
                             other similar things on the user level. */

```

```

    SOCK_CLOEXEC = 02000000,      /* Atomically set close-on-exec flag for the
                                   new descriptor(s). */
    SOCK_NONBLOCK = 00004000     /* Atomically mark descriptor(s) as
                                   non-blocking. */
};
- __protocol — имя протокола, обычно оно указывается только для символьных сокетов
SOCK_RAW, для остальных здесь можно указать 0 и имя протокола будет выбрано
автоматически:

```

```

#include <netinet/in.h>
/* Standard well-defined IP protocols. */
enum
{
    IPPROTO_IP = 0, /* Dummy protocol for TCP. */
    IPPROTO_HOPOPTS = 0, /* IPv6 Hop-by-Hop options. */
    IPPROTO_ICMP = 1, /* Internet Control Message Protocol. */
    IPPROTO_IGMP = 2, /* Internet Group Management Protocol. */
    IPPROTO_IPIP = 4, /* IPIP tunnels (older KA9Q tunnels use 94). */
    IPPROTO_TCP = 6, /* Transmission Control Protocol. */
    IPPROTO_EGP = 8, /* Exterior Gateway Protocol. */
    IPPROTO_PUP = 12, /* PUP protocol. */
    IPPROTO_UDP = 17, /* User Datagram Protocol. */
    IPPROTO_IDP = 22, /* XNS IDP protocol. */
    IPPROTO_TP = 29, /* SO Transport Protocol Class 4. */
    IPPROTO_DCCP = 33, /* Datagram Congestion Control Protocol. */
    IPPROTO_IPV6 = 41, /* IPv6 header. */
    IPPROTO_ROUTING = 43, /* IPv6 routing header. */
    IPPROTO_FRAGMENT = 44, /* IPv6 fragmentation header. */
    IPPROTO_RSVP = 46, /* Reservation Protocol. */
    IPPROTO_GRE = 47, /* General Routing Encapsulation. */
    IPPROTO_ESP = 50, /* encapsulating security payload. */
    IPPROTO_AH = 51, /* authentication header. */
    IPPROTO_ICMPV6 = 58, /* ICMPv6. */
    IPPROTO_NONE = 59, /* IPv6 no next header. */
    IPPROTO_DSTOPTS = 60, /* IPv6 destination options. */
    IPPROTO_MTP = 92, /* Multicast Transport Protocol. */
    IPPROTO_ENCAP = 98, /* Encapsulation Header. */
    IPPROTO_PIM = 103, /* Protocol Independent Multicast. */
    IPPROTO_COMP = 108, /* Compression Header Protocol. */
    IPPROTO_SCTP = 132, /* Stream Control Transmission Protocol. */
    IPPROTO_UDPLITE = 136, /* UDP-Lite protocol. */
    IPPROTO_RAW = 255, /* Raw IP packets. */
    IPPROTO_MAX
};

```

Отметим, что все перечисленные здесь константы вида `AF_*`, `SOCK_*`, `IPPROTO_*` — это не просто символьные препроцессорные константы, а численные значения, которые будут непосредственно **вписываться** в различные поля сетевых пакетов, передаваемых по сети.

Для созданного сокета определяется (<sys/socket.h>) целый ряд API операций над сокетом (показаны далеко не все, с полным перечнем крайне полезно познакомиться). Объект сокет в высшей степени подобен файловому дескриптору (а иногда и просто совпадает с ним), поэтому в вызовах (в прототипах в заголовочных файлах .h) он и указывается как файловый дескриптор.

```

/* Give the socket FD the local address ADDR (which is LEN bytes long). */
extern int bind(int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len) __THROW;

```

Функция `bind()` задаёт сокету локальный адрес (связывает) для выбранного семейства протокола. То, что конкретно представляет из себя этот адрес, зависит от самого протокола:

```
# define __CONST_SOCKADDR_ARG    const struct sockaddr*
```

Структура `sockaddr` может конкретизироваться совершенно по-разному для разных семейств протоколов, при вызове указатель преобразовывается к его типу, а 3-й параметр указывает длину адресной структуры для этого семейства протоколов. Например, для нескольких семейств:

```
#include <netinet/in.h>
struct sockaddr_in {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;           /* Port number. */
    struct in_addr sin_addr;      /* Internet address. */
    /* Pad to size of `struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) -
        __SOCKADDR_COMMON_SIZE -
        sizeof (in_port_t) -
        sizeof (struct in_addr)];
};
...
struct sockaddr_in6 {
    __SOCKADDR_COMMON (sin6_);
    in_port_t sin6_port;         /* Transport layer port # */
    uint32_t sin6_flowinfo;      /* IPv6 flow information */
    struct in6_addr sin6_addr;    /* IPv6 address */
    uint32_t sin6_scope_id;      /* IPv6 scope-id */
};

#include <linux/netlink.h>
struct sockaddr_nl {
    __kernel_sa_family_t nl_family; /* AF_NETLINK */
    unsigned short nl_pad;          /* zero */
    __u32 nl_pid;                  /* port ID */
    __u32 nl_groups;               /* multicast groups mask */
};

#include <linux/un.h>
struct sockaddr_un {
    __kernel_sa_family_t sun_family; /* AF_UNIX */
    char sun_path[UNIX_PATH_MAX];    /* pathname */
};
```

В случае **клиента**, в структуре адреса протоколов IP указывается тот **конкретный** хост (адрес и порт), к которому мы хотим получить связь. В случае **сервера**, для IPv4 сервер устанавливает в `bind()` **универсальный адрес**, указываемый константой `INADDR_ANY`, что означает: принимать запросы от любого IP. После успешного выполнения `bind()` сокет инициализирован готов к использованию.

Обычный вид кода, который производит создание и инициализацию сокета, имеет достаточно неизменный вид, подобный следующему (показан сервер UDP, но все прочие сервера и клиенты будут делать то же самое):

```
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    error("server: can't open datagram socket");
// Bind our local address so that the client can send to us.
struct sockaddr_in serv_addr;
bzero((char*)&serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port        = htons(SERV_UDP_PORT);
if (bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    error("server: can't bind local address");
```

```

...

/* Prepare to accept connections on socket FD.
   N connection requests will be queued before further requests are refused.
   Returns 0 on success, -1 for errors. */
extern int listen (int __fd, int __n) __THROW;

```

Функция `listen()` вызывается только **сервером** TCP, и выполняет 2 действия:

- преобразует неприсоединённый сокет (клиентский) в **пассивный**, запросы к которому начинают приниматься ядром;
- вторым аргументом этой функции задаётся максимальное число соединений, которые ядро может помещать в очередь этого сокета;

Функция `listen()` вызывается после `socket()` и `bind()`, перед `accept()`.

```

/* Await a connection on socket FD.
   When a connection arrives, open a new socket to communicate with it,
   set *ADDR (which is *ADDR_LEN bytes long) to the address of the connecting
   peer and *ADDR_LEN to the address's actual length, and return the
   new socket's descriptor, or -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW. */
extern int accept (int __fd, __SOCKADDR_ARG __addr,
                  socklen_t *__restrict __addr_len);

```

Функция `accept()` вызывается **сервером** TCP после для возвращения следующего полностью **установленного** соединения из очереди устанавливаемых соединений (пассивный сокет превращается в **присоединённый**).

```

/* Open a connection on socket FD to peer at ADDR (which LEN bytes long).
   For connectionless socket types, just set the default address to send to
   and the only address from which to accept transmissions.
   Return 0 on success, -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW. */
extern int connect (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len);

```

Функция `connect()` используется **клиентом** TCP для установления **соединения** с TCP сервером.

После завершения работы с соединённым сокетом соединение нужно разорвать, для чего сокет закрыть. Для разрыва TCP соединения, если не быть слишком придирчивым, вполне годится стандартная функция `close()`:

```

#include <unistd.h>
/* Close the file descriptor FD.
   This function is a cancellation point and therefore not marked with
   __THROW. */
extern int close (int __fd);

```

Но API сокетов предоставляет нам и другую альтернативу, которая позволяет более тонко управлять разрывом соединения:

```

/* Shut down all or part of the connection open on socket FD.
   HOW determines what to shut down:
   SHUT_RD   = No more receptions;
   SHUT_WR   = No more transmissions;

```

```

        SHUT_RDWR = No more receptions or transmissions.
        Returns 0 on success, -1 for errors.  */
extern int shutdown(int __fd, int __how) __THROW;

```

Здесь мы можем отдельно закрывать только считывающую или только записывающую половину соединения.

Обменные операции

После выполнения всяких подготовительных мероприятий с сокетом, сокет готов к записи в него данных, и считывания из него данных (сокет, обычно, полнодуплексный канал, позволяющий двунаправленные операции). Для чтения-записи данных сокета вполне достаточно, особенно для TCP (потокowego) сокета, использовать традиционные POSIX обменные функции файловых дескрипторов:

```

#include <unistd.h>
/* Read NBYTES into BUF from FD.  Return the
   number read, -1 for errors or 0 for EOF.
   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t read(int __fd, void *__buf, size_t __nbytes) __wur;
/* Write N bytes of BUF to FD.  Return the number written, or -1.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t write(int __fd, const void *__buf, size_t __n) __wur;

```

Более того, функция будет так же возвращать нулевое значение как признак EOF по закрытию сокета противоположным концом соединения.

Точно так же (как на файловых дескрипторах) на сокетах можно организовать мультиплексное ожидание ввода используя `select()` или `poll()`.

И точно так же, можно (и нужно!) организовывать **тайм-ауты** ожидания на сокетах в операциях `connect()`, `read()`, `select()` и `poll()`, `recvfrom()`, `recv()`, ... : POSIX вызовом `alarm()` или использованием асинхронных таймеров.

Если же этих традиционных способов вам окажется не достаточно, то API сокетов предоставляет ещё много специфических вариантов обменных функций:

```

#include <sys/socket.h>
/* Send N bytes of BUF to socket FD.  Returns the number sent or -1.
   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t send(int __fd, const void *__buf, size_t __n, int __flags);
/* Read N bytes into BUF from socket FD.
   Returns the number read or -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t recv(int __fd, void *__buf, size_t __n, int __flags);

```

Эта пара обменных операций (все они симметричные) предполагает дополнительный, 4-й параметр: `__flags` — нулевое значение, или формируемая в результате логического OR битовая маска флагов, уточняющая характер операции:

```

#include <bits/socket.h>
/* Bits in the FLAGS argument to `send', `recv', et al.  */
enum {
    MSG_OOB          = 0x01,          /* Process out-of-band data.  */
    MSG_PEEK         = 0x02,          /* Peek at incoming messages.  */
    MSG_DONTROUTE    = 0x04,          /* Don't use local routing.  */
#ifdef __USE_GNU
    /* DECnet uses a different name.  */
    MSG_TRYHARD      = MSG_DONTROUTE,

```

```

# define MSG_TRYHARD    MSG_DONTROUTE
#endif

MSG_CTRUNC      = 0x08,      /* Control data lost before delivery.  */
MSG_PROXY       = 0x10,      /* Supply or ask second address.  */
MSG_TRUNC       = 0x20,
MSG_DONTWAIT    = 0x40,      /* Nonblocking IO.  */
MSG_EOR         = 0x80,      /* End of record.  */
MSG_WAITALL     = 0x100,     /* Wait for a full request.  */
MSG_FIN         = 0x200,
MSG_SYN         = 0x400,
MSG_CONFIRM     = 0x800,     /* Confirm path validity.  */
MSG_RST         = 0x1000,
MSG_ERRQUEUE    = 0x2000,    /* Fetch message from error queue.  */
MSG_NOSIGNAL    = 0x4000,    /* Do not generate SIGPIPE.  */
MSG_MORE        = 0x8000,    /* Sender will send more.  */
MSG_WAITFORONE  = 0x10000,   /* Wait for at least one packet to return.*/
MSG_FASTOPEN    = 0x20000000, /* Send data in TCP SYN.  */
MSG_CMSG_CLOEXEC = 0x40000000 /* Set close_on_exit for file
                                descriptor received through
                                SCM_RIGHTS.  */

};

```

Из них особое внимание стоит обратить на MSG_OOB: отправка или получение **приоритетных** (внеполосовых) данных в TCP соединении.

```

#include <sys/socket.h>
/* Send N bytes of BUF on socket FD to peer at address ADDR (which is
   ADDR_LEN bytes long). Returns the number sent, or -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t sendto(int __fd, const void *__buf, size_t __n,
                      int __flags, __CONST_SOCKADDR_ARG __addr,
                      socklen_t __addr_len);

/* Read N bytes into BUF through socket FD.
   If ADDR is not NULL, fill in *ADDR_LEN bytes of it with the address of
   the sender, and store the actual size of the address in *ADDR_LEN.
   Returns the number of bytes read or -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t recvfrom(int __fd, void *__restrict __buf, size_t __n,
                       int __flags, __SOCKADDR_ARG __addr,
                       socklen_t *__restrict __addr_len);

```

Эта пара операций ввода-вывода **аналогичны** стандартным операциям read() и write(), но требуют 3-х дополнительных параметров:

- __flags — битовые флаги, которые объяснены выше;
- __addr — struct sockaddr, адрес получателя или отправителя, с кем происходит обмен;
- __addr_len — размер этой адресной структуры;

Последние 2 аргумента recvfrom() аналогичны последним аргументам accept(): кто отправил датаграмму (при UDP), или кто инициировал соединение (при TCP). Последние 2 аргумента sendto() аналогичны последним аргументам connect(): структура адреса заполняется адресом протокола того места куда отправляется датаграмма (при UDP), или с которым будет устанавливаться соединение (при TCP).

Такие прототипы операций recvfrom() и sendto() делают их **удобными** для использования в UDP, но все обменные операции в равной мере применимы **ко всем видам сокетов**.

Ещё один вид операций ввода-вывода — векторные операции:

```

#include <sys/uio.h>

```



```

/* Read data from file descriptor FD, and put the result in the
   buffers described by IOVEC, which is a vector of COUNT 'struct iovec's.
   The buffers are filled in the order specified.
   Operates just like 'read' (see <unistd.h>) except that data are
   put in IOVEC instead of a contiguous buffer.
   This function is a cancellation point and therefore not marked with
   __THROW. */
extern ssize_t readv(int __fd, const struct iovec *__iovec, int __count) __wur;

/* Write data pointed by the buffers described by IOVEC, which
   is a vector of COUNT 'struct iovec's, to file descriptor FD.
   The data is written in the order specified.
   Operates just like 'write' (see <unistd.h>) except that the data
   are taken from IOVEC instead of a contiguous buffer.
   This function is a cancellation point and therefore not marked with
   __THROW. */
extern ssize_t writev(int __fd, const struct iovec *__iovec, int __count) __wur;

```

Они позволяют использовать для чтения или записи один или **более** (вектор) буферов с помощью одного вызова функции. Такие операции называются операциями распределяющего чтения (scatter read) и объединяющей записи (gather write).

Второй параметр этих функций — указатель на массив структур `iovec`:

```

#include <bits/uio.h>
/* Structure for scatter/gather I/O. */
struct iovec {
    void *iov_base;    /* Pointer to data. */
    size_t iov_len;    /* Length of data. */
};

```

А последний параметр `readv()` и `writev()` — это размерность этого массива. Максимально допустимый размер вектора зависит от типа операционной системы (POSIX 1.g) и её версии, и определяется там же:

```

/* Size of object which can be written atomically.
   This macro has different values in different kernel versions. The
   latest versions of the kernel use 1024 and this is good choice. Since
   the C library implementation of readv/writev is able to emulate the
   functionality even if the currently running kernel does not support
   this large value the readv/writev call will not fail because of this. */
#define UIO_MAXIOV    1024

```

Ну и наконец, самая общая (но как всегда и сложная) форма обменных операций — на случай, если предыдущие не обеспечивают желаемое поведение в тонких деталях:

```

#include <sys/socket.h>
/* Send a VLEN messages as described by VMESSAGES to socket FD.
   Returns the number of datagrams successfully written or -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW. */
extern int sendmmsg (int __fd, struct mmsghdr *__vmessages,
                    unsigned int __vlen, int __flags);

/* Receive a message as described by MESSAGE from socket FD.
   Returns the number of bytes read or -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW. */
extern ssize_t recvmsg (int __fd, struct msghdr *__message, int __flags);

```

Большинство аргументов этих функций скрыто в структуре:

```

#include <bits/socket.h>

```

```

/* Structure describing messages sent by `sendmsg' and received by `recvmsg'. */
struct msghdr {
    void *msg_name;           /* Address to send to/receive from. */
    socklen_t msg_namelen;    /* Length of address data. */
    struct iovec *msg_iov;    /* Vector of data to send/receive into. */
    size_t msg_iovlen;        /* Number of elements in the vector. */
    void *msg_control;        /* Ancillary data (eg BSD filedesc passing). */
    size_t msg_controllen;    /* Ancillary data buffer length.
                               !! The type should be socklen_t but the
                               definition of the kernel is incompatible
                               with this. */
    int msg_flags;            /* Flags on received message. */
};

```

Параметры сокета

API сокетов включает ряд функций, дающих уточнённую информацию о сокете, или управляющих особенностями поведения сокета. Важнейшей из этих возможностей есть:

```

/* Put the current value for socket FD's option OPTNAME at protocol level LEVEL
   into OPTVAL (which is *OPTLEN bytes long), and set *OPTLEN to the value's
   actual length. Returns 0 on success, -1 for errors. */
extern int getsockopt(int __fd, int __level, int __optname,
                     void *__restrict __optval,
                     socklen_t *__restrict __optlen) __THROW;

/* Set socket FD's option OPTNAME at protocol level LEVEL
   to *OPTVAL (which is OPTLEN bytes long).
   Returns 0 on success, -1 for errors. */
extern int setsockopt(int __fd, int __level, int __optname,
                     const void *__optval, socklen_t __optlen) __THROW;

```

Функциями `getsockopt()` и `setsockopt()` считываются и устанавливаются значения многочисленных свойств сокета. Параметр `__fd` должен быть открытым дескриптором сокета.

Параметр `__level` указывает каким протокольным уровнем должен интерпретироваться параметр (`SOL_SOCKET`, `IPPROTO_IP`, `IPPROTO_IPV6`, `IPPROTO_TCP`, ... см. выше):

```

#define SOL_SOCKET      1
...

```

Параметр `__optname` указывает имя параметра к которому относится вызов:

```

#include <asm-generic/socket.h>
#define SO_DEBUG        1
#define SO_REUSEADDR    2
#define SO_TYPE         3
#define SO_ERROR        4
#define SO_DONTROUTE    5
#define SO_BROADCAST    6
#define SO_SNDBUF       7
#define SO_RCVBUF       8
#define SO_SNDBUFFORCE  32
#define SO_RCVBUFFORCE  33
#define SO_KEEPALIVE    9
...

```

Параметр `__optval` указывает адрес переменной, куда помещается считанное значение (`getsockopt()`) или которая содержит новое значение (`setsockopt()`). Тип этой переменной может быть различным, в зависимости от `__optname`, но для многих параметров это `int`.

Примеры изменения использования `setsockopt()` для изменений свойств сокета или изменения адаптационных механизмов TCP:

1. Разрешить быстрый перезапуск сервера (или восстановление упавшего), не ожидая истечения состояния TIME-WAIT (порядка 2-х минут):

```
const int on = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
```

Такой вызов должен предшествовать вызову `bind()` (привязке сокета).

2. Отключение алгоритма Нэйгла (объединения коротких последовательных сегментов в один более длинный), что допустимо для LAN:

```
const int on = 1;
setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, &on, sizeof(on));
```

Вспомогательные Функции того же (информационного) предназначения:

```
#include <sys/socket.h>
/* Put the local address of FD into *ADDR and its length in *LEN. */
extern int getsockname(int __fd, __SOCKADDR_ARG __addr,
                      socklen_t *__restrict __len) __THROW;
/* Put the address of the peer connected to socket FD into *ADDR
   (which is *LEN bytes long), and its actual length into *LEN. */
extern int getpeername(int __fd, __SOCKADDR_ARG __addr,
                      socklen_t *__restrict __len) __THROW;
```

Первая из них (`getsockname()`) возвращает локальный, а вторая (`getpeername()`) — удалённый адресную структуру протокола, связанный на данный момент с сокетом.

Использование сокетного API

Базовые схемы построения обмена в клиентских и серверных кодах, и для UDP и для TCP остаются практически постоянными от проекта к проекту, хотя в деталях могут существенно различаться⁹. Рассмотрим эти базовые схемы (все примеры кодов находятся в каталоге архива под именем `echo-cli-serv`)...

Общие определения:

common.h :

```
#ifndef __common_h
#define __common_h

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>
#include <sys/socket.h> /* basic socket definitions */
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAXLINE 4096 /* max text line length */
static char sendline[MAXLINE], /* write buffer */
           recvline[MAXLINE + 1]; /* read buffer */

void err_dump(const char *, ...);
```

⁹ За основу примеров этой главы взяты примеры из книги У.Р.Стивенса [3].

```

void err_sys(const char *, ...);

void str_cli(register FILE*, register int sockfd);
void str_echo(int sockfd);

void dg_cli(FILE* fp, int sockfd,
            struct sockaddr* pserv_addr,
            int servlen);
void dg_echo(int sockfd,
            struct sockaddr* pcli_addr,
            int maxclilen);

#endif

```

Здесь вы можете определить IP (и порты) адрес используемого сервера, локального или удалённого в глобальной сети ... но для тестирования работоспособности всех клиент-серверных приложений вполне достаточно, в большинстве случаев, и петлевого интерфейса:

inet.h :

```

/* Definitions for TCP and UDP client/server programs. */
#include "common.h"

#define SERV_UDP_PORT 60000
#define SERV_TCP_PORT 60000
// #define SERV_HOST_ADDR "192.168.1.13" /* host addr for server */
#define SERV_HOST_ADDR "127.0.0.1" /* host addr for server */

ssize_t readline(int fd, void *vptr, size_t maxlen);
ssize_t writen(int fd, const void *vptr, size_t n);

```

Все показанные дальше 4 вариант (включая UNIX сокеты) сделаны предельно просто, так чтобы они могли быть использованы как базовые шаблоны для расширения в конкретные приложения. Везде в них:

- Клиент ожидает ввода строки с терминала;
- Получив строку он пересылает её серверу...
- Сервер получив строку молча ретранслирует её назад клиенту...
- Клиент выводит полученное на терминал для сравнения на адекватность;
- После чего весь цикл повторяется до бесконечности...

P.S. Обратите внимание, что для портов TCP и UDP здесь выбрано одно и то же численное значение, для того чтобы лишний раз подтвердить что это совершенно разные вещи, и они не мешают взаимно работе друг друга.

Каждое приложение ниже (клиент и сервер каждого типа) разделены на части: часть формирующая сокет (вызывающая) и часть выполняющая цикл операций ввода-вывода. Так листинги короче и обозримее.

UDP клиент-сервер

Клиент посылающий UDP запросы и диагностирующий ретранслируемые ответы:

dgcli.c :

```

void dg_cli(FILE* fp, int sockfd,
            struct sockaddr* pserv_addr, // ptr to appropriate sockaddr_XX structure
            int servlen) {               // actual sizeof(*pserv_addr)

    int n;
    while (fgets(sendline, MAXLINE, fp) != NULL) {
        n = strlen(sendline);
        if (sendto(sockfd, sendline, n, 0, pserv_addr, servlen) != n)

```

```

        err_dump("dg_cli: sendto error on socket");
    // Now read a message from the socket and
    // write it to our standard output.
    n = recvfrom(sockfd, recvline, MAXLINE, 0,
        (struct sockaddr*)0, (socklen_t*)0);
    if (n < 0)
        err_dump("dg_cli: recvfrom error");
    recvline[n] = 0;    /* null terminate */
    fputs(recvline, stdout);
}
if (ferror(fp))
    err_dump("dg_cli: error reading file");
}

```

udpcli.c :

```

#include "inet.h"

int main(int argc, char* argv[]) {
    int sockfd;
    /* Open a UDP socket (an Internet datagram socket). */
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        err_dump("client: can't open datagram socket");
    struct sockaddr_in serv_addr;
    /* Fill in the structure "serv_addr" with the address
    * of the server that we want to send to. */
    bzero((char*)&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port        = htons(SERV_UDP_PORT);
    dg_cli(stdin, sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    close(sockfd);
    exit(0);
}

```

Эхо-сервер UDP, ретранслирующий запросы клиента:

dgecho.c :

```

#include "inet.h"

void dg_echo(int sockfd,
    struct sockaddr* pcli_addr, /* ptr to appropriate sockaddr_XX structure */
    int maxclilen) {           /* sizeof(*pcli_addr) */
    for (;;) {
        socklen_t clilen = maxclilen;
        int n = recvfrom(sockfd, recvline, MAXLINE, 0, pcli_addr, &clilen);
        if (n < 0) err_dump("dg_echo: recvfrom error");
        if (sendto(sockfd, recvline, n, 0, pcli_addr, clilen) != n)
            err_dump("dg_echo: sendto error");
    }
}

```

udpserver.c :

```

#include "inet.h"

int main(int argc, char* argv[]) {
    int sockfd;
    /* Open a UDP socket (an Internet datagram socket). */

```

```

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        err_dump("client: can't open datagram socket");
    struct sockaddr_in serv_addr;
    /* Fill in the structure "serv_addr" with the address
     * of the server that we want to send to. */
    bzero((char*)&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port        = htons(SERV_UDP_PORT);
    dg_cli(stdin, sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    close(sockfd);
    exit(0);
}

```

Эта пара в работе:

```

$ ./udpserv
...
$ ./udpcli
UDP клиент посылает
UDP клиент посылает
...

```

Важно! ... И очень поучительно для понимания: измените в файле `inet.h` значение IP адреса сервере `SERV_HOST_ADDR` на адрес отсутствующий (недоступный) в вашей локальной сети, или вообще бессмысленный IPv4, лишь бы он был синтаксически верно записан... При этом UDP-клиент будет бодро отсылать датаграммы «в небо» и бесконечно ожидать оттуда ретрансляции, а приложение никак не будет фиксировать ошибки в происходящем. В противоположность, показанный далее TCP-клиент сразу же заявит о недостижимости сервера. Об этом уже упоминалось ранее, но то было на словах, а здесь — возможность простой живой иллюстрации.

TCP клиент-сервер

Поскольку TCP — потоковый протокол, то полезно предварительно написать функции записи в поток и чтения из потока (именно потому, что в TCP нет «пакетов» и размер чтения-записи за одну операцию может быть произвольным):

writen.c :

```

#include "common.h"

/* Write "n" bytes to a descriptor */
ssize_t writen(int fd, const void *vptr, size_t n) {
    size_t nleft;
    ssize_t nwritten;
    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, vptr, nleft)) <= 0) {
            if (errno == EINTR) nwritten = 0;      /* and call write() again */
            else return(-1);                      /* error */
        }
        nleft -= nwritten;
        vptr += nwritten;
    }
    return(n);
}

```

readln.c :

```

#include "common.h"

```

```

static ssize_t my_read(int fd, char *ptr) {
    static int read_cnt = 0;
    static char *read_ptr;
    static char read_buf[MAXLINE];
    if (read_cnt <= 0) {
again:
        if ((read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR) goto again;
            return(-1);
        } else if (read_cnt == 0)
            return(0);
        read_ptr = read_buf;
    }
    read_cnt--;
    *ptr = *read_ptr++;
    return(1);
}

ssize_t readline(int fd, void *vptr, size_t maxlen) {
    int n, rc;
    char c, *ptr;
    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
        if ((rc = my_read(fd, &c)) == 1) {
            *ptr++ = c;
            if (c == '\n') break; /* newline is stored, like fgets() */
        } else if (rc == 0) {
            if (n == 1) return(0); /* EOF, no data read */
            else break; /* EOF, some data was read */
        } else return(-1); /* error, errno set by read() */
    }
    *ptr = 0; /* null terminate like fgets() */
    return(n);
}

```

Клиент посылающий TCP запросы и диагностирующий получаемые ответы:

strcli.c :

```

#include "common.h"

void str_cli(register FILE* fp, register int sockfd) {
    int n;
    while (fgets(sendline, MAXLINE, fp) != NULL) {
        n = strlen(sendline);
        if (written(sockfd, sendline, n) != n)
            err_sys("str_cli: written error on socket");
        /* Now read a line from the socket and
         * write it to our standard output. */
        n = readline(sockfd, recvline, MAXLINE);
        if (n < 0) err_dump("str_cli: readline error");
        recvline[n] = 0; /* null terminate */
        fputs(recvline, stdout);
    }
    if (ferror(fp))
        err_sys("str_cli: error reading file");
}

```

tcpcli.c :

```
#include "inet.h"

int main(int argc, char* argv[]) {
    int sockfd; // TCP сокет
    struct sockaddr_in serv_addr; // заполнить структуру адреса сервера
    bzero((char*)&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);
    /* Open a TCP socket (an Internet stream socket). */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("client: can't open stream socket");
    /* Connect to the server. */
    if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
        err_sys("client: can't connect to server");
    str_cli(stdin, sockfd); // цикл обменов с сервером
    close(sockfd);
    exit(0);
}
```

Эхо-сервер TCP, ретранслирующий запросы клиента (присутствующий в архиве сервер реализован как параллельный, обслуживающий много запросов, здесь же он показан в схематичном последовательном виде):

strecho.c :

```
#include "common.h"

void str_echo(int sockfd) {
    int n;
    for (;;) {
        n = readline(sockfd, recvline, MAXLINE);
        if (n == 0) return; /* connection terminated */
        else if (n < 0) err_dump("str_echo: readline error");
        if (writen(sockfd, recvline, n) != n)
            err_dump("str_echo: writen error");
    }
}
```

tcpserv.c :

```
#include "inet.h"

int main(int argc, char* argv[]) {
    int sockfd; // TCP сокет
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_dump("server: can't open stream socket");
    struct sockaddr_in serv_addr; // инициализировать униадресом
    bzero((char*)&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(SERV_TCP_PORT);
    if (bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
        err_dump("server: can't bind local address");
    listen(sockfd, 5);
    for (;;) { // цикл по подключения
        socklen_t clilen = sizeof(serv_addr);
        int childpid, newsockfd;
```



```

newsockfd = accept(sockfd, (struct sockaddr*)&serv_addr, &clilen);
if (newsockfd < 0)          // ожидать соединения
    err_dump("server: accept error");
if ((childpid = fork()) < 0)
    err_dump("server: fork error");
else if (childpid == 0) { // обрабатывать в дочернем процессе
    close(sockfd);         // закрыть копию прослушивающего сокета
    str_echo(newsockfd);   // ретранслировать полученные данные
    exit(0);               // завершить дочерний процесс
}
close(newsockfd);         // в родительском процессе
}
}

```

Смотрим эту пару в работе:

```

$ ./tcpserv
...
$ ./tcpcli
TCP клиент посылает
TCP клиент посылает
...

```

Взаимодействие запрос-ответ

Чаще всего (но это не всегда так) клиент-серверное взаимодействие в TCP реализует схему запрос-ответ, когда клиент отправляет серверу запрос, и ожидает на него ответа, а сервер обслуживает этот запрос, подготавливает на него ответ и возвращает ответ ожидающему клиенту.

В примере выше показано именно такое взаимодействие. TCP взаимодействие запрос-ответ может реализовываться в двух вариантах:

1. Когда клиент открывает соединение (`connect()` со стороны клиента и `accept()` со стороны сервера), а затем **периодически** отправляет через это соединение (соединённый сокет) запросы, и через это же соединение сервер последовательно возвращает клиенту ответы на запросы. Это ровно та схема, которая показана выше. По этой схеме работают множество сетевых служб, примерами могут быть протоколы: Telnet, FTP и др. Эта схема хоть и очевидна, но чаще всё же используется другая схема...

2. Другая схема состоит в том, что клиент открывает **новое** соединение на каждый запрос. Через это установленное соединение клиент пересылает запрос, и через него же сервер возвращает результат. Возвратив результат на запрос сервер немедленно **закрывает** соединение (как вариант, соединение может закрывать клиент **после** получения ответа от сервера). Эта схема используется в Интернет ещё чаще, чем предыдущая, примером её использования есть HTTP (запросы клиента GET или POST). В такой схеме фрагмент кода сервера, показанный выше, трансформируется в ещё более простой код:

```

...
while (1) {                                // цикл по подключения
    int len = sizeof(serv_addr),
        newsockfd = accept(sockfd, (struct sockaddr*)&serv_addr, &len);
    if (newsockfd < 0) err_dump(...);       // ожидать соединения
    int n = readline(newsockfd, recvline, MAXLINE); // чтение от клиента
    if (n < 0) err_dump(...);
    if (writen(newsockfd, recvline, n) != n) // ретрансляция
        err_dump(...);
    close(newsockfd);                       // закрыть после ответа
}

```

Главное отличие здесь состоит в исключении цикла по поступающим запросам клиента — каждый запрос-ответ является **законченным** актом взаимодействия, и завершается закрытием соединения.

Клиент-сервер в UNIX домене

Для сравнения, в архиве примеров приведены клиент и эхо-сервер, работающие в семействе протоколов UNIX домена — это взаимодействие в пределах локального компьютера. Такое сравнение способствует гораздо более глубокому пониманию вопросов сетевого программирования. Подобным образом взаимодействуют графические (GUI) пользовательские приложения Linux с сервером графической подсистемы X11 и оконными менеджерами (вообще то говоря, сетевыми приложениями и протоколом). Показаны пары приложений (клиент и сервер) как для UDP (`unixdgcli.c` и `unixdgserv.c`), так и для TCP (`unixstrcli.c` и `unixstrserv.c`).

Их работа (тестирование) будет выглядеть аналогично показанному выше:

```
$ ./unixstrserv
...
$ ./unixdgserv
...
$ ./unixstrcli
UNIX сокет поток
UNIX сокет поток
...
$ ./unixdgcli
UNIX датаграмм поток
UNIX датаграмм поток
...
```

Управляющие операции

Функции `ioctl()` традиционно являлись системным интерфейсом управления для всего, что не попадало в какую-либо другую чётко определённую категорию. Стандарт POSIX постепенно избавляется от вызова `ioctl()` в различных ситуациях, заменяя её функциями-обёртками с стандартизированной функциональностью.

Тем не менее, `ioctl()` может использоваться во многих случаях получения информации от сетевого стека, или управлением функциональностью сокета. Функция определена как:

```
int ioctl(int fd, int request, void* arg);
```

Функция всегда применяется к файловому дескриптору или сокету. Вид 3-го аргумента полностью определяется целочисленным кодом операции `request` (часто это указатель на числовое значение, но может быть и сложная структура). Для считывания и изменения значения одного и того же параметра используются разные коды `ioctl()` (код определяет направление передачи данных). Определения некоторых кодов, относящиеся к области сокетов, можно найти в файле `<linux/sockios.h>`:

```
...
/* Socket configuration controls. */
#define SIOCGIFNAME    0x8910          /* get iface name          */
#define SIOCSIFLINK    0x8911          /* set iface channel        */
#define SIOCGIFCONF    0x8912          /* get iface list          */
#define SIOCGIFFLAGS    0x8913          /* get flags                */
#define SIOCSIFFLAGS    0x8914          /* set flags                */
...
```

Работающие примеры использования операций `ioctl()` можно видеть в подкаталоге `ioctl` каталога `ufd` архива примеров кода:

```

$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp2s14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT
group default qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlp8s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default
qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff

$ ./prifinfo inet4 0
lo: <UP LOOP >
    IP addr: 127.0.0.1
enp2s14: <UP BCAST MCAST >
    IP addr: 192.168.1.5
    broadcast addr: 192.168.1.255

```

Классы обслуживания сервером

Системам UNIX свойственна клиент-серверная архитектура, эта архитектура имеет в UNIX много десятилетий историю, и многие крупные проекты построены именно по этой архитектуре. Именно поэтому, для клиент-серверной архитектуры изучено много вариантов того, как работает обслуживающий сервер. Эти варианты рассмотрены ниже.

Рассмотрение сосредоточено вокруг TCP серверов, так как они наиболее часто используют какие-либо формы распараллеливания обслуживания, для серверов TCP это особенно актуально, так как именно на этом протоколе строятся, главным образом, высоконагруженные сервера массового обслуживания. Но и для сервера UDP могут быть также реализованы параллельные формы обслуживания.

Различные варианты показаны в каталоге архива xservers. В проекте реализован специализированный TCP клиент (файл cli.cc), который посылает требуемое число раз¹⁰ запросы к нужному серверу (определяется выбором порта), принимает от него ответ, и тут же разрывает соединение (по такой примерно схеме обрабатываются запросы в HTTP протоколе). Сервера представляют собой простые ретрансляторы, но выполненные в различной технике.

Клиент измеряет время (точнее – **число тактов** частоты процессора, это высокая точность наносекундного диапазона¹¹) между отправкой запроса серверу и приходом ответа от него. Для фиксации числа тактов с момента старта процессора используется измерение аппаратного счётчика RDTSC, присутствующего во всех современных моделях процессоров семейства x86. Код записан с использованием инлайнового ассемблера проекта GCC:

rdtsc.c :

```

uint64_t rdtsc(void) {
    union sc {
        struct { uint32_t lo, hi; } r;
        uint64_t f;
    } sc;
    __asm__ __volatile__ ("rdtsc" : "=a"(sc.r.lo), "=d"(sc.r.hi));
    return sc.f;
}

// вариант для старых GCC:
// __asm__ __volatile__ (".byte 0x0f, 0x31" : "=a"(sc.r.lo), "=d"(sc.r.hi));

```

¹⁰ Серия запросов от клиента делается для усреднения результата и для того (как будет видно далее), чтобы исключить (или наоборот учесть, выделить) эффекты кэширования памяти.

¹¹ Инструмент для такого хронометража собран в статически компоновку библиотеку libdiag.a, все коды для неё представлены в архиве, но здесь они обсуждаться не будут.

```
// вариант не только для 32-бит i686:
// asm volatile ("rdtsc" : "=A" (x));
```

Каждый запрос к серверу представляет собой случайное число, генерируемое клиентом, в символьной форме. Ретранслированный сервером ответ сверяется с запросом для дополнительного контроля. Все показанные программы показаны в упрощённых вариантах: не везде сделана полная обработка ошибочных ситуаций, и сознательно не включена обработка сигнала SIGCHLD, которая должна препятствовать появлению «зомби» процессов.

Код клиента мы не обсуждаем (он приведен в архиве), но относительно его опций запуска: -a — IP адрес сервера (по умолчанию это localhost), -p — значение TCP порта подключения (по умолчанию это 51000, что соответствует простому последовательному серверу) и -n — число запросов к серверу в серии (по умолчанию 10).

Программный код выполнен в нотации языка C++ (хотя специфические объектные особенности C++, за исключением потокового ввода-вывода C++ и не использованы — всё то же легко выписать на классическом C, но оно выглядит несколько более объёмным, и это одна из причин выбора языка иллюстраций). Я собираю и показываю иллюстрации с компилятором и библиотекой:

```
$ gcc --version
gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

(В заметно более ранних версиях GCC были отличия в путевых именах заголовочных файлов, но это легко решаемые проблемы.)

Часть общих определений и функций этой иллюстрации вынесены в общие файлы, они используются всеми вариантами серверов, и их общее использование сильно упрощает сравнительное изучение собственно кодов серверов:

common.h :

```
#if !defined(__COMMON_H)
#define __COMMON_H

#include <iostream>
#include <iomanip>
using namespace std;
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include "libdiag.h"

#define EOK 0

const int PORT = 51000,
        SINGLE_PORT = PORT,          // 51001
        FORK_PORT = PORT + 1,
        FORK_LARGE_PORT = PORT + 2,
        PREFORK_PORT = PORT + 3,
        XINETD_PORT = PORT + 4,      // 51004
        THREAD_PORT = PORT + 5,
        THREAD_POOL_PORT = PORT + 6,
        PRETHREAD_PORT = PORT + 7,
        QUEUE_PORT = PORT + 8;      // 51008
const int MAXLINE = 40;
```

```

// критическая ошибка ...
void errx(const char *msg, int err = EOK);
// ретранслятор тестовых пакетов TCP
void retrans(int sc);
// создание и подготовка прослушивающего сокета
int getsocket(in_port_t);
extern int debug; // уровень отладочного вывода сервера
// параметры строки запуска сервера
void setv(int argc, char *argv[]);
#endif
common.c :

#include "common.h"

// диагностика ошибки ...
void errx(const char *msg, int err) {
    perror(msg);
    if (err != EOK) errno = err;
    exit(EXIT_FAILURE);
}

// ретранслятор тестовых пакетов TCP
static char chdata[MAXLINE];
void retrans(int sc) {
    int rc = read(sc, chdata, MAXLINE);
    if (rc > 0) {
        rc = write(sc, chdata, strlen(chdata) + 1);
        if (rc < 0) perror("write data failed");
    }
    else if (rc < 0) { perror("read data failed"); return; }
    else if (rc == 0) { cout << "client closed connection" << endl; return; };
    return;
}

// создание и подготовка прослушивающего сокета
static struct sockaddr_in addr;
int getsocket(in_port_t p) {
    int rc = 1, ls;
    if (-1 == (ls = socket(AF_INET, SOCK_STREAM, 0)))
        errx("create stream socket failed");
    if (setsockopt(ls, SOL_SOCKET, SO_REUSEADDR, &rc, sizeof(rc)) != 0)
        errx("set socket option failed");
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(p);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(ls, (struct sockaddr*)&addr, sizeof(sockaddr)) != 0)
        errx("bind socket address failed");
    if (listen(ls, 25) != 0)
        errx("put socket in listen state failed");
    cout << "waiting on port " << p << " ..." << endl;
    return ls;
}

// уровень отладочного вывода сервера
int debug = 0;
void setv(int argc, char *argv[]) {
    debug = (argc > 1 && 0 == strcmp(argv[1], "-v")) ? 1 : 0;
}

```

```

    if (debug) cout << "verbose mode" << endl;
}

```

При тестировании можно запускать клиент и сервера на одном хосте (сервер localhost по умолчанию), на отдельных хостах LAN (чтобы клиент и сервер не разделяли производительность одних и тех же процессоров при временных замерах), или удалённо в глобальной сети. Но временные задержки реакции сервера намного ниже, чем время распространения по физической сети, поэтому сравнения проведём именно на петлевом интерфейсе.

Последовательный сервер

Код простейшего последовательного сервера:

ech0.cc :

```

// последовательный ретранслятор тестовых пакетов TCP
int main(int argc, char *argv[]) {
    int ls = getsocket(SINGLE_PORT), rs;
    setv(argc, argv);
    while (true) {
        if ((rs = accept(ls, NULL, NULL)) < 0)
            errx("accept error");
        retrans(rs);
        close(rs);
        if (debug) cout << "*" << flush;
    }
    exit(EXIT_SUCCESS);
}

```

Выполнение:

```

$ sudo nice -n-19 ./ech0
waiting on port 51000 ...
^C
$ sudo nice -n-19 ./cli -p 51000 -n25
host: localhost, TCP port = 51000, number of echoes = 25
time of reply - Cycles [usec.] :
201615[84]      68259[28]      61590[25]      58374[24]      56883[23]
58092[24]      57909[24]      57522[23]      57519[23]      58440[24]
57117[23]      56571[23]      57057[23]      55872[23]      59031[24]
56148[23]      56808[23]      51144[21]      53736[22]      52761[21]
52398[21]      52269[21]      50538[21]      52053[21]      51636[21]
Mean: 22.875 | Mean square deviation: 1.61536

```

Параллельный сервер

Следующий вариант - это сервер, который на протяжении нескольких десятилетий считается классической реализацией параллельного сервера: когда по поступлению очередного запроса (accept()) порождается (вызовом fork()) новый обслуживающий процесс (клон родительского процесса), который и занимается обслуживанием поступившего запроса. Родительский же процесс при этом возвращается в режим прослушивания следующих поступающих запросов:

ech1.cc :

```

// ретранслятор с fork
int main(int argc, char *argv[]) {
    int ls = getsocket(FORK_PORT), rs;
    setv(argc, argv);
    while (true) {

```

```

    if ((rs = accept(ls, NULL, NULL)) < 0)
        errx("accept error");
    pid_t pid = fork();
    if (pid < 0)
        errx("fork error");
    if (pid == 0) {
        close(ls);
        retrans(rs);
        close(rs);
        if (debug) cout << "*" << flush;
        exit(EXIT_SUCCESS);
    }
    else close(rs);
}
exit(EXIT_SUCCESS);
}

```

Выполнение:

```

$ sudo nice -n-19 ./ech1
waiting on port 51001 ...
^C
$ sudo nice -n-19 ./cli -p 51001 -n25
host: localhost, TCP port = 51001, number of echoes = 25
time of reply - Cycles [usec.] :
754818[314]    538815[224]    656835[273]    568614[236]    506487[211]
438210[182]    636324[265]    426072[177]    556410[231]    479442[199]
631098[262]    461304[192]    480411[200]    599235[249]    476529[198]
395757[164]    387363[161]    415428[173]    485223[202]    764427[318]
521814[217]    783915[326]    406083[169]    556893[232]    534963[222]
Mean: 220.125 | Mean square deviation: 43.8095

```

Такой сервер может обслуживать достаточно много запросов одновременно (при условии, что у него остаётся на то достаточно ресурсов процессоров и памяти). Но, как и следовало ожидать, реактивность (пауза перед обслуживанием) у него значительно хуже.

Предварительное клонирование процесса

Получается, что для серверов, работающих на высоко интенсивных потоках запросов, с традиционным fork-методом всё не так хорошо со скоростью реакции... Но можно поменять порядок вызовов `fork()` и `accept()` местами – создать заранее некоторый пул обслуживающих процессов, каждый из которых до прихода клиентского запроса будет заблокирован на `accept()` (все `accept()` на одном и том же пассивном сокете, что не предусмотрено спецификацией, но это работает!). А после отработки очередного клиентского запроса заблаговременно создать новый обслуживающий процесс. Эта техника описана в литературе как «предварительный fork» или `pre-fork`. Меняем код сервера:

ech11.cc :

```

const int NUMPROC = 3;
// ретранслятор с предварительным fork (prefork)
int main(int argc, char *argv[]) {
    int ls = getsocket(PREFORK_PORT), rs;
    setv(argc, argv);
    for (int i = 0; i < NUMPROC; i++) {
        if (fork() == 0) {
            int rs;
            while (true) {
                if ((rs = accept(ls, NULL, NULL)) < 0)
                    errx("accept error");
            }
        }
    }
}

```

```

        retrans(rs);
        close(rs);
        if (debug)
            cout << i << flush;
        delay(250); // пауза 250 usec.
    }
}
}
for (int i = 0; i < NUMPROC; i++)
    wait(NULL);
exit(EXIT_SUCCESS);
}

```

Выполнение:

```

$ sudo nice -n-19 ./ech11
waiting on port 51003 ...
^C
$ sudo nice -n-19 ./cli -p 51003 -n25
host: localhost, TCP port = 51003, number of echoes = 25
time of reply - Cycles [usec.] :
121971[50]    100428[41]    96906[40]    64575[26]    72657[30]
62886[26]    61074[25]    59622[24]    56355[23]    56481[23]
58227[24]    55269[23]    54381[22]    58335[24]    55287[23]
67938[28]    56145[23]    53421[22]    63234[26]    58638[24]
55389[23]    57414[23]    55443[23]    54297[22]    54039[22]
Mean: 25.4167 | Mean square deviation: 4.94062

```

Здесь цифры весьма близкие к простому последовательному серверу.

Создание потока по запросу

Строим параллельный сервер (файл ech2.cc), но вместо параллельных клонов процессов теперь будем порождать параллельные потоки в том же адресном пространстве:

ech2.cc :

```

void* echo(void* ps) {
    int sc = *(int*)ps;
    sched_yield();
    retrans(sc);
    close(sc);
    if (debug)
        cout << "*" << flush;
    return NULL;
}

// пертранслятор с pthread_create по запросу
int main(int argc, char *argv[]) {
    int ls = getsocket(THREAD_PORT), rs;
    setv(argc, argv);
    while (true) {
        pthread_t tid;
        if ((rs = accept(ls, NULL, NULL)) < 0)
            errx("accept error");
        if (pthread_create(&tid, NULL, &echo, &rs) != EOK)
            errx("thread create error");
        sched_yield();
    }
    exit(EXIT_SUCCESS);
}

```



```
}
```

Выполнение:

```
$ sudo nice -n-19 ./ech2
waiting on port 51005 ...
^C
$ sudo nice -n-19 ./cli -p 51005 -n25
host: localhost, TCP port = 51005, number of echoes = 25
time of reply - Cycles [usec.] :
1015857[423]    222273[92]    309009[128]    208569[86]    220194[91]
326541[136]    196122[81]    189402[78]    189324[78]    183285[76]
182238[75]     184515[76]    184845[77]    191820[79]    197787[82]
259284[108]    217791[90]    194625[81]    215502[89]    193584[80]
188184[78]     209352[87]    192684[80]    198282[82]    320907[133]
Mean: 89.2917 | Mean square deviation: 17.7728
```

Пул потоков

Точно так, как мы это делали с предварительным созданием клона процесса (ech11.cc), мы можем создать сервер, который будет предварительно создавать несколько потоков, которые будут заблокированы на `accept()` в ожидании запросов на обслуживание. Мы, фактически, поменяли местами вызовы `pthread_create()` и `accept()` в предыдущей схеме:

ech22.cc :

```
static int ntr = 3;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

void* echo(void* ps) {
    int sc = *(int*)ps, rs;
    sched_yield();
    if ((rs = accept(sc, NULL, NULL)) < 0)
        errx("accept error");
    retrans(rs);
    close(rs);
    pthread_mutex_lock(&mutex);
    ntr++;
    pthread_cond_signal(&condvar);
    pthread_mutex_unlock(&mutex);
    if (debug)
        cout << pthread_self() << '.' << flush;
    delay(250); // пауза 250 usec.
    return NULL;
}

// ретранслятор с предварительным pthread_create()
int main(int argc, char *argv[]) {
    int ls = getsocket(PRETHREAD_PORT), rs;
    setv(argc, argv);
    while (true) {
        pthread_t tid;
        if (pthread_create(&tid, NULL, &echo, &ls) != EOK)
            errx("thread create error");
        sched_yield();
        pthread_mutex_lock(&mutex);
        ntr--;
        while (ntr <= 0)
            pthread_cond_wait(&condvar, &mutex);
        pthread_mutex_unlock(&mutex);
    }
}
```

```

    }
    exit(EXIT_SUCCESS);
}

```

Здесь `ассерт()` перенесен в обрабатывающий поток. Какой конкретно из потоков будет разблокирован для обработки при получении запроса — непредсказуемо! Для синхронизации использована условная переменная, но могут применяться любые из синхронизирующих примитивов.

Выполнение:

```

$ sudo nice -n-19 ./ech22
waiting on port 51007 ...
^C
$ sudo nice -n-19 ./cli -p 51007 -n25
host: localhost, TCP port = 51007, number of echoes = 25
time of reply - Cycles [usec.] :
221172[92]      79551[33]      83268[34]      70563[29]      96792[40]
61968[25]      69990[29]      68184[28]      69705[29]      69486[28]
68427[28]      69159[28]      60072[25]      60039[25]      60069[25]
59787[24]      79071[32]      67590[28]      59730[24]      58710[24]
68250[28]      67662[28]      58119[24]      81150[33]      167340[69]
Mean: 30 | Mean square deviation: 8.97682

```

Последовательный сервер с очередью обслуживания

Есть ещё один класс серверов, который при определённых условиях может оказаться оптимальнее параллельных. Это последовательный сервер с очередью обслуживания.

ech4.cc :

```

static queue<int> events;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* reply(void*) {
    int rsq;
    while (true) {
        if (events.empty())
            continue;
        pthread_mutex_lock(&mutex);
        rsq = events.front();
        events.pop();
        if (debug)
            cout << '-' << rsq << '.' << flush;
        pthread_mutex_unlock(&mutex);
        retrans(rsq);
        close(rsq);
    }
}

// последовательный ретранслятор с очередью обслуживания
int main(int argc, char *argv[]) {
    int ls = getsocket(Queue_PORT);
    setv(argc, argv);
    pthread_t tid;
    if (pthread_create(&tid, NULL, &reply, NULL) != EOK)
        errx("thread create error");
    while (true) {
        int rs = accept(ls, NULL, NULL);
        if (rs < 0) errx("accept error");
        pthread_mutex_lock(&mutex);

```

```

    events.push(rs);
    if (debug)
        cout << '+' << rs << '.' << flush;
    pthread_mutex_unlock(&mutex);
}
exit(EXIT_SUCCESS);
}

```

Выполнение:

```

$ sudo nice -n-19 ./ech4
waiting on port 51008 ...
^C
$ sudo nice -n-19 ./cli -p 51008 -n25
host: localhost, TCP port = 51008, number of echoes = 25
time of reply - Cycles [usec.] :
140265[58]    101799[42]    56643[23]    58818[24]    58521[24]
55710[23]    56742[23]    52230[21]    55062[22]    60906[25]
62130[25]    56286[23]    56343[23]    56433[23]    42876[17]
51561[21]    50826[21]    52536[21]    48429[20]    53763[22]
52680[21]    50244[20]    49935[20]    48507[20]    53337[22]
Mean: 22.75 | Mean square deviation: 4.39934

```

Как понятно при внимательном рассмотрении, архитектура с входной очередью обслуживания может быть расширена также на **любую параллельную** реализацию сервера. Это может быть очень перспективным направлением развития для малых, экономных и встраиваемых систем, где нужно препятствовать интенсивному разбазариванию ресурсов (за счёт бесконтрольного создания дочерних процессов или потоков).

Суперсервер и сокетная активация

Есть ещё один способ построения и запуска сетевых серверов, сочетающий простоту кодирования с экономичностью решения для бюджетных архитектур. Поскольку в нагруженной системе может быть достаточно большое число (несколько десятков) различных серверов, ожидающих подключений на разных портах, находящихся в пассивном ожидании и потребляющих при этом ресурсы (память), то в UNIX была давно была предложена другая техника активации серверов — по запросу. Это реализуется использованием суперсервера и сокетной активации¹². Философия сокетной активации состоит в том:

- Запущенный суперсервер пассивно **прослушивает** **весь** поддерживаемый (конфигурированный) диапазон портов UDP и TCP.
- При появлении активности (запросе от клиента) на каком либо из этих портов, суперсервер **запускает** программу, приписанную (в конфигурационных файлах) в качестве сервера для этого протокола. Этой программой сервера может быть как штатная реализация, так и ваше собственное приложение.
- Весь ввод-вывод из сети при этом продолжает приниматься и отправляться суперсервером, но он перенаправляет сетевой ввод-вывод на **стандартные потоки** ввода-вывода (SYSIN и SYSOUT) запущенной программы сервера.

Первым исторически из суперсерверов, наиболее известными, и используемым до сегодня на малых и встраиваемых архитектурах являются `inetd`. В десктопных и серверных архитектурах более часто используется следующее поколение `xinetd`. Большинство функциональности сокетной активации включено также в поддержку новой системы управления загрузкой и сервисами `systemd`, пришедшей на смену традиционной системе `init`.

¹² Это способ очень в духе UNIX и очень широко используемый в UNIX различными проектами.

Все из суперсерверов опираются на перечисление **зарегистрированные** сетевых служб (протоколов, серверов) в системном файле /etc/services:

```
$ cat /etc/services | wc -l
11176
$ cat /etc/services
...
echo          7/tcp
echo          7/udp
...
daytime       13/tcp
daytime       13/udp
...
ftp           21/tcp
ftp           21/udp          fsp fspd
ssh           22/tcp          # The Secure Shell (SSH) Protocol
ssh           22/udp          # The Secure Shell (SSH) Protocol
telnet        23/tcp
telnet        23/udp
...
```

Различие суперсерверов проявляется в их конфигурировании.

- Непосредственно **прослушиваемые** inetd порты записаны в его конфигурационном файле /etc/inetd.conf, по принципу: один сервис — одна строка конфигурации. В этой строке от 6-ти до 11-ти **позиционных** параметров, в строго предопределённом порядке, разделённых пробелами, никакие переносы строки не допускаются.
- Конфигурации xinetd более обстоятельны, описываются не одной строкой **позиционных** параметров на сервис, как для inetd, а целым блок **ключевых** параметров. Используемых параметров довольно много, большинство из них — опциональные, а основные очень похожи и соответствуют полям строки конфигурации inetd. Эти конфигурации записываются файлами в каталоге /etc/xinetd.d (чаще по одному сервису на файл, но может быть и по несколько сервисов одним файлом — все файлы этого каталога читаются **последовательно** как одно целое). Здесь каждый сервис конфигурируется одной **записью** (заклѳюченной в блок скобками { . . . }).
- Управляющие файлы systemd (управляющие стартом и остановом всех служб Linux) размещаются в каталоге /usr/lib/systemd/system. Для сокетной активации службы создаются 2 стартовых файла, на примере SSH это sshd.socket и sshd@.service: первый содержит описание сокета и порта, а второй — правил старта сервера. Формат параметров этих файлов подобен xinetd.

Рассмотрим запуск посредством сокетной активации сервера (ретранслятора), аналогичного рассматриваемым выше. Большим достоинством этого метода есть то, что сервер может быть выполнен быстро, на **любых** языках, компилирующих или скриптовых (ниже показаны несколько реализаций: C, C++, bash, JavaScript). Вот несколько примеров, любой из которых мы можем использовать (всѳ тот же каталог архива xservers):

ech3c.c :

```
int main(int argc, char *argv[]) {
    char data[MAXLINE];
    while (1)
        write(STDOUT_FILENO, data, read(STDIN_FILENO, data, MAXLINE));
    exit(EXIT_SUCCESS);
}
```

ech3cc.cc :

```

int main(void) {
    char buf[MAXLINE];
    // простейшая ретрансляция ...
    while (true) {
        if ((cin >> buf).eof()) break;
        cout << buf << endl;
    }
    return EXIT_SUCCESS;
}

```

ech3.sh :

```

#!/bin/bash
while [TRUE]
do
    read buf
    if [[${#buf} -eq 0]]    # ^D - конец ввода
    then break
    fi
    echo $buf
done

```

ech3.js :

```

#!/usr/bin/js
while (true) {
    var buf = readline();
    if (buf === null) {        // ^D - конец ввода
        break;
    }
    print(buf);
}

```

Как легко видеть, сервер в этом варианте приобретает простейший вид, и его (любой) легко оттестировать в локальном запуске:

```

$ ~/ech3
1
1
12
12
123
123
^D

```

За простоту программного кода сервера нужно платить некоторой усложнённостью системной конфигурации для его запуска, для этого нужно:

1. При запуске сервера должно указываться **абсолютное путевое имя** программы сервера, поэтому копируем программу куда-то в легкодоступное место (в экспериментах — в домашний каталог пользователя):

```

$ cp ech3.sh $HOME
$ ls ~/ech*
/home/Olej/ech3.sh

```

2. Добавить запись конфигурации xinetd в каталог /etc/xinetd.d отдельным файлом (в примере используем xinetd, остальные суперсервера конфигурируются подобно):

```

# cat /etc/xinetd.d/ech3
service ech3

```

```
{
    disable = no
    protocol = tcp
    wait = no           # параллельный сервер
    user = 0lej         # имя от которого запускать
    server = /home/0lej/ech3 # путь к программе
}
```

3. При каждой правке конфигураций заставить перечитать новые конфигурации. Это можно сделать либо послав запущенному серверу сигнал SIGHUP:

```
# ps -A | grep inetd
7408 ?          00:00:00 xinetd
# kill -SIGHUP 7408
```

Либо это можно сделать полностью перезапустив суперсервер:

```
# systemctl restart xinetd.service
# systemctl status xinetd.service
xinetd.service - Xinetd A Powerful Replacement For Inetd
Loaded: loaded (/usr/lib/systemd/system/xinetd.service; enabled)
Active: active (running) since Бс 2014-06-15 23:33:41 EEST; 42min ago
...
```

Теперь у нас всё готово к испытаниям полученного сервера:

```
# sudo nice -n19 ./cli -a 192.168.1.5 -p51004 -n20
host: 192.168.1.5, TCP port = 51004, number of echoes = 20
time of reply - Cycles [usec.] :
11242486[4333] 15039250[5797] 10533778[4060] 10174720[3922] 21456950[8271]
14830070[5716] 10340546[3986] 12170304[4691] 16235328[6258] 46813980[18046]
10834944[4176] 42702314[16461] 9714874[3745] 9897594[3815] 8937908[3445]
10455986[4030] 10009446[3858] 35143576[13547] 24846692[9578] 13429070[5176]
```

Здесь задержки, естественно, гораздо больше (на 1-2 порядка), чем у рассмотренных ранее «нативных» реализаций, но для очень многих проектов это не является критическим фактором, а простота и **гибкость** перенастроек (через конфигурационные файлы) перекрывают этот недостаток.

На что хотелось бы обратить внимание в завершение рассмотрения сокетной активации? **Во-первых**, на то, что хотя ваша собственная программа **сервера** и работает (как обычная консольная программа-фильтр) с входным потоком SYSIN (дескриптор 0) и выходным потоком SYSOUT (дескриптор 1), но программа здесь может применять к этим **потокам** также весь API, применяемый для работы с сетевыми сокетами: `getpeerbyname(0, ...)`, `getsockopt(1, ...)`, `setsockopt(1, ...)`, `send(1, ...)`, `recv(0, ...)`, `sendto(1, ...)`, `recvfrom(0, ...)`, ...

Ниже показано как, достаточно необычно, может выглядеть **многопоточный** UDP-сервер, работающий с запуском по сокетной активации: в обычной конфигурации UDP сервер обрабатывает датаграммы последовательно, что препятствует дальнейшему прослушиванию порта до завершения обработки текущего запроса (архив `xinetd` подкаталог `udp-connected`). Здесь запускаемый `xinetd` экземпляр сервера создаёт соединённый UDP-сокет (упоминался ранее) по поступившему запросу, обработка (сколь угодно продолжительная) и ответ в этот сокет осуществляются в отдельном дочернем процессе:

common.h :

```
#ifndef _COMMON_H
#define _COMMON_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

```
#include <errno.h>
#include <netdb.h>

#define MAXLEN 1500
#define SDELIM " > "

#endif
```

udpsocserv.c :

```
#include <syslog.h>
#include <sys/wait.h>

#include "common.h"

// добавить запись UDP порта в /etc/services и в конфигурацию в /etc/xinetd.d
int main(void) {
    char rbuf[MAXLEN];
    openlog(NULL, LOG_NDELAY, LOG_USER);
    syslog(LOG_NOTICE, "server [%d] start", getpid());
    while (1) {
        struct sockaddr_in adr;
        socklen_t len = sizeof(struct sockaddr_in);
        int n = recvfrom(STDIN_FILENO, rbuf, MAXLEN, 0, // чтение из SYSIN для xinetd:
                        (struct sockaddr*)&adr, &len);
        if (n >= 0) syslog(LOG_NOTICE, "server read %d byte", n);
        if (n <= 0) {
            if (n < 0) syslog(LOG_ERR, "recvfrom error: %m"), exit(EXIT_FAILURE);
            break;
        }
        rbuf[n] = '\0';
        int sc = socket(AF_INET, SOCK_DGRAM, 0); // open a UDP socket
        if (sc < 0) syslog(LOG_ERR, "socket error: %m"), exit(EXIT_FAILURE);
        if (connect(sc, (struct sockaddr*)&adr, sizeof(struct sockaddr_in)) < 0)
            syslog(LOG_ERR, "connect error: %m"), exit(EXIT_FAILURE);
        pid_t pid = fork();
        if (0 == pid) { // обработка в дочернем процессе
            close(STDIN_FILENO);
            close(STDOUT_FILENO);
            sleep(1); // имитация времени обработки
            char wbuf[MAXLEN + 8];
            sprintf(wbuf, "[%d]s%s", getpid(), SDELIM, rbuf);
            if (write(sc, wbuf, strlen(wbuf)) < 0) // ретрансляция данных
                syslog(LOG_ERR, "write error: %m"), exit(EXIT_FAILURE);
            exit(EXIT_SUCCESS); // завершение процесса
        }
        else if (pid > 0) {
            if (n != 1) waitpid(-1, NULL, WNOHANG); // продолжение работы
            else { // пустая "\n" строка от клиента
                wait(NULL);
                exit(EXIT_SUCCESS);
            }
        }
        else syslog(LOG_ERR, "fork error: %m"), exit(EXIT_FAILURE);
    }
    syslog(LOG_NOTICE, "server [%d] exit", getpid());
    closelog();
    exit(EXIT_SUCCESS);
}
```

Клиент для экспериментов с такими серверами (в архиве их несколько) — мы не можем использовать в качестве тестового клиента программу `telnet` как для TCP, потому что это UDP и в этом случае для каждого проекта нужно писать свою клиентскую часть:

udpcli.c :

```
#include <arpa/inet.h>
#include "common.h"

void dg_cli(FILE* fp, int sockfd,
            struct sockaddr* pserv_addr, // ptr to appropriate sockaddr structure
            int servlen) {              // actual sizeof(*pserv_addr)
    int n;
    char line[MAXLEN];
    while (fgets(line, MAXLEN, fp) != NULL) {
        n = strlen(line);
        if (sendto(sockfd, line, n, 0, pserv_addr, servlen) != n)
            printf("send: %m\n"), exit(EXIT_FAILURE);
        if ((n = recvfrom(sockfd, line, MAXLEN, 0,
                          (struct sockaddr*)NULL, (socklen_t*)NULL)) < 0)
            printf("recv: %m\n"), exit(EXIT_FAILURE);
        line[n] = '\0'; // null terminate
        fputs(line, stdout);
        if (strstr(line, SDELIM) == NULL) continue;
        if (strlen(strstr(line, SDELIM)) == strlen(SDELIM) + 1)
            printf("server exit\n");
    }
}

int main(int argc, char* argv[]) {
    char serv_host_addr[16] = "127.0.0.1"; // host addr for server
    int serv_udp_port = 50010;           // server UDP port
    if (argc > 1) strcpy(serv_host_addr, argv[1]);
    if (argc > 2) serv_udp_port = atoi(argv[2]);
    printf("UDP server: %s:%d\n", serv_host_addr, serv_udp_port);
    int sockfd;
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) // open a UDP socket
        printf("socket: %m\n"), exit(EXIT_FAILURE);
    struct sockaddr_in serv_addr;
    bzero((char*)&serv_addr, sizeof(serv_addr)); // fill address structure
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(serv_host_addr);
    serv_addr.sin_port = htons(serv_udp_port);
    dg_cli(stdin, sockfd, (struct sockaddr*)&serv_addr, sizeof(struct sockaddr_in));
    close(sockfd);
    exit(EXIT_SUCCESS);
}
```

Здесь принятый пакет суперсервер передал серверу через `STDIN_FILENO` (`fd = 0`), но тут же по сокетному адресу принятого сообщения создаётся дубликат соединённого сокета `sc`, и вся работа с этим сокетом и подготовка ответа на запрос производится в копии процесса (`fork()`):

```
$ ./udpcli 127.0.0.1 50011
UDP server: 127.0.0.1:50011
1
[11489] > 1
12
[11490] > 12
```



```

123
[11491] > 123

[11492] >
server exit
1234
[11494] > 1234

[11495] >
server exit
^C

```

Сервер в ретранслируемом сообщении от клиента (в заголовке) сообщает свой PID. Пример сделан так, что при вводе в клиенте (для передачи серверу) пустой строки (Enter), текущий экземпляр сервера завершается. Но по следующему сообщению от клиента xinetd запустит новый экземпляр сервера.

Что можно в заключение сказать о сокетной активации?

Что **во-первых**, поскольку программа сервер в этой схеме работает как фильтр (вход-выход), то в качестве сервера под управлением суперсервера может исполняться (через транзитный запускающий уровень, как дочернее приложение) практически любая утилита из набора штатных программ Linux, например консольные клиенты запросов PostgreSQL или MySQL. (такое решение приведено в архиве примера child).

А **во-вторых**, что запуск сервера посредством сокетной активации требует кропотливой конфигурации и настройки, достаточно сложны в отладке. Но такой способ того стоит! Особо обратите внимание при отработке такого способа на то, чтобы контролируемые суперсервером порты не были закрыты сетевым файрволом хоста — в данной технологии это сложно диагностируемая ситуация.

Независимые POSIX CLI сетевые тестеры

В сетевых приложениях (проектах), как ни в какой другой тематике программирования, в качестве разнообразных программ-тестеров наилучшим способом будет использование разнообразных текстовых (консольных, CLI) утилит и приложений, следующих соглашениям стандарта POSIX. Благо, в Linux (в стандартных репозиториях) таких приложений великие сотни, если не тысячи.

О том, что это наилучшая методика тестирования сетевых приложений уже было сказано в предыдущих главах. Почему наилучшая? Потому что завершённые Linux приложения, даже если они не фигурируют формально в перечне POSIX-совместимых, то они следуют букве и духу стандартов POSIX.

В предыдущем разделе мы отработывали несколько эхо-серверов протокола TCP, и к ним для тестирования создали соответствующего клиента. Но клиент не является необходимым! (И был представлен только для демонстрации кода.) Сейчас мы посмотрим только несколько из множества консольных приложений, которые можно и нужно использовать в качестве клиентов для тестирования.

Запустим один (любой) из представленных ранее ретранслирующих эхо-серверов:

```

$ ./ech0
waiting on port 51000 ...

```

А далее станем забрасывать его сообщениями от разных утилит (без какого-либо перезапуска сервера):

```

$ echo строка с переводом строки | nc localhost 51000
строка с переводом строки
$ telnet localhost 51000
Trying 127.0.0.1...

```

```

Connected to localhost.
Escape character is '^]'.
строка из telnet
строка из telnet
Connection closed by foreign host.
$ telnet localhost 51000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ещё одна строка
ещё одна строка
Connection closed by foreign host.

```

Замечания о telnet

Но в отношении такого популярного, удобного и мощного инструмента как `telnet`, в качестве тестера сетевых обменов, нужно соблюдать определённую осторожность, и всегда иметь в уме следующее:

- Клиент `telnet` может работать в двух режимах, переключаемых командой `mode` из его набора команд— построчном (`mode line`) и посимвольном (`mode char`). Для наших целей представляет интерес построчный режим, и он устанавливается по умолчанию при старте программы.
- Сам протокол `telnet` (RFC-854 и целый ряд последующих) использует передачу некоторых последовательностей управляющих символов (в построчном режиме) непосредственно в потоке TCP-соединения.
- Стандарт `telnet` — это один из самых старых сетевых стандартов, разрабатывался начиная с 1973 года, когда о мультязычности и UNICODE ещё никто не помышлял, кодирование UTF-8 будет разработано Кеном Томпсоном и Робби Пайком только к 1992 году, а текстовой информацией считалось только то, что укладывалось в кодировку ASCII, а ещё точнее, в её первую половину, то есть в символы латинского алфавита.
- И поэтому может стать большой неожиданностью, что при передаче от `telnet` символов многобайтных символов UTF-8 любого языка, символы в потоке передачи (именно передачи) будут искажаться.

Чтобы не быть голословным, подвергнет сказанное короткой проверке. И, чтобы не грешить на ошибки реализации, в качестве ответного эхо-сервера будем использовать реализацию исключительно средствами самих утилит системы (об этом подробно будет рассказано позже):

```
$ nc -l -p 50001 -c '/bin/cat'
```

И подключение `telnet` в качестве консольного клиента:

```

$ telnet 127.0.0.1 50001
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
1 23 456
1 23 456
call echo
call echo
русская строка
у
я
х
0

```

```
и снова ...
и снова ...
^]
telnet> quit
Connection closed.
```

Как видите, некоторые русские литеры либо прерывают эхо-тракт (нет эха), либо искажаются в нём. Я нашёл таких литер 3: у, я, х ... на самом деле их, может, и больше.

И, наконец, для контроля, подключимся к тому же серверу любым другим клиентом, например, всё тем же nc в качестве клиента:

```
$ nc 127.0.0.1 50001
русская строка
русская строка
у
у
я
я
х
х
^C
```

Расширенные операции ввода-вывода

В качестве **расширенных** (по сравнению с блокирующими `read()`, `write()` и всех из их многочисленных вариаций) операций обычно в обсуждениях обобщённо называют: расширенные операции ввода-вывода. К этой части обычно относят рассмотрение: `select()`, `pselect()`, `poll()`, `epoll()`, асинхронного ввода-вывода и подобных вопросов. Пожалуй, самую ясную и строгую классификацию моделей ввода-вывода в UNIX дал У. Р. Стивенс:

*Прежде чем начать описание функций `select` и `poll`, мы должны вернуться назад и уяснить основные различия между **пятью** моделями ввода-вывода, доступными нам в Unix:*

- *блокируемый ввод-вывод;*
- *неблокируемый ввод-вывод;*
- *мультиплексирование ввода-вывода (функции `select` и `poll`);*
- *ввод-вывод, управляемый сигналом (сигнал `SIGIO`);*
- *асинхронный ввод-вывод (функции `POSIX.1 aio_`).*

Все эти возможные модели осуществления обменных операций, что часто упускается из виду, относятся к операциям **над любыми** обменными объектами в программном коде: символьными потоками терминального ввода-вывода, файловыми дескрипторами, сетевыми сокетами... Но только на сетевых сокетах наиболее отчётливо проявляются отличительные стороны всех режимов, и для сетевых сокетов расширенные режимы ввода-вывода наиболее часто используются на практике. Так, на файловых дескрипторах или дескрипторах символьных устройств мы можем часто наблюдать блокирование на операциях `read()`, но гораздо реже — на операциях `write()` (в частности, и из-за операций кэширования записи на диск). Но на сокетах операция `write()` столь же часто может переходить в заблокированное состояние, когда сетевой стек ещё не готов отправить передаваемые данные.

Блокируемый ввод — это самый часто используемый, и самый известный вариант, когда выполняется операция `read()`. Все рассматривавшиеся раньше операции обмена на сокетах были блокирующими. Эта модель ввода-вывода не нуждается в детальных комментариях. А все прочие перечисленные модели реализации операций ввода вывода последовательно рассмотрены далее.

Примеры реализации

Далее будут рассматриваться все оставшиеся (помимо блокируемых операций) модели ввода-вывода. Все примеры, относящиеся к расширенным моделям обмена находятся в каталоге примеров `ufd`. Большинство из примеров заимствованных из книги У. Р. Стивенс [3], но претерпели некоторые изменения в связи с прошедшим временем со времени написания книги (больше 25 лет).

Все подкаталоги этого архива содержат образцы и клиентов и серверов, демонстрирующих рассматриваемую модель. Сервера представляют собой ретрансляторы строк получаемых от клиента (эхо-сервера) на порт (по умолчанию) 9877, определяемый в заголовочном файле `unp.h`:

```
/* Define some port number that can be used for client-servers */
#define SERV_PORT      9877                /* TCP and UDP client-servers */
#define SERV_PORT_STR  "9877"             /* TCP and UDP client-servers */
```

Такой порт выбран автором примеров произвольно (из области приватных портов), и может быть изменён, при желании, на любой другой приемлемый.

Неблокируемый ввод-вывод

При неблокируемом вводе-выводе не ожидает обязательно наличия данных (или возможности вывода) — результат выполнения операции, либо невозможность её выполнения в данный момент определяется по анализу кода возврата. Перевод сокета в режим неблокирующего чтения выполняется вызовом `fcntl()` с соответствующими параметрами (дескриптор всегда создаётся вызовами `open()`, `socket()`, `accept()` в **блокируемом** режиме операций).

Схематично обработка происходит следующим образом:

```
int fdi = open(...);                // открыли: фаловый дескриптор, сокет,
pipe, ...
int cur_flg = fcntl(fdi, F_GETFL);   // чтение должно быть в режиме O_NONBLOCK
if (-1 == fcntl(fdi, F_SETFL, cur_flg | O_NONBLOCK))
    error(...), exit(EXIT_FAILURE);
...
while (1) {
    int n = read(fdi, buf, buflen);
    if (n > 0) {
        ...                          // считаны данные ... обработка
    }
    else if (0 == n)                  // EOF — конец данных
        break;
    else if (-1 == n) {
        if (EAGAIN == errno) {        // данные не готовы
            printf("not ready!\n");
            usleep(300);
            continue;                 // после паузы повторить
        }
        else
            error(...), exit(EXIT_FAILURE); // невозстановимая ошибка
    }
}
close(fdi);
```

Замечания к примерам

Большая подборка примеров, относящихся к неблокирующему вводу-выводу применительно к сетевым сокетах находится в каталог `ufd/nonblock`.

На одном из хостов LAN запускаем ретранслирующий сервер:

```
$ ./tcpservselect03
```

```
listening socket readable
^C
```

На другом хосте LAN выполняем программу клиента, который отправляет строки, полученные из входного потока (которым может быть, например, и файл), серверу, и затем воспроизводит ответ, ретранслированный сервером:

```
$ ./tcpcli01 192.168.1.5
12345
12345
qwer
qwer
www
www
^D
Завершено
```

Всё, относящееся к неблокируемому вводу, реализовано в исходном файле `strclinonb.c`. В файле `strclifork.c` приведена ещё одна, альтернативная, более простая реализация клиента. За подробными объяснениями кода, если он непонятен, следует обратиться к [3].

Особенно полезно наблюдать работу подобных клиент-серверных приложений, анализируя параллельно дампы сетевого трафика, захватываемый сетевым сниффером `tcpdump`. Это можно делать как на хосте сервера, так и клиента:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
group default qlen 1000
    link/ether a0:1d:48:f4:93:5c brd ff:ff:ff:ff:ff:ff
3: wlo1: <NO-CARRIER,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
DORMANT mode DORMANT group default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff

$ sudo tcpdump -i em1 tcp and port 9877
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on em1, link-type EN10MB (Ethernet), capture size 65535 bytes
12:56:23.568017 IP modules.52421 > notebook.9877: Flags [S], seq 269087598, win 29200,
options [mss 1460,sackOK,TS val 19530115 ecr 0,nop,wscale 7], length 0
12:56:23.568292 IP notebook.9877 > modules.52421: Flags [S.], seq 894792783, ack
269087599, win 28960, options [mss 1460,sackOK,TS val 19525763 ecr 19530115,nop,wscale 7],
length 0
12:56:23.568355 IP modules.52421 > notebook.9877: Flags [.], ack 1, win 229, options
[nop,nop,TS val 19530115 ecr 19525763], length 0
12:56:27.878584 IP modules.52421 > notebook.9877: Flags [P.], seq 1:7, ack 1, win 229,
options [nop,nop,TS val 19534425 ecr 19525763], length 6
12:56:27.878842 IP notebook.9877 > modules.52421: Flags [.], ack 7, win 227, options
[nop,nop,TS val 19530074 ecr 19534425], length 0
12:56:28.569058 IP notebook.9877 > modules.52421: Flags [P.], seq 1:7, ack 7, win 227,
options [nop,nop,TS val 19530764 ecr 19534425], length 6
12:56:28.569157 IP modules.52421 > notebook.9877: Flags [.], ack 7, win 229, options
[nop,nop,TS val 19535116 ecr 19530764], length 0
12:56:38.990624 IP modules.52421 > notebook.9877: Flags [P.], seq 7:12, ack 7, win 229,
options [nop,nop,TS val 19545537 ecr 19530764], length 5
12:56:38.990859 IP notebook.9877 > modules.52421: Flags [.], ack 12, win 227, options
[nop,nop,TS val 19541186 ecr 19545537], length 0
12:56:38.991150 IP notebook.9877 > modules.52421: Flags [P.], seq 7:12, ack 12, win 227,
options [nop,nop,TS val 19541186 ecr 19545537], length 5
12:56:38.991208 IP modules.52421 > notebook.9877: Flags [.], ack 12, win 229, options
[nop,nop,TS val 19545538 ecr 19541186], length 0
12:56:42.414878 IP modules.52421 > notebook.9877: Flags [P.], seq 12:16, ack 12, win 229,
options [nop,nop,TS val 19548962 ecr 19541186], length 4
```

```

12:56:42.415136 IP notebook.9877 > modules.52421: Flags [P.], seq 12:16, ack 16, win 227,
options [nop,nop,TS val 19544610 ecr 19548962], length 4
12:56:42.415189 IP modules.52421 > notebook.9877: Flags [.], ack 16, win 229, options
[nop,nop,TS val 19548962 ecr 19544610], length 0
12:56:44.014763 IP modules.52421 > notebook.9877: Flags [F.], seq 16, ack 16, win 229,
options [nop,nop,TS val 19550562 ecr 19544610], length 0
12:56:44.015015 IP notebook.9877 > modules.52421: Flags [F.], seq 16, ack 17, win 227,
options [nop,nop,TS val 19546210 ecr 19550562], length 0
12:56:44.015073 IP modules.52421 > notebook.9877: Flags [.], ack 17, win 229, options
[nop,nop,TS val 19550562 ecr 19546210], length 0
^C
17 packets captured
17 packets received by filter
0 packets dropped by kernel

```

Обратите внимание на соответствие длин передаваемых строк (в терминале ввода) и на указание длины передаваемых данных в пакетах, перехватываемых снифером.

Конечно, можно изучать выполнение клиента и локально, на том же хосте, на котором выполняется и сервер:

```

$ ./tcpcli01 127.255.255.254
это петлевой интерфейс
это петлевой интерфейс
^D
Завершено

```

Однако, такое изучение поведения предлагаемых примеров может быть менее информативно.

Мультиплексирование ввода-вывода

Один из самых старых API POSIX:

```

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

```

И его более поздний эквивалент:

```

int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            const struct timespec *timeout, sigset_t *sigmask);

```

Различия:

- `select()` использует тайм-аут в виде `struct timeval` (с секундами и микросекундами), а `pselect()` использует `struct timespec` (с секундами и наносекундами);
- `select()` может обновить параметр `timeout`, чтобы сообщить, сколько времени осталось. Функция `pselect()` не изменяет этот параметр;
- `select()` не содержит параметра `sigmask`, и ведет себя как `pselect()` с параметром `sigmask`, равным `NULL`. Если этот параметр `pselect()` не равен `NULL`, то `pselect()` сначала замещает текущую маску сигналов на ту, на которую указывает `sigmask`, затем выполняет `select()`, после чего восстанавливает исходную маску сигналов.

Параметр тайм-аута может задаваться несколькими способами:

- `NULL`, что означает ожидать вечно;
- ожидать инициированное структурой значение времени;
- не ожидать вообще (программный опрос — `polling`), когда структура инициализируется значением `{0, 0}`.

Функции возвращают значение больше нуля — число готовых к операции дескрипторов, ноль — в случае истечения тайм-аута, и отрицательное значение при ошибке.

Примечание: Готовность дескриптора функция `select()` возвращает по поступлению на дескриптор **первого** доступного байта. Если вы ожидаете **блок** данных некоторого размера, то функция чтения,

непосредственно следующая за `select()`, может вернуть число считанных байт **меньше**, чем вы можете ожидать.

Вводится понятие набора дескрипторов, и макросы для работы с набором дескрипторов:

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

С готовностью дескрипторов чтения и записи `readfds`, `writfds` — относительно ясно интуитивно. Очень важно, что вариантом срабатывания исключительной ситуации `exceptfds` на дескрипторе сетевых сокетов — является получение внеполосовых (**приоритетных**) данных TCP, что очень широко используется в реализациях (конечных автоматов) сетевых протоколов (например SIP, VoIP сигнализаций PRI, SS7 — на линиях E1/T1, E3/T3, ...).

Примечание: Большинство UNIX систем имеют определение численной константы с именем `FD_SETSIZE` — максимального размера набора дескрипторов, но её численное значение сильно зависит от констант периода компиляции совместимости с стандартами (такими, например, как `__USE_XOPEN2K`, ...).

Ещё один вариант мультиплексирования ввода-вывода — функция `poll()`. Представление набора дескрипторов заменено на массив структур вида:

```
struct pollfd {
    int fd;           /* файловый дескриптор */
    short events;     /* запрошенные события */
    short revents;    /* возвращенные события */
};
```

Здесь: `fd` — открытый файловый дескриптор, `events` — набор битовых флагов запрошенных событий для этого дескриптора, `revents` — набор битовых флагов возвращенных событий для этого дескриптора (из числа запрошенных, или `POLLERR`, `POLLHUP`, `POLLNVAL`). Часть возможных битов, описаны в `<sys/poll.h>`:

```
#define POLLIN      0x0001 /* Можно читать данные */
#define POLLPRI     0x0002 /* Есть срочные данные */
#define POLLOUT     0x0004 /* Запись не будет блокирована */
#define POLLERR     0x0008 /* Произошла ошибка */
#define POLLHUP     0x0010 /* Разрыв соединения */
#define POLLNVAL    0x0020 /* Неверный запрос: fd не открыт */
```

Ещё некоторая часть относящихся констант описаны в `<asm/poll.h>`: `POLLRDNORM`, `POLLRDBAND`, `POLLWRNORM`, `POLLWRBAND` и `POLLMSG`.

Сам вызов оперирует с массивом таких структур, по одному элементу на каждый интересующий дескриптор:

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfd, int timeout);
```

Здесь: `ufds` - сам массив структур, `nfd` - его размерность, `timeout` - тайм-аут в миллисекундах (ожидание при положительном значении, немедленный возврат при нулевом, бесконечное ожидание при значении, заданном специальной константой `INFTIM`, которая определена просто как отрицательное значение).

Пример того, как используются (и работают) вызовы `select()` и `poll()` - позаимствованы из [3] (каталог `ufd`), оригиналы кодов У. Стивенса несколько изменены (оригиналы относятся к 1998г. и проверялись на совершенно других UNIX того периода). Примеры достаточно объёмные (это полные версии программ TCP клиентов и серверов), поэтому ниже показаны только фрагменты примеров, непосредственно относящиеся к вызовам `select()` и `poll()`, а также примеры того, что реально эти примеры выполняются и как это происходит (вызовы

функций в коде показаны как у У. Стивенса — с большой буквы, вызов этот — это полный аналог соответствующего вызова API, но обрамлённый выводом сообщения о роде ошибки, если она возникнет):

tcpservselect01.c (TCP ретранслирующий сервер на select()):

```
...
int                nready, client[FD_SETSIZE];
fd_set            rset, allset;
socklen_t         cliilen;
struct sockaddr_in cliaddr, servaddr;
...
listenfd = Socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(SERV_PORT);
Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
Listen(listenfd, LISTENQ);
maxfd = listenfd;                /* initialize */
maxi = -1;                       /* index into client[] array */
for (i = 0; i < FD_SETSIZE; i++)
    client[i] = -1;              /* -1 indicates available entry */
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
for (;;) {
    rset = allset;               /* structure assignment */
    nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);
    if (FD_ISSET(listenfd, &rset)) { /* new client connection */
        connfd = Accept(listenfd, (SA*)&cliaddr, &cliilen);
    }
}
...
```

tcpservpoll01.c (TCP ретранслирующий сервер на poll()):

```
...
struct pollfd      client[OPEN_MAX];
struct sockaddr_in cliaddr, servaddr;
...
listenfd = Socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(SERV_PORT);
Bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
Listen(listenfd, LISTENQ);
client[0].fd = listenfd;
client[0].events = POLLRDNORM;
for (i = 1; i < OPEN_MAX; i++)
    client[i].fd = -1;           /* -1 indicates available entry */
maxi = 0;                       /* max index into client[] array */
for (;;) {
    nready = Poll(client, maxi + 1, INFTIM);
    if (client[0].revents & POLLRDNORM) { /* new client connection */
        for (i = 1; i < OPEN_MAX; i++)
            if (client[i].fd < 0) {
                client[i].fd = connfd; /* save descriptor */
                break;
            }
    }
}
...
    client[i].events = POLLRDNORM;
```



```

        if (i > maxi)
            maxi = i;
    ...

```

Замечания к примерам

Как выполнять эти примеры и на что обратить внимание? Запускаем выбранный нами сервер (позже мы остановим его по Ctrl+C), все сервера этого раздела прослушивают фиксированный порт 9877, и являются для клиента ретрансляторами данных, получаемых на этот порт:

```

$ ./tcpservselect01
...
^C
или
$ ./tcpservpoll01
...
^C

```

В том, что сервер прослушивает порт и готов к работе, убеждаемся, например, так:

```

$ netstat -a | grep :9877
tcp        0      0 *:9877          *:9877          LISTEN

```

К серверу подключаемся клиентом (из того же архива примеров), и вводим строки, которые будут передаваться на сервер и ретранслироваться обратно:

```

$ ./tcpcli01 192.168.1.5
1 строка
1 строка
2 строка
2 строка
последняя
последняя
^C

```

Указание IP адреса сервера (не имени!) в качестве параметра запуска клиента — обязательно. Клиентов одновременно может быть много — сервера параллельные. Во время выполнения клиента можно увидеть состояние сокетов — клиентского и серверных, прослушивающего и присоединённого (клиент не закрывает соединение после обслуживания каждого запроса, как, например, сервер HTTP):

```

$ netstat -a | grep :9877
tcp        0      0 *:9877          *:9877          LISTEN
tcp        0      0 localhost:46783 localhost:9877   ESTABLISHED
tcp        0      0 localhost:9877  localhost:46783 ESTABLISHED

```

Ввод-вывод управляемый сигналом

В этом случае на сетевом сокете включается режим управляемого сигналом ввода-вывода, и устанавливается обработчик сигнала при помощи `sigaction()`. Когда UDP дейтаграмма готова для чтения, генерируется сигнал SIGIO. Обработать данные можно в обработчике сигнала вызовом `recvfrom()`. Пример того, как это работает, заимствован из [3], и находится в каталоге архива `ufd/sigio`, он слишком громоздкий для детального обсуждения, но может быть изучен и в коде и в работе. Краткая сводка о запуске примера:

Запуск ретранслирующего сервера UDP (в конце выполнения останавливаем его по Ctrl+C):

```

$ ./udpserv01
^C

```

Убедиться, что сервер готов и прослушивает порт, можно так:

```
$ netstat -a | grep :9877
udp          0          0  *:9877      *:*
```

Запуск клиента:

```
$ ./udpccli01 192.168.1.5
qweqert
qweqert
134534256
134534256
^D
```

Примечание: Особо интересен запуск (например из скрипта) нескольких одновременно (6) клиентов, которые плотным потоком шлют серверу на ретрансляцию большое число строк (у У. Стивенса — 3645 строк). После этого серверу можно послать сигнал **SIGHUP**, по которому он выведет гистограмму, которая складывалась по числу одновременно читаемых дейтаграмм:

```
$ ps -A | grep udp
2692 pts/12   00:00:00 udpserve01
$ kill -HUP 2692
$ ./udpserve01
cntread[0] = 0
cntread[1] = 8
cntread[2] = 0
cntread[3] = 0
cntread[4] = 0
cntread[5] = 0
cntread[6] = 0
cntread[7] = 0
cntread[8] = 0
^C
```

Асинхронный ввод-вывод

Асинхронный ввод-вывод добавлен только в редакции стандарта POSIX.1g (1993 г., одно из расширений реального времени). В вызове `aio_read()` даётся указание ядру начать операцию ввода-вывода, и указывается, каким сигналом уведомить процесс о завершении операции (включая копирование данных в пользовательский буфер). Вызывающий процесс не блокируется. Результат операции (например, полученная UDP дейтаграмма) может быть обработан, например, в обработчике сигнала. Разница с предыдущей моделью, управляемой сигналом, состоит в том, что в той модели сигнал уведомлял о возможности начала операции (вызове операции чтения), а в асинхронной модели сигнал уведомляет уже о завершении операции чтения в буфер пользователя.

Всё, что относится к асинхронному вводу-выводу в Linux описано в `<aio.h>`. Управляющий блок асинхронного ввода-вывода — видны все поля, которые обсуждались выше:

```
struct aiocb {
    /* Asynchronous I/O control block. */
    int aio_fildes; /* File descriptor. */
    int aio_lio_opcode; /* Operation to be performed. */
    int aio_reqprio; /* Request priority offset. */
    volatile void *aio_buf; /* Location of buffer. */
    size_t aio_nbytes; /* Length of transfer. */
    struct sigevent aio_sigevent; /* Signal number and value. */
    ...
}
```

Того же назначения блок для 64-битных операций:

```
struct aiocb64 {
    ...
}
```

И некоторые операции (в качестве примера):

```
int aio_read(struct aiocb *__aiocbp);
int aio_write(struct aiocb *__aiocbp);
```

Может быть инициализировано выполнение целой **цепочки** асинхронных операций (длиной `__nent`):

```
int lio_listio(int __mode,
               struct aiocb* const list[__restrict_arr],
               int __nent, struct sigevent *__restrict __sig);
```

Как и для потоков `pthread_t`, асинхронные операции значительно легче породить, чем позже остановить... для чего также потребовался отдельный API:

```
int aio_cancel(int __fildes, struct aiocb *__aiocbp);
```

Можно предположить, что каждая асинхронная операция выполняется как отдельный поток, у которого не циклическая функция потока.

Символьный сокет¹³

Символьные сокет (raw sockets) используются для обеспечения обмена на уровне протоколов, которые не поддерживаются транспортным уровнем (TCP, UDP, SCTP, ...). Средствами такого сокета можно работать с протоколами ICMP, IGMP, или даже организовывать обмен IPv4 датаграммами с собственным полем протокола IPv4, которое не обрабатывается ядром Linux (8-битовое поле пакета, характерные значения которого 1 — ICMP, 2 — IGMP, 6 — TCP, 17 — UDP: константы из `<netinet/in.h>` вида `IPPROTO_*`, которые мы уже встречали раньше). С помощью символьного сокета вообще можно построить свой собственный заголовок IPv4 при помощи параметра сокета `IP_HDRINCL`:

```
...
int on = 1, fd = socket(AF_INET, SOCK_RAW, 0);
setsockopt(fd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
...
```

Только **привилегированный** пользователь (`root`) может создавать символьные сокеты. Сокет создаётся вызовом такого типа:

```
int fd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

Третьим параметром указано протокол сетевого уровня, константы которых указаны в `<netinet/in.h>` (выше уже показывался их полный перечень).

Вывод в символьный сокет может производиться всё теми же вызовами `sendto()`, `sendmsg()`, или `write()`, `writenv()` и `send()`, если сокет уже присоединён.

Ввод из символьного сокета производится теми же, уже рассмотренными, функциями чтения из сокета. Но есть достаточно много исключений и особенностей в том, какие IP датаграммы и как передаются символьному сокету, например:

- Пакеты UDP и TCP **никогда** не передаются на символьный сокет;
- Все IGMP пакеты и большинство ICMP пакетов передаются на символьный пакет только **после того**, как ядро заканчивает обработку этих сообщений;
- Все IP датаграммы с таким значением поля протокола, которое не понимается ядром, передаются на символьный сокет;

Подобных особенностей довольно много и они, порой, могут привести в замешательство.

¹³ Ещё для символьного сокета разные авторы используют названия неструктурированный сокет и сырой сокет.

Детально работа с символьными сокетами описана в [3, 4], где приведены коды иллюстрационных приложений для программ ping и traceroute, работающих с символьными сокетами.

Канальный уровень

Возможен даже доступ к пакетам канального (MAC) уровня, но это уже совершенно возможность Linux, не предусмотренная какими-либо стандартами и не представленная в других системах UNIX — сокет :

```
int fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL));
```

В результате такого вызова будут из такого сокета возвращаться кадры для всех протоколов, получаемых канальным уровнем.

Третьим параметром вызова должна быть не нулевая константа, задающая тип кадра Ethernet, который будет отбираться фильтром. Если нужны все кадры IPv4, определяем сокет так:

```
int fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_IP));
```

Могут быть полезными в качестве последнего параметра такие константы как: ETH_P_ARP, ETH_P_IPV6.

Другим, более универсальным, средством перехвата и фильтрации пакетов канального уровня является свободно доступная библиотека BPF (BSD Packet Filter) — libpcap:

```
$ aptitude search libpcap
p  libpcap-dev                - библиотека разработчика для libpcap (переходный пакет).
p  libpcap-dev:i386           - библиотека разработчика для libpcap (переходный пакет).
i  libpcap0.8                 - системный интерфейс для захвата пакетов из пользовательского
p  libpcap0.8:i386            - системный интерфейс для захвата пакетов из пользовательского
p  libpcap0.8-dev             - development library and header files for libpcap0.8
p  libpcap0.8-dev:i386        - development library and header files for libpcap0.8
$ ls -ll /lib/x86_64-linux-gnu/libpcap*
-rw-r--r-- 1 root root 310592 map 24 2022 /lib/x86_64-linux-gnu/libpcap.so.0.8
-rw-r--r-- 1 root root 310592 map 24 2022 /lib/x86_64-linux-gnu/libpcap.so.1.10.1
```

Библиотека libpcap присутствует практически во всех POSIX операционных системах. На ней работает такой известный сетевой снифер как tcpdump. Библиотека реализует весьма обширный API таких вызовов как pcap_open_live(), pcap_open_live(), pcap_compile(), pcap_setfilter(), pcap_datalink() и др.

Для более детального изложения доступа к кадрам канального уровня можно обратиться к книгам [3, 4].

Современные языки программирования

Весь предыдущий программный код ранее показывался на языке C, или, временами, на C++, но только в части упрощения представления кода, при использовании всё того же POSIX API языка C. Код клиент-серверных иллюстраций (в начале главы) заимствован из книг У. Р. Стивенса, написанных в в 90-х годах XX века, и только выверен и трансформирован в части совместимости с нюансами современного Linux. Код относительно классов серверного обслуживания, далее, публиковался автором в отдельных публикациях 2000-х, и относится к этому периоду. Все эти коды сознательно не модифицировались, чтобы лишний раз сделать акцент на том, что абстракция низкоуровневого сетевого программирования сокетов (а именно нижним уровнем мы и занимаемся) является настолько фундаментальной и оказалась настолько успешной, что в неизменном виде используется на протяжении 30 лет.

Все более поздние языки программирования высокого уровня, естественно, предоставляют все механизмы работы с этими фундаментальными абстракциями клиент-серверной

парадигмы. Обеспечивается это подключаемыми компонентами каждой языковой системы — в каждой из них это может называться по-разному: библиотеки, пакеты, модули, крейты... И делается это с определённым изяществом, упрощающим низкоуровневое сокетное программирование. Обратите внимание, с какой лаконичностью и как просто описывается эта низкоуровневая модель в сравнении с классикой C/C++.

Ещё одной особенностью всех современных (относительно новых) языков программирования (в отличие от классики) есть то, что они по умолчанию работают с текстовой информацией в представлении UNICODE в кодировке UTF-8. А это значит что и их сетевые компоненты исходно мультилингвистические, и допускают в своём трафике любые языки мира без каких-либо дополнительных ухищрений. Но это, тем не менее, всё должно проверяться каждый раз в ваших приложениях.

И, наконец, ещё одной (и важнейшей) характерной особенностью всех современных языковых сред высокого уровня является то, что помимо подключаемых компонент для обеспечения низкоуровневого сокетного программирования TCP/UDP, для их сред предоставляются, параллельно, и другие компоненты, для простой и быстрой разработки приложений, работающих на верхних этажах протоколов прикладных уровней (HTTP, HTTPS, SIP ...), а иногда и целый букет таких альтернативных компонент, на выбор. А это позволяет не заниматься вручную прописыванием деталей протоколов этих уровней, а удачно выбрать уже готовую реализацию из предоставляемых.

Ниже приводятся несколько кратких иллюстраций из области клиент-серверных технологий на современных языках программирования высокого уровня (всё это находится в подкаталогах каталога `new_lang` в архиве примеров)... Все TCP приложения, на разных языках, по работе построены внешне совершенно единообразно (с приведением к единообразности пришлось повозиться — механизмы языков существенно разные):

- Сервер при старте сообщением-приглашением сообщает порт, который он собирается обслуживать (примеров много, и все они должны различаться портом TCP, тут не сложно запутаться);
- Клиент выводит приглашение на ввод с терминала строки, которую он намеревается транслировать серверу;
- Полученное от сервера эхо клиент выводит на терминал для сверки с исходной строкой;
- Работа клиента завершается (так чтобы сокет корректно закрылся с его стороны) одним из 3-х вариантов ввода в качестве исходной строки: 1). ввод Ctrl+C, 2). ввод Ctrl+D (конец потока, EOF в терминологии POSIX), 3). просто ввод пустой строки нажатием Enter.

Языки программирования для последующей иллюстрации выбирались не случайным образом, здесь представлены только те языки, которые, по мнению автора, имеют существенные преимущества (каждый свои) для создания сетевых проектов. Поэтому я возьму на себя смелость предварить каждый язык коротким упоминанием, по субъективному мнению, именно его преимуществ.

Python

Python — это достаточно старый язык: Python 2.0 (линия версий), выпущен в октябре 2000 года. Долгое время в Linux сосуществовали (из соображений переносимости системных утилит) две независимых линии версий: 2.X и 3.X (которые синтаксически не совместимые). И только к ноябрю 2022 года все основные дистрибутивы Linux завершили окончательный переход к линейке версий 3.X как умалчиваемой. Полный переход с Python 2 на Python затянулся на 14 лет (с 2008 по 2022).

С этого времени интерес для вас должен представлять только Python версий 3.X:

```
$ python -v
Python 3.10.12
```

В августе 2022 года Python занял первое место в рейтинге языков программирования от TIOBE Software.

Выбор Python как инструмента разработки для сетевых проектов будет иметь несколько главных преимуществ:

1. Огромное число разнообразных пакетов, наработанных за 20 лет, для любой целевой потребности. Очень многие вещи вам не придётся писать вручную;
2. Скорость разработки: Python называют языком быстрой разработки. Опубликованы материалы, что время разработки проекта (или прототипа проекта) на Python от 4-х до 10-ти раз короче, чем аналогичная разработка на C/C++;
3. Код Python не требует компиляции, и может начать выполняться немедленно. Для небольших проектов (утилит) это может оказаться большим преимуществом.

Теперь вернёмся к реализации TCP эхо-сервера и его клиента...

Последовательный TCP эхо-сервер:

echo_serv.py :

```
#!/usr/bin/python3
import socket

HOST = "127.0.0.1"          # loopback interface address (localhost)
PORT = 52500                # port to listen with Python

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    try:
        s.bind((HOST, PORT))
    except OSError:
        print("address already in use")
        quit()
    s.listen()
    while True:
        print('waiting on port', PORT)
        try:
            conn, addr = s.accept()
        except KeyboardInterrupt:
            print()
            quit()
        with conn:
            while True:
                data = conn.recv(1024)
                if not data:          # клиент закрыл сокет
                    break
                conn.sendall(data)
            conn.close()
```

Симметричный к нему клиент:

echo_cli.py :

```
#!/usr/bin/python3
import socket
import sys

HOST = "127.0.0.1" # The server's hostname or IP address
PORT = 52500       # the port used by the Python server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    try:
        # пробуем присоединиться
```

```

s.connect((HOST, PORT))
except ConnectionRefusedError:
    print("no server available")
    quit()
while True:
    sys.stdout.write("> ")
    sys.stdout.flush()
    try:
        str = sys.stdin.readline()
    except KeyboardInterrupt:      # ^C - завершение работы
        print("\r")
        break
    if str == '':                  # ^D - завершение работы
        sys.stdout.write('\r')
        break
    if "\n" == str:               # Enter - завершение работы
        break
    s.sendall(bytes(str, "UTF-8"))
    data = s.recv(1024)
    print(" ", data.decode("UTF-8"), end='')

```

Завершается (корректно) работа клиента несколькими способами: Ctrl+C, Ctrl+D (конец потока ввода в понятиях UNIX), просто ввод пустой строки Enter.

Большая часть этого кода занимается обработкой некоторых ошибок (не всех), чтобы предотвратить неприятные сообщения Python при ошибочных действиях пользователя с этими приложениями. Например, при попытке запуска 2-го экземпляра сервера:

```

$ ./echo_serv.py
address already in use

```

Или при запуске клиента когда сервер не запущен:

```

$ ./echo_cli.py
no server available

```

Работает клиент-серверная пара так:

```

$ ./echo_cli.py
> 123 456
    123 456
> русская строка
    русская строка

```

При этом сервер фиксирует коротким сообщением подключение каждого нового клиента (сервер последовательный):

```

$ ./echo_serv.py
waiting on port 52500
waiting on port 52500
waiting on port 52500
waiting on port 52500
...

```

Как уже обсуждалось ранее, можно и нужно, помимо собственных, для тестирования (совместимости, как минимум) использовать независимые CLI утилиты системы:

```

$ nc 127.0.0.1 52500
123 4567
123 4567
русская строка
русская строка
^C

```

```
$ socat tcp4:127.0.0.1:52500 STDOUT
```

```
123 456 7
```

```
123 456 7
```

```
русская строка
```

```
русская строка
```

```
^C
```

```
$ telnet 127.0.0.1 52500
```

```
Trying 127.0.0.1...
```

```
Connected to 127.0.0.1.
```

```
Escape character is '^['.
```

```
12 34 56
```

```
12 34 56
```

```
строка
```

```
строка
```

```
строка 2
```

```
строка 2
```

```
^]
```

```
telnet> quit
```

```
Connection closed.
```

И к вопросу мультязычности реализации канала, и её проверки:

```
$ ./echo_cli.py
```

```
> नमस् ॐ
```

```
नमस् ॐ
```

```
> Dobrý den
```

```
Dobrý den
```

```
> لسلام عليكم
```

```
لسلام عليكم
```

```
> . ᱫᱽᱨᱟᱹᱭ
```

```
. ᱫᱽᱨᱟᱹᱭ
```

```
> こんにちは
```

```
こんにちは
```

```
> 今日は
```

```
今日は
```

```
> 안녕하세요
```

```
안녕하세요
```

```
> 你好
```

```
你好
```

Код TCP сокетного программирования совершенно другой (в сравнении с классикой C/C++), но вся логика работы с сокетами остаётся неизменной.

Python и HTTP¹⁴

Говоря о сетевом программировании и о Python, в частности, просто нельзя не упомянуть о роли Python в разработке WEB проектов, и тех возможностях которые здесь предоставляются. За многие годы развития Python (сначала версии 2, а затем и 3) в его репозиториях пакетов накопилось несколько десятков альтернативных (кому что больше нравится) пакетов для работы с протоколом прикладного уровня HTTP, формирования и получения запросов, работы с основным форматом WEB JSON и т. д.

Не столько для конкретизации и обсуждения, а сколько для обсуждения простоты и лаконичности готовых к использованию инструментов, предоставляемых Python разработчику, рассмотрим пример только одного¹⁵ из таких свободно доступных пакетов для написания разнообразных HTTP-клиентов — пакет Requests:

¹⁴ Эта часть подготовлена по итогам замечательной (по охвату при её компактности, и её новизне) публикации:

«The best Python HTTP clients for 2022», ссылка на которую приведена в конце текста.

¹⁵ В архиве примеров кодов к книге приведено использование этого, и ещё 7 разнородных пакетов Python для аналогичных целей.

requests.get.py :

```
$ cat requests.get.py
import requests

r = requests.get('https://swapi.dev/api/starships/9/')
print(r.json())
```

Это программа в 2 строки, которая запрашивает информацию (запрос GET протокола HTTP), подобным образом, с любого удалённого ресурса Интернет (начиная от разрешения его URL имени в его IP ...), и далее производящая разбор полученного ответа в JSON формат. При ручном программировании подобное может вылиться в страницу-другую убоистого кода...

И как это работает:

```
$ python3 requests.get.py
{'name': 'Death Star', 'model': 'DS-1 Orbital Battle Station',
 'manufacturer': 'Imperial Department of Military Research, Sienar Fleet Systems',
 'cost_in_credits': '1000000000000', 'length': '120000', 'max_atmosphering_speed': 'n/a',
 'crew': '342,953', 'passengers': '843,342', 'cargo_capacity': '1000000000000',
 'consumables': '3 years', 'hyperdrive_rating': '4.0', 'MGLT': '10',
 'starship_class': 'Deep Space Mobile Battlestation', 'pilots': [],
 'films': ['https://swapi.dev/api/films/1/'], 'created': '2014-12-10T16:36:50.509000Z',
 'edited': '2014-12-20T21:26:24.783000Z', 'url': 'https://swapi.dev/api/starships/9/'}
```

А следующая иллюстрация (рука на подымается называть всё это примерами) показывает отработку запроса POST протокола HTTP, позволяющего передать свой блок информации (data — и всё тот же JSON формат¹⁶) удалённому ресурсу Интернет, и получить на неё обратно ответ:

requests.get.py :

```
$ cat requests.post.py
import requests

data = {
    "name": "Obi-Wan Kenobi",
    "height": "182",
    "mass": "77",
    "hair_color": "auburn, white",
    "skin_color": "fair",
    "eye_color": "blue-gray",
    "birth_year": "57BBY",
    "gender": "male"
}

r = requests.post('https://httpbin.org/post', json=data)
print(r.json())
```

Вот, собственно, и всё. И вот ответ удалённого ресурса:

```
$ python3 requests.post.py
{'args': {}, 'data': '{"name": "Obi-Wan Kenobi", "height": "182", "mass": "77",
 "hair_color": "auburn, white", "skin_color": "fair", "eye_color": "blue-gray",
 "birth_year": "57BBY", "gender": "male"}', 'files': {}, 'form': {},
 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate, br',
 'Content-Length': '177', 'Content-Type': 'application/json', 'Host': 'httpbin.org',
```

¹⁶ В запросе POST это может быть всё что угодно, вплоть до бинарного исполнимого кода, который предлагается исполнить на удалённом Интернет ресурсе. В запросе GET тоже, вообще-то говоря, можно передать некоторую уточняющую информацию к запросу, но в этом случае: а) она исключительно текстовая, б) достаточно ограниченного объёма и в) дописывается непосредственно к строке самого запроса.

```
'User-Agent': 'python-requests/2.28.2',
'X-Amzn-Trace-Id': 'Root=1-6586f104-66289eff3a49a66d5f792948'},
'json': {'birth_year': '57BBY', 'eye_color': 'blue-gray', 'gender': 'male',
'hair_color': 'auburn, white', 'height': '182', 'mass': '77', 'name': 'Obi-wan Kenobi',
'skin_color': 'fair'}, 'origin': '45.9.31.32', 'url': 'https://httpbin.org/post'}
```

Go

Всё ещё интереснее в языке Go... При использовании Go вы получаете 2 самых существенных преимущества:

1. Go предлагает, достаточно уникальную, модель организации лёгких исполняющихся параллельно ветвей (горутин, сопрограмм). Эта модель допускает создание сотен, или даже тысяч, параллельно выполняющихся ветвей. И выполняющихся с высокой эффективностью. Это путь к созданию высоко нагруженных приложений! И реализуются эти механизмы в Go с необычайной лёгкостью (что мы увидим вскоре). Это особенно актуально на многопроцессорных и многоядерных архитектурах, на которых Go может на 100% использовать возможности десятков процессоров.
2. Код на Go обладает высокой надёжностью, в смысле возникновения неожиданных сбоев в работе программ. Главным фактором здесь является то, что в Go реализована работа с памятью со сборкой мусора. И вы не можете динамически создавать и уничтожать объекты в памяти по своему усмотрению — этим занимается исполняющая система Go. А львиная доля ошибок в программном коде — это ошибки в работе с памятью.

Сделаем такой же, как ранее, последовательный TCP эхо-сервер:

echo_serv.go :

```
package main
import "net"
import "fmt"
import "bufio"

func main() {
    ln, _ := net.Listen("tcp", ":51500") // Устанавливаем прослушивание порта
    for {
        fmt.Println("waiting on port 51500")
        conn, _ := ln.Accept()          // Открываем серверный порт
        for {                             // Запускаем цикл прослушивания
            // Будем прослушивать все сообщения разделенные \n
            message, err := bufio.NewReader(conn).ReadString('\n')
            if err != nil {                // Клиент разорвал коннект
                conn.Close()
                break
            }
            conn.Write([]byte(message))    // Отправить обратно клиенту
        }
    }
}
```

Симметричный к нему клиент:

echo_cli.go :

```
package main
import "net"
import "fmt"
import "bufio"
import "os"
import "io"
```

```

import "strconv"

func main() {
    port := "51500"
    if len(os.Args) > 1 {
        _, err := strconv.Atoi(os.Args[1])
        if err != nil {
            fmt.Print("Ошибочный порт")
            return
        } else {
            port = os.Args[1]
        }
    }
    fmt.Println("connect to port " + port)
    conn, err := net.Dial("tcp", "127.0.0.1:" + port) // Подключаемся к сокету
    if err != nil {
        fmt.Println("no server available"); return
    }
    for {
        reader := bufio.NewReader(os.Stdin) // Чтение данных из stdin
        fmt.Print("> ")
        text, err := reader.ReadString('\n')
        if err != nil {
            if err != io.EOF { fmt.Print(err) }
            fmt.Println()
            return
        }
        if len(text) == 1 { break }
        fmt.Fprintf(conn, text + "\n") // Отправляем в socket
        message, _ := bufio.NewReader(conn).ReadString('\n') // Чтение ответа
        fmt.Print(" " + message)
    }
}

```

В отличие от предыдущего, этот клиент может, в качестве параметра командной строки, получить номер TCP порта, чтобы позволить ему работать с разными TCP-серверами.

Компиляция приложений традиционная:

```

$ go build echo-cli.go
$ go build echo-serv.go
$ ls -l | grep ^-rwx
-rwxrwxr-x 1 olej olej 2747626 дек 16 15:29 echo-cli
-rwxrwxr-x 1 olej olej 2778961 дек 16 15:29 echo-serv

```

Этот эхо-сервер последовательно фиксирует подключения к нему разнообразных клиентов:

```

$ ./echo-serv
waiting on port 51500
waiting on port 51500
waiting on port 51500
waiting on port 51500
waiting on port 51500
^C
$ ./echo-cli
connect to port 51500
> 123
123
> 2345
2345

```

```
> русская
русская
> ^C
```

И проверка совместимости:

```
$ nc 127.0.0.1 51500
123
123
3456
3456
^C
$ socat tcp4:127.0.0.1:51500 STDOUT
123
123
0987
0987
^C
$ telnet 127.0.0.1 51500
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
123
123
4568
4568
^]
telnet> quit
Connection closed.
```

Но это не всё... Мы все слышали, что язык Go реализует совершенно новую, уникальную модель параллелизма, лёгких сопрограмм, с огромной нагрузочной способностью, позволяющей легко создавать и исполнять сотни, или даже даже тысячи, параллельно выполняющихся ветвей.

Преобразуем наш последовательный эхо-сервер в параллельный (как это делалось при рассмотрении классов серверов): сервер сможет обслуживать независимо и одновременно много клиентов. Код такого сервера:

echo servm.go :

```
package main
import "net"
import "fmt"
import "bufio"

func main() {
    ln, _ := net.Listen("tcp", ":51501") // Устанавливаем прослушивание порта
    for {                                // Цикл прослушивания
        fmt.Println("waiting on port 51501")
        conn, _ := ln.Accept()          // Открываем серверный порт
        go func(cn net.Conn) {
            defer func() { cn.Close() }()
            for {                          // Запускаем цикл подключений
                // Будем прослушивать все сообщения разделенные \n
                message, err := bufio.NewReader(cn).ReadString('\n')
                if err != nil {            // Клиент разорвал коннект
                    return
                }
                cn.Write([]byte(message))  // Отправить обратно клиенту
            }
        }(conn)
    }
}
```

```

    }(conn)
  }
}

```

Здесь проявляется всё изящество нового языка Go, вобравшего в себя весь опыт эксплуатации за 50 лет (и все накопленные проблемы) языков C и C++:

- Обслуживание каждого клиента производится в отдельной, создаваемой для него оператором `go` лёгкой сопрограмме (иногда говорят как о горутинах)...
- В качестве тела выполняющейся сопрограммы оператор `go` получает тут же, инлайн, описанную анонимную, не именованную, лямбда-функцию...
- Которая тут же, в точке описании, и начинает выполняться с параметром вызова `conn`.
- Завершающие действия этой анонимной функции определены не непосредственно в последовательном коде, а в операторе завершения `defer`, тело которого тоже описано анонимной лямбда-функцией, и которое будет выполняться при `return` из функции сопрограммы.

И теперь мы можем испытать это в деле... Поведение сервера теперь немного изменилось: он выдаёт однострочную подсказку (для новых подключений) после подключения каждого клиента, независимо от того, что ранее подсоединившиеся клиенты всё ещё, возможно, продолжают работать:

```

$ ./echo-servm
waiting on port 51501
waiting on port 51501
waiting on port 51501
...
^C

```

А клиенты, сколько бы их не было, обслуживаются сервером независимо, без интерференции, наложения, или искажения их трафиков (в текст сообщений специально внесён префикс, чтобы различать к чьему потоку относятся эти сообщения):

```

$ ./echo-cli 51501
connect to port 51501
> A_клиент
  A_клиент
> A_1234567
  A_1234567
> A_строка
  A_строка
$ ./echo-cli 51501
connect to port 51501
> B_клиент
  B_клиент
> B_987654
  B_987654
> B_строка
  B_строка

```

И точно то же самое с независимыми клиентами из числа сетевых утилит Linux:

```

$ ./echo-cli 51501
connect to port 51501
> A_123
  A_123
$ nc 127.0.0.1 51501
B_123
B_123

```

```
$ socat tcp4:127.0.0.1:51501 STDOUT
C_123
C_123
$ telnet 127.0.0.1 51501
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
D_123
D_123
```

Rust

Дальше нас ожидает такой модный язык системного программирования как Rust! Язык Rust был официально представлен компанией Mozilla, спонсирующей разработку Rust, в 2010 году.

Выбор Rust в качестве инструментария сетевого проекта даст вам главное преимущество — высочайшую надёжность и отсутствие скрытых ошибок. Это обеспечивается, главным образом, 2-мя факторами:

- Если результатом операции (например, вызов функции) может иметь несколько альтернативных исходов (нормальный результат, грубая ошибка операции, неожиданный результат...), то вы обязаны описать действие на каждую альтернативу без исключений. Если реакция на некоторый исход не описан кодом, то программа просто не пройдёт компиляцию. Контроль жесточайший...
- Для работы с памятью (а это главный источник ошибок) реализована модель владения: объект данных в операции (вызов функции, присвоение, ...) не копируется, а передаётся во владение, старый владелец объекта при этом теряет доступ к нему. Таким образом, каждый программный объект представлен в памяти только в одном экземпляре, а это упрощает контроль за созданием и уничтожением объектов в программе.

Поскольку все языки программирования последнего поколения (скажем, последних 15 лет происхождения) безусловно поддерживают (каждый по-своему, по-разному) парадигму параллельного выполнения, то для Rust эхо-сервер будет показан сразу в многопоточном параллельном исполнении:

echo_serv.rs :

```
use std::io::{Read, Write};
use std::net::{TcpListener, TcpStream};

fn handle_client(mut stream: TcpStream) {
    loop {
        let mut buffer = [0; 120];
        match stream.read(&mut buffer) {
            Err(_) => { //клиент закрыл сокет
                return;
            }
            _ => (),
        }
        match stream.write(&mut buffer) {
            Err(_) => { //клиент закрыл сокет
                return;
            }
            _ => (),
        }
    }
}
```

```

fn main() {
    const PORT : &str = "53500";
    let listener = match
        TcpListener::bind("127.0.0.1:".to_string() + PORT) {
            Ok(listener) => listener,
            Err(_) => {
                println!("серверный порт занят");
                return;
            }
        };
    println!("waiting on port {}", PORT);
    for stream in listener.incoming() {
        println!("waiting on port {}", PORT); // новый клиент
        match stream {
            Ok(stream) => {
                std::thread::spawn(move || handle_client(stream));
            }
            Err(e) => {
                eprintln!("ошибка соединения: {}", e);
            }
        }
    }
}

```

И параллельный ему клиент:

echo cli.rs :

```

use std::io;
use std::io::prelude::*;
use std::net::TcpStream;

fn main() -> std::io::Result<()> {
    let mut stream = match TcpStream::connect("127.0.0.1:53500") {
        Ok(stream) => stream,
        _ => {
            println!("no server available");
            return Ok(());
        },
    };
    loop {
        print!("> ");
        io::stdout().flush()?;
        let mut req = String::new();
        io::stdin()
            .read_line(&mut req)
            .expect("ошибка ввода пользователя");
        if req.is_empty() { // ^D - завершение работы
            println!("");
            break;
        }
        if req.len() == 1 { // Enter - завершение работы
            break;
        }
        match stream.write(&req.as_bytes()) {
            Ok(..) => (),
            Err(e) => println!("Error: {}", e)
        }
        let mut buffer = [0; 120]; // буфер чтения
    }
}

```

```

        match stream.read(&mut buffer) {
            Ok(..) => (),
            Err(e) => println!("Error: {}", e),
        }
        let rep = String::from_utf8_lossy(&buffer[..]);
        print!("{}", rep);
    }
    Ok(())
}

```

Для компиляции отдельных (однофайловых) простых приложений нам нет необходимости использовать такой мощный менеджер проектов языка Rust под именем `cargo`, а можно напрямую воспользоваться компилятором `rustc`, так же входящим в инсталляцию проекта Rust:

```

$ rustc echo_serv.rs
$ rustc echo_cli.rs
$ ls -l | grep ^-rwx
-rwxrwxr-x 1 olej olej 4311104 дек 21 10:58 echo_cli
-rwxrwxr-x 1 olej olej 4361552 дек 21 10:58 echo_serv

```

Вот как выглядит выполнение этого многопоточного эхо-сервера:

```

$ ./echo_serv
waiting on port 53500
waiting on port 53500
waiting on port 53500
^C
$ ./echo_cli
> 1 234 567
1 234 567
> русская строка
русская строка
> у
у
> я
я
> x
x
>

```

А вот как с этим же эхо-сервером работают, в параллель с показанным, независимые клиенты из числа утилит Linux (с артефактами от `telnet`, о которых говорилось раньше):

```

$ nc 127.0.0.1 53500
#1: it is nc
#1: it is nc
#1: русская строка
#1: русская строка
#1: 123 45 6
#1: 123 45 6
^C
$ nc 127.0.0.1 53500
#2: one more nc
#2: one more nc
#2: клиент nc
#2: клиент nc
#2: 9 87 654
#2: 9 87 654
^C

```



```

$ telnet 127.0.0.1 53500
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
string
string
строка
строка
ещё одна строка
ещё одна строка
русская строк
русская строка
у
я
х
0
^]
telnet> quit
Connection closed.

```

Эхо-сервер на shell

Часто в отработке и тестировании сетевых проектов может понадобиться независимый эхо-сервер TCP, на какой-то хитрый порт, работающий на локальном хосте, или где-то в сети, локальной или глобальной. Но такой сервер легко смастерить вообще без глубокого программирования, средствами shell (производных от ash: bash, dash ...) и утилит Linux.

Вариант 1-й ... вот и весь сервер:

```
$ nc -l -k -p 50000 -c 'xargs -n1 echo'
```

А вот его клиент:

```

$ nc 127.0.0.1 50000
слово
слово
два слова
два
слова
"два слова"
два слова
"целых три слова"
целых три слова
1 2
1
2
"1 2 3 4 5 6 7"
1 2 3 4 5 6 7
^C

```

Вот тот же эхо-сервер на удалённом хосте в LAN:

```

$ ip -4 a s enp3s0
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
    group default qlen 1000
    inet 192.168.1.241/16 brd 192.168.255.255 scope global dynamic noprefixroute enp3s0
        valid_lft 100388sec preferred_lft 100388sec
$ nc -l -p 50000 -c 'xargs -n1 echo'

```

И коннект к нему по сети:

```
$ nc 192.168.1.241 50000
123
123
"два русских слова"
два русских слова
"1 2 3 4 5 6 7"
1 2 3 4 5 6 7
^C
```

Всё замечательно работает! Но особенностью этого варианта есть то, что многословные строки (разделённые пробелами) он «бьёт» а отдельные слова. И поэтому такие строки нужно обрамлять кавычками.

А потому ... вариант 2-й:

```
$ nc -l -p 50001 -c '/bin/cat'
$ nc 127.0.0.1 50001
12345
12345
два слова
два слова
русская строка
русская строка
^C
```

И сразу же — вариант 3-й:

```
$ nc -l -p 50002 -e '/bin/cat'
$ nc 127.0.0.1 50002
123
123
45 67 89
45 67 89
русская строка
русская строка
^C
```

Особенностью всех вариантов есть то, что из 2-х реализаций NetCat (nc), присутствующих на сегодня практически во всех дистрибутивах Linux (о чём уже говорилось раньше при рассмотрении сетевых инструментов), это работает только с традиционной реализацией, а не реализацией OpenBSD¹⁷:

```
$ aptitude search netcat | grep ^i
i netcat-openbsd - TCP/IP swiss army knife
i netcat-traditional - инструмент для TCP/IP4 на все случаи жизни
```

И, возможно, нужно будет переключиться на эту реализацию системой управления альтернативами:

```
$ sudo update-alternatives --config nc
```

Есть 2 варианта для альтернативы nc (предоставляет /bin/nc).

Выбор	Путь	Приор	Состояние
* 0	/bin/nc.openbsd	50	автоматический режим
1	/bin/nc.openbsd	50	ручной режим
2	/bin/nc.traditional	10	ручной режим

Нажмите «enter», чтобы не менять текущий выбор[*], или введите нужное число: 2

¹⁷ Почти наверняка всё то же работает и с реализацией OpenBSD, но с другими опциями запуска — реализации несовместимы по синтаксису и протоколам. Но я этого не проверял...

update-alternatives: используется /bin/nc.traditional для предоставления /bin/nc (nc) в ручном режиме

Последнее неудобство всех рассмотренных вариантов состоит в том, что при завершении клиента такой последовательный эхо-сервер тоже завершается, как реакция на закрытие сокета. Но этому делу можно помочь если сервер запускать, например, таким скриптом:

```
$ while true; do nc -l -p 50000 -c 'xargs -n1 echo'; done;
```

При этом эхо-сервер остаётся в поднятом состоянии при многочисленных подключениях клиентов, до тех пор, пока вы сами принудительно не остановите его.

Источники использованной информации

[1] Йон Снейдерс, «Эффективное программирование TCP/IP», «ДМК Пресс», 2009

Йон Снейдерс, «Эффективное программирование TCP/IP. Библиотека программиста» - СПб.: «Питер», 2001, 320 стр., ISBN 5-318-00453-9

Йон Снейдерс, «Эффективное программирование TCP/IP», «ДМК Пресс», 2009



[2] У. Р. Стивенс, «UNIX: Разработка сетевых приложений», СПб.: «Питер», 2003, ISBN 5-318-00535-7, стр. 1088

Полный архив примеров кодов к этой книге может быть взят здесь:

<http://www.kohala.com/start/unp.tar.Z>



[3] У. Стивенс, Б. Феннер, Э. Рудофф, «UNIX: Разработка сетевых приложений», СПб.: «Питер», 2006, ISBN: 5-94723-991-4, стр. 1040



[4] У. Р. Стивенс, «UNIX: взаимодействие процессов», СПб.: «Питер», 2003, ISBN: 5-318-00534-9, стр. 576.



[5] W. Richard Stevens' Home Page (ресурс полного собрания книг и публикаций У.Р.Стивенс):
<http://www.kohala.com/start/>

[6] Natasha Samoylenko, Python для сетевых инженеров, яз. рус., мая 31, 2023, 745 страниц —
<https://pyneng.readthedocs.io/ru/latest/index.html>

[7] Ian Wootten, The best Python HTTP clients for 2022 —
<https://www.scrapingbee.com/blog/best-python-http-clients/>

[8] РеPy, репозиторий пакетов Python — <https://www.pepy.tech/>

[9] Go Documentation — <https://go.dev/doc/>

[10] Изучить Rust — <https://www.rust-lang.org/ru/learn>

[11] Язык программирования Rust. Установка —
<https://doc.rust-lang.ru/book/ch01-01-installation.html>

Драйверы сетевых устройств в Linux (ядро)

Linux — операционная система с **монолитным** ядром. Альтернативой монолитному ядру являются **микроядерные** операционные системы (которых создано достаточно мало). Проблемой всех моноядерных операционных систем является: как расширить функциональность ядра, динамически подгружая к нему новые компоненты? В Linux такие компоненты ядра называются модулями, и загружаясь они связываются с API ядра и его структурами данных **по абсолютным адресам** размещения. Все **драйверы** Linux являются модулями. Все имена (функции API, переменные, ...) ядра, можно видеть в псевдофайле `/proc/kallsyms`:

```
$ cat /proc/kallsyms | wc -l
88304
```

Как видно, число имён (объектов) ядра составляет несколько десятков тысяч (это число определяется **версией ядра**). Функции API ядра (которые в наибольшей мере интересуют разработчика) отмечены в этом списке 'T': сегмент текста (то есть кода) и экспортируемое имя (большая буква 'T' — внешнее имя):

```
$ cat /proc/kallsyms | head -n 100 | tail -n 10
c0401a30 T start_thread
c0401a80 T thread_saved_pc
c0401aa0 T __show_regs
c0401cc0 T release_thread
c0401ce0 T copy_thread
c0401f60 T __switch_to
c04022b0 T get_wchan
c0402350 T restore_sigcontext
c0402460 T setup_sigcontext
c0402530 t do_signal
```

Число, стоящее в начале каждой строке — это **абсолютный** адрес для вызова этой функции ядра. Но и число доступных функций API ядра впечатляет:

```
$ cat /proc/kallsyms | grep T | wc -l
15341
```

Введение в модули ядра

Для простейшего знакомства с техникой написания модулей ядра Linux проще не вдаваться в пространные объяснения, но создать простейший модуль (код такого модуля интуитивно понятен всякому программисту), собрать его и наблюдать исполнение. Вот с такого образца простейшего модуля ядра (архив `first_hello.tgz`) мы и начнём рассмотрение:

hello_printk.c :

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej@front.ru>");

static int __init hello_init(void) {
    printk("Hello, world!");
    return 0;
}

static void __exit hello_exit(void) {
```

```

    printk("Goodbye, world!");
}

module_init(hello_init);
module_exit(hello_exit);

```

Сборка модуля

Для сборки созданного модуля используем скрипт сборки Makefile, который будет с минимальными изменениями повторяться при сборке **любых** модулей ядра (он использует макросы подготовленные разработчиками ядра):

Makefile :

```

CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
DEST = /lib/modules/$(CURRENT)/misc

TARGET = hello_printk
obj-m    := $(TARGET).o

default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    @rm -f *.o *.cmd *.flags *.mod.c *.order
    @rm -f *.*.cmd *.symvers *~ *.~* TODO.*
    @rm -fR .tmp*
    @rm -rf .tmp_versions

```

От модуля к модулю в различных проектах будет меняться только переменная скрипта: hello_printk — это имя собираемого модуля и имя исходного файла кода hello_printk.c. Делаем сборку модуля ядра, выполняя команду:

```

$ make
make -C /lib/modules/2.6.32.9-70.fc12.i686.PAE/build
M=/home/olej/2011_WORK/Linux-kernel/examples
make[1]: Entering directory `/usr/src/kernels/2.6.32.9-70.fc12.i686.PAE'
  CC [M]  /home/olej/2011_WORK/Linux-kernel/examples/own-modules/1/hello_printk.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/olej/2011_WORK/Linux-kernel/examples/own-modules/1/hello_printk.mod.o
  LD [M]  /home/olej/2011_WORK/Linux-kernel/examples/own-modules/1/hello_printk.ko
make[1]: Leaving directory `/usr/src/kernels/2.6.32.9-70.fc12.i686.PAE'

```

На этом модуль создан. Начиная с ядер 2.6 расширение файлов модулей сменено с *.o на *.ko:

```

$ ls *.ko
hello_printk.ko

```

Форматом **файла** модуля является обычный **объектный** ELF формат (.o), но дополненный в таблице внешних имён некоторыми дополнительными именами, такими как : __mod_author5, __mod_license4, __mod_srcversion23, __module_depends, __mod_vermagic5, ... которые определяются специальными модульными макросами. Изучаем **файл** модуля командой:

```

$ modinfo ./hello_printk.ko
filename:      hello_printk.ko
author:        Oleg Tsiliuric <olej@front.ru>

```

```
license:      GPL
srcversion:   83915F228EC39FFCBAF99FD
depends:
vermagic:    2.6.32.9-70.fc12.i686.PAE SMP mod_unload 686
```

Точки входа и завершения

Любой модуль должен иметь объявленные функции **входа** (инициализации) модуля и его **завершения** (не обязательно, может отсутствовать). Функция инициализации будет вызываться (после проверки и соблюдения всех достаточных условий) при выполнении команды `insmod` для модуля. Точно так же, функция завершения будет вызываться при выполнении команды `rmmod`.

Функция инициализации имеет прототип и объявляется именно как функция инициализации макросом `module_init()`, как это было сделано с только-что рассмотренном примере:

```
static int __init hello_init(void) {
    ...
}
module_init(hello_init);
```

Функция завершения, совершенно симметрично, имеет прототип, и объявляется макросом `module_exit()`, как было показано:

```
static void __exit hello_exit(void) {
    ...
}
module_exit(hello_exit);
```

Примечание: Обратите внимание: функция завершения по своему прототипу не имеет возвращаемого значения, и, поэтому, она даже не может сообщить о невозможности каких-либо действий, когда она уже начала выполняться. Идея состоит в том, что система при `rmmod` сама проверит допустимость вызова функции завершения, и если они не соблюдены, просто не вызовет эту функцию.

Показанные выше соглашения по объявлению функций инициализации и завершения являются общепринятыми. Но существует ещё один не документированный способ описания этих функций: воспользоваться непосредственно их **предопределёнными** именами, а именно `init_module()` и `cleanup_module()`. Это может быть записано так:

```
int init_module(void) {
    ...
}
void cleanup_module(void) {
    ...
}
```

При такой записи необходимость в использовании макросов `module_init()` и `module_exit()` отпадает, а использовать квалификатор `static` с этими функциями нельзя (они должны быть известными внешними именами при связывании модуля с ядром).

Конечно, такая запись никак не способствует улучшению читаемости текста, но иногда может существенно сократить рутину записи, особенно в коротких иллюстративных примерах.

Вывод диагностики модуля

Для диагностического вывода из модуля используем вызов `printk()`. Он настолько подобен по своим правилам и формату общеизвестному из пользовательского пространства `printf()`, что даже не требует дополнительного описания. Отметим только некоторые тонкие особенности `printk()` относительно `printf()`:

Сам вызов `printk()` и все сопутствующие ему константы и определения найдёте в файле определений `/lib/modules/`uname -r`/build/include/linux/kernel.h`:

```
asmlinkage int printk(const char * fmt, ...)
```

Первому значащему параметру (форматной строке) **может** предшествовать (а может и не предшествовать) константа квалификатор, определяющая уровень сообщений. Определения констант для 8 уровней сообщений, записываемых в вызове `printk()` вы найдёте в файле `printk.h`:

```
#define KERN_EMERG      "<0>"    /* system is unusable          */
#define KERN_ALERT      "<1>"    /* action must be taken immediately */
#define KERN_CRIT       "<2>"    /* critical conditions          */
#define KERN_ERR         "<3>"    /* error conditions             */
#define KERN_WARNING     "<4>"    /* warning conditions           */
#define KERN_NOTICE      "<5>"    /* normal but significant condition */
#define KERN_INFO        "<6>"    /* informational                */
#define KERN_DEBUG       "<7>"    /* debug-level messages         */
```

Предшествующая константа не является отдельным параметром (не отделяется запятой!), и (как видно из определений) представляет собой символьную строку определённого вида, которая **конкатенируется** с первым параметром (являющимся, в общем случае, **форматной** строкой). Если такая константа не записана, то устанавливается уровень вывода этого сообщения по умолчанию.

Загрузка модулей

Наш модуль при загрузке/выгрузке выводит сообщение посредством вызова `printk()`. Этот вывод направляется на **текстовую консоль**. При работе в **терминале** (в графической системе X11) вывод не попадает в терминал, но его можно видеть в файле системного журнала `/var/log/messages`.

```
$ sudo insmod hello_printk.ko
$ lsmod | head -n2
Module                Size  Used by
hello_printk          557    0
$ sudo rmmod hello_printk
$ lsmod | head -n2
Module                Size  Used by
vfat                  6740   2
$ dmesg | tail -n2
Hello, world!
Goodbye, world!
$ sudo cat /var/log/messages | tail -n3
Mar  8 01:44:14 notebook ntpd[1735]: synchronized to 193.33.236.211, stratum 2
Mar  8 02:18:54 notebook kernel: Hello, world!
Mar  8 02:19:13 notebook kernel: Goodbye, world!
```

Последними 2-мя командами показаны 2 основных метода визуализации сообщений ядра (занесенных в системный журнал): утилита `dmesg` и прямое чтение файла журнала `/var/log/messages`. Они имеют несколько отличающийся формат: файл журнала содержит метки времени поступления сообщений, что иногда бывает нужно. Кроме того, прямое чтение файла журнала требует, в некоторых дистрибутивах, наличия прав `root`.

Утилита `insmod` получает **имя файла модуля**, и пытается загрузить его без проверок взаимосвязей. Утилита `modprobe` сложнее: ей передаётся или **универсальный идентификатор**, или непосредственно **имя модуля**. Если `modprobe` получает универсальный идентификатор, то она сначала пытается найти соответствующее имя модуля в файле `/etc/modprobe.conf` (устаревшее), или в файлах `*.conf` каталога `/etc/modprobe.d`, где каждому универсальному идентификатору поставлено в соответствие имя модуля (в строке `alias ...`, смотри `modprobe.conf(5)`).

Далее, по имени модуля утилита `modprobe`, по содержимому файла :

```
$ ls -l /lib/modules/`uname -r`/*.dep
-rw-r--r-- 1 root root 206131 Mar  6 13:14
/lib/modules/2.6.32.9-70.fc12.i686.PAE/modules.dep
```

- пытается установить зависимости запрошенного модуля: модули, от которых зависит запрошенный, будут загружаться утилитой прежде него. Сам файл зависимостей `modules.dep` формируется командой :

```
# depmod -a
```

Той же командой (время от времени) мы обновляем и большинство других файлов `modules.*` этого каталога:

```
$ ls /lib/modules/`uname -r`
build          modules.block      modules.inputmap    modules.pcimap      updates
extra          modules.ccwmap      modules.isapnpmap    modules.seriomap     vds
kernel         modules.dep         modules.modesetting  modules.symbols      weak-
updates
misc           modules.dep.bin     modules.networking   modules.symbols.bin
modules.alias   modules.drm         modules.ofmap        modules.usbmap
modules.alias.bin modules.ieee1394map modules.order         source
```

Интересующий нас файл содержит строки вида:

```
$ cat /lib/modules/`uname -r`/modules.dep
...
kernel/fs/ubifs/ubifs.ko: kernel/drivers/mtd/ubi/ubi.ko kernel/drivers/mtd/mtd.ko
...
```

Каждая строка файла зависимостей (`modules.dep`) содержит: а). модули, от которых зависит данный (например, модуль `ubifs` зависит от 2-х модулей `ubi` и `mtd`), и б). полные пути к файлам всех модулей. После этого загрузить модули не представляет труда, и непосредственно для этой работы включается (по каждому модулю последовательно) утилита `insmod`.

Примечание: если загрузка модуля производится непосредственно утилитой `insmod`, указанием ей **имени файла модуля**, то утилита никакие зависимости не проверяет, и, если обнаруживает неразрешённое имя — завершает загрузку аварийно.

Утилита `rmmod` выгружает ранее загруженный модуль, в качестве параметра утилита должна получать **имя модуля** (не **имя файла модуля**). Если в системе есть модули, зависящие от выгружаемого (счётчик ссылок использования модуля больше нуля), то выгрузка модуля не произойдёт, и утилита `rmmod` завершится аварийно.

Совершенно естественно, что все эти утилиты: `insmod`, `modprobe`, `depmod`, `rmmod` — слишком кардинально влияют на поведение системы, и для своего выполнения, естественно, требуют права `root`.

Параметры загрузки модуля

Модулю при его загрузке могут быть переданы значения параметров — здесь наблюдается полная аналогия (по смыслу, но не по формату) с передачей параметров пользовательскому процессу из командной строки через массив `argv[]`.

Для каждого параметра определяется переменная-параметр, далее имя этой переменной указывается в макросе `module_param()`. Подобный макрос должен быть записан **для каждого** предусмотренного параметра, и должен последовательно определить: а). имя (параметра и переменной), б). тип значения этой переменной, в). права доступа к параметру, отображаемому как путевое имя в системе `/sys`.

Значения параметрам могут быть установлены **во время загрузки** модуля через `insmod` или `modprobe`, последняя команда также может прочитать значение параметров из своего файла конфигурации (`/etc/modprobe.conf`) для загрузки модулей.

Обработка входных параметров модуля обеспечивается макросами (описаны в `<linux/moduleparam.h>`), вот основные (там же есть ещё ряд мало употребляемых), два из них приводятся с полным определением через другие макросы (что добавляет понимания):

```
module_param_named(name, value, type, perm)
#define module_param(name, type, perm) \
    module_param_named(name, name, type, perm)
module_param_string(name, string, len, perm)
module_param_array_named(name, array, type, nump, perm)
#define module_param_array(name, type, nump, perm) \
    module_param_array_named(name, name, type, nump, perm)
```

Но даже из этого подмножества употребляются чаще всего только два: `module_param()` и `module_param_array()` (детально понять работу макросов можно реально выполняя обсуждаемый ниже пример).

Примечание: Последним параметром `perm` указаны права доступа (например, `S_IRUGO | S_IWUSR`), относящиеся к имени параметра, отображаемому в подсистеме `/sys`, если нас не интересует имя параметра отображаемое в `/sys`, то хорошим значением для параметра `perm` будет 0.

Для параметров модуля в макросе `module_param()` могут быть указаны следующие типы:

- `bool`, `invbool` - булева величина (`true` или `false`) - связанная переменная должна быть типа `int`. Тип `invbool` инвертирует значение, так что значение `true` приходит как `false` и наоборот.

- `charp` - значение **указателя** на `char` - выделяется память для строки, заданной пользователем (не нужно предварительно распределять место для строки), и указатель устанавливается соответствующим образом.

- `int`, `long`, `short`, `uint`, `ulong`, `ushort` - базовые целые величины разной размерности; версии, начинающиеся с `u`, являются беззнаковыми величинами.

В качестве входного параметра может быть определён и массив выше перечисленных типов (макрос `module_param_array()`).

Пример, показывающий большинство приёмов использования параметров загрузки модуля (в архиве каталог `params`) показан ниже:

mod_params.c :

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/string.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej@front.ru>");

static int iparam = 0;          // целочисленный параметр
module_param(iparam, int, 0);

static int k = 0;              // имена параметра и переменной различаются
module_param_named(nparam, k, int, 0);

static bool bparam = true;     // логический инверсный параметр
module_param(bparam, invbool, 0);

static char* sparam;           // строчный параметр
module_param(sparam, charp, 0);
```

```

#define FIXLEN 5
static char s[FIXLEN] = ""; // имена параметра и переменной различаются
module_param_string(cparam, s, sizeof(s), 0); // копируемая строка

static int aparam[] = { 0, 0, 0, 0, 0 }; // параметр - целочисленный массив
static int arnum = sizeof(aparam) / sizeof(aparam[0]);
module_param_array(aparam, int, &arnum, S_IRUGO | S_IWUSR);

static int __init mod_init(void) {
    int j;
    char msg[40] = "";
    printk("=====\n");
    printk("iparam = %d\n", iparam);
    printk("nparam = %d\n", k);
    printk("bparam = %d\n", bparam);
    printk("sparam = %s\n", sparam);
    printk("cparam = %s {%d}\n", s, strlen(s));
    sprintf(msg, "aparam [ %d ] = ", arnum);
    for (j = 0; j < arnum; j++) sprintf(msg + strlen(msg), " %d ", aparam[j]);
    printk("%s\n", msg);
    printk("=====\n");
    return -1;
}

module_init(mod_init);

```

В коде этого модуля присутствуют две вещи, которые могут показаться непривычными программисту на языке C, нарушающие стереотипы этого языка, и поначалу именно в этом порождающие ошибки программирования в собственных модулях:

- отсутствие резервирования памяти для символьного параметра `sparam`¹⁸;
- и динамический размер параметра-массива `aparam`. (динамически изменяющийся после загрузки модуля);
- при этом этот динамический размер **не может** превысить статически зарезервированную максимальную размерность массива (такая попытка вызывает ошибку).

Но и то, и другое, хотелось бы надеяться, достаточно разъясняется демонстрируемым кодом примера.

Для сравнения - выполнение загрузки модуля с параметрами по умолчанию (без указания параметров), а затем с переопределением значений всех параметров:

```

# sudo insmod mod_params.ko
insmod: ERROR: could not insert module ./mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[14562.245812] =====
[14562.245816] iparam = 0
[14562.245818] nparam = 0
[14562.245820] bparam = 1
[14562.245822] sparam = (null)
[14562.245824] cparam = {0}
[14562.245828] aparam [ 5 ] =  0  0  0  0  0
[14562.245830] =====

```

¹⁸ Объявленный в коде указатель просто устанавливается на строку, размещённую где-то в параметрах запуска программы загрузки. При этом остаётся открытым вопрос: а если **после** отработки инсталляционной функции, **резидентный** код модуля обратится к такой строке, к чему это приведёт? Я могу предположить, что к критической ошибке, а вы можете проверить это экспериментально.

```
# insmod mod_params.ko iparam=3 nparam=4 bparam=1 sparam=str1 cparam=str2 aparam=5,4,3
insmod: ERROR: could not insert module mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[15049.389328] =====
[15049.389336] iparam = 3
[15049.389338] nparam = 4
[15049.389340] bparam = 0
[15049.389342] sparam = str1
[15049.389345] cparam = str2 {4}
[15049.389348] aparam [ 3 ] = 5 4 3
[15049.389350] =====
```

При этом массив `aparam` получил и новую размерность `argnum`, и его элементам присвоены новые значения.

Вводимые параметры загрузки и их значения в команде `insmod` жесточайшим образом контролируются (хотя, естественно, всё проконтролировать абсолютно невозможно), потому как модуль, загруженный с ошибочными значениями параметров, который становится составной частью ядра — это угроза целостности системы. Если **хотя бы один** из параметров признан некорректным, загрузка модуля не производится. Вот как происходит контроль для некоторых случаев:

```
# insmod mod_params.ko aparam=5,4,3,2,1,0
insmod: ERROR: could not insert module mod_params.ko: Invalid parameters
# echo $?
1
$ dmesg | tail -n2
[15583.285554] aparam: can only take 5 arguments
[15583.285561] mod_params: `5' invalid for parameter `aparam'
```

Здесь имела место попытка заполнить в массиве `aparam` число элементов большее, чем его зарезервированная размерность (5).

Попытка загрузки модуля с указанием параметра с именем, не предусмотренным в коде модуля, в некоторых конфигурациях (Fedora 14) приведёт к ошибке не распознанного параметра, и модуль не будет загружен:

```
$ sudo /sbin/insmod ./mod_params.ko zparam=3
insmod: error inserting './mod_params.ko': -1 Unknown symbol in module
$ dmesg | tail -n1
mod_params: Unknown parameter `zparam'
```

Но в других случаях (Fedora 20) такой параметр будет просто проигнорирован, а модуль будет нормально загружен:

```
# insmod mod_params.ko ZZparam=3
insmod: ERROR: could not insert module mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[15966.050023] =====
[15966.050026] iparam = 0
[15966.050029] nparam = 0
[15966.050031] bparam = 1
[15966.050033] sparam = (null)
[15966.050035] cparam = {0}
[15966.050039] aparam [ 5 ] = 0 0 0 0 0
[15966.050041] =====
```

К таким (волатильным) возможностям нужно относиться с большой осторожностью!

```
# insmod mod_params.ko iparam=qwerty
insmod: ERROR: could not insert module ./mod_params.ko: Invalid parameters
```

```
$ dmesg | tail -n1
[16625.270285] mod_params: `qwerty' invalid for parameter `iparam'
```

Так выглядит попытка присвоения не числового значения числовому типу.

```
# insmod mod_params.ko cparam=123456789
insmod: ERROR: could not insert module mod_params.ko: No space left on device
$ dmesg | tail -n2
[16960.871302] cparam: string doesn't fit in 4 chars.
[16960.871309] mod_params: `123456789' too large for parameter `cparam'
```

А здесь была превышена максимальная длина для строки-параметра, передаваемой копированием.

Подсчёт ссылок использования

Одним из важных (и очень путанных по описаниям) понятий из сферы модулей есть подсчёт ссылок использования модуля. Счётчик ссылок является внутренним полем структуры описания модуля и, вообще то говоря, является слабо доступным пользователю непосредственно. При загрузке модуля начальное значение счётчика ссылок нулевое. При загрузке любого следующего модуля, который использует имена (импортирует), экспортируемые данным модулем, счётчик ссылок данного модуля инкрементируется. Модуль, счётчик ссылок использования которого не нулевой, **не может быть выгружен** командой `rmmod`. Такая тщательность отслеживания сделана из-за критичности модулей в системе: некорректное обращение к несуществующему модулю **гарантирует** крах всей системы.

Смотрим такую простейшую команду:

```
$ lsmod | grep i2c_core
i2c_core                21732  5 videodev,i915,drm_kms_helper,drm,i2c_algo_bit
```

Здесь модуль, зарегистрированный в системе под именем (не имя файла!) `i2c_core` (имя выбрано произвольно из числа загруженных модулей системы), имеет текущее значение счётчика ссылок 5, и далее следует перечисление имён 5-ти модулей на него ссылающихся. До тех пор, пока эти 5 модулей не будут удалены из системы, удалить модуль `i2c_core` будет невозможно.

В чём состоит отмеченная выше путанность всего, что относится к числу ссылок модуля? В том, что в области этого понятия происходят постоянные изменения от ядра к ядру, и происходят они с такой скоростью, что литература и обсуждения не успевают за этими изменениями, а поэтому часто описывают какие-то несуществующие механизмы. До сих пор в описаниях часто можно встретить ссылки на макросы `MOD_INC_USE_COUNT()` и `MOD_DEC_USE_COUNT()`, которые увеличивают и уменьшают счётчик ссылок. Но эти макросы остались в ядрах 2.4. В ядре 2.6 и далее их место заняли функциональные вызовы (определённые в `<linux/module.h>`):

- `int try_module_get(struct module *module)` - увеличить счётчик ссылок для модуля (возвращается признак успешности операции);
- `void module_put(struct module *module)` - уменьшить счётчик ссылок для модуля;
- `unsigned int module_refcount(struct module *mod)` - вернуть значение счётчика ссылок для модуля;

В качестве параметра всех этих вызовов, как правило, передаётся константный указатель `THIS_MODULE`, так что вызовы, в конечном итоге, выглядят подобно следующему:

```
try_module_get(THIS_MODULE);
```

Таким образом, видно, что имеется возможность управлять значением счётчика ссылок из собственного модуля. Делать это нужно крайне обдуманно, поскольку если мы увеличим счётчик и симметрично его позже не уменьшим, то мы вообще не сможем выгрузить модуль (до перезагрузки системы), это один из путей возникновения в системе «перманентных» модулей, другая возможность их возникновения: модуль не имеющий в коде функции завершения. В некоторых случаях может оказаться нужным динамически изменять счётчик ссылок, препятствуя на время возможности выгрузки модуля. Это актуально, например, в функциях, реализующих операции `open()` (увеличиваем счётчик обращений) и `close()` (уменьшаем, восстанавливаем счётчик обращений) для драйверов устройств — иначе станет возможна выгрузка модуля, обслуживающего открытое устройство, а следующие обращения (из процесса пользовательского пространства) к открытому дескриптору устройства будут направлены в не инициализированную память!

И здесь возникает очередная путаница (которую можно наблюдать и по коду некоторых модулей): во многих источниках рекомендуется инкрементировать из собственного кода модуля счётчик использований при открытии устройства, и декрементировать при его закрытии. Это было актуально, наверное, когда-то, но с некоторой версии ядра (я не смог отследить с какой) это отслеживание делается автоматически при выполнении открытия/закрытия.

Структуры данных сетевого стека

Сетевая реализация построена так, чтобы не зависеть от конкретики протоколов. Основной структурой данных описывающей **сетевой интерфейс** (устройство) является `struct net_device`, к ней мы вернёмся позже, описывая устройство.

А вот **основной** структурой обмениваемых **данных** (между сетевыми уровнями), на движении экземпляров данных которой между сетевыми уровнями построена работа всей подсистемы — это есть буферы сокетов (определения в `<linux/skbuff.h>`). Буфер сокетов состоит из двух частей: данные управления `struct sk_buff`, и данные пакета (указываемые в `struct sk_buff` указателями `head` и `data`). Буферы сокетов всегда увязываются в очереди (`struct sk_queue_head`) посредством своих двух первых полей `next` и `prev`. Вот некоторые поля структуры, которые позволяют представить её структуру:

```
typedef unsigned char *sk_buff_data_t;
struct sk_buff {
    struct sk_buff *next; /* These two members must be first. */
    struct sk_buff *prev;
    ...
    sk_buff_data_t  transport_header;
    sk_buff_data_t  network_header;
    sk_buff_data_t  mac_header;
    ...
    unsigned char  *head,
                  *data;
    ...
};
```

Структура вложенности заголовков сетевых уровней в точности соответствует структуре инкапсуляции сетевых протоколов протоколов внутри друг друга, это позволяет обрабатывающему слою получать доступ к информации, относящейся только к нужному ему слою.

Экземпляры данных типа `struct sk_buff`:

- Возникают при поступлении очередного сетевого пакета (здесь нужно принимать во внимание возможность сегментации пакетов) из внешней физической среды распространения данных. Об этом событии извещает прерывание (IRQ), генерируемое сетевым адаптером. При этом создаётся (чаще извлекается из пула использованных)

экземпляр буфера сокета, заполняется данными из поступившего пакета и далее этот экземпляр передаётся **вверх** от сетевого слоя к слою, до приложения **прикладного уровня**, которое является получателем пакета. На этом экземпляр данных буфера сокета уничтожается (утилизируется).

- Возникают в среде приложения **прикладного уровня**, которое является отправителем пакета данных. Пакет отправляемых данных помещается в созданный буфер сокета, который начинает перемещаться вниз от сетевого слоя к слою, до достижения канального уровня L2. На этом уровне осуществляется физическая передача данных пакета через сетевой адаптер в среду распространения. В случае успешного завершения передачи (что подтверждается прерыванием, генерируемым сетевым адаптером, часто по той же линии IRQ, что и при приёме пакета) буфер сокета уничтожается (утилизируется). При отсутствии подтверждения отправки (IRQ) обычно делается несколько повторных попыток, прежде, чем принять решение об ошибке канала.

Прохождение экземпляра данных буфера сокета сквозь стек сетевых протоколов будет детально проанализировано далее.

Путь пакета сквозь стек протоколов

Теперь у нас достаточно деталей, чтобы проследить путь пакетов (буферов сокетов) сквозь сетевой стек, проследить то, как буфера сокетов возникают в системе, и когда они её покидают, а также ответить на вопрос, почему вышележащие протокольные уровни (будут рассмотрены чуть ниже) никогда не порождают и не уничтожают буферов сокетов, а только обрабатывают (или модифицируют) содержащуюся в них информацию (работают как фильтры). Итак, последовательность связей мы можем разложить в таком порядке:

Приём: традиционный подход

Традиционный подход состоит в том, что каждый приходящий сетевой пакет порождает аппаратное прерывание по линии IRQ адаптера, что и служит сигналом на приём очередного сетевого пакета и создание буфера сокета для его сохранения и обработки принятых данных. Порядок действий модуля сетевого интерфейса при этом следующий:

1. Читая конфигурационную область PCI адаптера сети при инициализации модуля, определяем линию прерывания IRQ, которая будет обслуживать сетевой обмен:

```
char irq;
pci_read_config_byte(pdev, PCI_INTERRUPT_LINE, &byte);
```

Точно таким же манером будет определена и область адресов ввода-адресов адаптера, скорее всего, через DMA ...

2. При инициализации сетевого интерфейса, для этой линии IRQ устанавливается обработчик прерывания `my_interrupt()` :

```
request_irq((int)irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id);
```

3. В обработчике прерывания, по приёму нового пакета из сети (то же прерывание может происходить и при завершении отправки пакета в сеть, здесь нужен анализ причины по чтению некоего аппаратного флага), создаётся (или запрашивается из пула используемых) новый экземпляр буфера сокетов:

```
static irqreturn_t my_interrupt(int irq, void *dev_id) {
    ...
    struct sk_buff *skb = kmalloc(sizeof(struct sk_buff), ...);
    // заполнение данных *skb чтением из портов сетевого адаптера
    netif_rx(skb);
    return IRQ_HANDLED;
}
```

Все эти действия выполняются не в самом обработчике верхней половины прерываний от сетевого адаптера, а в обработчике отложенного прерывания `NET_RX_SOFTIRQ` для этой линии. Последним действием является передача заполненного сокетного буфера вызову `netif_rx()` (или `netif_receive_skb()`) который и запустит процесс движения его (буфера) вверх по структуре сетевого стека (отметит отложенное программное прерывание `NET_RX_SOFTIRQ` для исполнения).

Приём: высокоскоростной интерфейс

Особенность природы сетевых интерфейсов состоит в том, что их активность носит взрывной характер: после весьма продолжительных периодов молчания возникают интервалы пиковой активности, когда сетевые пакеты (сегментированные на IP пакеты объёмы передаваемых данных) следуют сплошной плотной чередой. После такого пика активности могут снова наступать значительные промежутки полного отсутствия активности, или вялой активности на интерфейсе (обмен ARP пакетами для обновления информации разрешения локальных адресов и подобные виды активности). Современные Ethernet сетевые карты используют скорости обмена до 10 Gib/s, но уже даже при значительно ниже интенсивностях традиционный подход становится нецелесообразным: в периоды высокой плотности поступления пакетов:

- новые приходящие пакеты создают вложенные запросы IRQ нескольких уровней при ещё не обслуженном приёме текущего IRQ;
- асинхронное обслуживание каждого IRQ в плотном потоке создаёт слишком большие накладные расходы;

Поэтому был добавлен набор API для обработки таких плотных потоков пакетов, поступающих с высокоскоростных интерфейсов, который и получил название NAPI (New API¹⁹). Идея состоит в том, чтобы приём пакетов осуществлять не методом аппаратного прерывания, а методом **программного опроса** (polling), точнее, комбинацией этих двух возможностей:

- при поступлении **первого** пакета «пачки» инициируется прерывание IRQ адаптера (всё начинается как в традиционном методе)...
- в обработчике прерывания **запрещается** поступление дальнейших запросов прерывания с этой линии IRQ по приёму пакетов, IRQ с этой же линии по отправке пакетов могут продолжать поступать, таким образом, этот запрет происходит не программным запретом линии IRQ со стороны процессора, а записью управляющей информации в **аппаратные регистры** сетевого адаптера, адаптер должен предусматривать такое раздельное управление поступлением прерываний по приёму и передаче, но для современных высокоскоростных адаптеров это, обычно, соблюдается;
- после прекращения прерываний по приёму обработчик переходит в режим циклического считывания и обработки принятых из сети пакетов, сетевой адаптер при этом накапливает поступающие пакеты во внутреннем кольцевом буфере приёма, а считывание производится либо до полного исчерпания кольцевого буфера, либо до опеределённого порогового числа считанных пакетов (10, 20, ...), называемого бюджетом функции полинга;
- естественно, это считывание и обработка пакетов происходит не в собственно обработчике прерывания (верхней половине), а в его отсроченной (нижней) части;
- по каждому принятому в опросе пакету генерируется сокетный буфер для продвижения его по стеку сетевых протоколов вверх;
- после **завершения цикла** программного опроса, по его результатам устанавливается состояние завершения `NAPI_STATE_DISABLE` (если не осталось больше не сосчитанных пакетов в кольцевом буфере адаптера), или `NAPI_STATE_SCHED` (что

¹⁹ Естественно, до какого времени он будет «новым» неизвестно — до появления ещё более нового.

говорит, что устройство адаптера должно продолжать опрашиваться когда ядро следующий раз перейдёт к циклу опросов в отложенном обработчике прерываний).

- если результатом является `NAPI_STATE_DISABLE`, то после завершения цикла программного опроса восстанавливается разрешение генерации прерываний по линии IRQ приёма пакетов (записью в порты сетевого адаптера);

В реализующем коде модуля это укрупнённо должно выглядеть подобно следующему (при условии, что линия IRQ связана с аппаратным адаптером, как это описано для традиционного метода):

1. Реализатор обязан предварительно создать и зарегистрировать специфичную для модуля функцию опроса (poll-функцию), используя вызов (`<netdevice.h>`):

```
static inline void netif_napi_add(struct net_device *dev,
                                struct napi_struct *napi,
                                int (*poll)(struct napi_struct *, int),
                                int weight);
```

- где:

`dev` — это рассмотренная раньше структура зарегистрированного сетевого интерфейса;

`poll` — регистрируемая функция программного опроса, о которой ниже;

`weight` — относительный вес, приоритет, который придаёт разработчик этому интерфейсу, для 10Mib и 100Mib адаптеров здесь часто указано значение 16, а для 10Gib и 100Gib — значение 64;

`napi` — дополнительный параметр, указатель на специальную структуру, которая будет передаваться в каждый вызов функции `poll`, и где будет, по результату выполнения этой функции, заполняться поле `state` значениями `NAPI_STATE_DISABLE` или `NAPI_STATE_SCHED`, вид этой структуры должен быть (`<netdevice.h>`):

```
struct napi_struct {
    struct list_head poll_list;
    unsigned long state;
    int weight;
    int (*poll)(struct napi_struct*, int);
};
```

2. Зарегистрированная функция программного опроса (полностью зависящая от задачи и реализуемая в коде модуля) имеет подобный вид:

```
static int my_card_poll(struct napi_struct *napi, int budget) {
    int work_done; // число реально обработанных в цикле опроса сетевых пакетов
    work_done = my_card_input(budget, ...); // реализационно специфический приём пакетов
    if (work_done < budget) {
        netif_rx_complete(netdev, napi);
        my_card_enable_irq(...); // разрешить IRQ приёма
    }
    return work_done;
}
```

Здесь пользовательская функция `my_card_input()` в цикле пытается аппаратно сосчитать `budget` сетевых пакетов, и для каждого считанного сетевого пакета создаёт сокетный буфер и вызывает `netif_receive_skb()`, после чего этот буфер начинает движение по стеку протоколов вверх. Если кольцевой буфер сетевого адаптера исчерпан ранее `budget` пакетов (нет более наличных пакетов), то адаптеру разрешается возбуждать прерывания по приёму, а ядро вызовом `netif_rx_complete()` уведомляется, что отменяется отложенное программное прерывание `NET_RX_SOFTIRQ` для дальнейшего вызова функции опроса. Если же удалось сосчитать `budget` пакетов (в буфере адаптера, видимо, есть ещё не обработанные

пакеты), то опрос продолжится при следующем цикле обработки отложенного программного прерывания NET_RX_SOFTIRQ.

3. Обработчик аппаратного прерывания линии IRQ сетевого адаптера (активирующий при приходе **первого** сетевого пакета «пачки» активности) должен выполнять примерно следующее:

```
static irqreturn_t my_interrupt(int irq, void *dev_id) {
    struct net_device *netdev = dev_id;
    if (likely(netif_rx_schedule_prep(netdev, ...))) {
        my_card_disable_irq(...);           // запретить IRQ приёма
        __netif_rx_schedule(netdev, ...);
    }
    return IRQ_HANDLED;
}
```

Здесь ядро должно быть уведомлено, что новая порция сетевых пакетов готова для обработки. Для этого

- вызов `netif_rx_schedule_prep()` подготавливает устройство для помещения в список для программного опроса, устанавливая состояние в `NAPI_STATE_SCHED`;
- если предыдущий вызов успешен (а противное возникает только если NAPI уже активен), то вызовом `__netif_rx_schedule()` устройство помещается в список для программного опроса, в цикле обработки отложенного программного прерывания NET_RX_SOFTIRQ.

Вот, собственно, и всё относительно новой модели приёма сетевых пакетов. Здесь нужно держать в виду, что бюджет, что разово устанавливаемый в функции опроса (локальный бюджет), не должен быть чрезмерно большим. По крайней мере:

- Опрос не должен должен потреблять более одного системного тика (глобальная переменная `jiffies`), иначе это будет искажать диспетчеризацию потоков ядра;
- Бюджет не должен быть больше глобально установленного ограничения:

```
$ cat /proc/sys/net/core/netdev_budget
300
```

После каждого цикла опроса число обработанных пакетов (возвращаемых функцией опроса) вычитается из этого глобального бюджета, и если остаток меньше нуля, то обработчик программного прерывания NET_RX_SOFTIRQ останавливается.

Передача пакетов

Описанными выше действиями инициируется создание и движение сокетного буфера вверх по стеку. Движение же вниз (при отправке в сеть) обеспечивается по другой цепочке:

1. При инициализации сетевого интерфейса (это момент, который уже был назван выше в п.2), создаётся таблица операций сетевого интерфейса, одно из полей которой `ndo_start_xmit` определяет функцию передачи пакета в сеть:

```
struct net_device_ops ndo = {
    .ndo_open = my_open,
    .ndo_stop = my_close,
    .ndo_start_xmit = stub_start_xmit,
};
```

2. При вызове `stub_start_xmit()` должна обеспечить аппаратную передачу полученного сокета в сеть, после чего уничтожает (возвращает в пул) буфер сокета:

```
static int stub_start_xmit(struct sk_buff *skb, struct net_device *dev) {
    // ... аппаратное обслуживание передачи
```

```

dev_kfree_skb(skb);
return 0;
}

```

Реально чаще уничтожение отправляемого буфера будет происходить не при инициализации операции, а при её (успешном) завершении, что отслеживается **по той же линии IRQ**, что и приём пакетов из сети.

Часто задаваемый вопрос: а где же в этом процессе место (код), где реально создаётся содержательная информация, **помещаемая** в сокетный буфер, или где **потребляется** информация из принимаемых сокетных буферов? Ответ: не ищите такого места в пределах сетевого стека ядра — любая информация для отправки в сеть, или потребляемая из сети, возникает в поле зрения только на прикладных уровнях, в приложениях пространства пользователя, таких, например, как `ping`, `ssh`, `telnet` и великое множество других. Интерфейс из этого прикладного уровня в стек протоколов ядра обеспечивается известным POSIX API сокетов прикладного уровня.

Драйверы: сетевой интерфейс

Задача сетевого интерфейса — быть тем местом, в котором:

- создаются экземпляры структуры `struct sk_buff`, по каждому принятому из интерфейса пакету (здесь нужно принимать во внимание возможность сегментации IP пакетов), далее созданный экземпляр структуры продвигается по стеку протоколов вверх, до получателя пользовательского пространства, где он и уничтожается;
- исходящие экземпляры структуры `struct sk_buff`, порождённые где-то на верхних уровнях протоколов пользовательского пространства, должны отправляться (чаще всего каким-то аппаратным механизмом), а сами экземпляры структуры после этого — уничтожаться.

Более детально эти вопросы рассмотрены, при обсуждении прохождения пакетов сквозь стек сетевых протоколов. А пока наша задача — создание той конечной точки (интерфейса), где эти последовательности действий начинаются и завершаются.

Ниже показан пример простого создания и регистрации в системе нового сетевого интерфейса (примеры этого раздела заимствованы из и находятся в каталоге архива `net`):

network.c :

```

#include <linux/module.h>
#include <linux/netdevice.h>

static struct net_device *dev;

static int my_open(struct net_device *dev) {
    printk(KERN_INFO "Hit: my_open(%s)\n", dev->name);
    /* start up the transmission queue */
    netif_start_queue(dev);
    return 0;
}

static int my_close(struct net_device *dev) {
    printk(KERN_INFO "Hit: my_close(%s)\n", dev->name);
    /* shutdown the transmission queue */
    netif_stop_queue(dev);
    return 0;
}

/* Note this method is only needed on some; without it
   module will fail upon removal or use. At any rate there is a memory

```

```

        leak whenever you try to send a packet through in any case*/
static int stub_start_xmit(struct sk_buff *skb, struct net_device *dev) {
    dev_kfree_skb(skb);
    return 0;
}

static struct net_device_ops ndo = {
    .ndo_open = my_open,
    .ndo_stop = my_close,
    .ndo_start_xmit = stub_start_xmit,
};

static void my_setup(struct net_device *dev) {
    int j;
    printk(KERN_INFO "my_setup(%s)\n", dev->name);
    /* Fill in the MAC address with a phoney */
    for (j = 0; j < ETH_ALEN; ++j)
        dev->dev_addr[j] = (char)j;
    ether_setup(dev);
    dev->netdev_ops = &ndo;
}

static int __init my_init(void) {
    printk(KERN_INFO "Loading stub network module:....");
    dev = alloc_netdev(0, "fict%d", my_setup);
    if (register_netdev(dev)) {
        printk(KERN_INFO " Failed to register\n");
        free_netdev(dev);
        return -1;
    }
    printk(KERN_INFO "Succeeded in loading %s!\n", dev_name(&dev->dev));
    return 0;
}

static void __exit my_exit(void) {
    printk(KERN_INFO "Unloading stub network module\n");
    unregister_netdev(dev);
    free_netdev(dev);
}

module_init(my_init);
module_exit(my_exit);

MODULE_AUTHOR("Bill Shubert");
MODULE_AUTHOR("Jerry Cooperstein");
MODULE_AUTHOR("Tatsuo Kawasaki");
MODULE_DESCRIPTION("LDD:1.0 s_24/lab1_network.c");
MODULE_LICENSE("GPL v2");

```

Здесь нужно обратить внимание на вызов `alloc_netdev()`, который в качестве параметра получает шаблон (`%d`) имени нового интерфейса: мы задаём префикс имени интерфейса (`fict`), а система присваивает сама первый свободный номер интерфейса с таким префиксом. Обратите также внимание как в цикле заполнился фиктивным значением `00:01:02:03:04:05` MAC-адрес интерфейса, что мы увидим вскоре в диагностике.

Вся связь сетевого интерфейса с выполняемыми на нём операциями осуществляется через таблицу функций операций сетевого интерфейса (**net device operations**):

```

struct net_device_ops {

```

```

int                (*ndo_init)(struct net_device *dev);
void               (*ndo_uninit)(struct net_device *dev);
int                (*ndo_open)(struct net_device *dev);
int                (*ndo_stop)(struct net_device *dev);
netdev_tx_t        (*ndo_start_xmit) (struct sk_buff *skb,
                                     struct net_device *dev);

...
struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
...
}

```

В ядре 3.09, например, определено 39 операций в `struct net_device_ops`, (и около 50-ти операций в ядре 3.14), но реально разрабатываемые модули реализуют только некоторую малую часть из них.

Характерно, что в таблице операций интерфейса присутствует операция **передачи** сокетного буфера `ndo_start_xmit` в физическую среду, но вовсе нет операции **приёма** пакетов (сокетных буферов). Это совершенно естественно, как мы увидим вскоре: принятые пакеты (например в обработчике аппаратного прерывания IRQ) тут же передаются в очередь (ядра) принимаемых пакетов, и далее уже обрабатываются сетевым стеком. А вот выполнять операцию `ndo_start_xmit` — обязательно, хотя бы, как минимум, для вызова API ядра `dev_kfree_skb()`, который утилизирует (уничтожает) сокетный буфер после успешной (да и безуспешной тоже) операции передачи пакета. Если этого не делать, в системе возникнет слабо выраженная утечка памяти (с каждым пакетом), которая, в конечном итоге, рано или поздно приведёт к краху системы.

Теперь созданное нами выше (пока фиктивное) сетевое устройство уже можно установить в системе:

```

$ sudo insmod ./network.ko
$ dmesg | tail -n4
[ 7355.005588] Loading stub network module:....
[ 7355.005597] my_setup()
[ 7355.006703] Succeeded in loading fict0!
$ ip link show dev fict0
5: fict0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
$ sudo ifconfig fict0 192.168.56.50
$ dmesg | tail -n6
[ 7355.005588] Loading stub network module:....
[ 7355.005597] my_setup()
[ 7355.006703] Succeeded in loading fict0!
[ 7562.604588] Hit: my_open(fict0)
[ 7573.442094] fict0: no IPv6 routers present
$ ping 192.168.56.50
PING 192.168.56.50 (192.168.56.50) 56(84) bytes of data.
64 bytes from 192.168.56.50: icmp_req=1 ttl=64 time=0.253 ms
64 bytes from 192.168.56.50: icmp_req=2 ttl=64 time=0.056 ms
64 bytes from 192.168.56.50: icmp_req=3 ttl=64 time=0.057 ms
64 bytes from 192.168.56.50: icmp_req=4 ttl=64 time=0.056 ms
^C
--- 192.168.56.50 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.056/0.105/0.253/0.085 ms
$ ifconfig fict0
fict0    Link encap:Ethernet  HWaddr 00:01:02:03:04:05
         inet addr:192.168.56.50  Bcast:192.168.56.255  Mask:255.255.255.0
         inet6 addr: fe80::201:2ff:fe03:405/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0

```

```
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
```

Обратите внимание, как совершенно **произвольное значение** заполняется в структуре `net_device`, и устанавливается в качестве MAC (аппаратного) адреса созданного интерфейса (в функции `my_setup()`).

Как уже отмечалось выше, основу структуры описания сетевого интерфейса составляет структура `struct net_device`, описанная в `<linux/netdevice.h>`. При работе с сетевыми интерфейсами эту структуру стоит изучить весьма тщательно. Это очень крупная структура, содержащая не только описание аппаратных средств, но и конфигурационные параметры сетевого интерфейса по отношению к выше лежащим протоколам (пример взят из ядра 3.09):

```
struct net_device {
    char name[ IFNAMSIZ ] ;
    ...
    unsigned long mem_end; /* shared mem end */
    unsigned long mem_start; /* shared mem start */
    unsigned long base_addr; /* device I/O address */
    unsigned int irq; /* device IRQ number */
    ...
    unsigned mtu; /* interface MTU value */
    unsigned short type; /* interface hardware type */
    ...
    struct net_device_stats stats;
    struct list_head dev_list;
    ...
    /* Interface address info. */
    unsigned char perm_addr[ MAX_ADDR_LEN ]; /* permanent hw address */
    unsigned char addr_len; /* hardware address length */
    ...
}
```

Здесь поле `type`, например, определяет тип аппаратного адаптера с точки зрения ARP-механизма разрешения MAC адресов (`<linux/if_arp.h>`):

```
...
#define ARPHRD_ETHER 1 /* Ethernet 10Mbps */
...
#define ARPHRD_IEEE802 6 /* IEEE 802.2 Ethernet/TR/TB */
#define ARPHRD_ARCNET 7 /* ARCnet */
...
#define ARPHRD_IEEE1394 24 /* IEEE 1394 IPv4 - RFC 2734 */
...
#define ARPHRD_IEEE80211 801 /* IEEE 802.11 */
```

Здесь же заносятся такие совершенно аппаратные характеристики интерфейса (реализующего его физического адаптера), как, например, адрес базовой области ввода-вывода (`base_addr`), используемая линия аппаратного прерывания (`irq`), максимальная длина пакета для данного интерфейса (`mtu`)...

Детальный разбор огромного числа полей `struct net_device` (этой и любой другой сопутствующей) или их возможных значений — бессмысленный, хотя бы потому, что эта структура радикально изменяется от одной подверсии к другой подверсии ядра; такой разбор должен проводиться «по месту» на основе изучения названных выше заголовочных файлов.

Со структурой сетевого интерфейса обычно создаётся и связывается (кодом модуля) **приватная структура данных**, в которой пользователь может размещать произвольные собственные данные любой сложности, ассоциированные с интерфейсом. Это обычная

практика ядра Linux, и не только сетевой подсистемы. Указатель такой приватной структуры помещается в структуру сетевого интерфейса. Это особо актуально, если предполагается, что драйвер может создавать несколько сетевых интерфейсов (например, несколько идентичных сетевых адаптеров). Доступ к приватной структуре данных должен определяться **исключительно** специально определённой для того функцией `netdev_priv()`. Ниже показан возможный вид функции — это определение из ядра 3.09, но никто не даст гарантий, что в другом ядре оно радикально не поменяется:

```
/*      netdev_priv - access network device private data
 * Get network device private data
 */
static inline void *netdev_priv(const struct net_device *dev) {
    return (char*)dev + ALIGN(sizeof(struct net_device), NETDEV_ALIGN);
}
```

Примечание: Как легко видеть из определения, приватная структура данных дописывается **непосредственно в хвост** `struct net_device` - это обычная практика создания структур переменного размера, принятая в языке C начиная с стандарта C89 (и в C99). Но именно из-за этого, за счёт эффектов **выравнивания** данных (и его возможного изменения в будущем), не следует адресоваться к приватным данным непосредственно, а следует использовать `netdev_priv()`.

При начальном размещении интерфейса размер определённой пользователем приватной структуры передаётся первым параметром функции размещения, например так:

```
child = alloc_netdev(sizeof(struct priv), "fict%d", &setup);
```

После успешного выполнения размещения интерфейса приватная структура также будет размещена («в хвост» структуре `struct net_device`), и будет доступна по вызову `netdev_priv()`.

Все структуры `struct net_device`, описывающие доступные сетевые интерфейсы в системе, увязаны в единый связный список.

Примечание: В ядре Linux **все и любые** списочные связные структуры строятся на основе API кольцевых двухсвязных списков, структур данных единого вида `struct list_head` (поле `dev_list` в `struct net_device`).

Следующий пример визуализирует содержимого списка сетевых интерфейсов:

devices.c :

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/netdevice.h>

static int __init my_init(void) {
    struct net_device *dev;
    printk(KERN_INFO "Hello: module loaded at 0x%p\n", my_init);
    dev = first_net_device(&init_net);
    printk(KERN_INFO "Hello: dev_base address=0x%p\n", dev);
    while (dev) {
        printk(KERN_INFO
            "name = %6s irq=%4d trans_start=%12lu last_rx=%12lu\n",
            dev->name, dev->irq, dev->trans_start, dev->last_rx);
        dev = next_net_device(dev);
    }
    return -1;
}

module_init(my_init);
```

Выполнение (предварительно для убедительности загрузим ранее созданный модуль `network.ko`):

```
$ sudo insmod network.ko
$ sudo insmod devices.ko
insmod: error inserting 'devices.ko': -1 Operation not permitted
$ dmesg | tail -n8
Hello: module loaded at 0xf8853000
Hello: dev_base address=0xf719c400
name =    lo  irq=   0  trans_start=          0  last_rx=          0
name =  eth0  irq=  16  trans_start= 4294693516  last_rx=          0
name = wlan0  irq=   0  trans_start= 4294693412  last_rx=          0
name =  pan0  irq=   0  trans_start=          0  last_rx=          0
name = cipsec0 irq=   0  trans_start=    2459232  last_rx=          0
name = mynet0 irq=   0  trans_start=          0  last_rx=          0
```

Статистики интерфейса

Процессы, происходящие на сетевом интерфейсе, сложно явно наблюдать (в сравнении, скажем, с интерфейсами `/dev` или `/proc`). Поэтому очень важной характеристикой интерфейса становится накопленная статистика происходящих на нём процессов. Для накопления статистики работы сетевого интерфейса описана специальная структура (достаточно большая, определена там же в `<linux/netdevice.h>`, показано только начало структуры):

```
struct net_device_stats {
    unsigned long rx_packets;    /* total packets received */
    unsigned long tx_packets;    /* total packets transmitted */
    unsigned long rx_bytes;      /* total bytes received */
    unsigned long tx_bytes;      /* total bytes transmitted */
    unsigned long rx_errors;     /* bad packets received */
    unsigned long tx_errors;     /* packet transmit problems */
    ...
}
```

Поля такой структуры должны заполняться кодом модуля статистическими данными проходящих пакетов (при передаче пакета, например, инкрементируя `tx_packets`).

В пространство пользователя эту структуру возвращает функция `ndo_get_stats` в таблице операций `struct net_device_ops` (выше эти поля были специально показаны). Модуль должен реализовать такую собственную функцию и поместить её в `struct net_device_ops`. Это делается, если вы хотите получать статистики сетевого интерфейса пользователем вызовом `ifconfig`, или через интерфейс файловой системы `/proc`, как это ожидаемо и происходит для всех других сетевых интерфейсов:

```
$ ifconfig wlan0
wlan0    Link encap:Ethernet  HWaddr 00:13:02:69:70:9B
         inet addr:192.168.1.22  Bcast:192.168.1.255  Mask:255.255.255.0
         inet6 addr: fe80::213:2ff:fe69:709b/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:8658 errors:0 dropped:0 overruns:0 frame:0
         TX packets:9070 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:4240425 (4.0 MiB)  TX bytes:1318733 (1.2 MiB)
```

Где обычно размещается структура `net_device_stats`, которую мы предполагаем возвращать пользователю? Часто встречаются несколько вариантов:

1. Если модуль обслуживает только один конкретный сетевой интерфейс, то структура может размещаться на глобальном уровне кода модуля.


```
static struct net_device_stats stats;
...
static struct net_device_stats *my_get_stats(struct net_device *dev) {
    return &stats;
}
...
static struct net_device_ops ndo = {
    ...
    .ndo_get_stats = my_get_stats,
};
```

2. Часто структура статистики размещается как составная часть структуры **приватных данных** (о которой была речь выше), которую разработчик связывает с сетевым интерфейсом.

```
static struct net_device *my_dev = NULL;
struct my_private {
    struct net_device_stats stats;
    ...
};
...
static struct net_device_stats *my_get_stats(struct net_device *dev) {
    struct my_private *priv = (my_private*)netdev_priv(dev);
    return &priv->stats;
}
...
static struct net_device_ops ndo = {
    ...
    .ndo_get_stats = my_get_stats,
}
...
void my_setup(struct net_device *dev) {
    memset(netdev_priv(dev), 0, sizeof(struct my_private));
    dev->netdev_ops = &ndo;
}
int __init my_init(void) {
    my_dev = alloc_netdev(sizeof(struct my_private), "my_if%d", my_setup);
}
```

3. Наконец, может использоваться структура, включённая (имплементированная) непосредственно **в состав** определения интерфейса struct net_device.

```
...
static struct net_device_stats *my_get_stats(struct net_device *dev) {
    return &dev->stats;
}
```

Все эти три варианта использования показаны (для сравнения: файлы virt.c, virt1.c и virt2.c в каталоге архива virt).

Виртуальный сетевой интерфейс

В предыдущих примерах мы создавали сетевые интерфейсы, но они не осуществляли реально с физической средой передачи и приёма. Для выполнения такого уровня проработки нужно бы иметь реальное коммуникационное оборудование на PCI шине, что не всегда доступно. Но мы можем создать интерфейс, который будет перехватывать трафик сетевого ввода-вывода с другого, реально существующего в системе, интерфейса, и обеспечивать обработку этих потоков (каталог архива virt).

virt.c :

```

#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/moduleparam.h>
#include <net/arp.h>

#define ERR(...) printk(KERN_ERR "! " __VA_ARGS__)
#define LOG(...) printk(KERN_INFO "! " __VA_ARGS__)

static char* link = "eth0";
module_param(link, charp, 0);

static char* ifname = "virt";
module_param(ifname, charp, 0);

static struct net_device *child = NULL;

struct priv {
    struct net_device_stats stats;
    struct net_device *parent;
};

static rx_handler_result_t handle_frame(struct sk_buff **pskb) {
    struct sk_buff *skb = *pskb;
    if (child) {
        struct priv *priv = netdev_priv(child);
        priv->stats.rx_packets++;
        priv->stats.rx_bytes += skb->len;
        LOG("rx: injecting frame from %s to %s", skb->dev->name, child->name);
        skb->dev = child;
        return RX_HANDLER_ANOTHER;
    }
    return RX_HANDLER_PASS;
}

static int open(struct net_device *dev) {
    netif_start_queue(dev);
    LOG("%s: device opened", dev->name);
    return 0;
}

static int stop(struct net_device *dev) {
    netif_stop_queue(dev);
    LOG("%s: device closed", dev->name);
    return 0;
}

static netdev_tx_t start_xmit(struct sk_buff *skb, struct net_device *dev) {
    struct priv *priv = netdev_priv(dev);
    priv->stats.tx_packets++;
    priv->stats.tx_bytes += skb->len;
    if (priv->parent) {
        skb->dev = priv->parent;
        skb->priority = 1;
        dev_queue_xmit(skb);
        LOG("tx: injecting frame from %s to %s", dev->name, skb->dev->name);
        return 0;
    }
    return NETDEV_TX_OK;
}

```

```

}

static struct net_device_stats *get_stats(struct net_device *dev) {
    return &((struct priv*)netdev_priv(dev))->stats;
}

static struct net_device_ops crypto_net_device_ops = {
    .ndo_open = open,
    .ndo_stop = stop,
    .ndo_get_stats = get_stats,
    .ndo_start_xmit = start_xmit,
};

static void setup(struct net_device *dev) {
    int j;
    ether_setup(dev);
    memset(netdev_priv(dev), 0, sizeof(struct priv));
    dev->netdev_ops = &crypto_net_device_ops;
    for (j = 0; j < ETH_ALEN; ++j) // fill in the MAC address with a phoney
        dev->dev_addr[j] = (char)j;
}

int __init init(void) {
    int err = 0;
    struct priv *priv;
    char ifstr[40];
    sprintf(ifstr, "%s%s", ifname, "%d");
    child = alloc_netdev(sizeof(struct priv), ifstr, setup);
    if (child == NULL) {
        ERR("%s: allocate error", THIS_MODULE->name); return -ENOMEM;
    }
    priv = netdev_priv(child);
    priv->parent = __dev_get_by_name(&init_net, link); // parent interface
    if (!priv->parent) {
        ERR("%s: no such net: %s", THIS_MODULE->name, link);
        err = -ENODEV; goto err;
    }
    if (priv->parent->type != ARPHRD_ETHER && priv->parent->type != ARPHRD_LOOPBACK) {
        ERR("%s: illegal net type", THIS_MODULE->name);
        err = -EINVAL; goto err;
    }
    /* also, and clone its IP, MAC and other information */
    memcpy(child->dev_addr, priv->parent->dev_addr, ETH_ALEN);
    memcpy(child->broadcast, priv->parent->broadcast, ETH_ALEN);
    if ((err = dev_alloc_name(child, child->name))) {
        ERR("%s: allocate name, error %i", THIS_MODULE->name, err);
        err = -EIO; goto err;
    }
    register_netdev(child);
    rtnl_lock();
    netdev_rx_handler_register(priv->parent, &handle_frame, NULL);
    rtnl_unlock();
    LOG("module %s loaded", THIS_MODULE->name);
    LOG("%s: create link %s", THIS_MODULE->name, child->name);
    LOG("%s: registered rx handler for %s", THIS_MODULE->name, priv->parent->name);
    return 0;
err:
    free_netdev(child);
    return err;
}

```

```

}

void __exit exit(void) {
    struct priv *priv = netdev_priv(child);
    if (priv->parent) {
        rtnl_lock();
        netdev_rx_handler_unregister(priv->parent);
        rtnl_unlock();
        LOG("unregister rx handler for %s\n", priv->parent->name);
    }
    unregister_netdev(child);
    free_netdev(child);
    LOG("module %s unloaded", THIS_MODULE->name);
}

module_init(init);
module_exit(exit);

MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_AUTHOR("Nikita Dorokhin");
MODULE_LICENSE("GPL v2");
MODULE_VERSION("2.1");

```

Перехват **входящего** трафика родительского интерфейса здесь осуществляется установкой нового обработчика (функция `handle_frame()`) входящих пакетов для созданного интерфейса, вызовом `netdev_rx_handler_unregister()`, который появился в API ядра начиная с 2.6.36 (ранее это приходилось делать другими способами). При передаче **исходящего** сокетного буфера в сеть (функция `start_xmit()`) мы просто подменяем в структуре сокетного буфера интерфейс, через который физически должна производиться отправка.

Работа с таким интерфейсом выглядит примерно следующим образом:

- на **любой** существующий и работоспособный сетевой интерфейс:

```

$ ip addr show dev p7p1
3: p7p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:9e:02:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global p7p1
    inet6 fe80::a00:27ff:fe9e:202/64 scope link
    valid_lft forever preferred_lft forever

```

- устанавливаем новый виртуальный интерфейс и конфигурируем его (на IP подсеть, отличную от исходной подсети интерфейса p7p1):

```

$ sudo insmod virt2.ko link=p7p1
$ sudo ifconfig virt0 192.168.50.2
$ ifconfig virt0
virt0    Link encap:Ethernet  HWaddr 08:00:27:9E:02:02
         inet addr:192.168.50.2  Bcast:192.168.50.255  Mask:255.255.255.0
         inet6 addr: fe80::a00:27ff:fe9e:202/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:27 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:0 (0.0 b)  TX bytes:5027 (4.9 KiB)

```

- самый простой способ создать **ответный** конец (вам ведь нужно как-то тестировать свою работу?) для такой (192.168.50.2/24) подсети **на другом хосте** LAN, это создать **алиасный IP** для сетевого интерфейса этого удалённого хоста, по типу:

```

$ sudo ifconfig vboxnet0:1 192.168.50.1
$ ifconfig
...

```

```
vboxnet0 Link encap:Ethernet HWaddr 0A:00:27:00:00:00
        inet addr:192.168.56.1 Bcast:192.168.56.255 Mask:255.255.255.0
        inet6 addr: fe80::800:27ff:fe00:0/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:223 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 b) TX bytes:36730 (35.8 KiB)

vboxnet0:1 Link encap:Ethernet HWaddr 0A:00:27:00:00:00
        inet addr:192.168.50.1 Bcast:192.168.50.255 Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
```

(Здесь показан сетевой интерфейс гипервизора виртуальных машин VirtualBox, но точно то же можно проделать и с интерфейсом любого физического устройства).

- теперь из вновь созданного виртуального интерфейса мы можем проверить прозрачность сети посылкой ICMP:

```
$ ping 192.168.50.1
PING 192.168.50.1 (192.168.50.1) 56(84) bytes of data.
64 bytes from 192.168.50.1: icmp_req=1 ttl=64 time=0.371 ms
64 bytes from 192.168.50.1: icmp_req=2 ttl=64 time=0.210 ms
64 bytes from 192.168.50.1: icmp_req=3 ttl=64 time=0.184 ms
64 bytes from 192.168.50.1: icmp_req=4 ttl=64 time=0.242 ms
^C
--- 192.168.50.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.184/0.251/0.371/0.074 ms
```

- и далее создать для удалённого хоста сессию ssh (по протоколу TCP) через новый виртуальный интерфейс:

```
$ ssh 192.168.50.2
Nasty PTR record "192.168.50.2" is set up for 192.168.50.2, ignoring
olej@192.168.50.2's password:
Last login: Tue Apr  3 10:21:28 2012 from 192.168.1.5
[olej@fedora16vm ~]$ uname -a
Linux fedora16vm.localdomain 3.3.0-8.fc16.i686 #1 SMP Thu Mar 29 18:33:55 UTC 2012 i686
i686 i386 GNU/Linux
[olej@fedora16vm ~]$ exit
logout
Connection to 192.168.50.2 closed.
$
```

Примечание: Чтобы не увеличивать сложность обсуждения, выше показан упрощённый пример модуля, который полностью **перехватывает** трафик родительского интерфейса, при этом он **замещает** родительский интерфейс (иначе нужно отдельно обрабатывать пакеты IP и пакеты разрешения адресов ARP). Среди архива примеров размещён и архив (virt-full), который корректно («прозрачно») анализирует адреса получателей и перенаправляет пакеты в нужное место.

Виртуальный сетевой интерфейс (созданный тем или иным способом) это очень мощный инструмент периода создания и отладки сетевых модулей, поэтому он заслуживает отдельного рассмотрения.

Протокол сетевого уровня

На этом уровне (L2) обеспечивается обработка таких протоколов, как: IP/IPv4/IPv6, IPX, ICMP, RIP, OSPF, ARP, или добавление оригинальных пользовательских протоколов. Для установки обработчиков сетевого уровня предоставляется API сетевого уровня (<linux/netdevice.h>):

```
struct packet_type {
    __be16 type; /* This is really htons(ether_type). */
    struct net_device *dev; /* NULL is wildcarded here */
};
```

```

    int (*func)(struct sk_buff*, struct net_device*, struct packet_type*, struct
net_device*);
    ...
    struct list_head list;
};
extern void dev_add_pack(struct packet_type *pt);
extern void dev_remove_pack(struct packet_type *pt);

```

Фактически, в протокольных модулях, как здесь, так и далее на транспортном уровне — мы должны **добавить фильтр**, через который проходят буфера сокетов **из входящего потока** интерфейса (исходящий поток реализуется проще, как показано в примерах ранее). Функция `dev_add_pack()` добавляет ещё один новый обработчик для пакетов заданного типа и выбранного интерфейса, реализуемый функцией `func()`. Функция **добавляет, но не замещает** существующий обработчик (в том числе и обработчик по умолчанию сетевой системы Linux). На обработку в функцию отбираются (попадают) те буфера сокетов, которые удовлетворяют критерию, заложенным в структуре `struct packet_type` (по типу протокола `type`, сетевому интерфейсу `dev`)

Примеры добавления собственных обработчиков сетевых протоколов находятся в каталоге архива `netproto`. Вот так может быть добавлен обработчик нового протокола сетевого уровня:

net_proto.c :

```

#include <linux/module.h>
#include <linux/init.h>
#include <linux/netdevice.h>

#define ERR(...) printk(KERN_ERR "! " __VA_ARGS__)
#define LOG(...) printk(KERN_INFO "! " __VA_ARGS__)

int test_pack_rcv(struct sk_buff *skb, struct net_device *dev,
                  struct packet_type *pt, struct net_device *odev) {
    LOG("packet received with length: %u\n", skb->len);
    kfree_skb(skb);
    return skb->len;
}

#define TEST_PROTO_ID 0x1234
static struct packet_type test_proto = {
    __constant_htons(ETH_P_ALL), // may be: __constant_htons(TEST_PROTO_ID),
    NULL,
    test_pack_rcv,
    (void*)1,
    NULL
};

static int __init my_init(void) {
    dev_add_pack(&test_proto);
    LOG("module loaded\n");
    return 0;
}

static void __exit my_exit(void) {
    dev_remove_pack(&test_proto);
    LOG(KERN_INFO "module unloaded\n");
}

module_init(my_init);
module_exit(my_exit);

```

```
MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_LICENSE("GPL v2");
```

Примечание: Самая большая сложность с подобными примерами — это то, какими средствами вы будете его тестировать для нестандартного протокола, когда операционная система, возможно, не знает такого сетевого протокола, и не имеет утилит обмена в таком протоколе...

Выполнение такого примера:

```
$ sudo insmod net_proto.ko
$ dmesg | tail -n6
module loaded
packet received with length: 74
packet received with length: 60
packet received with length: 66
packet received with length: 241
packet received with length: 52
$ sudo rmmod net_proto
```

В этом примере обработчик протокола перехватывает (фильтрует) **все** пакеты (константа ETH_P_ALL) на всех сетевых интерфейсах. В случае собственного протокола здесь должна бы быть константа TEST_PROTO_ID (но для такого случая нам нечем оттестировать модуль). Очень большое число идентификаторов протоколов (Ethernet Protocol ID's) находим в `<linux/if_ether.h>`, некоторые наиболее интересные из них, для примера:

```
#define ETH_P_LOOP    0x0060 /* Ethernet Loopback packet */
...
#define ETH_P_IP      0x0800 /* Internet Protocol packet */
...
#define ETH_P_ARP     0x0806 /* Address Resolution packet */
...
#define ETH_P_PAE     0x888E /* Port Access Entity (IEEE 802.1X) */
...
#define ETH_P_ALL     0x0003 /* Every packet (be careful!!!) */
...
```

Здесь же находим описание заголовка Ethernet пакета, который помогает в заполнении структуры `struct packet_type` :

```
struct ethhdr {
    unsigned char h_dest[ETH_ALEN]; /* destination eth addr */
    unsigned char h_source[ETH_ALEN]; /* source ether addr */
    __be16 h_proto; /* packet type ID field */
} __attribute__((packed));
```

Написанный нами пример модуля порождает ряд вопросов:

- Можно ли установить несколько обработчиков потока пакетов (для одного или различающихся типов протоколов type)?
- В каком порядке будут вызываться функции-обработчики при поступлении пакета?
- Как специфицировать один отдельный сетевой интерфейс, к которому должен применяться фильтр?
- Какую роль играет вызов `kfree_skb()` (в функции-фильтре обработки протокола `test_pack_rcv()`), и какой сокетный буфер (или его копия) при этом освобождается?

Для уяснения этих вопросов нам необходимо собрать на базе предыдущего другой пример (показаны только отличающиеся фрагменты кода):

net_proto2.c :

```

...
static int debug = 0;
module_param(debug, int, 0);

static char* link = NULL;
module_param(link, charp, 0);

int test_pack_rcv_1(struct sk_buff *skb, struct net_device *dev,
                    struct packet_type *pt, struct net_device *odev) {
    int s = atomic_read(&skb->users);
    kfree_skb(skb);
    if (debug > 0)
        LOG("function #1 - %p => users: %d->%d\n", skb, s, atomic_read(&skb->users));
    return skb->len;
}

int test_pack_rcv_2(struct sk_buff *skb, struct net_device *dev,
                    struct packet_type *pt, struct net_device *odev) {
    int s = atomic_read(&skb->users);
    kfree_skb(skb);
    if (debug > 0)
        LOG("function #2 - %p => users: %d->%d\n", skb, s, atomic_read(&skb->users));
    return skb->len;
}

static struct packet_type
test_proto1 = {
    __constant_htons(ETH_P_IP),
    NULL,
    test_pack_rcv_1,
    (void*)1,
    NULL
},
test_proto2 = {
    __constant_htons(ETH_P_IP),
    NULL,
    test_pack_rcv_2,
    (void*)1,
    NULL
};

static int __init my_init(void) {
    if (link != NULL) {
        struct net_device *dev = __dev_get_by_name(&init_net, link);
        if (NULL == dev) {
            ERR("%s: illegal link", link);
            return -EINVAL;
        }
        test_proto1.dev = test_proto2.dev = dev;
    }
    dev_add_pack(&test_proto1);
    dev_add_pack(&test_proto2);
    if (NULL == test_proto1.dev) LOG("module %s loaded for all links\n", THIS_MODULE-
>name);
    else LOG("module %s loaded for link %s\n", THIS_MODULE->name, link);
    return 0;
}

static void __exit my_exit(void) {

```



```

    if (test_proto2.dev != NULL) dev_put(test_proto2.dev);
    if (test_proto1.dev != NULL) dev_put(test_proto1.dev);
    dev_remove_pack(&test_proto2);
    dev_remove_pack(&test_proto1);
    LOG("module %s unloaded\n", THIS_MODULE->name);
}

```

Здесь, собственно, выполняется абсолютно то же, что и в предыдущем варианте, но мы устанавливаем одновременно **два** новых дополнительных обработчика для пакетов IPv4 (ETH_P_IP), причём устанавливаем именно в последовательности: `test_pack_rcv_1()`, а затем `test_pack_rcv_2()`. Смотрим что из этого получается... Загружаем модуль:

```

$ sudo insmod net_proto2.ko link=p7p1
$ dmesg | tail -n1
[ 403.339591] ! module net_proto2 loaded for link p7p1

```

И далее (конфигурировав интерфейс на адрес 192.168.56.3) с другого хоста выполняем:

```

$ ping -c3 192.168.56.3
PING 192.168.56.3 (192.168.56.3) 56(84) bytes of data.
64 bytes from 192.168.56.3: icmp_req=1 ttl=64 time=0.668 ms
64 bytes from 192.168.56.3: icmp_req=2 ttl=64 time=0.402 ms
64 bytes from 192.168.56.3: icmp_req=3 ttl=64 time=0.330 ms
--- 192.168.56.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.330/0.466/0.668/0.147 ms

```

В системном журнале мы найдём:

```

$ dmesg | tail -n7
[ 403.339591] ! module net_proto2 loaded for link p7p1
[ 420.305824] ! function #2 - eb6873c0 => users: 2->1
[ 420.305829] ! function #1 - eb6873c0 => users: 2->1
[ 421.306302] ! function #2 - eb687c00 => users: 2->1
[ 421.306308] ! function #1 - eb687c00 => users: 2->1
[ 422.306289] ! function #2 - eb687180 => users: 2->1
[ 422.306294] ! function #1 - eb687180 => users: 2->1

```

Отсюда видно, что обработчики срабатывают в порядке обратном их установке (который установленный позже — тот срабатывает раньше), но срабатывают **все**. Они получают в качестве аргумента адрес одной и той же копии буфера сокета. Относительно `kfree_skb()` мы должны обратиться к исходному коду реализации ядра (файл `net/core/skbuff.c`):

```

void kfree_skb(struct sk_buff *skb) {
    if (unlikely(!skb))
        return;
    if (likely(atomic_read(&skb->users) == 1))
        smp_rmb();
    else if (likely(!atomic_dec_and_test(&skb->users)))
        return;
    trace_kfree_skb(skb, __builtin_return_address(0));
    __kfree_skb(skb);
}

```

Вызов `kfree_skb()` будет реально освобождать буфер сокета только в случае `skb->users == 1`, при всех остальных значениях он будет только декрементировать `skb->users` (счётчик использования). Для уточнения происходящего мы можем закомментировать вызовы `kfree_skb()` в двух обработчиках выше, и наблюдать при этом:

```

$ dmesg | tail -n7
[11373.754524] ! module net_proto2 loaded for link p7p1
[11398.930057] ! function #2 - ed3dfc00 => users: 2->2

```

```
[11398.930061] ! function #1 - ed3dfc00 => users: 3->3
[11399.929838] ! function #2 - ed3dfb40 => users: 2->2
[11399.929843] ! function #1 - ed3dfb40 => users: 3->3
[11400.929522] ! function #2 - ed3df480 => users: 2->2
[11400.929527] ! function #1 - ed3df480 => users: 3->3
```

Если функция обработчик не декрементирует `skb->users` по завершению вызовом `kfree_skb()`, то, после окончательного вызова обработки по умолчанию, буфер сокета не будет уничтожен, и в системе будет наблюдаться **утечка памяти**. Она незначительная, но неумолимая, и, в конце концов, приведёт к краху системы. Проверку на отсутствие утечки памяти (по этой причине, или по любой иной) предлагается проверить приёмом по сетевому каналу значительного объёма данных (несколько гигабайт), с фиксацией состояния памяти командой `free`. Для этого можно с успехом использовать утилиту `nc` (`network cat`):

— на тестируемом узле запустить скрипт `./client`:

```
LIMIT=10000000
for ((a=1; a <= LIMIT ; a++))
do
    rm -a z.txt
    nc 192.168.56.1 12345 > z.txt
    sleep 1
done
```

— на стороннем узле сети запустить скрипт `./server`

```
dd if=/dev/zero of=z.txt bs=1M count=1
LIMIT=10000000
for ((a=1; a <= LIMIT ; a++))
do
    cat z.txt | nc -l 12345
done
```

(здесь в скриптах: 192.168.56.1 — IP адрес узла где работает `server`, 12345 — номер TCP порта, через который `nc` предписывается передавать данные, при желании, можно использовать и UDP, добавив к `nc` опцию `-u`)

— и подождать некоторое время, порядка 10-20 минут:

```
$ free
```

	total	used	free	shared	buffers	cached
Mem:	768084	260636	507448	0	23392	129108
-/+ buffers/cache:		108136	659948			
Swap:	1540092	0	1540092			

```
$ ifconfig p7p1
p7p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.56.3 netmask 255.255.255.255 broadcast 192.168.56.3
    inet6 fe80::a00:27ff:fe08:9abd prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:08:9a:bd txqueuelen 1000 (Ethernet)
    RX packets 2017560 bytes 3048762624 (2.8 GiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 398156 bytes 26283474 (25.0 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Ещё раз о виртуальном интерфейсе

Ранее рассматривалось создание виртуального сетевого интерфейса (использующего трафик реального физического, родительского) средствами `netdev_rx_handler_register()`. Но такой способ не свободен от некоторых недостатков: он появился хронологически достаточно поздно, в API ядра начиная с версии 2.6.36, не со всяким ядром он может быть использован,

кроме того, он в некоторой степени громоздкий и путанный. Но это же можно сделать и другим способом, не зависящим от версий используемого ядра, используя протокольные механизмы сетевого уровня (L3). Кроме того, это хороший реалистичный пример использования протокольных фильтров. Пример подобной реализации расположен в архиве virt-proto.

Примечание: В названном архиве представлены два варианта модуля: упрощённый модуль virtl.ko (lite вариант), интерфейс (virt0) которого **замещает** родительский сетевой интерфейс, и полный вариант virt.ko, который анализирует сетевые фреймы (и ARP и IP4 протоколов), и затрагивает только трафик, к его интерфейсу относящийся. Разница состоит в том, что на время загрузки упрощённого модуля работа родительского интерфейса прекращается, а при загрузке полного варианта оба интерфейса работают одновременно и независимо. Но код полного модуля гораздо более громоздкий, а для понимания принципов он ничего не добавляет. Ниже детально рассмотрен упрощённый вариант, не скрывающий принципы, и только позже мы в пару слов коснёмся полного варианта: код его и протокол испытаний приведены в архиве, поэтому его детализация не вызывает сложностей.

virtl.c :

```
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/inetdevice.h>
#include <linux/moduleparam.h>
#include <net/arp.h>
#include <linux/ip.h>

#define ERR(...) printk(KERN_ERR "! " __VA_ARGS__)
#define LOG(...) printk(KERN_INFO "! " __VA_ARGS__)
#define DBG(...) if (debug != 0) printk(KERN_INFO "! " __VA_ARGS__)

static char* link = "eth0";
module_param(link, charp, 0);

static char* ifname = "virt";
module_param(ifname, charp, 0);

static int debug = 0;
module_param(debug, int, 0);

static struct net_device *child = NULL;
static struct net_device_stats stats; // статическая таблица статистики интерфейса
static u32 child_ip;

struct priv {
    struct net_device *parent;
};

static char* strIP(u32 addr) { // диагностика IP в точечной нотации
    static char saddr[MAX_ADDR_LEN];
    sprintf(saddr, "%d.%d.%d.%d",
            (addr) & 0xFF, (addr >> 8) & 0xFF,
            (addr >> 16) & 0xFF, (addr >> 24) & 0xFF
    );
    return saddr;
}

static int open(struct net_device *dev) {
    struct in_device *in_dev = dev->ip_ptr;
```

```

    struct in_ifaddr *ifa = in_dev->ifa_list;      /* IP ifaddr chain */
    LOG("%s: device opened", dev->name);
    child_ip = ifa->ifa_address;
    netif_start_queue(dev);
    if (debug != 0) {
        char sdbg[40] = "";
        sprintf(sdbg, "%s:", strIP(ifa->ifa_address));
        strcat(sdbg, strIP(ifa->ifa_mask));
        DBG("%s: %s", dev->name, sdbg);
    }
    return 0;
}

static int stop(struct net_device *dev) {
    LOG("%s: device closed", dev->name);
    netif_stop_queue(dev);
    return 0;
}

static struct net_device_stats *get_stats(struct net_device *dev) {
    return &stats;
}

// передача фрейма
static netdev_tx_t start_xmit(struct sk_buff *skb, struct net_device *dev) {
    struct priv *priv = netdev_priv(dev);
    stats.tx_packets++;
    stats.tx_bytes += skb->len;
    skb->dev = priv->parent;    // передача в родительский (физический) интерфейс
    skb->priority = 1;
    dev_queue_xmit(skb);
    DBG("tx: injecting frame from %s to %s with length: %u",
        dev->name, skb->dev->name, skb->len);
    return 0;
}

static struct net_device_ops net_device_ops = {
    .ndo_open = open,
    .ndo_stop = stop,
    .ndo_get_stats = get_stats,
    .ndo_start_xmit = start_xmit,
};

// приём фрейма
int pack_parent(struct sk_buff *skb, struct net_device *dev,
                struct packet_type *pt, struct net_device *odev) {
    skb->dev = child;          // передача фрейма в виртуальный интерфейс
    stats.rx_packets++;
    stats.rx_bytes += skb->len;
    DBG("tx: injecting frame from %s to %s with length: %u",
        dev->name, skb->dev->name, skb->len);
    kfree_skb(skb);
    return skb->len;
}

static struct packet_type proto_parent = {
    __constant_htons(ETH_P_ALL), // перехватывать все пакеты: ETH_P_ARP & ETH_P_IP
    NULL,
    pack_parent,

```

```

        (void*)1,
        NULL
    };

int __init init(void) {
    void setup(struct net_device *dev) { // вложенная функция (GCC)
        int j;
        ether_setup(dev);
        memset(netdev_priv(dev), 0, sizeof(struct priv));
        dev->netdev_ops = &net_device_ops;
        for (j = 0; j < ETH_ALEN; ++j) // заполнить MAC фиктивным адресом
            dev->dev_addr[j] = (char)j;
    }
    int err = 0;
    struct priv *priv;
    char ifstr[40];
    sprintf(ifstr, "%s%s", ifname, "%d");
    child = alloc_netdev(sizeof(struct priv), ifstr, setup);
    if (child == NULL) {
        ERR("%s: allocate error", THIS_MODULE->name); return -ENOMEM;
    }
    priv = netdev_priv(child);
    priv->parent = __dev_get_by_name(&init_net, link); // родительский интерфейс
    if (!priv->parent) {
        ERR("%s: no such net: %s", THIS_MODULE->name, link);
        err = -ENODEV; goto err;
    }
    if (priv->parent->type != ARPHRD_ETHER && priv->parent->type != ARPHRD_LOOPBACK) {
        ERR("%s: illegal net type", THIS_MODULE->name);
        err = -EINVAL; goto err;
    }
    memcpy(child->dev_addr, priv->parent->dev_addr, ETH_ALEN);
    memcpy(child->broadcast, priv->parent->broadcast, ETH_ALEN);
    if ((err = dev_alloc_name(child, child->name))) {
        ERR("%s: allocate name, error %i", THIS_MODULE->name, err);
        err = -EIO; goto err;
    }
    register_netdev(child); // зарегистрировать новый интерфейс
    proto_parent.dev = priv->parent;
    dev_add_pack(&proto_parent); // установить обработчик фреймов для родителя
    LOG("module %s loaded", THIS_MODULE->name);
    LOG("%s: create link %s", THIS_MODULE->name, child->name);
    return 0;
err:
    free_netdev(child);
    return err;
}

void __exit exit(void) {
    dev_remove_pack(&proto_parent); // удалить обработчик фреймов
    unregister_netdev(child);
    free_netdev(child);
    LOG("module %s unloaded", THIS_MODULE->name);
    LOG("=====");
}

module_init(init);
module_exit(exit);

```

```
MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_LICENSE("GPL v2");
MODULE_VERSION("3.6");
```

От рассмотренных уже ранее примеров код отличается только тем, что после регистрации нового сетевого интерфейса (`virt0`) он выполняет вызов `dev_add_pack()`, предварительно установив в структуре `packet_type` поле `dev` на указатель родительского интерфейса: только с этого интерфейса входящий трафик будет перехватываться определённой в структуре функцией `pack_parent()`. Эта функция фиксирует статистику интерфейса и, самое главное, **подменяет** в сокетном буфере указатель родительского интерфейса на виртуальный. Обратная подмена (виртуального на физический) происходит в функции отправки фрейма `start_xmit()`, но это не отличается от того, что мы видели ранее. Вот как это работает:

- на тестируемом компьютере загружаем модуль и конфигурируем его:

```
$ ip address
...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:52:b9:e0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.21/24 brd 192.168.1.255 scope global eth0
    inet6 fe80::a00:27ff:fe52:b9e0/64 scope link
        valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:0f:13:6d brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.102/24 brd 192.168.56.255 scope global eth1
    inet6 fe80::a00:27ff:fe0f:136d/64 scope link
        valid_lft forever preferred_lft forever
$ sudo insmod ./virt.ko link=eth1 debug=1
$ sudo ifconfig virt0 192.168.50.19
$ sudo ifconfig virt0
virt0    Link encap:Ethernet  HWaddr 08:00:27:0f:13:6d
         inet addr:192.168.50.19  Bcast:192.168.50.255  Mask:255.255.255.0
         inet6 addr: fe80::a00:27ff:fe0f:136d/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:46 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:0 (0.0 B)  TX bytes:8373 (8.1 KiB)
```

(показана статистика с нулевым числом принятых байт на интерфейсе).

- на тестирующем компьютере создаём алиасный IP для тестируемой подсети (192.168.50.0/24) и можем осуществлять трафик на созданный интерфейс:

```
$ sudo ifconfig vboxnet0:1 192.168.50.1
$ ping 192.168.50.19
PING 192.168.50.19 (192.168.50.19) 56(84) bytes of data.
64 bytes from 192.168.50.19: icmp_req=1 ttl=64 time=0.627 ms
64 bytes from 192.168.50.19: icmp_req=2 ttl=64 time=0.305 ms
64 bytes from 192.168.50.19: icmp_req=3 ttl=64 time=0.326 ms
^C
--- 192.168.50.19 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.305/0.419/0.627/0.148 ms
```

- на этом же (тестирующем) компьютере (ответной стороне) очень содержательно наблюдать трафик (в отдельном терминале), фиксируемый снифером `tcpdump`:

```

$ sudo tcpdump -i vboxnet0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on vboxnet0, link-type EN10MB (Ethernet), capture size 65535 bytes
...
18:41:01.740607 ARP, Request who-has 192.168.50.19 tell 192.168.50.1, length 28
18:41:01.741104 ARP, Reply 192.168.50.19 is-at 08:00:27:0f:13:6d (oui Unknown), length 28
18:41:01.741116 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 1, length 64
18:41:01.741211 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 1, length 64
18:41:02.741164 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 2, length 64
18:41:02.741451 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 2, length 64
18:41:03.741163 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 3, length 64
18:41:03.741471 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 3, length 64
18:41:06.747701 ARP, Request who-has 192.168.50.1 tell 192.168.50.19, length 28
18:41:06.747715 ARP, Reply 192.168.50.1 is-at 0a:00:27:00:00:00 (oui Unknown), length 28

```

Теперь коротко, в два слова, о том, как сделать полноценный виртуальный интерфейс, работающий только со своим трафиком, и не нарушающий работу родительского интерфейса (то, что делает полная версия модуля в архиве). Для этого необходимо:

- объявить **два** отдельных обработчика протоколов (для протоколов разрешения имён ARP и собственно для протокола IP):

```

// обработчик фреймов ETH_P_ARP
int arp_pack_rcv(struct sk_buff *skb, struct net_device *dev,
                 struct packet_type *pt, struct net_device *odev) {
    ...
    return skb->len;
}

static struct packet_type arp_proto = {
    __constant_htons(ETH_P_ARP),
    NULL,
    arp_pack_rcv, // фильтр протокола ETH_P_ARP
    (void*)1,
    NULL
};

// обработчик фреймов ETH_P_IP
int ip4_pack_rcv(struct sk_buff *skb, struct net_device *dev,
                 struct packet_type *pt, struct net_device *odev) {
    ...
    return skb->len;
}

static struct packet_type ip4_proto = {
    __constant_htons(ETH_P_IP),
    NULL,
    ip4_pack_rcv, // фильтр протокола ETH_P_IP
    (void*)1,
    NULL
};

```

- и оба их зарегистрировать в функции инициализации модуля:

```

    arp_proto.dev = ip4_proto.dev = priv->parent; // перехват только с родительского
интерфейса
    dev_add_pack(&arp_proto);
    dev_add_pack(&ip4_proto);

```

- каждый из обработчиков должен осуществлять **подмену** интерфейса только для тех фреймов, IP получателя которых совпадает с IP интерфейса ...

- а два отдельных обработчика удобны тем, что заголовки фреймов ARP и IP имеют совершенно разный формат, и выделять IP назначения в них приходится по-разному (весь полный код показан в архиве примера).

Используя такой полновесный модуль, можно открыть к хосту, например, одновременно две параллельные сессии SSH на разные интерфейсы, родительский и виртуальный (использующие разные IP), которые будут в параллель реально использовать единый общий физический интерфейс:

```

$ ssh olej@192.168.50.17
olej@192.168.50.17's password:
Last login: Mon Jul 16 15:52:16 2012 from 192.168.1.9
...
$ ssh olej@192.168.56.101
olej@192.168.56.101's password:
Last login: Mon Jul 16 17:29:57 2012 from 192.168.50.1
...
$ who
olej      tty1          2012-07-16 09:29 (:0)
olej      pts/0          2012-07-16 09:33 (:0.0)
olej      pts/1          2012-07-16 12:22 (192.168.1.9)
olej      pts/4          2012-07-16 15:52 (192.168.1.9)
olej      pts/6          2012-07-16 17:29 (192.168.50.1)
olej      pts/7          2012-07-16 17:31 (192.168.56.1)

```

Последняя показанная команда (who) выполняется уже в сессии SSH, то есть на том самом удалённом хосте, к которому и фиксируется командой два **независимых** подключения из двух различных подсетей (последние две строки вывода), которые на самом деле представляют один хост.

Протокол транспортного уровня

На этом уровне (L4) обеспечивается обработка таких протоколов, как: UDP, TCP, SCTP, DCCP ... их число возрастает. Протоколы транспортного уровня (протоколы IP) описаны в <linux/in.h> :

```

/* Standard well-defined IP protocols. */
enum {
    IPPROTO_IP = 0, /* Dummy protocol for TCP */
    IPPROTO_ICMP = 1, /* Internet Control Message Protocol */
    IPPROTO_IGMP = 2, /* Internet Group Management Protocol */
    ...
    IPPROTO_TCP = 6, /* Transmission Control Protocol */
    ...
    IPPROTO_UDP = 17, /* User Datagram Protocol */
    ...
    IPPROTO_SCTP = 132, /* Stream Control Transport Protocol */
    ...
    IPPROTO_RAW = 255, /* Raw IP packets */
}

```


Для установки обработчика протоколов транспортного уровня существует API `<net/protocol.h>` :

```
struct net_protocol {
    // This is used to register protocols
    int (*handler)(struct sk_buff *skb);
    void (*err_handler)(struct sk_buff *skb, u32 info);
    int (*gso_send_check)(struct sk_buff *skb);
    struct sk_buff *(*gso_segment)(struct sk_buff *skb, int features);
    struct sk_buff *(*gro_receive)(struct sk_buff **head, struct sk_buff *skb);
    int (*gro_complete)(struct sk_buff *skb);
    unsigned int no_policy:1,
               netns_ok:1;
};
int inet_add_protocol(const struct net_protocol *prot, unsigned char num);
int inet_del_protocol(const struct net_protocol *prot, unsigned char num);
```

Здесь 2-й параметр вызова функций (num) как раз и есть константа набора констант вида `IPPROTO_*`.

Эта схема в общих чертах напоминает то, как это же делалось на сетевом уровне: каждый пакет проходит через функцию-фильтр, где мы можем анализировать или изменять отдельные поля сокета буфера, отображающего пакет.

Пример модуля, устанавливающего протокол:

trn_proto.c :

```
#include <linux/module.h>
#include <linux/init.h>
#include <net/protocol.h>

int test_proto_rcv(struct sk_buff *skb) {
    printk(KERN_INFO "Packet received with length: %u\n", skb->len);
    return skb->len;
}

static struct net_protocol test_proto = {
    .handler = test_proto_rcv,
    .err_handler = 0,
    .no_policy = 0,
};

#ifdef PROTO IPPROTO_ICMP
#ifdef PROTO IPPROTO_TCP
#define PROTO IPPROTO_RAW
static int __init my_init(void) {
    int ret;
    if ((ret = inet_add_protocol(&test_proto, PROTO)) < 0) {
        printk(KERN_INFO "proto init: can't add protocol\n");
        return ret;
    }
    printk(KERN_INFO "proto module loaded\n");
    return 0;
}

static void __exit my_exit(void) {
    inet_del_protocol(&test_proto, PROTO);
    printk(KERN_INFO "proto module unloaded\n");
}

module_init(my_init);
```

```

module_exit(my_exit);

MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_LICENSE("GPL v2");

```

Вот как будет выглядеть работа модуля для протокола IPPROTO_RAW:

```

$ sudo insmod trn_proto.ko
$ lsmod | head -n2
Module                Size  Used by
trn_proto              780    0
$ cat /proc/modules | grep proto
trn_proto 780 0 - Live 0xf9a26000
$ ls -R /sys/module/trn_proto
/sys/module/trn_proto:
holders  initstate  notes  refcnt  sections  srcversion
...
$ sudo rmmod trn_proto
$ dmesg | tail -n60 | grep -v ^audit
proto module loaded
proto module unloaded

```

Но если вы попытаетесь установить (добавить!) обработчик для уже **обрабатываемого** (установленного) протокола (например, IPPROTO_TCP), то получите ошибку:

```

$ sudo insmod trn_proto.ko
insmod: error inserting 'trn_proto.ko': -1 Operation not permitted
$ dmesg | tail -n60 | grep -v ^audit
proto init: can't add protocol
$ lsmod | grep proto
$

```

Здесь возникает уже названная раньше сложность:

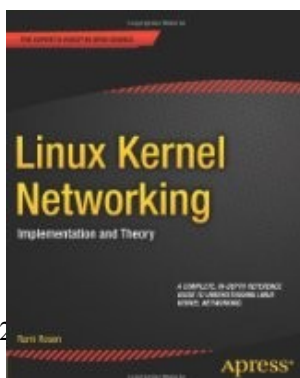
- Если вы планируете обрабатывать новый (или не использующийся в системе) протокол, то для его тестирования в системе нет инструментов, и прежде нужно подумать о том, чтобы создать тестовые приложения прикладного уровня.
- Если пытаться моделировать работу нового протокола под видом уже существующего (например, IPPROTO_UDP), то вам прежде понадобится удалить существующий обработчик, чем можно радикально нарушить работоспособность системы (например, для IPPROTO_UDP разрушить систему разрешения доменных имён DNS).

Источники использованной информации

[1] Rami Rosen, «Linux Kernel Networking: Implementation and Theory», New York, «Apress», 2014, ISBN: 978-1-4302-6196-4, p. 612 —

<http://www.foxebook.net/linux-kernel-networking-implementation-and-theory/>

(там же книгу можно свободно скачать)



[2] С. Сэтчэлл, Х. Клиффорд, «Linux IP Stacks в комментариях», К.: «ДиаСофт», 2001, ISBN: 966-7393-83-6, 288 стр.



[3] «Linux Network Configuration» —

<http://www.yolinux.com/TUTORIALS/LinuxTutorialNetworking.html#CONFIGFILES>

[4] Олег Цилюрик, статьи в редакциях разных лет (различающиеся):

— «Сеть IP - когда писать программы лень»

<http://smartbox.jinr.ru/qnx.org.ru/article10.html>, 2002

<http://rus-linux.net/MyLDP/algol/Simple-TCP-programming.html>, 2012

— «Сервер TCP / IP ... много серверов хороших и разных»

http://www.cta.ru/online/online_progr-nets.htm — журнал «СТА» («Современные Технологии Автоматизации»), Москва, 2003

<http://rus-linux.net/MyLDP/algol/analiz-variantov-realizacii-TCP-IP-servera-01.html>, 2012

Эти же статьи в книге : Д. Алексеев, Е. Видревич, А. Волков, Е. Горошко, М. Горчак, Р. Жавнис, Д. Сошин, О. Цилюрик, А. Чиликин, Практика работы с QNX, М.: КомБук, 2004, 432 стр., ISBN: 5-94740-009-X



Книгу можно скачать здесь: <http://9knig.ru/os/19431-praktika-raboty-s-qnx..html>

[5] LXR (Linux Cross-Referencer) ресурсы перекрёстного анализа исходных кодов ядра Linux

<http://lxr.free-electrons.com/source/>

<http://lxr.linux.no>

<http://lxr.missinglinkelectronics.com/linux>

<http://lxr.oss.org.cn/>

[6] Олег Цилюрик, Расширения ядра Linux: драйверы и модули, ВНУ-СПб, март 2023, ISBN: 978-5-9775-1719-5, 688 стр.



За границами привычного Интернет

«Нот - 7, клавиш рояля - 88, пальцев пианиста - 10, и наука пока не может объяснить эти цифры», - сообщил нашему корреспонденту пьяный сторож консерватории.

Обычно обсуждение завершает изложение некоторого предмета, как его итог. Но в этом разделе мы поступим в точности до наоборот: начнём с обсуждения мотивации возникновения целой грозды экспериментальных проектов, возникших за последние годы **над** транспортным механизмом Интернет... и только затем выборочно рассмотрим обхорно некоторые из таких проектов.

Здесь будет рассмотрено несколько альтернативных проектов, выбранных достаточно произвольно, но представляющих те тенденции, которые отчётливо проявляются в большинстве новых сетевых проектах. Иногда высказывается мнение что такие проекты продвигаются хакерами, злоумышленниками в сети, и/или людьми параноидально озабоченными безопасностью. Если это и так, то в самой своей малой части. А в оставшейся, большей части смыслов, и это мнение разделяется автором — направления развития таких проектов это «Интернет завтрашнего дня». Поэтому далее мы и рассмотрим подробно несколько из таких проектов.

Обсуждение

Классический Интернет представляет собой мощную организованную систему. Но некоторые пользователи и разработчики считают что он заорганизован, управляемый разнообразными комитетами и комиссиями, возникшими за 4 десятилетия, и с имеющими место ограничениями, которые начинают накладывать государственные законодательные органы в рамках своих территориальных компетенций.

Обширный управленческий аппарат, помимо всего прочего, несёт в себе потенциал мелких но частых сбоев в работе за счёт несогласованности и ошибочности действий.

Но развивается целый ряд альтернативных проектов, использующих только транспортный механизм ТСП/IP сети, но не требующих его централизованного управления и его управляющих органов (комиссий и комитетов). Они функционируют **над** транспортным механизмом сети ТСП/IP, используя этот транспортный механизм Интернет, но выходя где-то за рамки и соглашения Интернет. Поэтому я в этой части назвал их (совершенно условно) «надсистемы».

Альтернативные проекты апробируют модели «самонастраивающихся» механизмов конфигурирования и функционирования сети.

Альтернативные проекты носят характер экспериментальных разработок, но они концентрируются вокруг достаточно ограниченного числа основных направлений развития. Поэтому их интересно и полезно знать, хотя бы в форме отдельных конкретных реализаций. При множестве экспериментальных проектов, которые постоянно возникают новые и угасают существующие, их тематики группируются вокруг относительно немногочисленных целей:

- Охрана приватности. Такие проекты направлены не только на защищённость самого содержимого сетевых обменов, но и на анонимизацию получателя, целевых ресурсов, и сокрытие направлений и самого факта осуществления трафика. Эту деятельность некоторые рассматривают как расширение сферы охранения персональной информации.
- Децентрализация. За долгие годы развития Интернет сложилась развитая инфраструктура учреждений, организаций и комитетов, обеспечивающих деятельность сети: провайдеры, хостеры, регистраторы... Они обеспечивают слаженное функционирование разных сторон сети, но, иногда, и вносят избыточную заорганизованность. Кроме того, их деятельность коммерческая и часто услуги

довольно дорого стоят потребителям. В этой части экспериментальные проекты пытаются исключить избыточные регламентирующие звенья.

- Живучесть. Во многом, из-за определённой централизованности организации Интернет, при нарушении согласованности действий различных организующих сторон, возможны перебои работы сети, или даже полное нарушение её работоспособности, на отдельных территориальных, государственных, ведомственных или других сегментах сети. Эксперименты в части адаптивных изменений топологии сети и направлены на уменьшение подобных рисков.
- Упрощение настроек. Много десятилетнее постоянное усложнение фрагментов сети (локальных сетей в составе Интернет) привело к большой сложности их конфигураций, и требует на сегодня даже в средней руки организациях содержать сетевых администраторов как отдельную профессиональную группу. Экспериментальные проекты пытаются двигаться в сторону самоорганизующихся архитектур, которые гибко подключаются к существующим участникам сети, выбирают оптимальные маршруты и самонастраиваются внутри сети.

Далее рассмотрены выборочно только несколько альтернативных направлений развития, для того, чтобы составить поверхностное представление о **тенденциях** движения. Проекты для рассмотрения выбраны достаточно произвольно, но то что показано, кроме других критериев их отбора — это не уровень сырых экспериментов, а вполне работоспособные на сегодня реализации.

Мы не станем слишком углубляться в каждый из проектов, чтобы не слишком засорять изложение. Многие реализационные детали ниже сознательно опущены. Описание строится отчётливо со следующей направленностью: если кто хочет попробовать какой-то из проектов «руками» — то здесь представлено достаточно материала чтобы его развернуть и начать использовать... Всё остальное — в документации и в показанных в конце текста источниках информации.

Стек TOR

Проект TOR (The Onion Router) — свободное и открытое программное обеспечение для реализации второго (V2) и третьего (V3) поколения так называемой луковичной маршрутизации: ваши данные — это сердцевина луковицы, а их защита — слои вокруг. Это такая адаптивная система прокси серверов, которая позволяющая устанавливать анонимное сетевое соединение, защищённое от прослушивания²⁰.

Разработка системы началась в 1995 году. К началу 2000-х исходный код был опубликован свободной лицензией. Написана преимущественно на языке С. В октябре 2002 года впервые была развёрнута сеть маршрутизаторов, которая к концу 2003 года насчитывала около десяти сетевых узлов. По состоянию на 2016 год, по оценкам, число участников сети (включая ботов), превысило 2 миллиона.

Для анонимизации Тог, также как прокси и VPN, пропускает трафик через промежуточные серверы. Но Только в случае с Тог их не один, а три, и называется они узлами. Каждый пакет данных, передаваемый системой узлов TOR, проходит через три различных различных узла, которые выбираются случайным образом в начале сеанса, и могут быть изменены пользователем по ходу прохождения сеанса. Перед отправкой пакет последовательно шифруется тремя ключами: сначала — для третьего узла, потом — для второго и в конце — для первого. Когда первый узел получает пакет, он расшифровывает «верхний» слой шифра (аналогия с тем, как чистят луковицу) откуда и узнаёт, куда отправить пакет дальше. Второй и третий сервер поступают также. После 3-го прокси-сервера пакет поступает получателю в открытом виде.

20 В генерируемых Интернет-публикой обсуждениях у Тог «дурная репутация» — постоянно активируются слухи о компрометации Тог. Но за все 25 лет эксплуатации проекта не было указано ни одного конкретного случая вскрытия Тог обменов, всё остаётся только на уровне гипотетических предположений. У автора складывается устойчивое мнение (только личное мнение), что слухи о компрометации Тог возбуждаются именно самими теми кругами, защищённость от которых он и призван обеспечен: лицами, учреждениями и ведомствами.

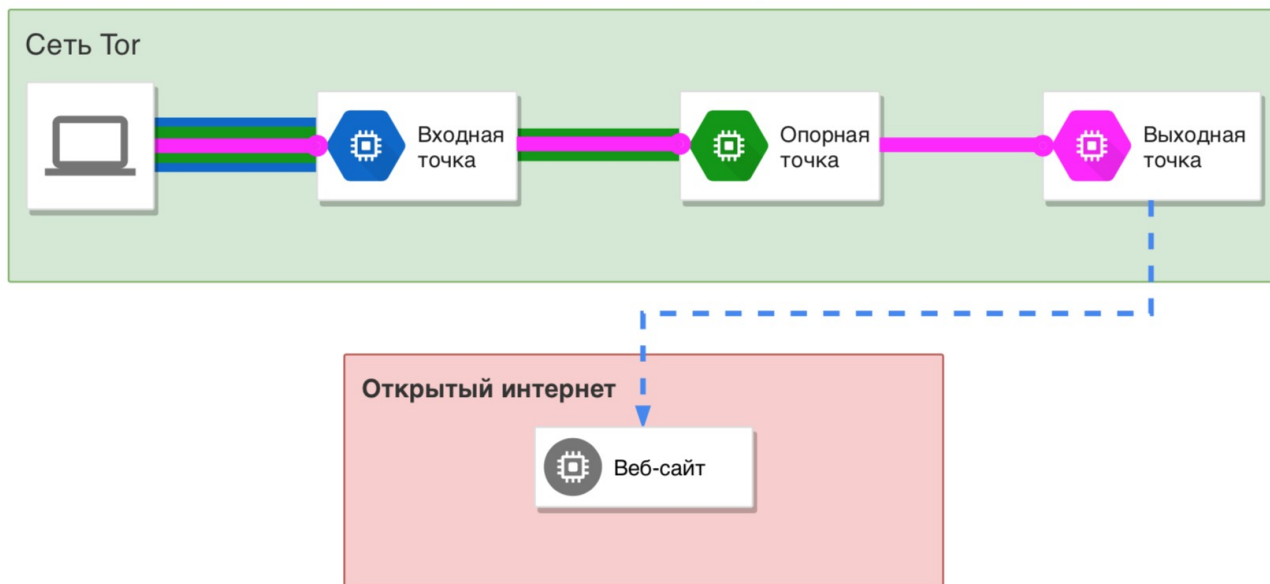


Рис. 5.1 Сеть TOR

TOR как прокси для любых служб

Для большинства конечных пользователей проект TOR известен и представляется как «законченное изделие» TOR-браузер, который произведен и регулярно обновляется на базе FireFox как отдельное изделие с 2008 года. И как конечный потребитель вы вполне можете, на этом уровне, использовать TOR-браузер, не углубляясь в структуру проекта. Но «под капотом» TOR-браузера (достаточно простого изделия) работает мощный сетевой стек TOR.

На самом деле, проект TOR гораздо шире в своих задумках и возможностях. И если нас интересует техническая сторона вопроса, то интерес представляет именно TOR как SOCKS-прокси — и вот этот базовый пакет Linux:

```
$ aptitude show tor
Пакет: tor
Версия: 0.4.6.10-1
Новый: да
Состояние: установлен
Установлен автоматически: да
Приоритет: необязательный
Раздел: universe/net
Сопровождающий: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Архитектура: amd64
Размер в распакованном виде: 5.400 k
Зависит: libc6 (>= 2.34), libcap2 (>= 1:2.10), libevent-2.1-7 (>= 2.1.8-stable), liblzma5 (>= 5.1.1alpha+20120614), libseccomp2 (>= 0.0.0-20120605), libssl3 (>= 3.0.0~alpha1), libsystemd0, libzstd1 (>= 1.4.0), zlib1g (>= 1:1.1.4), adduser, lsb-base
Рекомендует: logrotate, tor-geoipdb, torsocks
Предлагает: mixmaster, torbrowser-launcher, socat, apparmor-utils, nux, obfs4proxy
Конфликтует: libssl0.9.8 (< 0.9.8g-9)
Описание: анонимизирующая сеть работающая поверх TCP
Tor — система анонимного взаимодействия с установкой соединений и малым временем задержки.
```

Пользователи выбирают начало пути по сети узлов, и согласовывают «виртуальные цепи» через сеть, в которых каждому узлу известен предыдущий и следующий узел, а все остальные — не известны. Трафик, передаваемый по цепи, расшифровывается симметричным ключом в каждом узле, из которого получается информация о следующем узле.

Фактически, Tor позволяет создавать распределённую сеть ретранслирующих узлов. Пользователи пропускают свои TCP-данные (веб-трафик, ftp, ssh и т. д.) через цепь таких узлов-ретрансляторов, так что получателям, наблюдателям и самим маршрутизаторам становится трудно отследить источник и цель потока данных.

Этот пакет по умолчанию включает только клиентскую часть, но его можно настроить также для работы в качестве узла передачи и/или скрытого сервера.

Клиентские приложения могут использовать сеть Tor путём подключения к локальному прокси-серверу SOCKS. Если само приложение не имеет поддержки SOCKS, вы можете использовать клиент socks, такой как torsocks.

Обратите внимание, что Tor не занимается очисткой протоколов. Это значит, что существует опасность воздействия на протоколы приложений и связанные с ними программы, так что они выдадут информацию об источнике. Для решения этой проблемы Tor полагается на Torbutton и аналогичные очистители протоколов. Самый простой вариант работы с веб-сайтами через Tor - Tor Browser Bundle, самодостаточный пакет, состоящий из статической сборки Tor, Torbutton и Firefox с изменениями, устраняющими различные проблемы, связанные с защитой персональных данных.

Домашняя страница: <https://www.torproject.org/>

Установка пакета:

```
$ sudo apt install tor
Чтение списков пакетов... Готово
Построение дерева зависимостей
...
ureadahead will be reprofiled on next reboot
Обрабатываются триггеры для systemd (237-3ubuntu10.41) ...
```

Обращаем внимание на последнюю строку, упоминающую systemd — это показывает, что стек TOR устанавливается как сервис. Сразу после установки:

```
$ systemctl status tor
● tor.service - Anonymizing overlay network for TCP (multi-instance-master)
   Loaded: loaded (/lib/systemd/system/tor.service; enabled; vendor preset: enabled)
   Active: active (exited) since Thu 2023-04-27 11:58:14 EEST; 5 days ago
     Main PID: 545 (code=exited, status=0/SUCCESS)
        Tasks: 0 (limit: 18948)
       Memory: 0B
          CPU: 0
       CGroup: /system.slice/tor.service
```

```
anp 27 11:58:14 nvme systemd[1]: Starting Anonymizing overlay network for TCP (multi-instance-master)...
```

```
anp 27 11:58:14 nvme systemd[1]: Finished Anonymizing overlay network for TCP (multi-instance-master).
```

```
$ ps -A | grep tor
623 ?          00:00:16 tor
```

И вот SOCKS прокси, через который работают приложения TOR, прокси устанавливается на TCP порт 9050 (заметим, что инсталляция TOR-браузера устанавливает отдельный, параллельный прокси независимо на порт 9150, таким образом они могут работать независимо, не вредя друг другу):

```
$ sudo netstat -tunlp | grep tor
tcp        0      0 127.0.0.1:9051      0.0.0.0:*           LISTEN      1486/tor
tcp        0      0 127.0.0.1:9050      0.0.0.0:*           LISTEN      1486/tor
tcp        0      0 127.0.0.1:9151      0.0.0.0:*           LISTEN      20785/tor
tcp        0      0 127.0.0.1:9150      0.0.0.0:*           LISTEN      20785/tor
```


Проверить можем простейшим образом (1-я строка в тестировании — это прямой запрос, индицируется IP адрес, который динамически выделяет мой провайдер, а следующая строка — то же, но через SOCKS TOR):

```
$ curl check-host.net/ip
193.28.177.119
$ curl -x socks4://127.0.0.1:9050 check-host.net/ip
107.189.31.134
$ whois 107.189.31.134 | grep -i Country:
Country:          US
Country:          LU
```

И я оказываюсь где-то там ... в Луизиане, кажется... Но через некоторое время, без всякого вмешательства с нашей стороны, с того же компьютера, это может стать ... Германия:

```
$ curl -x socks4://127.0.0.1:9050 check-host.net/ip
212.21.66.6
$ whois 212.21.66.6 | grep -i Country:
country:          DE
country:          DE
```

Это TOR стек самопроизвольно сменил цепочку 3-х узлов (3-й из которых мы и видим в качестве IP, выводимого командой). Периодическая смена цепочек — это ещё одна особенность TOR. Но эту смену принудительно может делать и пользователь, в любой момент времени.

Совершенно не обязательно дожидаться пока стек TOR сам сменил цепочку узлов через которые вы используете сеть TOR — достаточно послать сервису TOR сигнал HUP чтобы он принудительно «подбросил» новую цепочку узлов.²¹ Так мы путешествуем со своим хостом по всему миру:

```
$ sudo killall -HUP tor
$ curl -x socks4://127.0.0.1:9050 check-host.net/ip && echo
178.218.144.99
$ whois 178.218.144.99 | grep -i Country:
country:          IT
$ sudo killall -HUP tor
$ curl -x socks4://127.0.0.1:9050 check-host.net/ip && echo
192.42.116.216
$ whois 192.42.116.216 | grep -i Country:
Country:          NL
country:          NL
country:          NL
$ sudo killall -HUP tor
$ curl -x socks4://127.0.0.1:9050 check-host.net/ip && echo
103.251.167.10
$ whois 103.251.167.10 | grep -i Country:
country:          ZZ
country:          ZZ
```

Мы проверили и продемонстрировали работу TOR SOCKS прокси на примере простейшей, даже тривиальной утилиты curl. Но точно таким же способом (используя локально работающий SOCKS прокси) вы можете организовать работу через сеть анонимизации TOR практически **любого** приложения вашего компьютера (о технике использования прокси было рассказано раньше, но она в полной мере относится и к этому случаю).

²¹ Также смена цепочки будет происходить, естественно и понятно, будет происходить при рестарте сервиса TOR командой systemctl. Ещё один, 3-й способ, через командный интерфейс TOR, будет показан позже.

Торификация терминальной сессии

Есть очень удобная возможность, которую мы (за неимением терминологии) назовём торификация: когда все сетевые CLI команды в терминале (если говорить строго, то в оболочке shell) осуществляют трафик через SOCKS5 прокси Tor, без явного на то указания. Достигается это терминальной командой:

```
$ . torsocks on
Tor mode activated. Every command will be torified for this shell.
```

Когда необходимость использования прокси для команд этого терминала отпадёт, мы отменим её симметричной командой:

```
$ . torsocks off
Tor mode deactivated. Command will NOT go through Tor anymore.
```

После этого весь трафик сетевых команд восстанавливается напрямую к IP получателя, минуя прокси.

Теперь посмотрим как это работает. В простейшем виде так — это IP нашего хоста как он видится из торифицированного терминала:

```
$ curl http://checkip.amazonaws.com/
199.195.250.165
$ whois 199.195.250.165 | grep -i country
Country:          US
```

А теперь снимем торификацию с терминала:

```
$ . torsocks off
Tor mode deactivated. Command will NOT go through Tor anymore.
$ curl http://checkip.amazonaws.com/
45.9.31.32
$ whois 45.9.31.32 | grep -i country
country:          UA
country:          UA
```

И мы сразу переместились на другой континент!

Точно так же (без явного указания использовать SOCKS прокси):

```
$ wget -qO- https://check.torproject.org/ | grep -i congratulations
Congratulations. This browser is configured to use Tor.
Congratulations. This browser is configured to use Tor.
```

Или, я покажу дальше проксированный коннект к серверному хосту, к которому прямой коннект от меня перекрыт стараниями провайдера (из политических соображений):

```
$ ping 90.156.230.27 -c3
PING 90.156.230.27 (90.156.230.27) 56(84) bytes of data.

--- 90.156.230.27 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2037ms
```

Дальше я показываю примеры успешных коннектов к этому «перекрытому» хосту, с использованием разных прикладных протоколов и консольных утилит...

Тестирование скорости обмена с хостом (сервер находится на удалении свыше 4000 километров). На сервере выполняется (работает файрвол, поэтому нужно явно указать один из ограниченного числа открытых TCP портов; обращаем внимание, что семейство IPv4 или IPv6 явно указывать не надо — они обслуживаются одновременно):

```
$ iperf3 -s -p8080
...
```

На клиенте (торсифицированном, прямой коннект запрещён):

```
$ iperf3 -c 90.156.230.27 -p8080 -4
Connecting to host 90.156.230.27, port 8080
[ 5] local 127.0.0.1 port 51846 connected to 90.156.230.27 port 8080
[ ID] Interval           Transfer     Bitrate        Retr  Cwnd
[ 5]  0.00-1.00   sec   7.50 MBytes  62.9 Mbits/sec    3   5.25 MBytes
[ 5]  1.00-2.00   sec   1.25 MBytes  10.5 Mbits/sec    1  1023 KBytes
...
- - - - -
[ ID] Interval           Transfer     Bitrate        Retr
[ 5]  0.00-10.00  sec  13.8 MBytes  11.5 Mbits/sec    9
[ 5]  0.00-10.51  sec   6.29 MBytes   5.02 Mbits/sec
                                sender
                                receiver
```

Точно так же, с прозрачным проксированием, работает приложение NetCat:

- На сервере запускаем — ожидание запроса снаружи на передачу файла:

```
$ ls -l www*
-rwx----- 1 olej olej 225 мая 13 2023 www.arch
-rwx----- 1 olej olej 198 мая 11 2023 www.arch.2
-rwx----- 1 olej olej 52 мая 11 2023 www.arch.rem
$ cat www.arch | nc -q0 -l -p 8080
...
```

- На клиенте выполняем запрос на получение этого файла:

```
$ nc -q0 90.156.230.27 8080 > file.nc
$ ls -l file.nc
-rw-rw-r-- 1 olej olej 225 дек 14 22:29 file.nc
```

Так же использует не указанный явно SOCKS прокси и более «навороченная» утилита сетевой передачи:

```
$ socat -V | head -n2
socat by Gerhard Rieger and contributors - see www.dest-unreach.org
socat version 1.7.4.1 on Mar 25 2022 09:51:32
```

- На сервере запускаем — ожидание запроса снаружи на передачу файла:

```
$ socat -u file:www.arch tcp4-listen:8080,reuseaddr
...
```

- На клиенте выполняем запрос на получение этого файла:

```
$ socat -u tcp4:90.156.230.27:8080 STDOUT > file.socat
$ ls -l file.*
-rw-rw-r-- 1 olej olej 225 дек 14 22:29 file.nc
-rw-rw-r-- 1 olej olej 225 дек 14 23:04 file.socat
```

Такая техника торификации терминальной сессии намного упрощает использование Тор-прокси.

А в завершение, добавим ложку дёгтя в эту благостную бочку мёда — далеко не все сетевые утилиты умеют неявно использовать SOCKS прокси ... например:

```
$ . torsocks on
Tor mode activated. Every command will be torified for this shell.
$ telnet 90.156.230.27
1702633197 ERROR torsocks[17796]: Unable to resolve. Status reply: 4 (in
socks5_recv_resolve_reply() at socks5.c:677)
Trying 90.156.230.27...
1702633330 ERROR torsocks[17796]: General SOCKS server failure (in
socks5_recv_connect_reply() at socks5.c:527)
telnet: Unable to connect to remote host: Connection refused
```

```
$ nmap 90.156.230.27
```

```
Starting Nmap 7.80 ( https://nmap.org ) at 2023-12-15 11:45 EET  
NSOCK ERROR [0.2930s] nsock_make_socket(): Socket trouble: Operation not permitted  
nmap: nsock_core.c:1259: nsock_pool_add_event: Assertion `nse->iod->sd >= 0' failed.  
Аварийный останов (образ памяти сброшен на диск)
```

Точно там же, не умеет неявно использовать Тор-прокси и такая мощная утилита (основная сетевая утилита администратора в Linux) как ssh, хотя она и умеет и работать через SOCKS-прокси, и даже, более того, сама создавать туннели для SOCKS-прокси. Но делается это в ней манипуляциями с опциями запуска, а не неявным использованием!

Ресурсы onion

Но проект TOR не ограничивается только предоставлением собственного транспортного механизма над TCP/IP транспортом Интернет! Ещё одним фактором зависимости ресурсов Интернет от централизации управления является то как происходит регистрация доменных имён в сети. В Интернет она производится рядом официально обозначенных регистраторов имён DNS, которым делегирована такая функция. Но регистратор как назначает соответствие доменного имени IP адресу, точно там же может и ликвидировать такое назначение — раз делегировать (исходя из различных мотиваций, чаще всего не зависящих от технических аспектов, что мы не станем здесь это обсуждать).

Для децентрализации именования проект TOR вводит своё onion-именование (в доменной зоне .onion), не требующее разрешения IP адресов через службы DNS.²² (Как мы увидим вскоре, вопрос независимости от DNS-регистраторов является предметом поиска целого ряда новых развиваемых сетевых проектов.)

Ещё одной стороной в организации и использовании .onion ресурсов есть то (и это для нас самый интересный аспект), что любой хост локальной сети с частным IPv4, выходящий в Интернет посредством NAT, и недоступный извне по своему IPv4 (что обсуждалось выше по тексту книги) — вполне доступен из Интернет по своему onion-имени.

Организовать onion-имя для любого своего хоста, на котором работает сервис tor (как показывалось выше) совсем несложно:

- Прописываем в конфигурационном файле /etc/tor/torrc строки (сделать это из-за прав файла можно только как root):

```
# grep -v ^# /etc/tor/torrc | grep -v ^$  
HiddenServiceDir /var/lib/tor/onion.service/  
HiddenServicePort 80 127.0.0.1:80  
HiddenServicePort 22 127.0.0.1:22
```

Во многих версиях подобные строки уже присутствуют в конфигурационном файле, но они закомментированы. Имя onion.service здесь произвольное, которое вам удобное, это не имеет особого значения. А следующие строки (HiddenServicePort) определяют порты к которым будет разрешено подключение onion, и то как оно перенаправляется к локальным сервисам сервера.

- Далее перезапустить сервис tor:
systemctl restart tor
- После чего в каталоге /var/lib/tor у вас появляется новый подкаталог вот с тем произвольным именем onion.service, которое мы и прописали в конфигурационном файле (всё остальное там присутствовало и до того):

```
$ sudo ls -l /var/lib/tor  
итого 36228  
-rw----- 1 debian-tor debian-tor 20442 июн 24 10:40 cached-certs  
-rw----- 1 debian-tor debian-tor 2469608 июл 21 19:37 cached-microdesc-consensus
```

²² Основная мотивация TOR в onion именах состоит именно в анонимизации (сокрытии) именно IP адресов Интернет ресурсов (по их известным именам). Но именно эта сторона вопроса нас не будет интересовать в обсуждении, а сконцентрируемся мы только на независимости от DNS-регистратора.

```
-rw----- 1 debian-tor debian-tor 24238084 июл 16 22:05 cached-microdescs
-rw----- 1 debian-tor debian-tor 10336114 июл 21 19:45 cached-microdescs.new
drwx--S--- 2 debian-tor debian-tor      4096 окт 12  2021 keys
-rw----- 1 debian-tor debian-tor          0 июл 21 19:45 lock
drwx--S--- 3 debian-tor debian-tor      4096 июл 21 19:45 onion.service
-rw----- 1 debian-tor debian-tor     15355 июл 21 19:47 state
```

- Этот новый подкаталог и содержит, помимо прочего, файл `hostname`, который и содержит `onion`-имя нового хоста TOR-сети:

```
$ sudo ls -l /var/lib/tor/onion.service
итого 16
drwx--S--- 2 debian-tor debian-tor 4096 июл 21 19:45 authorized_clients
-rw----- 1 debian-tor debian-tor   63 июл 21 19:45 hostname
-rw----- 1 debian-tor debian-tor   64 июл 21 19:45 hs_ed25519_public_key
-rw----- 1 debian-tor debian-tor   96 июл 21 19:45 hs_ed25519_secret_key
$ sudo cat /var/lib/tor/onion.service/hostname
jfdbieogcloucayp6o4twh337fryoc6phg5yzpuv6phontov6ijm7id.onion
```

И вы можете тут же проверить доступ по вновь созданному полному URL: с другого терминала того же хоста, с другого хоста локальной сети, или с удалённого хоста в Интернет (у меня в примере ниже показано подключение от сервера удалённого на 4000 километров именно к локальному хосту LAN за NAT провайдера):

```
$ torsocks ssh olej@jfdbieogcloucayp6o4twh337fryoc6phg5yzpuv6phontov6ijm7id.onion
olej@jfdbieogcloucayp6o4twh337fryoc6phg5yzpuv6phontov6ijm7id.onion's password:
...
```

И на хосте, к которому проходит `onion`-подключение, можем убедиться:

```
$ who
olej      tty7          2023-07-31 17:30 (:0)
olej      pts/4          2023-07-31 17:36 (127.0.0.1)
```

Вторая строка и показывает текущее `onion`-подключение, переадресованное к серверу SSH через локальный IP 127.0.0.1.

Доступ к таким `.onion` ресурсам осуществляется не только из TOR-браузера (хотя достаточно очевидно было, что именно это было основной целью введения таких ресурсов), но и из любого приложения, «умеющего» использовать SOCKS-прокси, как это рассматривалось только-что в предыдущем рассмотрении, например:

```
$ torsocks wget -q \
  http://flibustaongezhld6dibs2dps6vm4nvqg2kp7vgowbu76tzopgnhazqd.onion/ \
  -O - > flibusta.html
$ ls -l flibusta.html
-rw-rw-r-- 1 olej olej 46053 июл 21 11:30 flibusta.html
```

Важно чтобы переадресация любого порта, используемого таким приложением, была прописана в конфигурационном файле, как мы сделали это выше для портов 22 и 80 (хотя они могли бы быть любыми).

Несколько слов относительно URL `.onion` ресурсов:

1. Внешне неприглядный их вид («нечитабельный») не должен смущать — такие URL не выписываются вручную, а копируются ... что, впрочем, давно уже характерно и для «нативных» DNS-имён Интернет, как следствие их постоянного усложнения.
2. `.onion` имена («именная» их часть) формируются из публичных ключей шифрования, создававшихся при установке пакета (стека TOR).
3. В публикациях часто можно встретить упоминания о 16-символьных именах `.onion` ресурсов. Это относилось к реализации v2 (16 символов, короткие имена), которые не

поддерживаются в v3 (56 символов), начиная с версии TOR стека 0.4.6.8, это примерно конец 2021г.:

```
$ tor --version
Tor version 0.4.6.10.
...
```

4. Вот как хорошо, по-простому но обстоятельно, описывают это в обсуждениях сообщества:

К v3-сервису обращаются напрямую по ключу, а не по его хешу. То есть v3 — это ключ ed25519 (алгоритм цифровой подписи Эдвардса EdDSA на базе эллиптической кривой Curve25519), а v2 — это хеш ключа RSA-1024 (оба в base32), соответственно эти хэши должны распространяться с помощью DHT (распределённая таблица хешей), а для работы v2-сервисов нужны т.н. V2Dir.

v3 лишены подобных недостатков, то есть им не нужно распространение по DHT, их ключ (ed25519) намного более стойкий (~128 бит теоретической стойкости), чем у v2 (для 128 бит по стандартам NIST нужно не RSA-1024, а RSA-3072), а слабый и морально устаревший короткий хэш (80 бит) не используется вообще.

Вот что получается в итоге: меньше нагрузка на ретрансляторы сети Tor, которые выступают в роли V2Dir, поиск и проверка доступности onion-сервисов НАМНОГО быстрее и точнее, нет нужды ждать connection timeout и делать несколько попыток подключения, используемая в v3 криптография намного устойчивее, современнее и быстрее, а также считается невзламываемой на данный момент (128 бит, как говорится, хватит всем!), даже при наличии огромных вычислительных ресурсов.

v2 уже успешно подбирали, для этого даже не нужно взламывать/подбирать точную копию ключа того или иного сервиса — достаточно подобрать очень короткий хэш, и мы получим два сервиса с разными ключами, но одинаковым именем, что запустит V2Dir и позволит противнику перехватывать подключения к v2-сервису.

К этому времени мы достаточно знаем для практического использования TOR

На этом мы

Управляющий порт TOR

Проекту TOR около 25 лет. Проект TOR очень объёмный и весьма сложный. Совершенно естественно, что для таких продуктов должен был быть создан инструментарий диагностики и управления. Для этого разработчики предусмотрели управляющий порт TOR. С конфигурационным файлом по умолчанию этот порт закрыт, и его нужно объявить и открыть ... чем мы и займёмся чуть позже. А вот после выполнения этого, мы ещё раз взглянем на открытые порты:

```
$ sudo netstat -tunlp | grep tor
tcp        0      0 127.0.0.1:9051        0.0.0.0:*           LISTEN      1486/tor
tcp        0      0 127.0.0.1:9050        0.0.0.0:*           LISTEN      1486/tor
tcp        0      0 127.0.0.1:9151        0.0.0.0:*           LISTEN      20785/tor
tcp        0      0 127.0.0.1:9150        0.0.0.0:*           LISTEN      20785/tor
```

Здесь показаны 2 стека TOR: собственно TOR как сервис systemd (с PID 1486) и стек развёрнутый TOR-браузером (запущен специально для наглядности, с PID 20785). Порт 9050, как мы знаем и его использовали — это, собственно, основной порт SOCKS-прокси сервиса. А вот порт 9051 — это и есть управляющий порт этого сервиса, если его открыть (а 9150 и 9151, соответственно, порты стека TOR-браузера).

Вот теперь мы можем рассмотреть как конфигурировать и разрешить использование управляющего порта. А только затем вернёмся к вопросу как ним воспользоваться. Мы добавляем в конфигурацию строку с параметром `ControlPort`:

```
$ grep -v ^# /etc/tor/torrc | grep -v ^$
ControlPort 9051
HashedControlPassword 16:D1D87CA61355A9A86091D3708A1B8C3604A2D3AC7DC14E8047283D6145
HiddenServiceDir /var/lib/tor/onion.service/
HiddenServicePort 8080 127.0.0.1:80
HiddenServicePort 22 127.0.0.1:22
```

После этого добавления (**любое** изменение `/etc/tor/torrc`, естественно, требует рестарта сервиса через: `systemctl restart tor` — не забываем это!) мы получим соответствующую строку (порт) в `netstat`. Но если мы попытаемся воспользоваться этим управляющим портом, то получим ошибку вида (Интернет полнится вопросом про это):

```
$ telnet 127.0.0.1 9051
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
AUTHENTICATE
515 Authentication failed: Wrong length on authentication cookie.
Connection closed by foreign host.
```

Требуется аутентификация! TOR предусматривает и описаны 3 возможных механизма аутентификация, определяемых в конфигурационном файле `/etc/tor/torrc` — мы рассмотрим только один из них, парольный. Для этого нам нужно получить хэш-код для своего пароля (не станем углубляться в алгоритмы по которым это получается). Как пишут:

Хэш-сумма пароля получается следующим образом: необходимо выполнить команду `tor --hash-password 'new_tor_password'` – в ответ получим сгенерированный хэш-код пароля.

Если мы проделываем это для своего локального изолированного компьютера за NAT, то мы можем позволить себе пустой пароль, вот так:

```
$ tor --hash-password ''
16:D1D87CA61355A9A86091D3708A1B8C3604A2D3AC7DC14E8047283D6145
```

И теперь мы должны изменить (или прописать заново) в `/etc/tor/torrc` строку с параметром `HashedControlPassword`, вставив как его значение полученную строку **полностью** — как это и можно видеть в выводе конфигурационного файла выше.

Дальше мы подключаемся к локальному TCP порту 9051 (часто для этого используют клиент `telnet`, но есть много и альтернативных утилит) для управления и диагностики, выполняя **команды** (после аутентификации — первой командой сеанса мы вводим строку `AUTHENTICATE`). Для примера ниже показан сеанс с выполнением одной из самых часто используемых команд — смены цепочки TOR:

```
$ curl --socks5 127.0.0.1:9050 http://checkip.amazonaws.com/
45.141.215.56
$ telnet 127.0.0.1 9051
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
AUTHENTICATE
250 OK
signal NEWNYM
250 OK
^]
telnet> quit
Connection closed.
$ curl --socks5 127.0.0.1:9050 http://checkip.amazonaws.com/
185.220.100.247
```

Как альтернатива использования `telnet` это может быть так:

```
$ echo -e 'PROTOCOLINFO\r\n' | nc 127.0.0.1 9051
250-PROTOCOLINFO 1
250-AUTH METHODS=COOKIE,SAFECOKIE,HASHEDPASSWORD COOKIEFILE="/run/tor/control.authcookie"
250-VERSION Tor="0.4.6.10"
250 OK
514 Authentication required.
```

Примеры выполнения некоторых команд управляющего порта (все они могут выполняться последовательно в одном сеансе `telnet`):

```
...
GETCONF HiddenServiceOptions
250-HiddenServiceDir=/var/lib/tor/onion.service/
250-HiddenServicePort=8080 127.0.0.1:80
250 HiddenServicePort=22 127.0.0.1:22
GETINFO version
250-version=0.4.6.10
250 OK
GETINFO config-file
250-config-file=/etc/tor/torrc
250 OK
GETINFO traffic/read
250-traffic/read=11586959
250 OK
GETINFO traffic/written
250-traffic/written=16980937
250 OK
GETINFO uptime
250-uptime=27793
250 OK
...
```

Перечень возможных аргументов (справку, с кратким описанием) команды `GETINFO`, которая даёт нам исчерпывающую информацию о стеке, можно получить командой — там свыше 200 вариантов:

```
GETINFO info/names
250+info/names=
accounting/bytes -- Number of bytes read/written so far in the accounting interval.
accounting/bytes-left -- Number of bytes left to write/read so far in the accounting interval.
accounting/enabled -- Is accounting currently enabled?
...
```

Команды могут не только запрашивать любую информацию стека, но и позволяет полностью управлять работой выполняющегося сервиса. Полный перечень команд описывается в спецификации TOR, ссылка на которую приведена в перечне источников.

Утилита Nux

Показанная выше техника использования управляющих портов TOR, отработанная за 25 лет, в высшей степени исчерпывающая («позволяет всё»), но в такой же степени трудоёмкая. Для более простого наблюдения в динамике, и некоторого управления стеком TOR создана визуальная (на `ncurses`) утилита, которая включена в стандартные репозитории большинства дистрибутивов Linux:

```
$ aptitude search nux | grep tor
p nux - terminal status monitor for tor
```



```
$ sudo apt install nux
```

Чтение списков пакетов... Готово

Построение дерева зависимостей... Готово

Чтение информации о состоянии... Готово

Следующие НОВЫЕ пакеты будут установлены:

nux

Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 39 пакетов не обновлено.

Необходимо скачать 67,5 кВ архивов.

После данной операции объём занятого дискового пространства возрастёт на 302 кВ.

Пол:1 <http://ubuntu.volia.net/ubuntu-archive> jammy/universe amd64 nux all 2.1.0-2.1 [67,5 кВ]

Получено 67,5 кВ за 1с (59,4 кВ/с)

Выбор ранее не выбранного пакета nux.

(Чтение базы данных ... на данный момент установлено 583394 файла и каталога.)

Подготовка к распаковке .../archives/nux_2.1.0-2.1_all.deb ...

Распаковывается nux (2.1.0-2.1) ...

Настраивается пакет nux (2.1.0-2.1) ...

Обрабатываются триггеры для man-db (2.10.2-1) ...

```
$ which nux
```

/usr/bin/nux

```
$ nux --version
```

nux version 2.1.0 (released January 12, 2019)

```
$ nux --help
```

Usage nux [OPTION]

Terminal status monitor for Tor relays.

```
-i, --interface [ADDRESS:]PORT  change control interface from 127.0.0.1:default
-s, --socket SOCKET_PATH        attach using unix domain socket if present,
                                SOCKET_PATH defaults to: /var/run/tor/control
-c, --config CONFIG_PATH        loaded configuration options, CONFIG_PATH
                                defaults to: /home/olej/.nux/config
-d, --debug LOG_PATH            writes all nux logs to the given location
-l, --log EVENTS                comma separated list of events to log
-v, --version                   provides version information
-h, --help                      presents this help
```

Example:

nux -i 1643 attach to control port 1643

nux -l we -c /tmp/cfg use this configuration file with 'WARN'/'ERR' events

Поскольку программа визуальная, то мы можем запускать её просто по имени — все остальные возможности мы получим из меню, которое вызывается по нажатию клавиши `m`:



Утилита в высшей степени интересная тем, что позволяет наблюдать поведение стека TOR в динамике. Вот, например, как выглядит трафик сквозь стек при просмотре онлайн ТВ в браузере FireFox, настроенном на использование SOCKS-прокси 127.0.0.1:9050

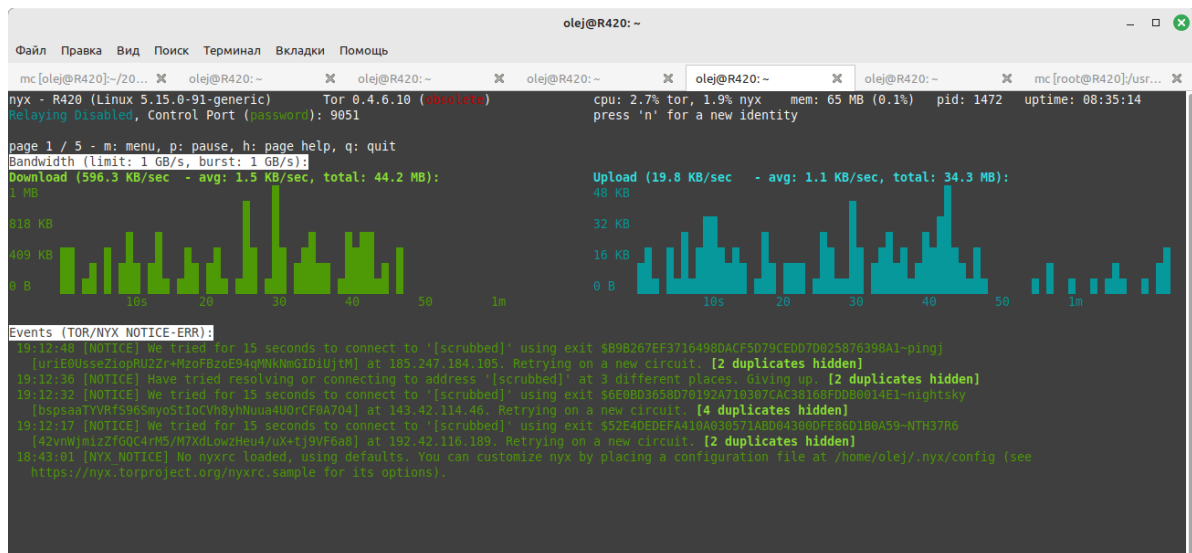


Рис. 5.3 Трафик стека TOR в динамике

Mesh-сети

Mesh-сети, которые до того неспешно обсуждались в академических кругах, привлекли к себе самое пристальное внимание после 2013-2014 годов, когда такие сети, неожиданно массово, были использованы участниками протестных настроений, «цветных революций», в массовых уличных акциях. Из новостных лент тех дней:

В Гонконге, участники "революции зонтиков" используют программу FireChat, что позволяет им быстро установить связь друг с другом по сети, поддерживаемой устройствами отдельных пользователей (не внешнего оборудования) - mesh networking. ... Бесплатное приложение FireChat для устройств на Android и iOS позволяет людям общаться между собой даже там, где не работает сотовая связь. Мессенджер устанавливает прямое соединение между двумя телефонами на расстоянии до 70 м. Но при большом скоплении подключенных к сети пользователей FireChat радиус действия мессенджера может быть намного больше: в пределах стадиона, парка или везде, где минимальное расстояние между двумя пользователями меньше 70 метров.

Ячеистая топология (mesh-сеть) — это распределенная, одноранговая, ячеистая сеть. Каждый узел в такой сети обладает такими же полномочиями как и все остальные, грубо говоря — все узлы в сети равны. Сети бывают самоорганизующиеся и настраиваемые, первый тип сетей при включении оборудования, которое его поддерживает, автоматически подключаются к существующим участникам, выбирают оптимальные маршруты и самонастраиваются внутри сети.

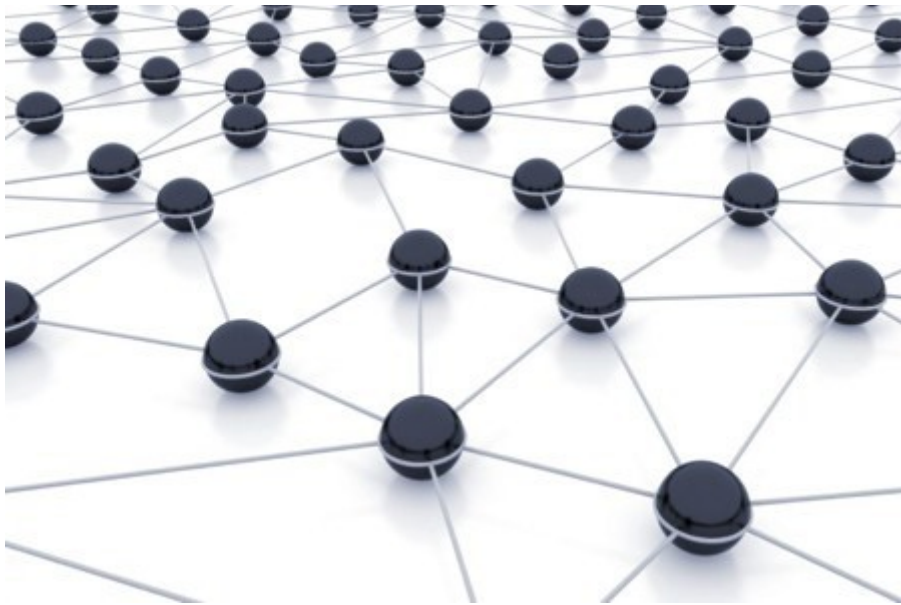


Рис. 5.4 Ячеистая (пиринговая, mesh) сеть

Такая организация сети, является достаточно сложной в реализации и, если она не является самоорганизующейся, то и в настройке тоже, однако, как предполагается, при такой топологии реализуется максимально возможная отказоустойчивость сети.

Первоначально, именно сеть мобильных, перемещающихся, гаджетов с беспроводной связью (WiFi) привлекла внимание к этой идее. Но позже оказалось что она отлично расширяется на ячеистую сеть узлов любой топологии, независимо от природы самих узлов, в том числе и на уже существующую транспортную сеть Интернет, в том числе и стационарных хостов в его составе.

Основной принцип, обеспечивающий высокую живучесть такой сети, состоит в её самоконфигурируемости: при потере отдельных узлов, или даже целых значительных фрагментов сети, сеть динамически (на ходу) меняет свою топологию, и изменяются траектории распространения IP пакетов даже в пределах одного сеанса связи. Большое количество связей обеспечивает широкий выбор маршрута трафика внутри сети — следовательно, обрыв одного соединения не нарушит функционирования сети в целом. (В какой-то мере, это, не по существу но качественно, напоминает переход от стационарных сетей коммутации каналов в проводной телефонии к той же коммутации каналов, но уже в сети мобильных сотовых станций).

Терминологически mesh-сети, пиринговые сети — это родовое наименование, целый класс, в котором известно уже несколько разработок. В качестве иллюстрации конкретной реализации mesh-сети, но уровня проекта уже пригодного для практического применения, мы рассмотрим один частный пример такой IPv6 сети Yggdrasil. Помимо деталей конкретной сети это даёт возможность вообще взглянуть на особенности ячеистых сетей.

Сеть Yggdrasil

Yggdrasil — это пиринговая сеть, использующая IPv6 адресацию, весь трафик в которой шифруется несимметричным кодированием (с приватными и публичными ключами). (Пиринговая — это ещё одно название ячеистых одноранговых сетей, в которых каждый хост связывается с несколькими известными ему пирами, соседями по сети, через которые хост организует свой адаптивный роутинг в сеть.)

Пара ключей шифрования (приватный и публичный) устанавливаются для хоста либо автоматически в момент инсталляции, либо могут быть изменены пользователем ручной записью их значений (более оптимальных) в конфигурационный файл. Адрес IPv6 хоста формируется как функция от этой пары ключей, и никем глобально не регламентируется.

Проект Yggdrasil использует адреса IPv6 из подсети 0200::/7²³, которые никак не пересекаются с «нативными» адресами (2000::/3) официально выделяемые IANA (RFC 1881, 3513, 4291). В этом легко убедиться воспользовавшись для представления полных IPv6 онлайн IPv6 Subnet Calculator (<https://www.vultr.com/resources/subnet-calculator-ipv6/>):

```
2000::/3 => 2000:0000:0000:0000:0000:0000:0000:0000 ...
3fff:ffff:ffff:ffff:ffff:ffff:ffff:ffff

0200::/7 => 0200:0000:0000:0000:0000:0000:0000:0000 ...
03ff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
```

Проект Yggdrasil свободный и с открытыми исходными кодами, код написан и развивается на современном языке программирования Go (что является дополнительным достоинством проекта). Некоторые дополнительные сервисные утилиты к нему написаны сообществом пользователей на языках C, Python и Rust, и будут коротко показаны далее.

Установка

Установку сети Yggdrasil, на протяжении всей истории существования проекта и поныне, можно было произвести одним из нескольких доступных способов:

1. Из стандартного репозитория вашего дистрибутива;
2. Из бинарного пакета, подготовленного разработчиками в GIT-репозитории проекта;
3. Сборкой из открытых исходных кодов (на том же GIT-репозитории);

Буквально в те же дни, когда вышло 1-е издание книги (30 ноября 2023г.), произошло (28 ноября 2023г.) давно анонсируемое и ожидавшееся событие: смена **семейства**²⁴ **версий** IPv6 mesh-сети Yggdrasil с 0.4.7 на 0.5.X (на сегодня доступна версия 0.5.3). Событие достаточно ординарное в мире свободных программных проектов ... да и в хронологии развития данного конкретного проекта. Но в данном случае нужно учесть следующее:

- Сетевые протоколы (и алгоритмы маршрутизации) этих версий **несовместимы** друг с другом. Хосты таких сетей не будут видеть друг друга. Нужно использовать либо одно семейство версий, либо другое ... а поскольку эффективность использования здесь зависит не только от ваших желаний, но и от множества других пользователей (контрагентов) разбросанных по всем просторам Интернет — то устанавливать целесообразно именно новую версию.
- Принципиально процедуры установки, настройки и использования внешне мало чем отличаются, и практически не изменились. Некоторые отличия наблюдаются в мелких деталях: различия в содержании конфигурационных файлов, в некоторых путевых именах каталогов, отличающемся внешнем виде диагностики... Некоторые отличия, необходимый в настройках, будут рассмотрены вскоре...
- При установке новой версии средствами пакетной системы дистрибутива (из DEB-пакета) поверх уже работающей старой версии, изменения будут **автоматически** внесены в конфигурационный файл (тут нужно отдать должное изобретательности разработчиков). А сам конфигурационный файл будет перемещён в новую дефолтную позицию (из старой /etc/yggdrasil.conf в новую /etc/yggdrasil/yggdrasil.conf). С большой вероятностью такая установка «поверх» сохранит работоспособность существующей и её параметры (значения ключей шифрования и, как следствие, адрес IPv6). Естественно, что в конфигурационном файле для этого должны быть переписаны адреса соседствующих пиров, вместо работающих в старой версии на работающих в новой.

Уже из этого предупреждения уже должно стать понятным, что названный первым, он же и самый простой, способ установки (из стандартного репозитория вашего дистрибутива) на

²³ Есть альтернативная проект развития (форк) Yggdrasil, с небольшими отличиями, под названием RiV-mesh, который эксплуатирует другой диапазон адресов: fc00::/7

²⁴ Только за 1-й календарный месяц с объявления семейства 0.5.X сменились версии 0.5.1, 0.5.2, 0.5.3 и 0.5.4 — это, со своей стороны, показывает как динамично развивается проект. Но эта смена младшей версии не должна смущать потребителя: все они протоколно совместимы.

некоторый период, пока идёт смена основного семейства версий и протокола (сейчас и впредь). Поэтому, первейшее что вы должны сделать ориентируясь на стандартную установку, это проверить соответствие версии. Но многие поставщики дистрибутивов, на удивление, очень быстро отрабатывают смену версии, вот, к примеру, Linux Mint всего через месяц после смены линейки 0.4.X на 0.5.X:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID:    Linuxmint
Description: Linux Mint 21.2
Release:          21.2
Codename:         victoria
$ aptitude show yggdrasil | grep Версия
Версия: 0.5.4
```

Но стандартная установка из репозитория (в части доведения до рабочего состояния) ничем не отличается от установки из готовых DEB-пакетов Linux, предоставляемых в каждом релизе разработчиками для нескольких аппаратных архитектур вместе с исходными кодами (все ссылки на ресурсы показаны в конце этой части). А поэтому к ней и переходим ... позже мы ещё вернёмся к сборке проекта из исходных кодов.

Для демонстрации установки из бинарного пакета автором сознательно был выбран в качестве оборудования экземпляр миниатюрного Orange Pi One:



Рис. 5.5 Миниатюрный хост сети Yggdrasil

Для такого (экзотического) выбора есть целый ряд оснований²⁵:

- Одноплатная архитектура SoC с 32-бит ARM процессором... (мы иллюстрируем охват проектом Yggdrasil разных процессорных архитектур, а позже увидим ещё и разнообразие поддерживаемых операционных систем);
- Минимальный размер — не более банковской кредитной карточки...

²⁵ Для разнообразных Intel x86 (i86, x86_64) установка проходит так же, или ещё даже проще.

- Минимальной стоимости (непосредственно от китайских производителей на AliExpress стоимость порядка \$9)...
- Один из представителей многочисленных подобных семейств SoC от китайских производителей (BananaPi, NanoPi, ZeroPi и мн. другие ... которые, примем на заметку, в разы дешевле подобных им, но более известных, раскрученных и модных европейских Raspberry Pi)...
- Более того, для эксперимента выбрана именно самая минимальная из существующих модель семейства Orange Pi, с загрузкой из SD-карточки:

```
$ inxi -MCDmxxx
```

```
Machine:   Type: ARM Device System: Xunlong Orange Pi One details:
           Allwinner sun8i Family rev: N/A serial: 02c000815fd5e717
Memory:    RAM: total: 491.7 MiB used: 291.1 MiB (59.2%)
           RAM Report: missing: Required program dmidecode not available
CPU:       Topology: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32 type:
           MCP arch: v7l rev: 5
           features: Use -f option to see features bogomips: 0
           Speed: 1008 MHz min/max: 480/1008 MHz Core speeds (MHz):
           1: 1008 2: 1008 3: 1008 4: 1008
Drives:    Local Storage: total: 14.84 GiB used: 7.16 GiB (48.2%)
           ID-1: /dev/mmcblk0 model: SB16G size: 14.84 GiB
           serial: 0x5baaa201 scheme: MBR
```

- Даже такой минималистичной конфигурации и её производительности вполне достаточно для установки полноценного Linux (в сборке Armbian, ссылка в источниках информации) и его долговременной работы в качестве IPv6 хоста или ретранслятора сети Yggdrasil:

```
$ uptime
```

```
14:11:28 up 15 days, 15:45, 2 users, load average: 0,06, 0,03, 0,01
```

Итак, приступим (на сегодня с сайта Armbian вы можете загрузить заметно более свежую версию Linux «23.8 Bookworm Kernel 6.1», но это не имеет, для наших целей, принципиального значения)...

```
$ lsb_release -a
```

```
No LSB modules are available.
Distributor ID: Debian
Description: Debian GNU/Linux 10 (buster)
Release: 10
Codename: buster
```

```
$ cat /etc/debian_version
```

```
10.13
```

Скачиваем с GIT-репозитория разработчиков самую свежую версию бинарного пакета Yggdrasil (для этого процессора нам нужна архитектура armhf):

```
$ wget https://github.com/yggdrasil-network/yggdrasil-go/releases/download/v0.5.2/yggdrasil-0.5.2-armhf.deb
```

```
--2023-11-13 22:08:33--
```

```
https://github.com/yggdrasil-network/yggdrasil-go/releases/download/v0.5.2/yggdrasil-0.5.2-armhf.deb
```

```
Распознаётся github.com (github.com)... 140.82.121.3
```

```
Подключение к github.com (github.com)|140.82.121.3|:443... соединение установлено.
```

```
HTTP-запрос отправлен. Ожидание ответа... 302 Found
```

```
...
```

```
HTTP-запрос отправлен. Ожидание ответа... 200 OK
```

```
Длина: 5754998 (5,5M) [application/octet-stream]
```

```
Сохранение в: «yggdrasil-0.5.2-armhf.deb»
```

```
2023-11-13 22:08:35 (5,08 MB/s) - «yggdrasil-0.5.2-armhf.deb» сохранён [5754998/5754998]
```



```
$ ls -l yggdrasil-0.5.2-armhf.deb
-rw-r--r-- 1 olej olej 5754998 ноя 6 11:33 yggdrasil-0.5.2-armhf.deb
```

Инсталляция из ДЕВ-пакета традиционная и проста:

```
$ sudo dpkg -i yggdrasil-0.5.2-armhf.deb
Выбор ранее не выбранного пакета yggdrasil.
(Чтение базы данных ... на данный момент установлено 179506 файлов и каталогов.)
Подготовка к распаковке yggdrasil-0.5.2-armhf.deb ...
Распаковывается yggdrasil (0.5.2) ...
Настраивается пакет yggdrasil (0.5.2) ...
Generating initial configuration file /etc/yggdrasil/yggdrasil.conf
Created symlink /etc/systemd/system/multi-user.target.wants/yggdrasil.service →
/lib/systemd/system/yggdrasil.service.
$ which yggdrasil
/usr/bin/yggdrasil
$ yggdrasil --version
Build name: yggdrasil
Build version: 0.5.2
```

Как и для многих программных продуктов, предполагающих запуск в качестве сервиса `systemd` (о чём отмечалось в предыдущих частях), одновременно с инсталляцией происходит и запуск сервиса (попытка ... в данном случае не успешная):

```
$ systemctl status yggdrasil
• yggdrasil.service - Yggdrasil Network
   Loaded: loaded (/lib/systemd/system/yggdrasil.service; enabled; vendor preset: enabled)
   Active: failed (Result: exit-code) since Mon 2023-11-13 22:10:58 EET; 53s ago
   Process: 21396 ExecStartPre=/sbin/modprobe tun (code=exited, status=1/FAILURE)
   Process: 21397 ExecStart=/usr/bin/yggdrasil -useconffile /etc/yggdrasil/yggdrasil.conf
   (code=exited, status=2)
   Main PID: 21397 (code=exited, status=2)

ноя 13 22:10:58 orangepine systemd[1]: yggdrasil.service: Service RestartSec=100ms
expired, scheduling restart.
ноя 13 22:10:58 orangepine systemd[1]: yggdrasil.service: Scheduled restart job, restart
counter is at 5.
ноя 13 22:10:58 orangepine systemd[1]: Stopped Yggdrasil Network.
ноя 13 22:10:58 orangepine systemd[1]: yggdrasil.service: Start request repeated too
quickly.
ноя 13 22:10:58 orangepine systemd[1]: yggdrasil.service: Failed with result 'exit-code'.
ноя 13 22:10:58 orangepine systemd[1]: Failed to start Yggdrasil Network.
```

Вас это не должно несколько смущать, потому что нам ещё предстоит, под конкретику своей локальной сети (или автономного компьютера) заполнить конфигурационный файл, который для нас уже любезно создала проделанная только-что инсталляция:

```
# ls -l /etc/yggdrasil/yggdrasil.conf
-rw-r----- 1 root yggdrasil 3503 ноя 18 16:48 /etc/yggdrasil/yggdrasil.conf
```

(Как и видно из флагов созданного файла, редактировать его можно только с правами `root` — авторы проекта не хотят чтобы в нём ковырялись посторонние!)

Безусловно необходимым действием над конфигурационным файлом — это будет заполнение в нём набора адресов (или URL-имён) нескольких внешних пиров. Напомню, что внешние пиры — это сторонние узлы, находящиеся где-то в сети `Yggdrasil`, посредством связи с которыми наш созданный хост станет членом сообщества `Yggdrasil`. В только-что созданном конфигурационном файле этот список ещё пуст:

```
# grep "Peers:" /etc/yggdrasil/yggdrasil.conf -A1 | grep -v ^#
Peers: [
]
```

Нам нужно для каждого внешнего пира (отдельной строкой, никакие комментарии сверх того не допускаются) указать: протокол, адрес/имя, порт... и обязательно для той линейки версий (совместимости) с которой сейчас работаем. Например так:

```
# grep " Peers:" /etc/yggdrasil/yggdrasil.conf -A8 | grep -v ^#
Peers: [
    tcp://193.93.119.42:14244
    tcp://178.132.161.124:65533
    tcp://y.zbin.eu:7743
    tcp://88.210.3.30:65533
    tcp://87.251.77.39:65533
    tcp://sin.yuetau.net:6642
    tcp://mima.localghost.org:1996
]
```

Про то как выбирать внешние пиры — будет подробно рассказано в дальнейшем, и это ключевой компонент успеха. Не нужно копировать пиры показанные выше в примере, удачный набор пиров зависит от вашей конкретно геолокации. (Пир с нативными IPv6 адресами, из общего числа доступных, также можно использовать только в том случае, если ваш Интернет-провайдер предоставляет вам IPv6 адресацию ... а это, как говорилось ранее, не сегодня пока ещё не часто.)

Если обсуждавшееся выше изменение в конфигурационном файле является безусловно необходимым, то альтернативным, но весьма желательным, будет ещё и изменение строки с значением PrivateKey — заменить значение (128 байтовая строка) устанавливаемое там автоматически при инсталляции, на значение обеспечивающее более «высокий IP». Такие значения могут быть найдены путём майнинга, о чём будет рассказано вскоре. Например:

```
# grep PrivateKey: /etc/yggdrasil.conf | grep -v ^#
PrivateKey:
1fb2819e5dc33c999764d28e78500abbf01068dacc177cf1eb69b39239333f6c00000004968b2543691a13b9cf9
f2c485ed9444731f3bf76299a6b2273f5128e
```

(Но это изменение значения ключа можно будет сделать и позже, в любое время ... только нужно держать в уме, что при этом изменится IPv6 вашего Yggdrasil хоста, который формируется именно из значений ключей.)

Примечание: Значение вашего приватного ключа — это ваша тайна, которую нужно хранить, во избежание коллизии IPv6 в сети (вашего и ещё кого-то другого). В новых версиях 0.5.X даже добавлена возможность хранить значение приватного ключа в зашифрованном виде в отдельном файле, вне конфигурационного файла сети Yggdrasil. На сегодня, младшие 64 байта 128-байтного приватного ключа — это публичный ключ шифрования этого хоста, и его не требуется явно указывать в конфигурационном файле (предыдущие версии, 0.4.X и ранее — требовали). Но если вы укажете в конфигурационном файле корректное значение PublicKey (на манер предыдущих версий) это не будет квалифицироваться как ошибка.

На этом всё ... И вы можете стартовать (или рестартовать) неудачно стартовавший ранее сервис:

```
$ sudo systemctl restart yggdrasil
```

```
$ systemctl --no-pager --full status yggdrasil
```

```
• yggdrasil.service - Yggdrasil Network
   Loaded: loaded (/lib/systemd/system/yggdrasil.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2023-11-13 22:41:04 EET; 2min 51s ago
   Process: 2067 ExecStartPre=/sbin/modprobe tun (code=exited, status=0/SUCCESS)
   Main PID: 2069 (yggdrasil)
   Tasks: 11 (limit: 905)
   Memory: 7.8M
   CGroup: /system.slice/yggdrasil.service
```



```
└─2069 /usr/bin/yggdrasil -useconffile /etc/yggdrasil/yggdrasil.conf
```

```
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected outbound:
22d:d3dd:3afe:9599:3da9:d89f:6ae:8401@193.93.119.42:14244, source 192.168.1.153:42672
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected outbound:
222:10d:b0b4:f237:5db9:e47c:a2e6:9ab9@178.132.161.124:65533, source 192.168.1.153:40028
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected inbound:
21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f@[fe80::13f5:9fe2:6393:bf4a%eth0]:50409, source
[fe80::ef45:854e:6776:cfe7%eth0]:37163
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected inbound:
21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f@[fe80::9bac:3791:1b79:7237%eth0]:54231, source
[fe80::ef45:854e:6776:cfe7%eth0]:37163
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected outbound:
225:124e:2a74:9966:d18d:2f19:adbfa950@88.210.3.30:65533, source 192.168.1.153:32798
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected outbound:
224:79c3:da07:db06:60ef:282d:f908:d97@87.251.77.39:65533, source 192.168.1.153:50600
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected outbound:
200:dc42::3fca:8a99:5d66:b660@77.247.225.234:7743, source 192.168.1.153:57580
ноя 13 22:41:05 orangeplane yggdrasil[2069]: 2023/11/13 22:41:05 Connected outbound:
200:b858:3f39:af76:83ae:bb1:ebb2:604e@149.28.135.180:6642, source 192.168.1.153:55876
ноя 13 22:41:10 orangeplane yggdrasil[2069]: 2023/11/13 22:41:10 Failed to connect to
[200:b1ff:9827:dc87:f067:c43b:d546:1dd7]:1996: dial tcp
[200:b1ff:9827:dc87:f067:c43b:d546:1dd7]:1996: i/o timeout
ноя 13 22:41:11 orangeplane yggdrasil[2069]: 2023/11/13 22:41:11 Connected outbound:
200:b1ff:9827:dc87:f067:c43b:d546:1dd7@38.54.87.34:1996, source 192.168.1.153:55582
```

Теперь всё стартовало успешно. Утилита диагностики и управления, устанавливаемая в составе пакета — `yggdrasilctl`:

```
# yggdrasilctl getself
Build name:          yggdrasil
Build version:       0.5.2
IPv6 address:        222:b303:b732:efb0:8bf7:a017:c2d3:bb6b
IPv6 subnet:         322:b303:b732:efb0::/64
Routing table size:  19
Public key:          00000000299f8919a209ee810bfd07a588928943d03d40f43d765d5052fcf0a9
```

yggdrasilctl getpeers

URI	State	Dir	IP Address	Uptime	RX	TX	Pr	Last Error
tcp://193.93.119.42:14244					Up	Out		
22d:d3dd:3afe:9599:3da9:d89f:6ae:8401				4m28s	4kb	3kb	0	-
tcp://178.132.161.124:65533					Up	Out		
222:10d:b0b4:f237:5db9:e47c:a2e6:9ab9				4m28s	2kb	1kb	0	-
tls://[fe80::13f5:9fe2:6393:bf4a%25eth0]					Up	In		
21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f				4m28s	1kb	1kb	0	-
tls://[fe80::9bac:3791:1b79:7237%25eth0]					Up	In		
21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f				4m28s	1kb	1kb	0	-
tcp://88.210.3.30:65533					Up	Out		
225:124e:2a74:9966:d18d:2f19:adbfa950				4m28s	2kb	1kb	0	-
tcp://87.251.77.39:65533					Up	Out		
224:79c3:da07:db06:60ef:282d:f908:d97				4m28s	2kb	1kb	0	-
tcp://y.zbin.eu:7743					Up	Out		
200:dc42::3fca:8a99:5d66:b660				4m28s	2kb	1kb	0	-
tcp://sin.yuetau.net:6642					Up	Out		
200:b858:3f39:af76:83ae:bb1:ebb2:604e				4m28s	2kb	1kb	0	-
tcp://mima.localghost.org:1996					Up	Out		
200:b1ff:9827:dc87:f067:c43b:d546:1dd7				4m22s	2kb	1kb	0	-

Все возможности этой утилиты смотрим так:

\$ sudo yggdrasilctl list

Command	Arguments	Description
addpeer	uri=..., interface=...	Add a peer to the peer list
debug_remotegetpeers	key=...	Debug use only
debug_remotegetself	key=...	Debug use only

debug_remotegettree	key=...	Debug use only
getmulticastinterfaces		Show which interfaces multicast is enabled on
getnodeinfo	key=...	Request nodeinfo from a remote node by its public key
getpaths		Show established paths through this node
getpeers		Show directly connected peers
getself		Show details about this node
getsessions		Show established traffic sessions with remote nodes
gettree		Show known Tree entries
gettun		Show information about the node's TUN interface
list		List available commands
removepeer	uri=..., interface=...	Remove a peer from the peer list

По итогу, мы получаем новый сетевой интерфейс `tun0` с адресом IPv6, чего и добивались:

```
$ ip a s tun0
6: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 65535 qdisc pfifo_fast state
UNKNOWN group default qlen 500
    link/none
    inet6 222:b303:b732:efb0:8bf7:a017:c2d3:bb6b/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::8670:f610:d41:1605/64 scope link stable-privacy
        valid_lft forever preferred_lft forever
```

И можем тут же проверить новый полученный IPv6 по отношению какого-то внешнего хоста, находящегося, возможно, за тысячи километров:

```
$ ping -c4 201:23b4:991a:634d:8359:4521:5576:15b7
PING 201:23b4:991a:634d:8359:4521:5576:15b7(201:23b4:991a:634d:8359:4521:5576:15b7) 56
data bytes
64 bytes from 201:23b4:991a:634d:8359:4521:5576:15b7: icmp_seq=1 ttl=64 time=649 ms
64 bytes from 201:23b4:991a:634d:8359:4521:5576:15b7: icmp_seq=2 ttl=64 time=113 ms
64 bytes from 201:23b4:991a:634d:8359:4521:5576:15b7: icmp_seq=3 ttl=64 time=196 ms
64 bytes from 201:23b4:991a:634d:8359:4521:5576:15b7: icmp_seq=4 ttl=64 time=127 ms

--- 201:23b4:991a:634d:8359:4521:5576:15b7 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 6ms
rtt min/avg/max/mdev = 112.851/271.020/648.701/220.299 ms
```

Или так (почему в Yggdrasil могут появляться адреса с ведущим хекстетом вида 3XX будет рассказано дальше):

```
$ ping -c4 321:c99a:91a1:cd2c::18
PING 321:c99a:91a1:cd2c::18(321:c99a:91a1:cd2c::18) 56 data bytes
64 bytes from 321:c99a:91a1:cd2c::18: icmp_seq=1 ttl=64 time=482 ms
64 bytes from 321:c99a:91a1:cd2c::18: icmp_seq=2 ttl=64 time=85.8 ms
64 bytes from 321:c99a:91a1:cd2c::18: icmp_seq=3 ttl=64 time=83.0 ms
64 bytes from 321:c99a:91a1:cd2c::18: icmp_seq=4 ttl=64 time=82.6 ms

--- 321:c99a:91a1:cd2c::18 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 5ms
rtt min/avg/max/mdev = 82.643/183.446/482.342/172.572 ms
```

Характерно, что для Yggdrasil, как для всякой пиринговой сети, 1-й пакет ICMP идёт в разы дольше по времени, чем последующие — это прокладывается маршрут к пункту назначения, а все последующие ICMP идут уже только повторяя этот маршрут.

Вот, собственно, и всё что нужно для того, чтобы получить работающую Yggdrasil сеть, как видно из состава устанавливаемых файлов, устанавливается сервис для запуска средствами `systemd`, утилиты `yggdrasil` и `yggdrasilctl`, и генерируется шаблон который будет использоваться в качестве конфигурационного файла `/etc/yggdrasil/yggdrasil.conf`.

```
$ apt contains yggdrasil
yggdrasil: /lib/systemd/system/yggdrasil-default-config.service
yggdrasil: /lib/systemd/system/yggdrasil.service
yggdrasil: /usr/bin/yggdrasil
yggdrasil: /usr/bin/yggdrasilctl
```

Далее, пользуясь случаем, мы рассмотрим сборку проекта из открытых исходных кодов, тем более, что, например, для RPM-дистрибутивов (да и для всех прочих пакетных систем других дистрибутивов, отличных от DEB-пакетной системы) это единственный доступный способ инсталляции новых версий.

```
$ cat /etc/fedora-release
Fedora release 38 (Thirty Eight)
```

Нужно иметь в системе инструментарий GoLang, установленный стандартным способом, в относительно свежей (не обязательно последней) стабильной версии (утверждается что достаточно версии 1.17 или старше):

```
$ go version
go version go1.21.3 linux/amd64
```

Закачиваем исходный код проекта (URL показан в команде):

```
$ git clone https://github.com/yggdrasil-network/yggdrasil-go.git
Клонирование в «yggdrasil-go»...
remote: Enumerating objects: 12692, done.
remote: Counting objects: 100% (829/829), done.
remote: Compressing objects: 100% (339/339), done.
remote: Total 12692 (delta 524), reused 728 (delta 460), pack-reused 11863
Получение объектов: 100% (12692/12692), 3.53 МБ | 967.00 КиБ/с, готово.
Определение изменений: 100% (8531/8531), готово.
```

Сборка (происходит достаточно быстро) ... вывод достаточно длинный, но его имеет смысл привести, для сверки на случай если что пойдёт не так:

```
$ cd yggdrasil-go/
$ ./build
Building: yggdrasil
go: downloading github.com/gologme/log v1.3.0
go: downloading github.com/hjson/hjson-go/v4 v4.3.0
go: downloading github.com/Arceliar/ironwood v0.0.0-20231104025256-ec84c695fc44
go: downloading golang.org/x/text v0.13.0
go: downloading golang.org/x/net v0.17.0
go: downloading github.com/Arceliar/phony v0.0.0-20220903101357-530938a4b13d
go: downloading golang.zx2c4.com/wireguard v0.0.0-20230223181233-21636207a675
go: downloading golang.org/x/crypto v0.14.0
go: downloading golang.org/x/sys v0.13.0
go: downloading github.com/quic-go/quic-go v0.39.3
go: downloading github.com/bits-and-blooms/bloom/v3 v3.3.1
go: downloading github.com/bits-and-blooms/bitset v1.5.0
go: downloading golang.org/x/exp v0.0.0-20221205204356-47842c84f3db
golang.org/x/crypto/internal/alias
golang.org/x/exp/constraints
github.com/quic-go/quic-go/internal/utils/ringbuffer
golang.org/x/net/internal/iana
golang.org/x/crypto/salsa20/salsa
github.com/kardianos/minwinsvc
github.com/gologme/log
github.com/yggdrasil-network/yggdrasil-go/src/version
github.com/hjson/hjson-go/v4
github.com/yggdrasil-network/yggdrasil-go/src/address
```

github.com/Arceliar/phony
github.com/bits-and-blooms/bitset
golang.org/x/crypto/curve25519
golang.org/x/crypto/internal/poly1305
golang.org/x/sys/cpu
github.com/hashicorp/go-syslog
github.com/Arceliar/ironwood/types
github.com/Arceliar/ironwood/encrypted/internal/e2c
github.com/quic-go/quic-go/internal/utils/linkedlist
github.com/quic-go/quic-go/internal/protocol
golang.org/x/crypto/blake2b
github.com/bits-and-blooms/bloom/v3
golang.org/x/crypto/nacl/secretbox
golang.org/x/crypto/chacha20
golang.org/x/crypto/hkdf
golang.org/x/exp/rand
golang.org/x/net/bpf
github.com/quic-go/quic-go/internal/utils
github.com/quic-go/quic-go/internal/qtls
github.com/quic-go/quic-go/internal/qerr
github.com/Arceliar/ironwood/network
github.com/quic-go/quic-go/quicvarint
golang.org/x/crypto/chacha20poly1305
golang.org/x/net/internal/socks
golang.org/x/crypto/nacl/box
github.com/quic-go/quic-go/internal/flowcontrol
golang.org/x/sys/unix
github.com/quic-go/quic-go/internal/wire
golang.org/x/text/encoding/internal/identifier
golang.org/x/net/proxy
golang.org/x/text/transform
golang.org/x/text/internal/utf8internal
github.com/Arceliar/ironwood/encrypted
golang.org/x/text/encoding
golang.org/x/text/runes
golang.org/x/text/encoding/internal
github.com/quic-go/quic-go/logging
golang.org/x/text/encoding/unicode
github.com/quic-go/quic-go/internal/logutils
github.com/quic-go/quic-go/internal/handshake
github.com/quic-go/quic-go/internal/congestion
github.com/yggdrasil-network/yggdrasil-go/src/config
github.com/quic-go/quic-go/internal/ackhandler
golang.zx2c4.com/wireguard/rwcancel
github.com/vishvananda/netns
golang.org/x/net/internal/socket
github.com/vishvananda/netlink/nl
golang.org/x/net/ipv6
golang.org/x/net/ipv4
github.com/vishvananda/netlink
golang.zx2c4.com/wireguard/tun
golang.org/x/net/icmp
github.com/quic-go/quic-go
github.com/yggdrasil-network/yggdrasil-go/src/core
github.com/yggdrasil-network/yggdrasil-go/src/ipv6rwc
github.com/yggdrasil-network/yggdrasil-go/src/admin
github.com/yggdrasil-network/yggdrasil-go/src/multicast
github.com/yggdrasil-network/yggdrasil-go/src/tun
github.com/yggdrasil-network/yggdrasil-go/cmd/yggdrasil

```
Building: yggdrasilctl
go: downloading github.com/mattn/go-runewidth v0.0.14
github.com/rivo/uniseg
github.com/mattn/go-runewidth
github.com/olekukonko/tablewriter
github.com/yggdrasil-network/yggdrasil-go/cmd/yggdrasilctl
```

Все необходимые модули Go подтягиваются по зависимостям.

И по итогу:

```
$ ls -l yggdrasil*
-rwxr-xr-x 1 olej olej 8347648 ноя  4 18:00 yggdrasil
-rwxr-xr-x 1 olej olej 5988352 ноя  4 18:00 yggdrasilctl
$ ./yggdrasil --version
Build name: yggdrasil-develop
Build version: 0.5.1-5-g6a9c90d
```

Настройка

Прежде чем запускать сервис, нам нужно будет подготовить его конфигурационный файл. (При установке из бинарного DEB-пакета мы ещё можем рассчитывать что за нас это сделали сборщики пакета, но при сборке из программного кода мы все настройки должны провести самостоятельно.)

Как минимум, первое, что мы должны сделать перед первичным запуском — это вписать адреса нескольких пиров (соседей) с которыми будет взаимодействовать хост, эти адреса вписывается в секцию Peers (как и откуда берутся эти адреса и как выбираются пиры мы поговорим позже), например:

```
# grep -v '^$|#' /etc/yggdrasil/yggdrasil.conf | grep " Peers:" -A8
Peers: [
    tcp://193.93.119.42:14244
    tcp://178.132.161.124:65533
    tcp://y.zbin.eu:7743
    tcp://88.210.3.30:65533
    tcp://87.251.77.39:65533
    tcp://sin.yuetau.net:6642
    tcp://mima.localghost.org:1996
]
```

Как уже сказано ранее, протоколы версий 0.4.X и 0.5.X несовместимы, а значит, набор пиров для этих версий будет **радикально отличаться**, должен быть подготовлен в соответствии с версией! Это принципиально важно (ссылки на ресурсы с регулярно обновляемыми адресами активных пиров, для каждой из версий, приведены в списке источников, в конце текста).

Как уже было сказано, уникальная пара ключей несимметричного шифрования генерируется при инсталляции пакета, они записываются в конфигурационный файл²⁶, именно они однозначно определяют сгенерированный IPv6 адрес хоста (и пока мы не будем отвлекаться на значения этих ключей):

```
# grep -v '^$|#' /etc/yggdrasil/yggdrasil.conf | grep Key:
PublicKey: 3cb7c934a4852779885762bb59781f9f54d771206460ca0d1c9f72311a7729f6
PrivateKey:
fa9f8209f5fe184a4632513676eef8e3d0b7ee798af11cb37d7ee8873eef37883cb7c934a4852779885762bb597
81f9f54d771206460ca0d1c9f72311a7729f6
```

(Любознательные читатели могут заметить, что в текущих версиях публичный ключ, шифрования трафика, имеет длину 64 байт, а приватный ключ представляет собственно 64 байт, плюс конкатенация таких же 64 байт публичного ключа — итого 128 байт. И в версиях 0.5.X PublicKey уже не обязан быть указан в конфигурационном файле, и при

²⁶ В версиях 0.5.X только один из двух.

автоматической трансформации конфигурационного файла из предыдущей версии он оттуда убирается. Но может и присутствовать при ручной конфигурации.)

Всё! В этой позиции мы можем запустить, если ещё, скажем, и не оптимизированную, но уже полноценно работающую пиринговую сеть Yggdrasil:

```
# systemctl status yggdrasil
● yggdrasil.service - yggdrasil
   Loaded: loaded (/lib/systemd/system/yggdrasil.service; disabled; vendor preset:
   enabled)
   Active: active (running) since Tue 2023-03-28 23:54:35 EEST; 19s ago
   Process: 43199 ExecStartPre=/sbin/modprobe tun (code=exited, status=0/SUCCESS)
   Main PID: 43200 (yggdrasil)
     Tasks: 16 (limit: 115786)
    Memory: 6.1M
       CPU: 93ms
   CGroup: /system.slice/yggdrasil.service
           └─43200 /usr/sbin/yggdrasil -useconffile /etc/yggdrasil/yggdrasil.conf

map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Interface MTU: 53049
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Your public key is
3cb7c934a4852779885762bb59781f9f54d771206460ca0d1c9f72311a7729f6
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Your IPv6 address is
202:1a41:b65a:dbd6:c433:bd44:ea25:343f
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Your IPv6 subnet is
302:1a41:b65a:dbd6::/64
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Connected TCP:
203:63fc:667d:b16c:8e78:a899:8d54:a5e4@193.111.114.28, source 192.168.1.14
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Connected TLS:
203:63fc:667d:b16c:8e78:a899:8d54:a5e4@193.111.114.28, source 192.168.1.14
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Connected TCP:
202:db60::9ce0:a73d:7498:d7ae@195.123.245.146, source 192.168.1.14
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Connected TLS:
200:4ac1:2516:a78:b43e:51e1:ab90:e2a2@54.37.137.221, source 192.168.1.14
map 28 23:54:35 R420 yggdrasil[43200]: 2023/03/28 23:54:35 Connected TCP:
218:71e5:78e4:8989:b71:db7f:7bf1:f1e1@94.130.176.250, source 192.168.1.14
map 28 23:54:36 R420 yggdrasil[43200]: 2023/03/28 23:54:36 Connected TLS:
218:71e5:78e4:8989:b71:db7f:7bf1:f1e1@94.130.176.250, source 192.168.1.14
```

В сообщениях старта сервиса нам уже указан адрес IPv6 который сгенерирован из пары ключей для этого хоста: 202:1a41:b65a:dbd6:c433:bd44:ea25:343f и адрес нашей подсети: 302:1a41:b65a:dbd6::/64. В нашей системе появится новый сетевой интерфейс tun0 с соответствующими параметрами:

```
$ ip a s dev tun0
4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc fq_codel state UNKNOWN
group default qlen 500
    link/none
    inet6 202:1a41:b65a:dbd6:c433:bd44:ea25:343f/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::4268:4330:6890:e026/64 scope link stable-privacy
        valid_lft forever preferred_lft forever
```

Теперь мы можем оперировать с новым интерфейсом IPv6 привычными сетевыми утилитами:

```
$ host howto.ygg.lib
howto.ygg.lib has IPv6 address 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c
$ ping -c5 -6 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c
PING 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c(222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c) 56 data
bytes
 64 bytes from 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c: icmp_seq=1 ttl=64 time=1310 ms
 64 bytes from 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c: icmp_seq=2 ttl=64 time=387 ms
 64 bytes from 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c: icmp_seq=3 ttl=64 time=175 ms
```

```

64 bytes from 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c: icmp_seq=4 ttl=64 time=173 ms
64 bytes from 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c: icmp_seq=5 ttl=64 time=228 ms

--- 222:a8e4:50cd:55c:788e:b0a5:4e2f:a92c ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4024ms
rtt min/avg/max/mdev = 172.586/454.564/1310.016/434.823 ms, pipe 2

```

К этому месту рассмотрения мы уже имеем полновесный хост сети Yggdrasil со всеми возможностями этой сети: IPv6 адресация со сплошным шифрованием трафика внутри сети. Всё дальнейшее описание будет касаться только **оптимизации** функционирования хоста, но никак не его работоспособности — работоспособность мы уже обеспечили. Обратите внимание, что нам не пришлось заниматься трудоёмкими настройками этой сети: маршрутизация и другие параметры... И сколько бы велик ни был размер нашей локальной сети, работу которой мы захотим распространить на пиринговую сеть, нам не придётся заниматься её администрированием.

Внимание: Сразу внесём ясность, что Yggdrasil использует IPv6 адресацию в диапазоне 0200::/7, который не рекомендуется IETF к использованию с 2004 года в качестве диапазона адресов для Интернет. Таким образом этот диапазон 0200::/7 не вызовет никаких конфликтов с IPv6 адресами, распределяемых для Интернет IANA. Но в «традиционном» IPv6 Интернет не будут доступны адреса Yggdrasil, так же как и Yggdrasil не маршрутизирует адреса не из своего диапазона (префикса) сети. В этом легко убедиться в вашей сети, для которой провайдер не предоставляет, как это чаще всего и бывает, IPv6 маршрутизацию: трафик прозрачно идёт, как в примере выше, на howto.ygg.lib — адрес Yggdrasil, но вот противоположный пример:

```

$ host ya.ru
ya.ru has address 5.255.255.242
ya.ru has address 77.88.55.242
ya.ru has IPv6 address 2a02:6b8::2:242
ya.ru mail is handled by 10 mx.yandex.ru.
$ ping -6 2a02:6b8::2:242
ping: connect: Сеть недоступна
$ traceroute 2a02:6b8::2:242
traceroute to 2a02:6b8::2:242 (2a02:6b8::2:242), 30 hops max, 80 byte packets
connect: Сеть недоступна

```

И ещё небольшое изменение, привнесённое версиями 0.5.X ... Утилита диагностики проекта yggdrasilctl (мы её уже видели при сборке из исходных кодов) взаимодействует с процессом yggdrasil через UNIX-сокеты, путь к которому указывается в конфигурационном файле. И в версиях 0.5.X он ожидается (если не прописан явно, то именно на этом пути):

```

$ grep "AdminListen: " /etc/yggdrasil.conf | grep -v \#
AdminListen: unix:///var/run/yggdrasil/yggdrasil.sock

```

(В версиях 0.4.X он определялся по другому пути `unix:///var/run/yggdrasil.sock`). И, как следствия (достаточно существенные), появление дополнительного **каталога** yggdrasil на пути сокета ведёт к соединяющему:

- **Ручной** запуск сети (для целей отладки, тестирования, диагностики...) с таким (новым) указанием пути сокета становится невозможным из-за отсутствия такого каталога. Как средство лечения (если вам нужен ручной запуск как средство, например, отладки и тестирования) — можно вручную создать такой каталог перед ручным запуском, примерно так:


```

$ sudo mkdir /var/run/yggdrasil
$ sudo yggdrasil -useconffile /etc/yggdrasil.conf
2023/11/04 18:09:34 Build name: yggdrasil-develop
2023/11/04 18:09:34 Build version: 0.5.1-5-g6a9c90d

```

```

2023/11/04 18:09:34 Your public key is
0000000262c2ad7ddb908877e0217a28dbf18e27c71f06cbdc449abc192a2e28
2023/11/04 18:09:34 Your IPv6 address is 21e:ce9e:a941:1237:bbc4:fef:42eb:9207
2023/11/04 18:09:34 Your IPv6 subnet is 31e:ce9e:a941:1237::/64
2023/11/04 18:09:34 UNIX admin socket listening on
/var/run/yggdrasil/yggdrasil.sock
2023/11/04 18:09:34 Interface name: tun0
...

```

- Сценарий запуска сервиса `systemd` (штатный способ использования), при ручной сборке и установке из исходников, должен быть дополнен 2-мя строками (при пакетной установке авторы предусмотрительно это сделали сами):

```

# cat /lib/systemd/system/yggdrasil.service
...
[Service]
...
RuntimeDirectory=yggdrasil
ReadWritePaths=/var/run/yggdrasil/ /run/yggdrasil/
...

```

При такой записи дополнительный каталог `/var/run/yggdrasil/` (на самом деле это ссылка на `/run/yggdrasil/`) **создаётся** при старте сервиса, и **удаляется** после его остановки.

Выбор внешних пиров для хоста²⁷

Первый вопрос который просится: откуда мы взяли адреса пиров (соседей), которые прописали в секции `Peers` конфигурации, и через которые хост и осуществляет свой роутинг? И как и какие пиры выбирать?

Понятно, что должны быть записаны несколько пиров наиболее доступных, самых быстрых (с наименьшей латентностью) со стороны вашего конфигурируемого хоста. А поэтому, понятно, нет никакого универсального набора таких пиров, а выбор зависит от вашей конкретно геолокации (местоположения вашего хоста), провайдера (обеспечивающего ваш трафик во внешний Интернет), топологии ваших ближайших соседей, и должен производиться **сугубо индивидуально**.

Хорошо... но из какого потенциального набора кандидатов производить выбор? Наверное, из-за важности этого выбора, сложилось несколько разнообразных источников для выбора адресов внешних пиров.

1. Фиксированный список. Это статический список, который сформирован и обновляется участниками проекта с некоторой периодичностью:

```

$ git clone https://github.com/yggdrasil-network/public-peers.git
Клонирование в «public-peers»...
remote: Enumerating objects: 3234, done.
remote: Counting objects: 100% (713/713), done.
remote: Compressing objects: 100% (385/385), done.
remote: Total 3234 (delta 466), reused 513 (delta 322), pack-reused 2521
Получение объектов: 100% (3234/3234), 756.35 КиБ | 2.84 МиБ/с, готово.
Определение изменений: 100% (1882/1882), готово.

```

```
$ tree public-peers
```

²⁷ Всё, что касается списков внешних пиров, или приложений, их тестирующих и отбирающих — это инициативы разработчиков из числа участников сообщества Yggdrasil, и то для каких семейств версий проекта (0.3.X, 0.4.X, 0.5.X) они относятся, в какие сроки они обновляются к версиям, и обновляются ли вообще — всё это определяется только желанием конкретного автора. Мы рассмотрим все известные проекты этой нацеленности (в виду критической важности этого аспекта), а проверить применимость их к конкретным версиям следует перед началом применения, «по месту».


```

public-peers
├── africa
│   └── south-africa.md
├── asia
│   ├── japan.md
│   ├── philippines.md
│   └── singapore.md
├── europe
│   ├── austria.md
│   ├── czechia.md
│   ├── finland.md
│   ├── france.md
│   ├── germany.md
│   ├── latvia.md
│   ├── moldova.md
│   ├── netherlands.md
│   ├── poland.md
│   ├── romania.md
│   ├── russia.md
│   ├── slovakia.md
│   ├── sweden.md
│   ├── switzerland.md
│   ├── ukraine.md
│   └── united-kingdom.md
├── mena
│   ├── saudi-arabia.md
│   └── turkey.md
├── north-america
│   ├── canada.md
│   └── united-states.md
├── other
│   ├── i2p.md
│   ├── lokinet.md
│   └── tor.md
├── README.md
└── south-america
    └── brazil.md

```

8 directories, 29 files

Как вы легко можете наблюдать, список представляет собой набор каталогов по континентам, файлы в которых вида *.md описывают доступные пиры по странам, например:

```

$ cat public-peers/europe/ukraine.md
# Ukraine peers

```

Add connection strings from the below list to the `Peers: []` section of your Yggdrasil configuration file to peer with these nodes.

```

* Bila Tserkva, operated by [ufm](ufm@ufm.lol)
  * `tcp://193.111.114.28:8080`
  * `tls://193.111.114.28:1443`

* Kiev, operated by [mvvpt](mvvpt0@bigmir.net)
  * `tcp://78.27.153.163:33165`
  * `tls://78.27.153.163:33166`
  * `tls://78.27.153.163:3785`
  * `tls://78.27.153.163:3784`
  * `tls://78.27.153.163:179`

```

Это один из самых коротких списков, по другим странам их списки могут составлять по несколько десятков пиров! Каждый узел содержит в описании 3 обязательных компонента: 1). протокол (tcp:, tls:, socks: quic:), 2). IP адрес (IPv4 или нативный IPv6, там где он поддерживается), 3). TCP порт (потому что все используемые протоколы являются вариациями потокового TCP). Именно в таком синтаксисе пиры и должны прописываться в конфигурационном файле Yggdrasil — каждый отдельной строкой, и никакие комментарии или пробелы в этой строке не допускаются и расцениваются как грубая синтаксическая ошибка.

Думаю, что логика использования этого источника совершенно понятна: для наиболее близкого территориального расположения выбираем несколько адресов пиров (пишут что 3-4 будет достаточно, но и 7-8 не помешает — некоторые из них имеют свойство «отваливаться» даже на интервале в 1-2 месяца). Хорошей практикой будет выбирать свой набор пиров **из разных стран** (может даже и континентов) — в нынешних условиях целые страны, из политических соображений, неожиданно принимают решения перекрыть IP-трафик с целых направлений, причём, при этом никак не уведомляя это и никак не мотивируя своих решений. А потом так же неожиданно восстанавливают трафик на отдельных направлениях. Более того, у разных провайдеров одной страны зоны перекрытия трафика могут различаться!

Каждый адрес из формируемого списка должен быть **обязательно** проверен на его текущую доступность (и скорость доступа), например, посредством ping — вчера адрес был доступен, а сегодня его владелец передумал: всё это частные инициативы участников сети Yggdrasil. Вот, к примеру, IP адрес пира, выбранный мною из самого свежего списка:

```
$ ping -c3 78.27.153.163
PING 78.27.153.163 (78.27.153.163) 56(84) bytes of data.

--- 78.27.153.163 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2026ms
```

По опыту: выявлять неработоспособность отдельного конкретного пира из 7-8 прописанных в конфигурационном файле Yggdrasil — это куда более хлопотно, чем предварительно проверить трафик к IP адресу пира²⁸ ещё на этапе формирования списка.

2. При переходе к версии 0.5.X (конец 2023 года) были сформированы 2 регулярно обновляющихся онлайн страниц URL внешних пиров (они приведены и ниже в перечне используемых источников информации), причём, отдельные страницы для пиров уходящих версий 0.4.X и идущих им на смену версий 0.5.X (версии протоколно несовместимы, поэтому пиры, пригодные для одной из версий, не годятся для другой):

- public peers (v0.4) : <https://publicpeers.neilalexander.dev/v0.4>
- public peers (v0.5) : <https://publicpeers.neilalexander.dev/v0.5>

Эти списки визуально наглядны, удобны для использования, но они формируются непосредственно из того GIT-репозитория, который показанного в предыдущем пункте, и информация, и здесь и там, обновляется регулярно, с часовым периодом. Вот что вы увидите в заголовке показанных выше онлайн страниц:

Online status is updated hourly. This page last updated at 10 Jan 24 12:00 UTC.

3. Сформированные списки для выбора — это здорово, но как отобрать 3-5-7 более-менее оптимальных пиров из нескольких (многих) десятков их, по доступности, а тем более по их латентности? Вот на сегодня число пиров, описанных в GIT-репозитории (и в дальнейшем это число имеет тенденцию возрастать):

```
$ cat public-peers/*/*.md | grep ^" *" | grep "://" | wc -l
```

²⁸ Не актуальность протокола или номера порта в записи пира встречается гораздо реже ... если встречается вообще.

Не ping-овать же несколько сот адресов, да ещё и неоднократно повторяя для достоверности?

Но тут нам на помощь приходит утилита проверки пригодности пиров и их ранжирования по латентности, написанная кем-то из сообщества Yggdrasil на Python:

```
$ git clone https://github.com/zhoreeq/peer_checker.py.git
Клонирование в «peer_checker.py»...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 10 (delta 3), reused 4 (delta 0), pack-reused 0
Получение объектов: 100% (10/10), готово.
Определение изменений: 100% (3/3), готово.
```

Скачанная утилита не требует никакой сборки (Python, интерпретация). Запуская просто без параметров получим подсказку по её использованию:

```
$ python ./peer_checker.py/peer_checker.py
Usage: ./peer_checker.py/peer_checker.py [path to public_peers repository on a disk]
I.e.: ./peer_checker.py/peer_checker.py ~/Projects/yggdrasil/public_peers
```

Если вы клонировали public-peers (списки из предыдущего пункта) и peer_checker в один и тот же каталог (а так и следует делать):

```
$ ls -ld *peer* | grep ^d
drwxrwxr-x 3 olej olej 4096 мар 28 2023 peer_checker.py
drwxrwxr-x 10 olej olej 4096 мар 28 2023 public-peers
```

... то вы можете запустить эту утилиту, «напустив» её на тот фиксированный список, который обсуждался и вы скачали ранее:

```
$ time python ./peer_checker.py/peer_checker.py ./public-peers/
Report date: Wed Jan 10 14:16:53 2024
Dead peers:

tcp://212.154.86.134:8800      mena/turkey.md
tls://212.154.86.134:4433    mena/turkey.md
tcp://ip6-antalya.ddns.net:8800      mena/turkey.md
tls://ip6-antalya.ddns.net:4433      mena/turkey.md
tls://158.101.229.219:17001         mena/saudi-arabia.md
tcp://158.101.229.219:17002         mena/saudi-arabia.md
tls://[2603:c023:8001:1600:35e0:acde:2c6e:b27f]:17001      mena/saudi-arabia.md
tcp://[2603:c023:8001:1600:35e0:acde:2c6e:b27f]:17002      mena/saudi-arabia.md
tls://[2001:41d0:304:200::ace3]:23108      europe/france.md
tls://[2001:41d0:2:c44a:51:255:223:60]:54232      europe/france.md
tcp://[2001:470:1f13:e56::64]:39565      europe/france.md
tls://[2001:470:1f13:e56::64]:39575      europe/france.md
tls://de-fsn-1.peer.v4.yggdrasil.chaz6.com:4444      europe/germany.md
tls://x-fra-0.sergeysedoy97.ru:65535      europe/germany.md
tls://87.251.77.39:65535      europe/germany.md
tls://[2a01:4f9:c010:664d::1]:61995      europe/finland.md
tls://[2a01:4f9:2a:60c::2]:18836      europe/finland.md
tcp://[2a05:9403::8b]:7743      europe/czechia.md
tls://[2a03:3b40:fe:ab::1]:993      europe/czechia.md
tls://[2a07:e01:105:444:c634:6bff:feb5:6e28]:7040      europe/sweden.md
tls://[2001:41d0:801:2000::233f]:28395      europe/united-kingdom.md
tls://[2a10:4740:40:0:2222:3f9c:b7cf:1]:43006      europe/united-kingdom.md
tcp://193.111.114.28:8080      europe/ukraine.md
```

...

Alive peers (sorted by latency):

URI	Latency (ms)	Location
tls://s-fra-0.sergeysedoy97.ru:65535	19.91	europa/germany.md
tcp://78.27.153.163:33165	20.621	europa/ukraine.md
tls://78.27.153.163:3784	22.605	europa/ukraine.md
tls://78.27.153.163:179	22.649	europa/ukraine.md
tls://78.27.153.163:3785	22.672	europa/ukraine.md
tls://78.27.153.163:33166	22.764	europa/ukraine.md
...		
tcp://sin.yuetau.net:6642	199.066	asia/singapore.md
tls://sin.yuetau.net:6643	199.098	asia/singapore.md
tls://44.234.134.124:443	199.775	north-america/united-states.md

real 0m5,495s
user 0m0,148s
sys 0m0,038s

Утилита требует на выполнение некоторое умеренное время (5-6 секунд), но она действительно сканирует по сети хосты из скачанного свежего списка. Утилита отсортировала «мёртвые» (Dead) адреса от «живых» (Alive), а последние, пригодные к использованию, отсортировала по времени латентности как она видится **из вашего местоположения** (это крайне важно! ... об этом чуть ниже).

Внимание: Множество адресов IPv6 в списке «мёртвых» может быть связано не только с их недоступностью, но и с тем (в большинстве), что ваш собственный **провайдер** Интернет просто не обеспечивает для вас трафик в нативном протоколе IPv6... А для русскоязычных территорий на сегодня большинство провайдеров и не спешит предоставлять клиентам IPv6 трафик. (Естественно, что в таком случае использовать пиры IPv6 недопустимо.)

Но гораздо удобнее сканированный список сразу сбросить в файл:

```
$ python ./peer_checker.py/peer_checker.py ./public-peers/ > yggdrasil_0_5.peers.txt
$ ls -l yggdrasil_0_5.peers.txt
-rw-r--r-- 1 olej olej 13273 янв 9 12:38 yggdrasil_0_5.peers.txt
```

А после этого для отбора использовать только последовательность «живых» пиров с наименьшей латентностью:

```
$ grep "Alive peers (sorted by latency):" yggdrasil_0_5.peers.txt -A20
Alive peers (sorted by latency):
```

URI	Latency (ms)	Location
tcp://78.27.153.163:33165	10.585	europa/ukraine.md
tls://78.27.153.163:33166	10.654	europa/ukraine.md
tls://78.27.153.163:3784	11.184	europa/ukraine.md
tls://78.27.153.163:3785	11.186	europa/ukraine.md
tls://78.27.153.163:179	11.307	europa/ukraine.md
tls://193.93.119.42:443	17.9	europa/ukraine.md
tcp://s-kiv-0.sergeysedoy97.ru:65533	22.364	europa/moldova.md
tls://s-mow-4.sergeysedoy97.ru:65534	22.555	europa/russia.md
tcp://s-mow-4.sergeysedoy97.ru:65533	22.577	europa/russia.md
tcp://s-mow-3.sergeysedoy97.ru:65533	22.82	europa/russia.md
tls://s-kiv-0.sergeysedoy97.ru:65534	22.855	europa/moldova.md
tls://pl1.servers.devices.cwinfo.net:11129	22.951	europa/poland.md
tls://s-led-0.sergeysedoy97.ru:65534	22.978	europa/russia.md
tls://s-ovb-1.sergeysedoy97.ru:65534	23.033	europa/russia.md

tls://s-kzn-0.sergeysedoy97.ru:65534	23.068	europe/russia.md
tcp://s-led-0.sergeysedoy97.ru:65533	23.091	europe/russia.md
tcp://s-kzn-0.sergeysedoy97.ru:65533	23.108	europe/russia.md
tcp://s-ovb-0.sergeysedoy97.ru:65533	23.166	europe/russia.md

А отбором (желательно из разных стран) из этого перечня пиров, для конфигурационного файла Yggdrasil, мы можем этот файл заполнить, например, так:

```
$ sudo grep " Peers:" /etc/yggdrasil/yggdrasil.conf -A9
```

```
Peers: [
  tls://s-fra-0.sergeysedoy97.ru:65535
  tls://pl1.servers.devices.cwinfo.net:11129
  tcp://195.123.245.146:7743
  tcp://62.210.85.80:39565
  tls://23.137.249.65:443
  tcp://curiosity.tdjs.tech:30003
  tls://65.21.57.122:61995
  tcp://srv.itrus.su:7991
]
```

Если же вы выполните точно ту же последовательность действий, но на сервере, к примеру, в Казахстане, за 4000 километров от показанного выше результата, то получите (на том же исходном списке пиров) совершенно другую, радикально отличающуюся, картину «ближних» пиров:

```
$ grep "Alive peers (sorted by latency):" yggdrasil_0_5.peers.txt -A20
```

```
Alive peers (sorted by latency):
```

URI	Latency (ms)	Location
tls://45.147.200.202:443	57.125	europe/russia.md
tcp://45.147.200.202:12402	60.728	europe/russia.md
tls://ygg-msk-1.averyan.ru:8362	61.855	europe/russia.md
tcp://ygg-msk-1.averyan.ru:8363	62.136	europe/russia.md
tcp://s-ovb-1.sergeysedoy97.ru:65533	63.264	europe/russia.md
tcp://s-mow-4.sergeysedoy97.ru:65533	64.155	europe/russia.md
tls://box.paulll.cc:13338	64.749	europe/russia.md
tcp://box.paulll.cc:13337	64.879	europe/russia.md
tls://x-mow-1.sergeysedoy97.ru:65534	65.668	europe/russia.md
tls://yggno.de:18227	65.757	europe/russia.md
tcp://178.20.41.3:65533	66.187	europe/russia.md
tcp://185.103.109.63:65533	66.239	europe/russia.md
tcp://188.225.9.167:18226	66.302	europe/russia.md
tls://188.225.9.167:18227	66.327	europe/russia.md
tls://s-kiv-0.sergeysedoy97.ru:65534	67.248	europe/moldova.md
tcp://s-led-0.sergeysedoy97.ru:65533	67.334	europe/russia.md
tcp://s-ovb-0.sergeysedoy97.ru:65533	67.378	europe/russia.md
tls://s-mow-4.sergeysedoy97.ru:65534	67.422	europe/russia.md

А сформированный список пиров, в конфигурационный файл, при этом может быть, например, таким:

```
$ sudo grep " Peers:" /etc/yggdrasil/yggdrasil.conf -A12 | grep -v ^# | grep -v ^$
```

```
Peers: [
  tcp://45.147.200.202:12402
  tcp://ygg-msk-1.averyan.ru:8363
  tls://yggno.de:18227
  tcp://178.20.41.3:65533
  tls://s-kiv-0.sergeysedoy97.ru:65534
  tls://fi1.servers.devices.cwinfo.net:61995
  tls://95.216.5.243:18836
```

```
    tls://aurora.devices.waren.io:18836
]
```

Сравните с тем, что показано выше для центрально-европейской части (то был восток Украины, Харьков)...

Это — к вопросу актуальности формирования индивидуального списка пиров, исходя из собственной географии, а не заимствование из готовых источников ... например, показанных выше!

Более того, и неоднократные эксперименты это показывают, что если вы со своим коллегой находитесь территориально и в 100 метрах друг от друга, но у вас разные **провайдеры** Интернет (которым вы платите за обеспечение своего трафика), то ранжированный по латентности список пиров у вас будет **различный**, и, как следствие, пиры для своего конфигурационного файла вам следует выбирать разные!

Примечание: Это приложение написано под синтаксис Python версии 3, и несовместимо, как это часто и бывает, с Python версии 2. Поэтому показанной выше командой оно будет успешно запускаться только в тех дистрибутивах, где команда python вызывает версию 3 ... а это было не так во многих дистрибутивах Linux выпуска ещё примерно до 2021-2022 годов. Если же в вашем дистрибутиве это не так, то вы можете доустановить пакет:

```
# apt install python-is-python3
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Следующие НОВЫЕ пакеты будут установлены:
  python-is-python3
Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 0
пакетов не обновлено.
Необходимо скачать 2.788 В архивов.
...
Подготовка к распаковке .../python-is-python3_3.9.2-2_all.deb ...
Распаковывается python-is-python3 (3.9.2-2) ...
Настраивается пакет python-is-python3 (3.9.2-2) ...
Обрабатываются триггеры для man-db (2.10.2-1) ...
```

Или можете явно указать версию Python в команде запуска:

```
$ python3 ./peer_checker.py/peer_checker.py ./public-peers/
...
```

4. Ещё один, свежий, инструмент получения списка готовых к использованию пиров:

```
$ git clone https://github.com/ygguser/peers_updater.git
Клонирование в «peers_updater»...
remote: Enumerating objects: 902, done.
remote: Counting objects: 100% (159/159), done.
remote: Compressing objects: 100% (84/84), done.
remote: Total 902 (delta 82), reused 129 (delta 72), pack-reused 743
Получение объектов: 100% (902/902), 157.31 КиБ | 598.00 КиБ/с, готово.
Определение изменений: 100% (540/540), готово.
```

Теперь это выписано на новом модном языке программирования Rust.

```
$ cd peers_updater
$ tree -L 2
.
├─ build.rs
├─ Cargo.toml
├─ CHANGELOG.md
└─ LICENSE
```

```

├─ README.md
├─ README_ru.md
├─ src
│   ├── cfg_file_read_write.rs
│   ├── clap_args.rs
│   ├── defaults.rs
│   ├── download_file.rs
│   ├── main.rs
│   ├── parsing_peers.rs
│   ├── peer.rs
│   ├── self_updating.rs
│   ├── tmpdir.rs
│   ├── unpack.rs
│   └─ using_api.rs

```

1 directory, 17 files

Примечание: Язык программирования Rust, без всякой иронии и преувеличения, является новым словом в высоко надёжном системном программировании. Но в данном случае задача чисто вычислительная и не требующая экстремальной устойчивости к ошибкам, так что здесь это скорее «проба пера»...

Если у вас уже установлена среда разработки языка программирования Rust, то сборка утилиты должна проходить «без сучка без задоренки» (с подтягиванием всего необходимого, крейтов²⁹ для сборки).

Примечание: Установка инструментальной системы Rust оригинальная, ни на что не похожая в сравнении с другими языковыми системами. Но она хорошо описана и, обычно, проходит легко и без проблем. Ссылка на подробную инструкцию по установке (на русском языке) уже была приведена ранее в списке источников информации к разделу, описывающему программирование сетевых приложений.

Ход сборки приложения `peers_updater` стоит показать полностью, хотя бы просто для того, чтобы видеть как динамически подгружаются все требуемые приложению крейты из сети:

```

$ time cargo build --release
    Updating crates.io index
  Downloaded foreign-types-shared v0.1.1
  Downloaded foreign-types v0.3.2
  Downloaded openssl-probe v0.1.5
  Downloaded openssl-macros v0.1.1
  Downloaded clap_lex v0.6.0
  Downloaded autocfg v1.1.0
  Downloaded untrusted v0.9.0
  Downloaded cfg-if v1.0.0
  Downloaded nu-json v0.84.0
  Downloaded anstyle v1.0.4
  Downloaded Adler v1.0.2
  Downloaded pkg-config v0.3.28
  Downloaded strsim v0.10.0
  Downloaded spin v0.9.8
  Downloaded quote v1.0.35
  Downloaded native-tls v0.2.11
  Downloaded crc32fast v1.3.2
  Downloaded once_cell v1.19.0
  Downloaded sct v0.7.1
  Downloaded byteorder v1.5.0

```

²⁹ В терминологии Rust крейт — это то что в Go (который тоже неоднократно упоминается в тексте) — модуль, а в других языках — пакет или библиотека. Всё это внешние компоненты сборки, со своей спецификой в каждом случае.

```

Downloaded bitflags v2.4.1
Downloaded getrandom v0.2.12
Downloaded clap v4.4.14
Downloaded proc-macro2 v1.0.76
Downloaded serde v1.0.195
Downloaded cc v1.0.83
Downloaded minreq v2.11.0
Downloaded log v0.4.20
Downloaded miniz_oxide v0.7.1
Downloaded num-traits v0.2.17
Downloaded flate2 v1.0.28
Downloaded unicode-ident v1.0.12
Downloaded openssl-sys v0.9.98
Downloaded zip v0.6.6
Downloaded memchr v2.7.1
Downloaded clap_builder v4.4.14
Downloaded rustls-webpki v0.101.7
Downloaded aho-corasick v1.1.2
Downloaded vcpkg v0.2.15
Downloaded syn v2.0.48
Downloaded regex v1.10.2
Downloaded webpki-roots v0.25.3
Downloaded openssl v0.10.62
Downloaded rustls v0.21.10
Downloaded regex-syntax v0.8.2
Downloaded regex-automata v0.4.3
Downloaded libc v0.2.152
Downloaded ring v0.17.7
Downloaded 48 crates (9.1 MB) in 45.38s (largest was `ring` at 4.1 MB)
Compiling libc v0.2.152
Compiling cfg-if v1.0.0
Compiling spin v0.9.8
Compiling untrusted v0.9.0
Compiling autocfg v1.1.0
Compiling crc32fast v1.3.2
Compiling serde v1.0.195
Compiling rustls v0.21.10
Compiling Adler v1.0.2
Compiling anstyle v1.0.4
Compiling log v0.4.20
Compiling minreq v2.11.0
Compiling regex-syntax v0.8.2
Compiling strsim v0.10.0
Compiling clap_lex v0.6.0
Compiling byteorder v1.5.0
Compiling peers_updater v0.3.3 (/home/olej/2024/Yggdrasil/peers_updater)
Compiling webpki-roots v0.25.3
Compiling once_cell v1.19.0
Compiling miniz_oxide v0.7.1
Compiling clap_builder v4.4.14
Compiling num-traits v0.2.17
Compiling flate2 v1.0.28
Compiling zip v0.6.6
Compiling cc v1.0.83
Compiling getrandom v0.2.12
Compiling regex-automata v0.4.3
Compiling ring v0.17.7
Compiling clap v4.4.14
Compiling regex v1.10.2

```



```

    Compiling nu-json v0.84.0
    Compiling rustls-webpki v0.101.7
    Compiling sct v0.7.1
    Finished release [optimized] target(s) in 1m 25s

real 1m25,098s
user 1m6,232s
sys 0m5,243s

```

Процесс загрузки крейтов по зависимостям занимает львиную долю всей сборки, последующая компиляция самого приложения, последним этапом, проходит практически мгновенно. (Повторные компиляции не требуют загрузки зависимостей, поэтому идут очень быстро.)

Примечание: Текущая версия проекта `peers_updater` потребует для своей сборки версии Rust (`cargo`) не ниже 1.70, иначе сборка завершится терминальной ошибкой (после 10 минут процессорной работы). Хотя ещё в апреле 2023 года эта же сборка делалась успешно версией 1.69. Не ленитесь обновлять версии инструментальной системы Rust!

```

$ cargo --version
cargo 1.75.0 (1d8b05cdd 2023-11-20)

$ ls -l target/release/peers_updater
-rwxrwxr-x 2 olej olej 1931688 янв 10 20:09 target/release/peers_updater

```

Выполнение:

```

$ cd target/release
$ time ./peers_updater -p > peers.txt
real    0m12,992s
user    0m0,044s
sys     0m0,057s

```

```
$ head -n20 peers.txt
```

URI	Region	Country	Latency
tcp://s-mow-3.sergeysedoy97.ru:65533	europe	russia	11
tls://s-mow-3.sergeysedoy97.ru:65534	europe	russia	11
tcp://s-mow-4.sergeysedoy97.ru:65533	europe	russia	12
tcp://s-led-0.sergeysedoy97.ru:65533	europe	russia	12
tls://s-led-0.sergeysedoy97.ru:65534	europe	russia	12
tcp://s-kiv-0.sergeysedoy97.ru:65533	europe	moldova	13
tls://s-kiv-0.sergeysedoy97.ru:65534	europe	moldova	13
tls://s-mow-4.sergeysedoy97.ru:65534	europe	russia	13
tcp://s-ovb-0.sergeysedoy97.ru:65533	europe	russia	13
tls://s-ovb-0.sergeysedoy97.ru:65534	europe	russia	13
tcp://s-ovb-1.sergeysedoy97.ru:65533	europe	russia	13
tls://s-ovb-1.sergeysedoy97.ru:65534	europe	russia	13
tcp://s-kzn-0.sergeysedoy97.ru:65533	europe	russia	13
tls://s-kzn-0.sergeysedoy97.ru:65534	europe	russia	13
tls://78.27.153.163:33166	europe	ukraine	15
tls://78.27.153.163:3785	europe	ukraine	15
tls://78.27.153.163:179	europe	ukraine	15
tcp://s-fra-0.sergeysedoy97.ru:65533	europe	germany	15

```

$ cat peers.txt | wc -l
131

```

Найдено 131 свежих пиров, ранжированных по латентности. Совсем неплохо!

5. Проверка того какие пиры реально использует ваш хост, после того как вы их прописали в конфигурации и перезапустили сервис. (Здесь каждый URI, как он назван в заголовке, представляется на терминале одной длинной строкой, но отобразить это в тексте невозможно.):

```
$ sudo yggdrasilctl getpeers
```

URI	State	Dir	IP Address	Uptime	RX	TX	Pr	Last Error
tcp://193.93.119.42:14244	6h37m46s	3mb	52mb 0	Up	Out	22d:d3dd:3afe:9599:3da9:d89f:6ae:8401	-	-
tls://[fe80::13f5:9fe2:6393:bf4a%25eth0]	6h37m46s	4mb	48kb 0	Up	In	21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f	-	-
tls://[fe80::9bac:3791:1b79:7237%25eth0]	6h37m46s	19mb	13mb 0	Up	In	21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f	-	-
tcp://178.132.161.124:65533	6h37m46s	103kb	92kb 0	Up	Out	222:10d:b0b4:f237:5db9:e47c:a2e6:9ab9	-	-
tcp://88.210.3.30:65533	6h37m46s	109kb	90kb 0	Up	Out	225:124e:2a74:9966:d18d:2f19:adbf:a950	-	-
tcp://87.251.77.39:65533	6h37m46s	111kb	89kb 0	Up	Out	224:79c3:da07:db06:60ef:282d:f908:d97	-	-
tcp://y.zbin.eu:7743	6h37m46s	119kb	100kb 0	Up	Out	200:dc42::3fca:8a99:5d66:b660	-	-
tcp://sin.yuetau.net:6642	6h37m46s	110kb	112kb 0	Up	Out	200:b858:3f39:af76:83ae:bb1:ebb2:604e	-	-
tcp://mima.localghost.org:1996	6m4s	4kb	8kb 0	Up	Out	200:b1ff:9827:dc87:f067:c43b:d546:1dd7	-	-

Особое внимание нужно уделить последней колонке, Last Error (в показанном примере там стоит прочерк): внешние пиры могут исчезать и переходить в нерабочее состояние... на то они и внешние.

Майнинг IPv6 адресов³⁰

С точки зрения защищённости, потенциально существует возможность, путём грандиозного перебора, подобрать такие значения PublicKey и PrivateKey, которые будут производить известный адрес, и тем создавать угрозу сетевого спуфинга (подмена, фальсификация адресата). По крайней мере, о такой гипотетической угрозе пишут в обсуждениях...

В Yggdrasil, где адрес IPv6 выводится математически, а топология не является ответственностью уполномоченных лиц, единственный сценарий спуфинга – подбор злоумышленником пары ключей PublicKey и PrivateKey, которые образуют известный адрес. Из логики алгоритма образования адреса (которая описана в приведенных в конце части источниках, но не входит в цель наших обсуждений) получается, что 2-й байт адреса (первый байт это ведущие «0x02») — это число первых непрерывно идущих нулевых бит PublicKey — повышенное число таких нулевых бит создают дополнительный фактор уникальности адреса, увеличивают численное значение 2-го байта IPv6 адреса, и усложняют задачу подбора.

Для рядовых рабочих станций это вообще не имеет существенного значения. А вот для серверов стараются подобрать (а позже занести в конфиг-файл) такую пару ключей, чтобы они давали максимальное число ведущих нулей в PublicKey. Процесс поиска таких значений достижим только путём случайного перебора, по логике вычислений соответствует поиску следующего блока для криптовалюты с хэшем меньше порога, а поэтому, по аналогии, также называется **майнингом** (на этом сходство и заканчивается). Процесс подбора ключей является крайне трудоёмким, с экспоненциально возрастающим интервалом времени для поиска следующих комбинаций. На самом высоко производительном оборудовании перебор может продолжаться (до тех пор пока вы не сочтёте его достаточным!) непрерывно

³⁰ Алгоритм формирования IPv6 исходя из значений ключей остаётся не зависящим от используемой версии проекта, на сегодня, по крайней мере. Поэтому найденные ключи (процесс требующий огромной вычислительной трудоёмкости) пригодны к использованию независимо от версий.

часами, а, возможно, и сутками. Для выполнения этой работы в Yggdrasil предложено несколько вариантов утилит.

Первая из таких утилит это SimpleYggGen-CPP:

```
$ git clone https://notabug.org/acetone/SimpleYggGen-CPP.git
Клонирование в «SimpleYggGen-CPP»...
remote: Counting objects: 1293, done.
remote: Compressing objects: 100% (527/527), done.
remote: Total 1293 (delta 748), reused 1281 (delta 740)
Получение объектов: 100% (1293/1293), 1.23 МиБ | 999.00 КиБ/с, готово.
Определение изменений: 100% (748/748), готово.
$ ls -l SimpleYggGen-CPP
итого 56
-rw-rw-r-- 1 olej olej 81 anp 3 21:19 CMakeLists.txt
-rw-rw-r-- 1 olej olej 34570 anp 3 21:19 LICENSE
-rw-rw-r-- 1 olej olej 2838 anp 3 21:19 README.md
drwxrwxr-x 4 olej olej 4096 anp 3 21:19 src
drwxrwxr-x 2 olej olej 4096 anp 3 21:19 src-qt
drwxrwxr-x 3 olej olej 4096 anp 3 21:19 test
```

Как следует и из названия, утилита написана на C++ и собирается с помощью Cmake, обычным для Cmake способом:

```
$ cd SimpleYggGen-CPP
$ mkdir _build && cd _build
$ cmake -G "Unix Makefiles" ..
-- The C compiler identification is GNU 11.3.0
-- The CXX compiler identification is GNU 11.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/olej/2023/Yggdrasil/SimpleYggGen-CPP/_build
$ make -j
[ 66%] Building CXX object src/CMakeFiles/sygcpp.dir/main.cpp.o
[ 66%] Building CXX object src/CMakeFiles/sygcpp.dir/parameters.cpp.o
[100%] Linking CXX executable sygcpp
[100%] Built target sygcpp

$ ls -l src/sygcpp
-rwxrwxr-x 1 olej olej 290464 anp 3 21:40 src/sygcpp
olej@R420:~/2023/Yggdrasil/SimpleYggGen-CPP/_build$ ls -l src/sygcpp
-rwxrwxr-x 1 olej olej 290464 anp 3 21:40 src/sygcpp
$ src/sygcpp --version
+-----+
|                               [ SimpleYggGen C++ 5.1-flow ]                               |
|                               EdDSA public key -> IPv6 -> Meshname                               |
|                               notabug.org/acetone/SimpleYggGen-CPP                               |
|                               |                               |                               |
|                               GPLV3 (c) 2021                               |
|                               |                               |                               |
+-----+
```

```
$ src/sygcpp --help
```

```
+-----+
|           Simple Yggdrasil address miner usage:  --help or -h           |
+-----+
[Mining modes]
  High addresses                                BY DEFAULT |
  IPv6 by pattern                               --ip | -i
  IPv6 by pattern + height                       --ip-high | -ih
  IPv6 by regular expression                     --regex | -r
  IPv6 by regular expression + height            --regex-high | -rh
  Meshname by pattern                           --mesh | -m
  Meshname by regular expression                 --mesh-regex | -mr
  Subnet brute force (300::/64)                  --brute-force | -b
[Main parameters]
  Threads count (maximum by default)             --threads | -t <value>
  String for pattern or regular expression        --pattern | -p <value>
  Start position for high addresses (14 by default) --altitude | -a <value>
[Extra options]
  Disable auto-increase in high mode             --increase-none | -in
  Disable logging to text file, stdout only       --logging-none | -ln
  Force display meshname domains                  --display-mesh | -dm
  Show PrivateKeys in full format in console      --full-pk | -fp
  Show the version of the miner                   --version | -v
[Meshname convertation]
  Convert IP to Meshname                         --tomes | -tm <value>
  Convert Meshname to IP                         --toip | -ti <value>
[Notes]
  Meshname domains use base32 (RFC4648) alphabet symbols.
  In meshname domain mining should use "=" instead ".meship" or ".meshname".
  Subnet brute force mode understand "3xx:" and "2xx:" patterns.
+-----+
```

Запуск на майнинг выглядит так (для всех рассматриваемых инструментов): сначала результативные строки идут довольно часто, потом всё реже и реже, ... продолжаем это до тех пор, пока нас устроит ведущее число нулей в PublicKey:

```
$ time src/sygcpp
```

```
SimpleYggGen was started without parameters.
The mining mode for high addresses will be launched automatically.
Use --help for usage information.
+-----+
|           [ SimpleYggGen C++ 5.1-flow ]           |
|           EdDSA public key -> IPv6 -> Meshname    |
|           notabug.org/acetone/SimpleYggGen-CPP    |
|           |                                       |
|           GPLV3 (c) 2021                        |
+-----+
Threads: 40, high addresses (214++), logging to text file.

Address: 216:813f:6186:1ced:92bb:9571:4780:936f
PublicKey: 000002fd813cf3c624da88d51d70fed920ae9c4687d188719176a9b3c98a276a
PrivateKey: 95660c4077f8dfb1c7ac04860ad0acfae400833be3a17b86b81b0cc8b521410c

kH/s: [___564] Total: [_____3200008] Found: [__1] Time: [0:00:00:07]
kH/s: [___570] Total: [_____6400012] Found: [__1] Time: [0:00:00:12]

Address: 218:47de:1d18:13ae:e5a5:c8f9:50b2:af76
PublicKey: 000000dc10f173f6288d2d1b8357a6a844fb0e6ce9a3910b05cd5178ec295ea3
PrivateKey: f93d3c5f9bd856b3101fb8d3e63ae84d5ad37bcc0744e7f0ee11a54fcab26f39
```

```
kH/s: [___567] Total: [_____9600007] Found: [__2] Time: [0:00:00:18]
```

```
Address: 21a:cdd9:136c:5e0e:25e9:3fcf:848d:e3cd
```

```
PublicKey: 0000002644dd92743e3b42d8060f6e43865b7835618c97b144130f943819a3be
```

```
PrivateKey: b7ab2dc8e4e3e5f335388a5e04f85422d399cb75c8c584b0aa318d1c7d5934ca
```

```
kH/s: [___569] Total: [_____12800004] Found: [__3] Time: [0:00:00:24]
```

```
kH/s: [___569] Total: [_____16000005] Found: [__3] Time: [0:00:00:29]
```

```
kH/s: [___567] Total: [_____19200007] Found: [__3] Time: [0:00:00:35]
```

```
kH/s: [___569] Total: [_____22400019] Found: [__3] Time: [0:00:00:41]
```

```
kH/s: [___569] Total: [_____25600022] Found: [__3] Time: [0:00:00:47]
```

```
kH/s: [___570] Total: [_____28800005] Found: [__3] Time: [0:00:00:52]
```

```
kH/s: [___570] Total: [_____32000006] Found: [__3] Time: [0:00:00:58]
```

```
kH/s: [___568] Total: [_____35200001] Found: [__3] Time: [0:00:01:04]
```

```
kH/s: [___571] Total: [_____38400005] Found: [__3] Time: [0:00:01:09]
```

```
kH/s: [___570] Total: [_____41600006] Found: [__3] Time: [0:00:01:15]
```

```
kH/s: [___569] Total: [_____44800027] Found: [__3] Time: [0:00:01:21]
```

```
Address: 21b:83ff:70cf:8a69:90d5:5a1f:7d6:339
```

```
PublicKey: 00000017c008f3075966f2aa5e0f829fcc68e06ebc7fcc38fb3c3bf6d8a4b939
```

```
PrivateKey: 9313b0cc394be3c108ad05886442d630fdc81539d475b804d5f42d9d7abf3815
```

...

Вторая, альтернативная, реализация для тех же целей, более поздняя — это `syg_go`, написана на языке Go (утверждается что она может быть при определённых условиях производительнее ... но я ничего такого не наблюдал):

```
$ git clone http://github.com/tdemin/syg_go
```

```
Клонирование в «syg_go»...
```

```
warning: переадресация на https://github.com/tdemin/syg_go/
```

```
remote: Enumerating objects: 140, done.
```

```
remote: Counting objects: 100% (140/140), done.
```

```
remote: Compressing objects: 100% (63/63), done.
```

```
remote: Total 140 (delta 75), reused 138 (delta 73), pack-reused 0
```

```
Получение объектов: 100% (140/140), 35.32 КиБ | 168.00 КиБ/с, готово.
```

```
Определение изменений: 100% (75/75), готово.
```

Построение исполнимой утилиты, если у вас установлена типовая среда разработки Go, не представляет особых проблем, но можно просто воспользоваться готовым релизом, скачав его отсюда: https://git.tdem.in/tdemin/syg_go/releases.

```
$ ls -l syg_go_0.2.0_Linux_x86_64.tar.gz
```

```
-rw-rw-r-- 1 olej olej 937653 anp 9 16:11 syg_go_0.2.0_Linux_x86_64.tar.gz
```

После разархивирования:

```
$ ls -l
```

```
итого 2176
```

```
-rw-r--r-- 1 olej olej 8747 июл 5 2021 LICENSE
```

```
-rw-r--r-- 1 olej olej 3097 июл 5 2021 README.md
```

```
-rwxr-xr-x 1 olej olej 2207744 июл 5 2021 syg_go
```

```
-rw-rw-r-- 1 olej olej 3912 anp 9 18:43 syg-ipv6-high.txt
```

```
$ ./syg_go -help
```

```
Usage of ./syg_go:
```

```
-highaddr
```

```
high address mining mode, excludes regex
```

```
-iter uint
```

```
per how many iterations to output status (default 100000)
```

```
-original
```

```

    use original Yggdrasil code
-regex string
    regex to match addresses against (default "::-")
-threads int
    how many threads to use for mining (default 40)
-version
    display version
$ ./syg_go --version
syg_go 0.2.0
Copyright (c) 2021 Timur Demin

```

Запуск майнинга, или как его ещё называют «поиск высоких адресов»... картина та же: сначала достаточно частые результаты, потом они всё реже и реже:

```

$ time ./syg_go -highaddr
2023/04/09 16:13:57 using syg_go vendored code
2023/04/09 16:13:57 starting mining higher addresses with 40 threads
2023/04/09 16:13:57 priv:
0156acc54893f6aa915ea4286bd74afaf07013066281d71a9625686da1548caff4a2742937f38b9a05dd6a3f5a3
1f86ce0a816ad60b3e02bc1fd2002c3ba2758 | pub:
f4a2742937f38b9a05dd6a3f5a31f86ce0a816ad60b3e02bc1fd2002c3ba2758 | ip:
200:16bb:17ad:9018:e8cb:f445:2b81:4b9c
2023/04/09 16:13:57 priv:
a71c18f9705e580105d5822809a9ba3972865fc0d9189dc60132d5bf2c015b49152ef90e31d460139ac8a0f6efa
c85db2418b4c3b9b0b85e5c8f9c3fe121c19e | pub:
152ef90e31d460139ac8a0f6efac85db2418b4c3b9b0b85e5c8f9c3fe121c19e | ip:
203:ad10:6f1c:e2b9:fec6:5375:f091:537
2023/04/09 16:13:57 priv:
45a258a60db77043fe9cc8e1d50e58404f2c11ca08af4875d0aad01d03fafdf3137ed3a4ce5b8ec7b2409e59428
1a7565eb88906a80d67ceb64d3b7d47c0dfdd4 | pub:
137ed3a4ce5b8ec7b2409e594281a7565eb88906a80d67ceb64d3b7d47c0dfdd4 | ip:
203:c812:c5b3:1a47:1384:dbf6:1a6b:d7e5
2023/04/09 16:13:57 priv:
d1c0ea03b1029412f179e97426514f3e6f20d68ee9e2ff24cd14fcacbc4c284d108bd227921b0a914d6597e04572
485cfa826123a33f30bb4821b80b29275aa82 | pub:
08bd227921b0a914d6597e04572485cfa826123a33f30bb4821b80b29275aa82 | ip:
204:e85b:b0db:c9ea:dd65:34d0:3f75:1b6f
2023/04/09 16:13:57 priv:
19fab30534a7e7825caa56754b49be7f18851410ae6f0d1b479c2867a27a87206eff8c6da8aa60e1e88ca96324
39a844e95ad60237b2ac4aa711c872d73eaf4 | pub:
06eff8c6da8aa60e1e88ca9632439a844e95ad60237b2ac4aa711c872d73eaf4 | ip:
205:4401:ce49:5d56:7c78:5dcd:5a73:6f19
2023/04/09 16:13:57 priv:
beca64b2bdc2448d547a7a3c660b8c60df2f92798a215afcc00a7cfee240d9a602d1bfc034113170237aac93c8c
49a783e2a3407c01f57112378aab6941cce78 | pub:
02d1bfc034113170237aac93c8c49a783e2a3407c01f57112378aab6941cce78 | ip:
206:9720:1fe5:f767:47ee:42a9:b61b:9db2
2023/04/09 16:13:57 priv:
8e273aa1c0e0657d554a3b6e2f4ffd8d44b214dbf5a297315824c775694b094b01417dda33aecf03bbba15038cf
f7bb87b7d80d4540e78520fadbd7f3c80360f | pub:
01417dda33aecf03bbba15038cfff7bb87b7d80d4540e78520fadbd7f3c80360f | ip:
207:be82:25cc:5130:fc44:45ea:fc73:84
2023/04/09 16:13:57 priv:
7d08cac7a607a9fee1dc933cce5fc17d67d03e8d50f6e5f134325e9d44967594009e84646ac803dd7a5bb57fd53
229ee8f261ea4b9446690154052e7697fa1f5 | pub:
009e84646ac803dd7a5bb57fd53229ee8f261ea4b9446690154052e7697fa1f5 | ip:
208:c2f7:372a:6ff8:450b:4895:55:9bac
2023/04/09 16:13:57 priv:
43d5662db521c5c3ae77058d7e63fff1e7ae72b911d29decdf64bebd04e0a61d00067dedd84de5357895bb86d5cb
eb759e51de174fedc4918b5aa0a1b6d2143f8 | pub:
0067dedd84de5357895bb86d5cbbeb759e51de174fedc4918b5aa0a1b6d2143f8 | ip:
209:6084:89ec:86b2:a1da:911e:4a8d:522
2023/04/09 16:13:57 priv:
9fb43928c805846fb76bdae35b9d8447378aa7c6358ae27fbc133611ee90d338001babb5412bb0ba6edd5367fb6
3610fe8ca7b90983d610b340cccb831e3f50f | pub:
001babb5412bb0ba6edd5367fb63610fe8ca7b90983d610b340cccb831e3f50f | ip:
20b:4544:abed:44f4:5912:2ac9:8049:c9ef
2023/04/09 16:13:57 priv:
58a98238e2cc4d8f15bdebfd2df09efc19d5878d9b909606b688d45c36ef0e3b000b28e81127a32c9d7b3734963

```

```

1992c227906721923a94a39097f326f41c559 | pub:
000b28e81127a32c9d7b37349631992c227906721923a94a39097f326f41c559 | ip:
20c:9ae2:fddb:b9a:6c50:9919:6d39:ccda
2023/04/09 16:13:57 priv:
c4401abdd63c73a44c66272c942358cc3427ff4ca5212ca8b25408aa93f8524d0000059274f225183a3600648a3
a80f22bbdf86ee9dcc33e876b946ae78b605c | pub:
0000059274f225183a3600648a3a80f22bbdf86ee9dcc33e876b946ae78b605c | ip:
215:9b62:c376:b9f1:727f:e6dd:715f:c375
2023/04/09 16:13:58 reached 100000 iterations
2023/04/09 16:13:58 reached 200000 iterations
2023/04/09 16:13:58 reached 300000 iterations
2023/04/09 16:13:59 reached 400000 iterations
2023/04/09 16:13:59 reached 500000 iterations
2023/04/09 16:14:00 reached 600000 iterations
2023/04/09 16:14:00 reached 700000 iterations
2023/04/09 16:14:01 reached 800000 iterations
2023/04/09 16:14:01 reached 900000 iterations
2023/04/09 16:14:01 reached 1000000 iterations
2023/04/09 16:14:02 reached 1100000 iterations
2023/04/09 16:14:02 reached 1200000 iterations
2023/04/09 16:14:02 priv:
afa07b2542645d585a9d842bd6b9bfa36cd1c1123e8aec29ba4381daaf4fd7230000020566de41a835e8c269010
b632a0f7a66b9a413f9b8f666cf0ac2350d90 | pub:
0000020566de41a835e8c269010b632a0f7a66b9a413f9b8f666cf0ac2350d90 | ip:
216:fd4c:90df:2be5:b9e:cb7f:7a4e:6af8
2023/04/09 16:14:03 reached 1300000 iterations
2023/04/09 16:14:03 reached 1400000 iterations
2023/04/09 16:14:04 reached 1500000 iterations
2023/04/09 16:14:04 reached 1600000 iterations
2023/04/09 16:14:04 reached 1700000 iterations
...

```

Чтобы не погрязнуть в этих миллионах пустых итераций, запускать можно так (и с сохранением найденных результатов в файл для последующего использования):

```

$ time ./syg_go -highaddr 2>/dev/null | tee syg-ipv6-high.txt
2023/04/09 18:02:43 using syg_go vendored code
2023/04/09 18:02:43 starting mining higher addresses with 40 threads
2023/04/09 18:02:43 priv:
0d5c209a915d47d263ca0aec484e991a2bc06c1a8284705c3b4e45a6fa8b4d0213007587070fad1192879f1310d
dbf6ef95bc0ffdd5ccdb3b4f8215abc67945dc | pub:
13007587070fad1192879f1310ddb6ef95bc0ffdd5ccdb3b4f8215abc67945dc | ip:
203:cff8:a78f:8f05:2ee6:d786:ece:f224
2023/04/09 18:02:43 priv:
e8446b97716fda08d135242934a2fee694162ff531d4bf4c3112cf3f9aa3118900934873b39eccb13981b7a7988
82e30143ead7a6d62605f34d120f2432bf6b8 | pub:
00934873b39eccb13981b7a798882e30143ead7a6d62605f34d120f2432bf6b8 | ip:
208:d96f:1898:c266:9d8c:fc90:b0ce:efa3
2023/04/09 18:02:43 priv:
9566f018fdee58eb807bd1117c10cb52897f588b6fd86383881e448326b177c3004344bf6ae5ebcad7721a77e19
b00fc5b8338148366da5a7c995eb690cf01bd | pub:
004344bf6ae5ebcad7721a77e19b00fc5b8338148366da5a7c995eb690cf01bd | ip:
209:f2ed:254:6850:d4a2:3796:2079:93fc
2023/04/09 18:02:43 priv:
dcd95c0c1761bf627e3a7db58af95d66422dd7f4cd5e29400f1c1deccef365f700223d49177bbda8af1fcd64ad0
4151cf332e0562c6b20065342d7ad4907d535 | pub:
00223d49177bbda8af1fcd64ad04151cf332e0562c6b20065342d7ad4907d535 | ip:
20a:ee15:b744:2212:ba87:194:da97:df57
2023/04/09 18:02:43 priv:
88a3b8e2a950ab0998a26d5f3846c08010e0b84c9cecb1174b90e6394e122bdc00020fcdfb850a3d066ba674783
db660e6c8cc66d86ccaeaaace4c641fc274c35 | pub:
00020fcdfb850a3d066ba674783db660e6c8cc66d86ccaeaaace4c641fc274c35 | ip:
20e:f819:23d:7ae1:7cca:2cc5:c3e1:24cf
2023/04/09 18:02:43 priv:
b7c63456cb51f7230b8a14ec207aef57811c5287523f7da7d46411630e78881c0001331c9bd3ca4b598be2117c3
77fe2298bd98789812bc509684957416664d4 | pub:

```

```

0001331c9bd3ca4b598be2117c377fe2298bd98789812bc509684957416664d4 | ip:
20f:cce3:642c:35b4:a674:1dee:83c8:801d
2023/04/09 18:02:43 priv:
5dc1f633f9c5f98e5844bca676a4ca084ba73839f8f7a63649dd4751974a42ac00007f0dfa34c5c1c700ff0fb83
2d676d3f27d95591ffffe7efab78cc721bbd76 | pub:
00007f0dfa34c5c1c700ff0fb832d676d3f27d95591ffffe7efab78cc721bbd76 | ip:
211:3c8:172c:e8f8:e3fc:3c1:1f34:a624
2023/04/09 18:02:43 priv:
4525c551d30b9630d828bdf97123c836c3b6547e7eba8827805ccd99a7bcb4c800006dce5b8b28b69fcbbf7689c
a585d3b403aa6c0f73ad4a84efa3ec0c15338 | pub:
00006dce5b8b28b69fcbbf7689ca585d3b403aa6c0f73ad4a84efa3ec0c15338 | ip:
211:48c6:91d3:5d25:80d1:225:d8d6:9e8b
2023/04/09 18:02:45 priv:
893ebeac1086b55c641a173e734ec95d510380d8be58414e9ed3915c4e549fd200006c0747664d74427440a20f0
5af138618188bf870ec369f92a3cafc373d42 | pub:
00006c0747664d74427440a20f05af138618188bf870ec369f92a3cafc373d42 | ip:
211:4fe2:e266:ca2e:f62e:fd77:c3e9:43b1
2023/04/09 18:02:45 priv:
5a2aec4bc93b126e51a1602859bbb91cca57c318ec5c5795c2070886d77ceba00001c6fa8d15428155d100a97b
d3d4ecd59a4f84cf2b40d4dae3b57302b108c | pub:
00001c6fa8d15428155d100a97bd3d4ecd59a4f84cf2b40d4dae3b57302b108c | ip:
213:3905:72ea:bd7e:aa2e:ff56:842c:2b13
2023/04/09 18:02:50 priv:
d559d3ed28ba5c9f69b33f8eeff9aa162e136f7d01ac949f3c7120d263c1d644000007a723d3e94063f7a38441
6a2edaf211744f2e29fe77ffedf1be3434288 | pub:
0000007a723d3e94063f7a384416a2edaf211744f2e29fe77ffedf1be3434288 | ip:
219:1637:b05:afe7:217:1eef:a574:4943
2023/04/09 18:04:08 priv:
2e63d58b5c37acaa3d0ce2cd05531f6e692df12d28dafa50fbd3548f70d31c6800000016544de8bc80c739b38de
7bcd77fc1dc98c892dd48f62e510b90cd64da | pub:
00000016544de8bc80c739b38de7bcd77fc1dc98c892dd48f62e510b90cd64da | ip:
21b:9abb:2174:37f3:8c64:c721:8432:8803
2023/04/09 18:27:23 priv:
076e23435b8c361818e6c7776e4a410d9e6c4a65ed98367ca7df8d8be1afc2c800000005d60d541432e5fbabad5
58d6cca02aaddbb3363fcd3a8421934e4d575 | pub:
00000005d60d541432e5fbabad558d6cca02aaddbb3363fcd3a8421934e4d575 | ip:
21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f
2023/04/09 18:43:17 priv:
8469aaef0296fb7d2c9a97008593c83d9b64b04a9b99f759eb105153ddafcd2f00000002a1150bb1ff15e07bb75
8c468f31e75cbf8d17bc246364377cf18eae3 | pub:
00000002a1150bb1ff15e07bb758c468f31e75cbf8d17bc246364377cf18eae3 | ip:
21e:af75:7a27:75:fc2:2453:9dc8:8670
^C
real 98m15,868s
user 2980m53,540s
sys 67m51,329s

```

Как и было обещано — наблюдаем: с каждым найденным результатом число ведущих нулей в `pub` становится всё больше и больше, соответствующий ему `ip` всё выше и выше, а интервал поиска следующей комбинации — всё длиннее и длиннее...

Найденные значения приватного и публичного ключ записываем в конфигурационный файл, заменяем в нём соответствующие строки:

```

$ sudo grep "Key:" /etc/yggdrasil/yggdrasil.conf | grep -v ^$
  PublicKey: 00001c6fa8d15428155d100a97bd3d4ecd59a4f84cf2b40d4dae3b57302b108c
  PrivateKey:
5a2aec4bc93b126e51a1602859bbb91cca57c318ec5c5795c2070886d77ceba00001c6fa8d15428155d100a97b
d3d4ecd59a4f84cf2b40d4dae3b57302b108c

```

Соответствующий им IPv6 адрес нам показан в майнинге только для справки, для сверки — он нигде не прописывается, и будет установлен автоматически, после перезапуска сервиса с новыми значениями ключей (если только где-то в конфигурации не допущена ошибка).

И, наконец, последний из рассмотренных инструментов поиска «высоких адресов» — находится непосредственно в самом дереве исходных кодов сети Yggdrasil (мы его

скачивали ранее при рассмотрении сборки проекта из исходных кодов, каталог `yggdrasil-go`), от авторов проекта³¹:

- код на языке Go:

```
$ ls -l cmd/genkeys
```

```
итого 4
```

```
-rw-rw-r-- 1 olej olej 1856 ноя 6 18:49 main.go
```

- его выполнение (GoLang в режиме непосредственного выполнения):

```
$ go run ./cmd/genkeys
```

```
Threads: 40
```

```
----- 4.413558ms
```

```
Priv:
```

```
5d7403012004d401253ed66a63f66f828de7f8a5b66d370918a71ee8adbb5e049be3229026ba3ff178e371b2676afe56ef907884f6e428aa973d6e3a256df903
```

```
Pub: 9be3229026ba3ff178e371b2676afe56ef907884f6e428aa973d6e3a256df903
```

```
IP: 200:c839:badf:b28b:801d:e39:1c9b:312a
```

```
----- 4.50002ms
```

```
Priv:
```

```
f42ca53112a216c3b069b58324046ac63cdf1b4a24a3b10302c54acd9373974635ea7fe9669b03586f98da958a523249972a4d5c5af07e709d7cd75dab235ba5
```

```
Pub: 35ea7fe9669b03586f98da958a523249972a4d5c5af07e709d7cd75dab235ba5
```

```
IP: 202:50ac:b4:cb27:e53c:8339:2b53:ad6e
```

```
...
```

```
----- 37m16.432034055s
```

```
Priv:
```

```
12077d252a0057e0b92c1a604a50063ff33e548be2fdb3db5f17147bc9118a440000000267272d8bbfc769d55f79aa92172f53c06e29e313d95cd8f9e5a8fcde
```

```
Pub: 0000000267272d8bbfc769d55f79aa92172f53c06e29e313d95cd8f9e5a8fcde
```

```
IP: 21e:cc6c:693a:201c:4b15:5043:2ab6:f468
```

```
...
```

Здесь показан фрагмент поиска, выполняющийся на промышленном сервере в 40 процессорных ядрах. Как и было отмечено ранее, **любой** из показанных инструментов требует очень высокой вычислительной трудоёмкости — даже на очень приличных процессорах, получения некоторого следующего («оптимального») значения может потребовать до нескольких суток непрерывной процессорной работы³². Разумно прогнозируйте порядки требуемых вам величин!

Yggdrasil в локальной сети

В LAN, где один из хостов уже подключен по IPv4 через внешние пиры, как показано было выше, для подключения следующих хостов Yggdrasil к P2P роумингу вовсе им не обязательно прописывать внешние пиры в секции `Peers` Конфигурационного файла. Внутри локальной сети хосты обнаруживают друг друга за счёт механизма Multicast Peer Discovery (MPL) — технология автоматического обнаружения пиров Yggdrasil в локальной сети по их link-local IPv6 адресам.

Когда в локальной сети Yggdrasil запущено на одном или нескольких устройствах, то благодаря этой технологии все остальные узлы обнаружат друг друга, и автоматически установят соединение между собой, прописывать их в секции `Peers` нет необходимости. Multicast Peer Discovery включено по-умолчанию. Открывает UDP порт 9001 (для прослушивания сигналов о существовании от других узлов).

Важно (и это часто является предметом неработоспособности и беспокойства): Multicast Peer Discovery в Yggdrasil работает только с link-local IPv6 адресами устройств в локальной сети (диапазон `fe80::/10`), с IPv4 это не работает! Таким образом вы должны проследить что на интерфейсах вашей локальной сети должна быть включена поддержка IPv6 и устройства

³¹ Я не могу сказать с какой версии проекта Yggdrasil в состав включено это средство, но оно должно завершить, думается, моду на изготовление инструментов «майнинга высоких адресов».

³² По субъективным ощущениям автора, все рассмотренные инструменты поиска (майнинга) обладают примерно соизмеримой скоростью нахождения значений.

должны иметь IPv6 link-local адреса в локальной сети (между ними должен работать ping). Поддержка IPv6 может быть запрещена, например, настройками Network Manager.

Проверяем что поддержка IPv6 на интерфейсе хоста включена (строка inet6):

```
$ ip a s dev enp3s0
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    link/ether 10:7b:44:47:a2:47 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.241/24 brd 192.168.1.255 scope global dynamic noprefixroute enp3s0
        valid_lft 106770sec preferred_lft 106770sec
    inet6 fe80::522d:d0bd:b221:a526/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

Теперь мы можем запускать хост Yggdrasil в такой вот конфигурации, вообще без указания внешних пиров:

```
$ sudo grep "Peers:" /etc/yggdrasil/yggdrasil.conf
Peers: []
InterfacePeers: {}

$ systemctl status --no-pager --full yggdrasil
• yggdrasil.service - yggdrasil
   Loaded: loaded (/etc/systemd/system/yggdrasil.service; enabled; vendor preset:
   enabled)
   Active: active (running) since Thu 2023-04-27 11:58:18 EEST; 6 days ago
   Main PID: 1302 (yggdrasil)
   Tasks: 12 (limit: 18948)
   Memory: 28.4M
   CPU: 23min 42.051s
   CGroup: /system.slice/yggdrasil.service
           └─1302 /usr/bin/yggdrasil -useconffile /etc/yggdrasil.conf
```

```
...
$ sudo yggdrasilctl getPeers
```

Port	Public Key	IP
Address	Uptime	URI
1	00000005d60d541432e5fbabad558d6cca02aaddbb3363fcd3a8421934e4d575 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f tls://[fe80::13f5:9fe2:6393:bf4a%enp3s0]:42099	157kb 0
2	00000005d60d541432e5fbabad558d6cca02aaddbb3363fcd3a8421934e4d575 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f tls://[fe80::9bac:3791:1b79:7237]:45769	74kb 0

Здесь видно, что в качестве пиров для этого хоста в LAN использованы только link-local IPv6 адреса его 2-х соседей по LAN. Но вот с самого хоста коннект к глобальному Yggdrasil хосту, находящемуся за тысячи километров, и здесь трафик маршрутизируется транзитом через 2 соседние хосты в LAN (fe80::13f5:9fe2:6393:bf4a%enp3s0 и fe80::9bac:3791:1b79:7237):

```
$ host ygg.linux-ru.lib
ygg.linux-ru.lib has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771

$ ping -6 -c3 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
PING 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771(221:58c9:9a6:99be:f3d:c1ac:2b5b:9771) 56 data
bytes
 64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=2 ttl=64 time=122 ms
 64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=3 ttl=64 time=122 ms

--- 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 ping statistics ---
 3 packets transmitted, 2 received, 33.3333% packet loss, time 2004ms
 rtt min/avg/max/mdev = 121.583/121.758/121.933/0.175 ms
```

Это и есть автоконфигурация достаточно сложной сети, о чём говорилось ранее.

Короткие адреса

Когда у вас работает сервис, то основные параметры работающей сети вы можете запросить командой:

```
$ sudo yggdrasilctl getSelf
Build name:      yggdrasil
Build version:   0.4.7
IPv6 address:    21c:4cbf:9e0b:2904:301c:1c7a:9715:62c0
IPv6 subnet:     31c:4cbf:9e0b:2904::/64
Coordinates:     [2 41 179 8]
Public key:      0000000d9a030fa6b7de7f1f1c2b4754e9fb66086cf971f7997576e842902ef4
```

Откуда в адресе сети возникло 31c:... вместо 21c:...? Маршрутизация в Yggdrasil, как уже было сказано работает по принципу сети IPv6 0200::/7, с префиксом 7. Это означает что самой сети соответствуют только старшие 7 бит. Явно выписав не указанные ноли в 1-м (старшем) хекстете, покажем выделением те биты, которые ответственны здесь за адрес сети: 021c => **0000 0010** ... — последний (младший) двоичный разряд старшего байта не попадает в префикс. Таким образом для Yggdrasil могут использоваться как адреса вида 02XX:... так и 03XX:... А раз это так, то мы можем над этим сетевым интерфейсом (а, вообще то говоря, над любым сетевым интерфейсом этого хоста, не только над интерфейсом смотрящим непосредственно в Интернет) выполнить операцию присвоения **любого** адреса из этой подсети (просто дописав к IPv6 произвольный последний хекстет):

```
$ ip a s dev tun0
18: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc pfifo_fast state
UNKNOWN group default qlen 500
    link/none
    inet6 21c:4cbf:9e0b:2904:301c:1c7a:9715:62c0/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::fad7:4892:ed54:d5a9/64 scope link stable-privacy
        valid_lft forever preferred_lft forever

$ sudo ifconfig tun0 inet6 add 31c:4cbf:9e0b:2904::2/64
$ ip a s dev tun0
18: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc pfifo_fast state
UNKNOWN group default qlen 500
    link/none
    inet6 31c:4cbf:9e0b:2904::2/64 scope global
        valid_lft forever preferred_lft forever
    inet6 21c:4cbf:9e0b:2904:301c:1c7a:9715:62c0/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::fad7:4892:ed54:d5a9/64 scope link stable-privacy
        valid_lft forever preferred_lft forever
```

Теперь интерфейс tun0 приобрёл дополнительный 300::/64 короткий адрес, и я могу с далёкого хоста Yggdrasil, за тысячи километров (из SSH сессии), тут же выполнить проверку этого:

```
$ ping -6 -c3 31c:4cbf:9e0b:2904::2
PING 31c:4cbf:9e0b:2904::2(31c:4cbf:9e0b:2904::2) 56 data bytes
64 bytes from 31c:4cbf:9e0b:2904::2: icmp_seq=1 ttl=64 time=680 ms
64 bytes from 31c:4cbf:9e0b:2904::2: icmp_seq=2 ttl=64 time=121 ms
64 bytes from 31c:4cbf:9e0b:2904::2: icmp_seq=3 ttl=64 time=121 ms

--- 31c:4cbf:9e0b:2904::2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 4ms
rtt min/avg/max/mdev = 121.131/307.450/679.974/263.414 ms
```

Такие адреса (300::/64), мало того, что они заметно короче и тем удобны для использования, но они позволяют сделать много удобных вещей, как и будет сделано в следующей главе.

Если же такое присвоение короткого адреса вы хотите сделать перманентным, сохранялось после перезагрузки, то **одним из** способов можно в конфиг `/etc/systemd/system/yggdrasil.service` дописать в конце секции `[Service]` дописать пару строк такого типа:

```
ExecStartPost=/usr/bin/sleep 2s
ExecStartPost=/usr/bin/ip address add 31c:4cbf:9e0b:2904::2/64 dev tun0
```

Команда `systemctl`, правда, при этом станет нещадно ругаться (красным текстом) примерно так:

```
# systemctl cat yggdrasil.service
# Warning: yggdrasil.service changed on disk, the version systemd has loaded is outdated.
# This output shows the current version of the unit's original fragment and drop-in files.
# If fragments or drop-ins were added or removed, they are not properly reflected in this
output.
# Run 'systemctl daemon-reload' to reload units.
...
```

Но как и следует из подсказки, выполняем:

```
# systemctl daemon-reload
```

И всё приходит в соответствие, наблюдаем:

```
# systemctl cat yggdrasil.service
...
[Service]
ExecStartPost=/usr/bin/sleep 2s
ExecStartPost=/usr/bin/ip address add 31c:4cbf:9e0b:2904::2/64 dev tun0
...
```

Можем теперь, пошагово, остановить сервис:

```
# systemctl stop yggdrasil
# ip a s dev tun0
Device "tun0" does not exist.
```

И сразу же убедиться в работоспособности после его поднятия:

```
# systemctl start yggdrasil
# ip a s dev tun0
21: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc pfifo_fast state
UNKNOWN group default qlen 500
    link/none
    inet6 31c:4cbf:9e0b:2904::2/64 scope global
        valid_lft forever preferred_lft forever
    inet6 21c:4cbf:9e0b:2904:301c:1c7a:9715:62c0/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::b97b:6249:ef2d:5dd/64 scope link stable-privacy
        valid_lft forever preferred_lft forever
```

И видим, что что при старте сервиса появляется дополнительная задержка 2 секунда, которую мы ввели (критически необходима, чтобы дать паузу сетевому интерфейсу подняться, величину задержки: 2-3-4 секунды подберите экспериментально).

Работа в Yggdrasil даже без установки клиента

Но показанное выше это ещё не всё. Если в локальной сети есть хост (хотя бы один) уже увязанный в Yggdrasil, то для других хостов этой LAN можно организовать роутинг в Yggdrasil даже вообще **не устанавливая** там никакого специального программного обеспечения, клиента ... не хлопоча о версиях, конфигурационных файлах и прочем. Более того, это можно сделать даже несколькими способами (точная ссылка в источниках информации). Но мне из этих способов кажется наиболее интересным способ организации роутинга нативным способом сети, вообще без привлечения каких-либо дополнительных программных инструментов. Что мы сейчас и сделаем...

Идея на качественном уровне («на пальцах») сводится к следующему:

1. Если устройство поддерживает протокол IPv6 в LAN (link-local), то его интерфейсу можно присвоить произвольный IPv6 из соответствующей подсети 300::/64;
2. Аналогичным образом, на хосте-шлюзе к сети Yggdrasil (транзитёре), физическому интерфейсу смотрящему внутрь LAN, также можно присвоить другой произвольный адрес подсети 300::/64;
3. В качестве шлюза по умолчанию для подключаемого хоста указываем этот 300::/64 адрес роутера;
4. А на хосте-шлюзе разрешаем (достаточно традиционным для Linux способом) **форвардинг** для протокола IPv6 между различными интерфейсами (собственно, интересует форвардинг между интерфейсом глядящим в Интернет, и интерфейсом глядящим в LAN ... при наличии большего числа интерфейсов).

Сначала сделаем подготовительную работу на активном хосте-шлюзе Yggdrasil:

```
$ ip a s tun0
3: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc pfifo_fast state
UNKNOWN group default qlen 500
    link/none
    inet6 21e:af75:7a27:75:fc2:2453:9dcb:8670/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::eeec:dd70:b702:96bd/64 scope link stable-privacy
        valid_lft forever preferred_lft forever
```

Адрес этого интерфейса в Yggdrasil 21e:af75:7a27:75:fc2:2453:9dcb:8670/7, тогда для подключения других устройств к этой подсети изнутри LAN необходимо им назначить адреса такого вида (подсеть): 31e:af75:7a27:75::/64 (старшую 2-ку исправил на 3-ку). Это могут быть адреса с **любыми** младшими позициями: 31e:af75:7a27:75::1, 31e:af75:7a27:75::2 ... и так далее. Обращаем внимание, что первая цифра адреса 2 изменена на 3, далее 3 секции с разделителем «:» - это обязательная часть префикса 64. Остальные 64 бита (4 последние секции) — произвольные числа от 0000 до ffff.

Итак, на активном хосте Yggdrasil присвоим интерфейсу глядящему в LAN произвольный адрес из этой подсети (алиас), скажем так:

```
# ip address add 31e:af75:7a27:75::1/64 dev eth0
# ip a s dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    link/ether b8:27:eb:7e:2c:a8 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.142/24 brd 192.168.1.255 scope global dynamic noprefixroute eth0
        valid_lft 107400sec preferred_lft 78260sec
    inet6 31e:af75:7a27:75::1/64 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::5b06:3ef5:5b91:15d3/64 scope link
        valid_lft forever preferred_lft forever
```

(Хорошей, оригинальной, идеей здесь может быть присвоение требуемого IPv6 петлевому интерфейсу lo 127.0.0.1. Идея смешная, но и это работает!)

На этом же хосте (это шлюз в Yggdrasil) проверяем и при необходимости разрешаем форвардинг IPv6 между всеми интерфейсами хоста, у меня это (было запрещено — рашрешили):

```
$ cat /proc/sys/net/ipv6/conf/all/forwarding
0
# sysctl -w net.ipv6.conf.all.forwarding=1
net.ipv6.conf.all.forwarding = 1
# cat /proc/sys/net/ipv6/conf/all/forwarding
1
```

Для того, чтобы эта установка сохранялась постоянно, после перезагрузки, проверим и при необходимости уберём комментарий (или допишем такую строку):

```
# grep net.ipv6.conf.all.forwarding /etc/sysctl.conf
net.ipv6.conf.all.forwarding=1
# grep -v ^# /etc/sysctl.conf | grep -v ^$
net.ipv6.conf.all.forwarding=1
```

Примечание: Иногда (не могу сказать с чем это связано) все эти манипуляции, проведенные для `ipv6.conf.all` не приносят желаемого результата для интересующих нас конкретных 2-х интерфейсов. Поэтому следует проверять, а при необходимости и выполнить такие действия индивидуально, для конкретных интерфейсов. В этом вам поможет вот эта подсказка:

```
$ ls -l /proc/sys/net/ipv6/conf/*/forwarding
-rw-r--r-- 1 root root 0 ноя 29 16:44 /proc/sys/net/ipv6/conf/all/forwarding
-rw-r--r-- 1 root root 0 ноя 29 16:44 /proc/sys/net/ipv6/conf/default/forwarding
-rw-r--r-- 1 root root 0 ноя 29 16:44 /proc/sys/net/ipv6/conf/enp3s0/forwarding
-rw-r--r-- 1 root root 0 ноя 29 16:44 /proc/sys/net/ipv6/conf/lo/forwarding
-rw-r--r-- 1 root root 0 ноя 28 07:34 /proc/sys/net/ipv6/conf/tun0/forwarding
$ cat /proc/sys/net/ipv6/conf/tun0/forwarding
1
```

На этом подготовку шлюза в Yggdrasil заканчиваем. Теперь на любом другом хосте LAN (подключаемом клиенте) присваиваем интерфейсу в LAN (если их несколько, например, проводной и WiFi, то любому) произвольный IPv6 из той же подсети `31e:af75:7a27:75::/64`:

```
$ ip a s dev eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
    altnam enp0s25
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute eno1
        valid_lft forever preferred_lft forever
    inet6 fe80::762:c6bf:9eaa:93a9/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
# ip address add 31e:af75:7a27:75::5/64 dev eno1
$ ip a s dev eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 70:71:bc:a3:c5:c0 brd ff:ff:ff:ff:ff:ff
    altnam enp0s25
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute eno1
        valid_lft forever preferred_lft forever
    inet6 31e:af75:7a27:75::5/64 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::762:c6bf:9eaa:93a9/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

И прописываем здесь для этого нового адреса маршрут (вот это самое важное действие) на хост-шлюз в Yggdrasil:

```
# ip -6 route add 0200::/7 via 31e:af75:7a27:75::1
# route -6 | grep eno1
```

Таблица маршрутизации ядра IPv6

Destination	Next Hop	Flag	Met	Ref	Use	If
31e:af75:7a27:75::/64	::	U	256	1	0	eno1
200::/7	31e:af75:7a27:75::1	UG	1024	1	0	eno1
fe80::/64	::	U	1024	4	0	eno1
nvidia/128	::	Un	0	3	0	eno1
nvidia/128	::	Un	0	6	0	eno1

```
ip6-mcastprefix/8          [::]          U      256 5      0 eno1
[::]/0                     [::]          !n     -1 1      0 lo
```

(Нас в этой таблице роутинга интересует 2-я строка.)

Всё! Проверяем ping на хост-роутер Yggdrasil:

```
$ ping -c3 31e:af75:7a27:75::1 -I eno1
PING 31e:af75:7a27:75::1(31e:af75:7a27:75::1) from 31e:af75:7a27:75::5 eno1: 56 data bytes
64 bytes from 31e:af75:7a27:75::1: icmp_seq=1 ttl=64 time=0.779 ms
64 bytes from 31e:af75:7a27:75::1: icmp_seq=2 ttl=64 time=0.834 ms
64 bytes from 31e:af75:7a27:75::1: icmp_seq=3 ttl=64 time=0.751 ms

--- 31e:af75:7a27:75::1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2052ms
rtt min/avg/max/mdev = 0.751/0.788/0.834/0.034 ms
```

И, наконец, ping на IPv6 хост Yggdrasil где-то далеко в глобальной сети:

```
$ host ygg.linux-ru.lib
ygg.linux-ru.lib has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
$ ping -c3 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
PING 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771(221:58c9:9a6:99be:f3d:c1ac:2b5b:9771) 56 data
bytes
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=1 ttl=64 time=901 ms
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=2 ttl=64 time=149 ms
64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=3 ttl=64 time=150 ms

--- 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 148.929/399.765/900.526/354.091 ms
```

Скоростные показатели

Если mesh-сети (пиринговые) настолько просты в конфигурировании и использовании, то возникает последний вопрос: насколько значительно в них снижается скорость передачи **в сравнении** со статически (традиционно) определённой топологией трассы прохождения. (А ещё и с тем, что весь трафик пиринговой сети шифруется несимметричным шифрованием).

Сравнение сделаем способом, который, чаще, применяют для оценки скоростных показателей в локальных сетях, используя утилиту `iperf`.

Первое сравнение сделаем на трассе **глобальной** протяжённости, пара хостов: сервер в Алматы, Казахстан и клиент в Харькове, Украина; расстояние по оценкам различных онлайн ресурсов 3990-4006 километров. Со стороны глобального сервера ограничивающим фактором будет то, что на нём, как это всегда принято, работает файрвол, предельно ограничивающий число открытых портов TCP/IPv4, и мы не можем использовать параметры утилиты `iperf` по умолчанию, а должны выбрать порт сервера: а). открытый файрволом, б). не используемый постоянно каким-либо сервисом, препятствующим открытию дублирующего серверного сокета и в). из числа не привилегированных (больше 1024) не требующих от нас привилегий `root`:

```
$ sudo ufw status
Status: active
```

To	Action	From
--	-----	----
OpenSSH	ALLOW	Anywhere
WWW Full	ALLOW	Anywhere
...		
8080	ALLOW	Anywhere
...		
8080 (v6) on tun0	ALLOW	Anywhere (v6)
...		

```

$ ip -4 a s dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    inet 90.156.230.27/24 brd 90.156.230.255 scope global dynamic eth0
        valid_lft 70867sec preferred_lft 70867sec

```

Перечисленным выше требованиям удовлетворяет, например, порт 8080, который и примем для экспериментов. Со стороны клиента оценим предварительно скорость к Интернет, которую предоставляет клиенту его провайдер, чтобы ориентироваться с масштабом цифр, которые предстоит оценивать:

```

$ speedtest-cli
Retrieving speedtest.net configuration...
Testing from Radio-Link LLC (193.28.177.117)...
Retrieving speedtest.net server list...
Selecting best server based on ping...
Hosted by SOLVER Ltd (Poltava) [129.44 km]: 19.14 ms
Testing download speed.....
Download: 48.33 Mbit/s
Testing upload speed.....
Upload: 49.78 Mbit/s

```

Теперь у нас проделаны все предварительные телодвижения, и можем перейти к оцениванию скорости обмена на трассе протяжённостью порядка 4000 километров...

Со стороны сервера:

```

$ iperf -s -p8080
-----
Server listening on TCP port 8080
TCP window size: 128 KByte (default)
-----
[ 4] local 90.156.230.27 port 8080 connected with 193.28.177.117 port 40776
[ ID] Interval      Transfer    Bandwidth
[ 4]  0.0-10.3 sec  57.3 MBytes  46.4 Mbits/sec
^C

```

Со стороны клиента:

```

$ iperf -c 90.156.230.27 -p8080
-----
Client connecting to 90.156.230.27, TCP port 8080
TCP window size: 85.0 KByte (default)
-----
[ 3] local 192.168.1.241 port 40776 connected with 90.156.230.27 port 8080
[ ID] Interval      Transfer    Bandwidth
[ 3]  0.0000-10.0049 sec  57.3 MBytes  48.0 Mbits/sec

```

Мы получили оценки скорости передачи практически совпадающие с тем лимитом который предоставляет клиенту его провайдер доступа в Интернет.

А теперь сделаем то же самое, но через пиринговую сеть Yggdrasil и используя протокол IPv6.

Со стороны сервера:

```

$ ip -6 a s dev tun0
3: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 state UNKNOWN qlen 500
    inet6 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::d86f:9577:d828:cb4d/64 scope link stable-privacy
        valid_lft forever preferred_lft forever

```



```

$ iperf -s -p8080 --ipv6_domain
-----
Server listening on TCP port 8080
TCP window size: 128 KByte (default)
-----
[ 4] local 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 port 8080 connected with
31c:4cbf:9e0b:2904::2 port 51824
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0-10.4 sec   55.5 MBytes   44.8 Mbits/sec
^C

```

Со стороны клиента:

```

$ iperf -c 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 -p8080 --ipv6_domain
-----
Client connecting to 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771, TCP port 8080
TCP window size: 1.25 MByte (default)
-----
[ 3] local 31c:4cbf:9e0b:2904::2 port 51824 connected with
221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 port 8080
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0000-10.0632 sec 55.5 MBytes   46.3 Mbits/sec

```

Сравним со скоростью прямого соединения IPv4 (классический Интернет) ... по оценкам, например, со стороны сервера: $44,8 / 46,4 = 0,965517241$, и это 96.5% от скорости IPv4, и это с учётом полного шифрования трафика на трассе IPv6!

В чём различие 2-х показанных выше экспериментов? Только ли в том, что используются разные протоколы IPv6 против IPv4? Нет! В случае использования IPv4 (традиционный Интернет) трафик всегда идёт по **статическому** маршруту, настроенному трудами десятков, а то и сотен, сетевых администраторов на всех промежуточных хостах трассы, прописывающих объёмные таблицы роутинга на своих хостах. (Это та трасса из десятков, обычно, промежуточных хостов, которые мы видим командой `traceroute`, но на сегодня команда эта стала мало информативная, поскольку администраторы большинства транзитных узлов, опасаясь за безопасность, настраивают их так, чтобы не отвечать на ICMP запросы.)

Трафик же IPv6 Yggdrasil идёт по **динамически** прокладываемой трассе, исходя из местоположения пиров сети в общей топологии (дереве) сети Yggdrasil. И эта топология дерева, время от времени, автоматически перестраивается в связи с появлением новых или исчезновением старых пиров. Это, коротко, и есть то, что выше упоминалось как автоконфигурация — никто из штатных системных администраторов не прикладывает к этому руки.

Но не стоит и обольщаться такими достаточно высокими скоростными показателями — это обеспечивается близостью пиринговых «соседей», которые вы прописываете **вручную** в конфигурации сети Yggdrasil. Если конфигурированные пиры двух хостов принадлежат близкой географической области, то трафик между ними будет идти по короткой трассе и на максимальных скоростях.

А дальше посмотрим как ведёт себя пиринговая сеть на максимально высоких, технически достижимых, скоростях — какие накладные расходы она привносит во внутренний трафик по локальной сети. Для этого используем всё того же клиента протестируем, но в качестве сервера подкинем ему его ближайшего соседа в локальной сети гигабитной скорости (1000 Mbits/sec):

```

$ ip -4 a s dev enp3s0
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    inet 192.168.1.138/24 brd 192.168.1.255 scope global dynamic noprefixroute enp3s0
        valid_lft 168102sec preferred_lft 168102sec

```

Теперь, при прямом IPv4 соединении в локальной сети, со стороны сервера:

```
$ iperf -s
-----
Server listening on TCP port 5001
TCP window size: 128 KByte (default)
-----
[ 4] local 192.168.1.138 port 5001 connected with 192.168.1.241 port 40382
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0000-10.0192 sec  1.10 GBytes   941 Mbits/sec
^C
```

Со стороны клиента:

```
$ iperf -c 192.168.1.138
-----
Client connecting to 192.168.1.138, TCP port 5001
TCP window size: 85.0 KByte (default)
-----
[ 3] local 192.168.1.241 port 40382 connected with 192.168.1.138 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0000-10.0057 sec  1.10 GBytes   943 Mbits/sec
```

Это очень хорошо согласуется с условиями эксперимента.

А теперь то же самое, но сквозь пиринговую сеть...

Сервер:

```
$ ip -6 a s dev tun0
4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 state UNKNOWN qlen 500
    inet6 201:617b:8699:5a7f:356b:1da6:a5d6:c960/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::879e:ccf9:5c8d:95b8/64 scope link stable-privacy
        valid_lft forever preferred_lft forever
$ iperf -s --ipv6_domain
-----
Server listening on TCP port 5001
TCP window size: 128 KByte (default)
-----
[ 4] local 201:617b:8699:5a7f:356b:1da6:a5d6:c960 port 5001 connected with
31c:4cbf:9e0b:2904::2 port 59208
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0000-10.0252 sec  1.06 GBytes   909 Mbits/sec
^C
```

То же с ответной стороны, со стороны клиента:

```
$ iperf -c 201:617b:8699:5a7f:356b:1da6:a5d6:c960 --ipv6_domain
-----
Client connecting to 201:617b:8699:5a7f:356b:1da6:a5d6:c960, TCP port 5001
TCP window size: 1.25 MByte (default)
-----
[ 3] local 31c:4cbf:9e0b:2904::2 port 59208 connected with
201:617b:8699:5a7f:356b:1da6:a5d6:c960 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0000-10.0002 sec  1.06 GBytes   912 Mbits/sec
```

И даже на этих скоростях мы получаем $909 / 941 = 0,965993624$ и это 96.5% от прямого традиционного IPv4 соединения на 1000 Mbits/sec.

А что в других ОС?

Огромным преимуществом проекта Yggdrasil есть то, что (кроме разнообразия поддерживаемых аппаратных архитектур, что было показывалось ранее) он портирован под разнообразные операционные системы, и кроме Linux это: Windows, FreeBSD, OpenBSD, MacOS ... практически во все широко используемые операционные системы. Это критически важно именно для сетевых проектов, потому что, кроме вас, важно какой совместимый инструментарий использует каждый ваш неизвестный корреспондент, где-то за тысячи километров на просторах Интернет.

Очень коротко взглянем на то как Yggdrasil отображается в специфику FreeBSD — ещё одной UNIX-подобной операционной системы, любимой POSIX-публикой кроме Linux... Это акцентирует внимание и на тех критических позициях в ходе запуска, на которые стоит обратить внимание, и контролировать, при запуске в любых других операционных системах, или их версиях и модификациях.

Пакетная система FreeBSD достаточно богата, в ней уже присутствует проект GoLang (язык программирования Go), и исполнимые файлы сети Yggdrasil, поэтому, собираются из исходных кодов так же легко, как и показано выше для дистрибутива Fedora. Но есть более специфичный для FreeBSD способ, который обеспечит более простую и корректную последующую настройку сети Yggdrasil, а именно — установка самой свежей³³ коллекции портов исходных кодов пакетов (если у вас есть определённый запас ресурсов и времени):

```
# git clone https://git.FreeBSD.org/ports.git /usr/ports
git clone https://git.FreeBSD.org/ports.git /usr/ports
Cloning into '/usr/ports'...
remote: Enumerating objects: 6010289, done.
remote: Counting objects: 100% (942/942), done.
remote: Compressing objects: 100% (126/126), done.
remote: Total 6010289 (delta 923), reused 822 (delta 816), pack-reused 6009347
Receiving objects: 100% (6010289/6010289), 1.14 GiB | 3.01 MiB/s, done.
Resolving deltas: 100% (3627831/3627831), done.
Updating files: 100% (156243/156243), done.
# du -hs /usr/ports
du -hs /usr/ports
2.5G    /usr/ports
```

После чего, смотрим порт Yggdrasil:

```
$ ls -l /usr/ports/net/yggdrasil/
total 14
-rw-r--r--  1 root wheel 714 Nov 23 14:45 Makefile
-rw-r--r--  1 root wheel 385 Nov 23 14:45 distinfo
drwxr-xr-x  2 root wheel  3 Nov 22 15:14 files
-rw-r--r--  1 root wheel 353 Nov 22 15:14 pkg-descr
$ cat /usr/ports/net/yggdrasil/distinfo
TIMESTAMP = 1698509866
SHA256 (go/net_yggdrasil/yggdrasil-v0.5.1/v0.5.1.mod) =
2039923698dd8b3f3a5a73a46bb57cd5e4b322edc9737923a193ecfee90add9e
SIZE (go/net_yggdrasil/yggdrasil-v0.5.1/v0.5.1.mod) = 1762
SHA256 (go/net_yggdrasil/yggdrasil-v0.5.1/v0.5.1.zip) =
ab082d066fbd825d0ea97a6bfa9dc80084ed220060e11c44db7880fa5723088c
SIZE (go/net_yggdrasil/yggdrasil-v0.5.1/v0.5.1.zip) = 151253
```

В итоге, мы получили порт Yggdrasil версии 0.5.1, в стандартной терминологии и позиции, принятых в этой операционной системе. И устанавливаем этот порт, опять же, согласно привычным традициям этой системы:

```
$ cd /usr/ports/net/yggdrasil
```

³³ Только в самой последней коллекции портов, на сегодня, присутствует проект версии 0.5.X, в последней стабильной (квартальной) коллекции портов — версия 0.4.7, и она будет несовместима по протоколам.

```
# time make
====> License LGPL3 accepted by the user
====> yggdrasil-0.5.1 depends on file: /usr/local/sbin/pkg - found
====> yggdrasil-0.5.1 depends on file: /usr/local/bin/go120 - found
====> yggdrasil-0.5.1 depends on package: ca_root_nss>0 - not found
====> Building/installing portconfig as it is required for the config dialog
====> Cleaning for portconfig-0.6.1
====> License BSD2CLAUSE accepted by the user
...
====> Compressing man pages (compress-man)
====> Staging rc.d startup script(s)
      183.11 real      80.31 user      177.23 sys
```

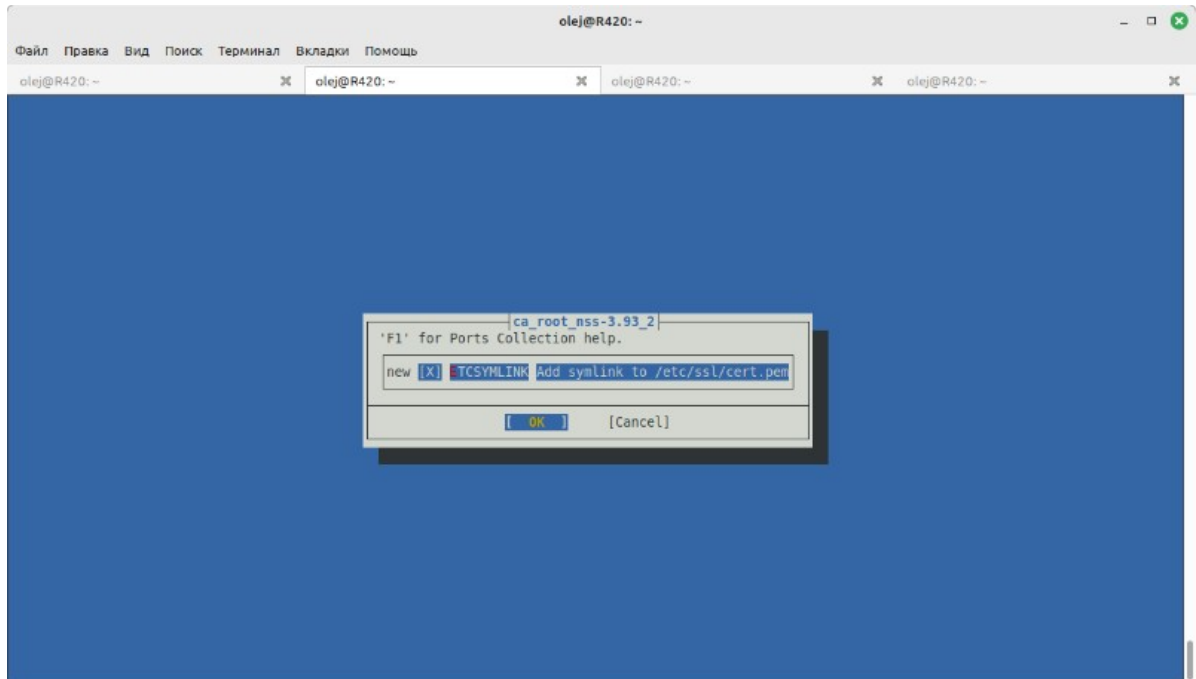


Рис. 5.6 Сборка из портов FreeBSD

По ходу этой сборки (всё-таки это больше 3-х минут, а у меня показана сборка на высокопроизводительном серверном процессоре) будут собираться все другие пакеты, требуемые **по зависимостям** (в этой согласованности — большое достоинство концепции коллекции портов FreeBSD). Во время сборки зависимых пакетов будет запрошены для выбора альтернативные конфигурации некоторых из этих пакетов. Запросы оформлены в системе ncurses (псевдографика в текстовом терминале), и выглядят в терминале подобно тому, как показано на рисунке.

Можно не вникая и не задумываясь согласиться с теми установками по умолчанию, которые при этом предлагаются.

После завершения компиляции и сборки, завершаем установку собранного и очистку побочных продуктов сборки (объектных файлов и подобных вещей):

```
# make install
====> Installing for yggdrasil-0.5.1
====> Checking if yggdrasil is already installed
====> Registering installation for yggdrasil-0.5.1
Installing yggdrasil-0.5.1...
# make clean
====> Cleaning for ca_root_nss-3.93_2
====> Cleaning for yggdrasil-0.5.1
$ which yggdrasil
```

```

/usr/local/bin/yggdrasil
$ which yggdrasilctl
/usr/local/bin/yggdrasilctl
$ yggdrasil --version
Build name: yggdrasil
Build version: 0.5.1

```

Если вы теперь попытаетесь стартовать сеть (либо вручную, как это делалось для Linux, или как сервис), то она либо просто не стартует вовсе, либо стартует (что удивительно), но не так как нам надо... Такое поведение всегда указывает на неадекватность конфигурационного файла, или просто на его отсутствие.

В FreeBSD (в отличие от Linux) этот сервис, после приведения в соответствие, будет стартовать из файла-скрипта `/usr/local/etc/rc.d/yggdrasil` (возьмём это на заметку — через 2 абзаца он нам понадобится):

```

$ ls -l /usr/local/etc/rc.d/yggdrasil
-rwxr-xr-x  1 root wheel 1876 Nov 23 15:38 /usr/local/etc/rc.d/yggdrasil

```

Заглянем в этот файл — в той части что там говорится на счёт конфигураций?:

```

$ grep "test ! -f" /usr/local/etc/rc.d/yggdrasil -A2
test ! -f /usr/local/etc/yggdrasil.conf && (
    logger -s -t yggdrasil "Generating new configuration file into
        /usr/local/etc/yggdrasil.conf"
    /usr/local/bin/yggdrasil -genconf > /usr/local/etc/yggdrasil.conf

```

Во-первых, о его местоположении, по умолчанию это: `/usr/local/etc/yggdrasil.conf`, что заметно отличается от состояния дел в Linux.

Во-вторых, что у нас (после инсталляции порта Yggdrasil) всё ещё нет такого файла вообще:

```

$ ls /usr/local/etc/yggdrasil.conf
ls: /usr/local/etc/yggdrasil.conf: No such file or directory

```

И в-третьих, что нам нужно — так это создать такой файл, и там же (выше) и подсказана команда которой это делается:

```

# yggdrasil -genconf > /usr/local/etc/yggdrasil.conf

```

(Собственно, эта команда описана и в документации Yggdrasil, но мы такой командой не пользовались ранее потому, что инсталляция в Linux из DEB-пакета сама создавала конфигурационный файл и в нужном месте. Команда потребует root полномочий, но только по причине прав доступа в `/usr/local/etc/`.)

После выполнения этой команды конфигурационный файл появился:

```

$ ls -l /usr/local/etc/yggdrasil.conf
-rw-r-----  1 root wheel 3553 Nov 23 22:02 /usr/local/etc/yggdrasil.conf

```

Но не станем спешить запускать IPv6 сеть с этим вновь созданным конфигурационным файлом — мы ведь помним, что, как минимум, туда нужно вписать (например, с помощью простейшего текстового редактора nano) адреса или имена внешних пиров сети. Например, так³⁴.

```

$ grep " Peers" /usr/local/etc/yggdrasil.conf -A8
Peers: [
    tcp://193.93.119.42:14244
    tcp://178.132.161.124:65533
    tcp://y.zbin.eu:7743

```

³⁴ Ещё раз напомним: конкретные адреса пиров в этом списке должны выбираться как функция вашей собственной географии, по принципу максимальной доступности, скорости.

```

tcp://88.210.3.30:65533
tcp://87.251.77.39:65533
tcp://sin.yuetau.net:6642
tcp://mima.localghost.org:1996
]

```

И если мы уже занялись конфигурационным файлом, то приостановимся и пристально рассмотрим те параметры (строки), которые **принципиально** отличаются от того, что мы видели в Linux, и без чего сеть нормально не станет работать в FreeBSD:

```

$ grep IfName /usr/local/etc/yggdrasil.conf
IfName: /dev/tun0
$ grep MTU /usr/local/etc/yggdrasil.conf
# Maximum Transmission Unit (MTU) size for your local TUN interface.
IfMTU: 32767

```

По поводу MTU мы можем найти на GIT самого проекта Yggdrasil (правда, в глубине обсуждений разработчиков):

The origin of the session MTU exchange is because the maximum MTU varies between different platforms, i.e. max 65535 works on macOS/Linux/Windows, 32767 on FreeBSD, 16384 on OpenBSD. It also occasionally makes sense to be able to reduce the MTU if you are routing your /64 routed subnet down a pipe with a lesser MTU, i.e. most links.

И, наконец, раз уж мы уже стали конфигурационный файл:

... команда создания конфиг-файла пропишет в него автоматически назначенное значение приватного ключа шифрования, что однозначно определяет и адрес IPv6 этого хоста. Но лучше вручную прописать туда собственное значение ключа (128 байт), найденное в процессе майнинга (как описывалось выше), для обеспечения более «высокого IP». Например³⁵ (всё это одна длинная строка в файле):

```

$ grep PrivateKey: /usr/local/etc/yggdrasil.conf | grep -v ^#
PrivateKey:
355fea539b2e20131581767ca20990741f8f10ef85af5ff81087132255b941d000000007be1ca278d73914f3c06
4afff91c9f18f4d4c16a07f619752fc30ffa9

```

Вот теперь всё приготовлено для запуска сети как сервиса FreeBSD (аналогично и ручной запуск, который проще и делается аналогично Linux):

```

$ sudo /usr/local/etc/rc.d/yggdrasil onestart
Password:
yggdrasil 1056 - - Starting yggdrasil
# yggdrasilctl getself
Build name:      yggdrasil
Build version:   0.5.1
IPv6 address:    21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f
IPv6 subnet:     31d:8a7c:aafa:f346::/64
Routing table size: 20
Public key:      00000005d60d541432e5fbabad558d6cca02aaddbb3363fcd3a8421934e4d575
# yggdrasilctl getpeers
URI          State Dir IP Address      Uptime      RX      TX      Pr      Last Error
tcp://193.93.119.42:14244  Up   Out  22d:d3dd:3afe:9599:3da9:d89f:6ae:8401  1h53m4s
94kb      14kb      0      -
tcp://88.210.3.30:65533    Up   Out  225:124e:2a74:9966:d18d:2f19:adbf:a950  1h53m4s
25kb      17kb      0      -
tcp://87.251.77.39:65533   Up   Out  224:79c3:da07:db06:60ef:282d:f908:d97    1h53m4s
37kb      17kb      0      -
tcp://y.zbin.eu:7743       Up   Out  200:dc42::3fca:8a99:5d66:b660           33m56s
12kb      6kb       0      -

```

³⁵ Во избежание дублирования вашего IPv6 с другими, во всей **всемирной** сети Yggdrasil, храните свои приватные ключи в тайне. Показанный здесь приватный ключ **не соответствует** показываемым далее публичному ключу и IPv6, и приведен только для иллюстрации как это выглядит.

```

tcp://178.132.161.124:65533    Up    Out    222:10d:b0b4:f237:5db9:e47c:a2e6:9ab9    1h53m4s
20kb    15kb    0      -
tcp://sin.yuetau.net:6642    Up    Out    200:b858:3f39:af76:83ae:bb1:ebb2:604e    1h53m4s
25kb    19kb    0      -
tcp://mima.localghost.org:1996 Up    Out    200:b1ff:9827:dc87:f067:c43b:d546:1dd7    1h52m58s
26kb    16kb    0      -

```

И проверяем ping на некоторый дальний хост Yggdrasil (возможно, совершенно другой процессорной архитектуры и любой операционной системы)

```

$ host linux-ru.ygg.at
linux-ru.ygg.at has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
$ ping -c3 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
PING6(56=40+8+8 bytes) 221:8f84:d3fb:143c:a91b:dd7:4532:c912 -->
221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
 16 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771, icmp_seq=0 hlim=64 time=907.963 ms
 16 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771, icmp_seq=1 hlim=64 time=158.966 ms
 16 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771, icmp_seq=2 hlim=64 time=164.971 ms

--- 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 ping6 statistics ---
 3 packets transmitted, 3 packets received, 0.0% packet loss
 round-trip min/avg/max/std-dev = 158.966/410.633/907.963/351.674 ms

```

Всё замечательно работает независимо от архитектур и операционных систем на тысяче-километровых трассах!

Ещё одна сеть: I2P

Ещё одна иллюстраций автономной, изолированной сети даркнет³⁶, которая является децентрализованной, самоорганизующейся и трафик в которой полностью шифрован. Коротко, чтобы не углубляться в детали, цитаты из информационных источников самой этой сети:

Проект I2P был начат в 2003 году для поддержки тех, кто заинтересован в новом нецензурируемом, анонимном средстве общения и распространения информации. ... I2P представляет собой защищенный протокол обмена данными, работающий поверх привычного протокола TCP/IP (на самом деле в первую очередь используется протокол UDP, а если его использовать не удастся, то TCP).

... TOR использует так называемый луковую маршрутизацию, когда каждое промежуточное звено снимает один слой шифрования, в результате последнее звено общается с конечным сервером открытым текстом... Протокол I2P работает по-другому, он использует чесночную маршрутизацию. В этом случае каждый пакет, который нужно передать (аналог — зубчик чеснока), шифруется, а затем упаковывается в большой пакет (чесночину), которая содержит еще несколько таких зубчиков-пакетов для передачи к разным узлам. Таким образом, когда пользователь получает чесночину, извлекает из нее зубчик-пакет, предназначенный для него самого, а остальные зубчики передает дальше. Так как все зубчики зашифрованы, то только тот, кому предназначен зубчик, знает, что с ним делать дальше.

Первоначально проект (i2p) был написан на языке Java, позже полностью переписан на языке C++ (проект i2pd) — мы далее будем рассматривать только последнюю из названных реализаций.

Запуск сети

Проект I2P, как достаточно «возрастной» (2003), реализован для большинства используемых операционных систем: Microsoft Windows, Linux, macOS, OpenBSD, FreeBSD и Android. Регулярно предоставляются готовые бинарные сборки для инсталляции (ссылки в конце

36 (*DarkNet*) — скрытая сеть, соединения которой устанавливаются только между доверенными пирами, иногда именующимися как «друзья», с использованием нестандартных протоколов и портов. Часто с термином даркнет ассоциируют отрицательные коннотации, связанные с противоправной деятельностью ... но нас здесь интересуют только вопросы технической организации.

текста). Но под Linux, к сожалению, предоставляются только `.rpm` пакеты. Если вы хотите поэкспериментировать с этой сетью, вы можете её легко собрать из исходных кодов:

```
$ git clone https://github.com/PurpleI2P/i2pd.git
Клонирование в «i2pd»...
remote: Enumerating objects: 36436, done.
remote: Counting objects: 100% (5220/5220), done.
remote: Compressing objects: 100% (254/254), done.
remote: Total 36436 (delta 5027), reused 5072 (delta 4960), pack-reused 31216
Получение объектов: 100% (36436/36436), 14.55 МБ | 5.69 МБ/с, готово.
Определение изменений: 100% (25982/25982), готово.
$ du -hs i2pd/
20M    i2pd/
```

Из зависимостей вам потребуется довольно очевидный «джентльменский набор» пакетов, который хорошо бы предварительно проверить и, при необходимости, доустановить:

```
$ aptitude search libboost-chrono-dev libboost-date-time-dev libboost-filesystem-dev \
  libboost-program-options-dev libboost-system-dev libboost-thread-dev libssl-dev
...
```

Конфигурируется сборка посредством Cmake:

```
$ cd i2pd/build/
$ cmake -DCMAKE_BUILD_TYPE=Release
CMake Warning:
  No source or binary directory provided. Both will be assumed to be the
  same as the current working directory, but note that this warning will
  become a fatal error in future CMake releases.

-- The CXX compiler identification is GNU 11.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info -- done
...
-- Configuring done
-- Generating done
-- Build files have been written to: /home/olej/2023/I2P/i2pd/build
```

И, собственно, сборка:

```
$ time make -j2
[ 2%] Building CXX object
CMakeFiles/libi2pd.dir/home/olej/2023/I2P/i2pd/libi2pd/Base.cpp.o
[ 2%] Building CXX object
CMakeFiles/libi2pdclient.dir/home/olej/2023/I2P/i2pd/libi2pd_client/AddressBook.cpp.o
[ 3%] Building CXX object
CMakeFiles/libi2pd.dir/home/olej/2023/I2P/i2pd/libi2pd/Blinding.cpp.o
...
real    5m11,959s
user    9m38,385s
sys     0m34,368s
```

Сборка достаточно напряжённая, даже на современных процессорах (что видно и по полному времени сборки):

```
$ uptime
19:23:22 up 9 days, 22:45, 8 users, load average: 84,47, 32,49, 12,04
```

И, в итоге:

```
$ ls -l i2pd
-rwxrwxr-x 1 olej olej 6033504 июл 19 19:35 i2pd
```


А инсталляцию собранного, чтобы не сильно разбираться в путях и каталогах, проще обеспечить так:

```
$ sudo make install
Consolidate compiler generated dependencies of target libi2pd
[ 54%] Built target libi2pd
Consolidate compiler generated dependencies of target libi2pdclient
[ 68%] Built target libi2pdclient
Consolidate compiler generated dependencies of target libi2pdlang
[ 90%] Built target libi2pdlang
Consolidate compiler generated dependencies of target i2pd
[100%] Built target i2pd
Install the project...
-- Install configuration: "Release"
-- Installing: /usr/local/lib/libi2pd.a
-- Installing: /usr/local/lib/libi2pdclient.a
-- Installing: /usr/local/lib/libi2pdlang.a
-- Installing: /usr/local/bin/i2pd
$ which i2pd
/usr/local/bin/i2pd
$ i2pd --version
i2pd version 2.48.0 (0.9.59)
Boost version 1.74.0
OpenSSL 3.0.2 15 Mar 2022
```

Необходимо ещё создать рабочий каталог с конфигурациями, и скопировать туда из каталога сборки (в принципе, вы можете создать рабочий каталог в любом месте, но если это дефальтное размещение \$HOME/.i2pd, то вам не придётся указывать его при запуске):

```
$ mkdir $HOME/.i2pd
$ cp -R contrib/certificates $HOME/.i2pd/
$ cp contrib/i2pd.conf $HOME/.i2pd/
$ cp contrib/tunnels.conf $HOME/.i2pd/
```

Сделанного нами достаточно для дефолтного запуска (в конфигурации по умолчанию) созданной сети...

```
$ i2pd
15:07:40@166/none - i2pd v2.48.0 (0.9.59) starting...
15:07:41@454/warn - Transports: 15 ephemeral keys generated at the time
15:07:43@402/warn - Profiling: No profile yet for
WVRdVfK3dShcomXxq0yDG0ym0m1WFR~0K7L9MXYUc9A=
...
```

И уже через некоторое непродолжительное время время (инсталляция при первом запуске подтягивает некоторые данные динамически из работающей сети, на это нужно время) вы можете приступить к использованию и управление сетью. Управление, по умолчанию, организовано через локальную WEB-консоль, с портом 7070 ... проверяем наличие такого порта:

```
$ telnet 127.0.0.1 7070
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
^]
telnet> quit
Connection closed.
```

И теперь это можно начинать использовать:

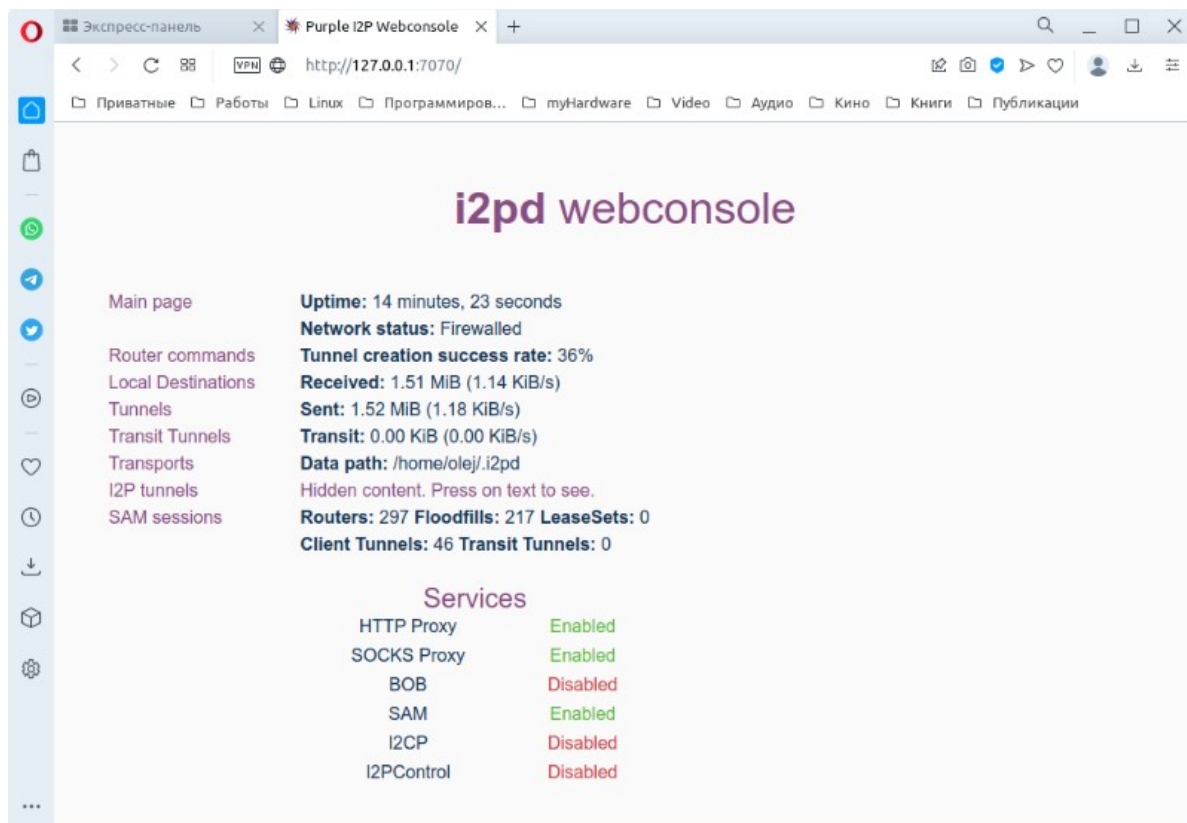


Рис. 5.7 WEB консоль конфигурирования I2P

Локальные характеристики (URL) созданного нами хоста смотрим здесь же в разделе «Local Destinations»:

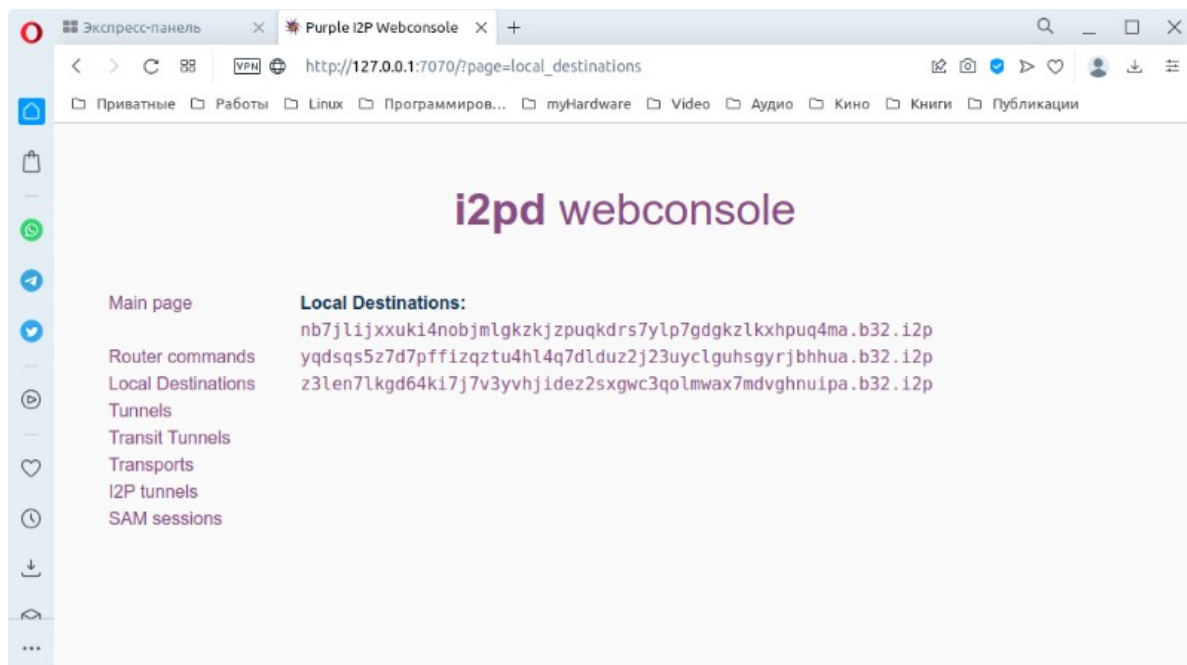


Рис. 5.8 URL хоста I2P

Использование сети

На рисунке выше виден общий вид «нативных» имён (URL) этой сети: к «именной» части (52 байт), формируемой из генерируемых публичных ключей, дописывается суффикс `.b32.i2p`.

В сети отсутствует как таковая служба разрешения имён, по типу DNS. Разрешение сетевых путей, и ближайших пиров, для построения динамических трасс, и целевых серверов (в том числе и коротких имён ресурсов вида, например, `zlibrary.i2p`) — разрешается своеобразно посредством из локального кэша `$HOME/.i2pd/addressbook`, который постоянно обновляется динамически из работающей сети:

```
$ ls -lR ~/.i2pd/addressbook/*/*.b32 | wc -l
799
$ ls -lR ~/.i2pd/addressbook/*/*.b32 | tail -n3
-rw-rw-r-- 1 olej olej 391 июл 17 16:58
/home/olej/.i2pd/addressbook/bz/zzls3ubai5wjfar4hskwqnh3vwwvrzoxsvcx64on2aogcxrvhoq.b32
-rw-rw-r-- 1 olej olej 391 июл 17 16:58
/home/olej/.i2pd/addressbook/bz/zzlsaymhcf1a7vibo3a223bybeecu3bd5z6rmw2u4y76maqu76q.b32
-rw-rw-r-- 1 olej olej 391 июл 17 16:58
/home/olej/.i2pd/addressbook/bz/zzlst7dauwprptpu2y7cxpetz4fl4jw73tivxhtnm7dla7m6teyq.b32
$ tail -n5 ~/.i2pd/addressbook/addresses.csv
zlibrary.i2p,zlib24th6ptyb4ibzn3tj2cndqafs6rhm4ed4gruxztaaco35lka
znc.str4d.i2p,ufkajv3stxpxlwgwwb2ae6oixdjircnbwog77qpxv7nt67rpcxq
znc.vern.i2p,vernd722rkd3v2o5ppu3aphmfhdthmillq5patn4fa7qv5ngiaq
zx.i2p,4ru5b25ixgz2wt2aqsksbr2iflv42gt6ta6qet7rv4et5rh2kq6fq
zzls.i2p,zzlst7dauwprptpu2y7cxpetz4fl4jw73tivxhtnm7dla7m6teyq
```

Вот как (очень по верхам) функционирует разрешение в этой сети:

Поскольку маршрутизаторы в основном располагаются на компьютерах участников сети, то их состав все время изменяется. Поэтому маршрутизаторы вынуждены постоянно поддерживать свой список другим маршрутизаторов в актуальном состоянии. Этот процесс называется «зондированием» (exploratory), заключающийся в посылке запросов со случайно выбранным 32-байтным адресом специальным маршрутизаторам, называемых floodfill. Предполагается что floodfill-маршрутизаторы обладают всей полнотой информации о сети. Помимо все прочего floodfill-маршрутизаторы постоянно сообщают друг другу информацию о найденных новых узлах.

Программное обеспечение сети при своей работе создаёт прокси-порты для доступа к (скрытым) ресурсам этой сети³⁷ (в том числе и по коротким именам):

По умолчанию, i2pd предоставляет для прикладных программ SOCKS и HTTP прокси. Прокси — это посредник. В нашем случае — посредник для выхода в скрытую сеть. По умолчанию SOCKS5 доступен по адресу 127.0.0.1:4447, HTTP-прокси — на порту 4444. Чтобы открыть веб-страницу, размещенную в сети I2P, в браузере необходимо установить соответствующие настройки.

```
$ chromium --proxy-server="socks://127.0.0.1:4447"
...
```

³⁷ Ресурсы I2P (сервера, или «концевые точки» в терминологии самой I2P) по своему предназначению в чём-то подобны `.onion` ресурсам TOR.

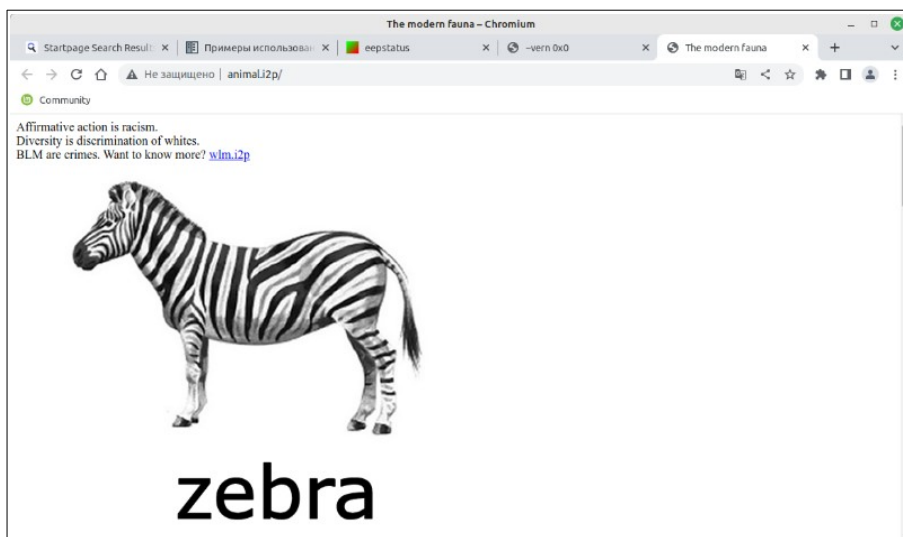


Рис. 5.9 Ресурс I2P (<http://animal.i2p/>) в стандартном браузере Chromium

Доступ к ресурсам извне

Если вы хотите организовать доступ сквозь сеть I2P к своим хостам, к типовым сервисам работающим на этих хостах, то в I2P это реализуется прописыванием туннелей в конфигурационных файлах (на клиенте и сервере доступа, соответственно):

```
$ ls -l ~/.i2pd/tunnels.conf
-rw-rw-r-- 1 olej olej 812 июл 20 09:55 /home/olej/.i2pd/tunnels.conf
```

Самым актуальным для сетевых служб (по крайней мере так это принято в UNIX/Linux) является протокол SSH — вот для соединений по нему мы и станем настраивать связь:

На сервере файл туннелей пропишем так (имя секции SSHD и файл ключа `ssh.dat` не имеют значения для работы — при первом корректном подключении файл `ssh.dat` будет создан автоматически, а в дальнейшем только использоваться):

```
$ grep -v ^# ~/.i2pd/tunnels.conf | grep -v ^$
[SSHD]
type = server
host = 127.0.0.1
port = 22
inport = 22
keys = ssh.dat
gzip = false
```

После чего сервер (после перезапуска, естественно) в его административной WEB консоли видится³⁸ так:

³⁸ Язык административной WEB консоли вы можете менять на русский непосредственно в меню консоли «Команды роутера», выпадающий список «Изменения языка».

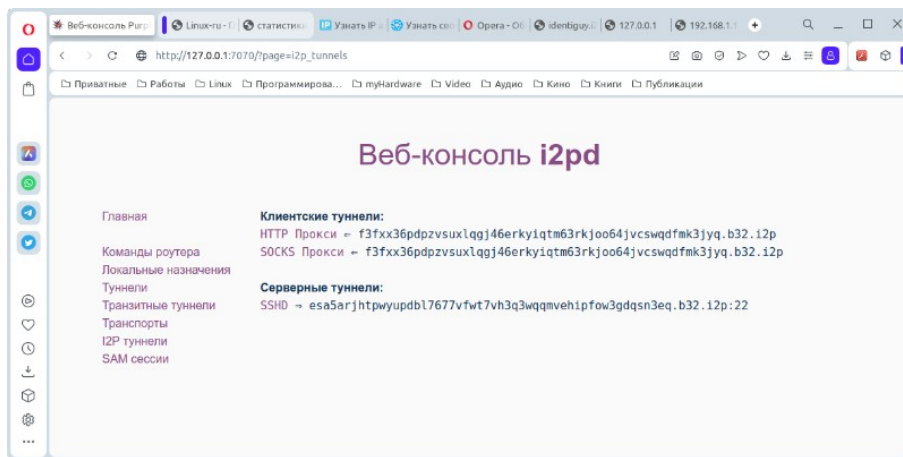


Рис. 5.10 Админпанель сервера

Здесь мы получили «нативный» URL *.b32.i2p сети I2P для подключений SSH.

На клиенте (встречной стороне SSH соединения) файл туннелей пропишем так (здесь также имена секции и файла ключа не имеют значения для работоспособности, а вот port — это и будет тот порт, из которого SSH-подключения будут переадресоваться в туннель I2P):

```
$ grep -v ^# ~/.i2pd/tunnels.conf | grep -v ^$
[ssh-R420]
type = client
address = 127.0.0.1
port = 2222
destinationport = 22
keys = ssh-R420.dat
destination = esa5arjhtpwyupdbl7677vfw7vh3q3wqqmvehipfow3gdqsn3eq.b32.i2p
```

В качестве значения destination указан именно тот I2P адрес, который даёт нам диагностика сервера для «Серверные туннели». Диагностика на клиенте (после перезапуска i2pd) теперь нам покажет (хотя она нам теперь и не особо нужна):

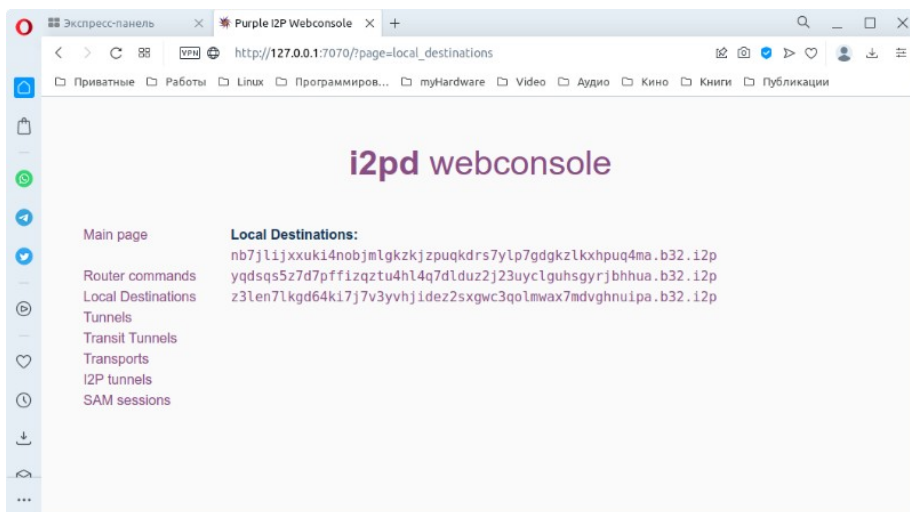


Рис. 5.11 Админпанель клиента

Теперь всё готово, и я могу подключиться по SSH через канал I2P:

```
$ ssh olej@127.0.0.1 -p 2222
The authenticity of host '[127.0.0.1]:2222 ([127.0.0.1]:2222)' can't be established.
ED25519 key fingerprint is SHA256:pfDAbUDI94rxh0McGbr4s0fmUepZALm9XweXUa/otxo.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[127.0.0.1]:2222' (ED25519) to the list of known hosts.
olej@127.0.0.1's password:
...
```

И теперь в созданной терминальной сессии я могу поинтересоваться: «А кто это к нам подключен?»:

```
$ who
olej      tty7          2023-07-20 08:50 (:0)
olej      pts/15        2023-07-20 10:35 (127.109.217.232)
```

Вот 2-я строка и является строкой I2P соединения, с петлевого адреса (туннеля?) ... но какого-то несколько хитрого...

На этом мы можем и закончить знакомство с сетью I2P ... Замечу только, в качестве примечания, что скорость сети I2P на дальних WAN соединениях (через Интернет, за сотни и тысячи километров) будет в разы ниже, чем рассмотренной ранее сети Yggdrasil.

Альтернативные DNS

Регистрация домена

Выше, говоря о инструментах сети, достаточно подробно рассмотрена иерархия зон DNS с технической точки зрения. Сейчас самое время взглянуть на эту же иерархию с организационной точки зрения, в каких формах она сложилась за 40 лет существования Интернет.

Домен — это (грубо говоря) имя в Интернете. Домены организованы в дерево. Корень дерева — зона «.», как можно сказать, это домен нулевого уровня. Его потомки — несколько десятков доменов первого (корневого) уровня: com, net, biz, org, ru, info, mobi, tv, ua, так называемые, TLD (Top-Level Domains). Для узла дерева, домена второго уровня linux-ru.com, например, все его непосредственные потомки (www.linear-ru.com, mail.linear-ru.com, foobar.linear-ru.com, ... о которых говорят как о субдоменах) образуют **зону** linux-ru.com.

У каждой зоны первого уровня (TLD) есть реестр, её координатор. Это организация, которая хранит, обслуживает и предоставляет доступ для регистраторов к центральной базе данных доменов этой зоны. Некоторые организации обслуживают несколько родственных зон, как, например, зоны SU, RU и РФ. Реестр продаёт регистраторам право доступа к зоне. Реестр — это сугубо техническая организация, предоставляющая доступ для регистраторов. В зоне RU реестром является РосНИИРОС (<http://www.ripn.su/>), а самым крупным регистратором — RUcenter (<https://www.nic.ru/>).

Реселлеры регистраторов — это реселлеры, как и с любым товаром: покупают подешевле, продают подороже. Уже реселлеры работают с розничными клиентами, людьми и организациями. Реселлеры продают доменные имена в аренду потребителям по своим собственным ценам, по существующим правилам потребитель может оплачивать аренду только на ограниченный срок (для большинства зон). По правилам зоны RU, например, этот срок составляет один год, через год пользователь обязан продлевать аренду доменного имени. Домен продают именно конечному потребителю, это, практически, его вещь (на срок аренды), в реестре хранится информация о владельце домена. В случае проблем с регистратором, можно перенести домен к другому, при этом, на всех этапах переноса, владельцем домена является только его покупатель. Вот, очень грубо, такой огромный базар...

Задача всей иерархии регистрации доменных имён со времён её создания было — построить **надёжную** дублирующуюся, резервированную систему баз данных хранящих соответствие

«имя — адрес» (другой побочной целью было и есть — создание целого поля бизнес интересов для тысяч его участников, обеспечивающих существование структур Интернет без их внешнего финансирования).

Блокчейн как регистратор домена

Но на время создания всей иерархической системы хранения и регистрации доменных имён (лет 40 назад) не была ещё изобретена техника блокчейн (blockchain, изначально block chain — цепочка из блоков) для доверенного хранения данных любой природы.

Впервые (октябрь 2008 года) термин появился как название полностью реплицированной распределённой базы данных криптовалюты Bitcoin, а дальше и всех десятков (или сотен?) разнообразных криптовалют. Однако технология цепочек блоков, как позже оказалось, может быть распространена на любые взаимосвязанные информационные блоки любой природы, для надёжного, не подлежащего изменению и фальсификации, распределённого хранения любой информации. И расценивается многими как «величайшее изобретение Интернет».

Рассмотрение техники блокчейна не входит в предмет нашего рассмотрения, но очень грубо (только до уровня достаточного для понимания дальнейшего изложения) это выглядит так:

- Информация хранится (защита) в некоторых блоках данных (block), которые упорядочены в связный список (цепочку, chain)...
- Связь между блоками обеспечивается не только их последовательной нумерацией, но и тем, что каждый блок содержит свою собственную хеш-сумму (по какому-то из алгоритмов) и хеш-сумму предыдущего блока.
- Изменение любой информации в блоке изменит его хеш-сумму. Чтобы соответствовать правилам построения цепочки, изменения хеш-суммы нужно будет записать в следующий блок, что вызовет изменения уже его собственной хеш-суммы. И так далее... При этом все предыдущие блоки не затрагиваются.
- Если изменяемый блок последний в цепочке, то внесение изменений может не потребовать существенных усилий. Но если после изменяемого блока уже сформировано продолжение, то изменение может оказаться крайне трудоёмким процессом.
- Копии цепочки блоков (экземпляры) хранятся независимо друг от друга на различных разнесённых по миру компьютерах, а любое изменение в одной из копий (транзакция) должно быть подтверждено и отражено (распространено) по всем копиям цепочки блоков.
- Что делает практически невозможным несанкционированное изменение информации уже занесённой в блоки данных, кроме последнего.

Таким образом, техника блокчейна (независимо от привязки к его конкретной реализации) позволяет хранить и обменивать ценности без необходимости доверия или центрального органа. Применительно к DNS, становится актуальной задача построения и распространения альтернативной DNS-сети, записи в которой:

- Управляются исключительно владельцем и никем другим.
- Могут быть свободно предоставлены в сети любому запросившему эту информацию.
- Не могут быть подменены злоумышленниками.
- Не могут быть заблокированы посредством административного ресурса.
- Не могут быть отозваны (разделегированы) никаким органом регистрации домена.

На настоящее время автору известны как минимум две независимые системы регистрации доменных имён построенные на децентрализованном блокчейне, это: криптовалюта EmerCoin и система Alfis. Ниже я просто изложу свой опыт использования альтернативных DNS, заимствованный из публикаций приведенных в списке используемых источников информации в конце этой части.

EmerCoin

Программная реализация EmerCoin наследуется от реализации Bitcoin, но не является, фактически, финансово-денежной системой, а является системой надёжного хранения для различных видов информации. Одним из таких видов и являются DNS-записи.

Логика использования укрупнённо выглядит так: вы, как владелец программного **кошелька** EmerCoin создаёте в нём адресные записи DNS типов A (адрес IPv4), AAAA (адрес IPv6) или MX (запись электронной почты), а любой другой пользователь EmerCoin в мире может сосчитать и использовать эту информацию в своих целях.

DNS-сеть от EmerCoin полностью децентрализована, и не имеет какого-либо управляющего сайта, доменными записями в системе управляет только владелец доменного имени. В настоящее время система поддерживает четыре корневых зоны доменных имён:

- *.coin — всё связанное с монетами и коммерцией;
- *.emc — всё связанное с проектом EmerCoin;
- *.lib — От слов Library и Liberty — то есть библиотеки и свобода;
- *.bazar — различные сервисы купли/продажи.

Любой заинтересованный в использовании DNS-сети от EmerCoin должен начать с того что скачать свободную реализацию кошелька EmerCoin в виде готовых пакетов (<https://github.com/emercoin/emercoin/releases/> или <https://sourceforge.net/projects/emercoin/files/>) — и это неплохой для практического использования вариант, сборки делаются тщательно, и для разных платформ (на сегодня это версия 0.8.3, пригодная для Windows, Linux и OpenBSD).

Другой вариант, как всегда, это самостоятельно собрать из исходников. Мы будем рассматривать именно этот последний вариант:

```
$ git clone https://github.com/emercoin/emercoin.git
Клонирование в «emergoin»...S/EmerCoin/2$
remote: Enumerating objects: 147918, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 147918 (delta 1), reused 9 (delta 1), pack-reused 147909
Получение объектов: 100% (147918/147918), 129.16 МиБ | 3.68 МиБ/с, готово.
Определение изменений: 100% (103956/103956), готово.
```

Или такой командой (репозитории в примерах клонировались в разное время, поэтому цифры могут незначительно отличаться, это не должно смущать):

```
$ git clone git@github.com:emergoin/emercoin.git
Клонирование в «emergoin»...
remote: Enumerating objects: 148663, done.
remote: Counting objects: 100% (721/721), done.
remote: Compressing objects: 100% (365/365), done.
remote: Total 148663 (delta 385), reused 660 (delta 356), pack-reused 147942
Получение объектов: 100% (148663/148663), 130.54 МиБ | 4.43 МиБ/с, готово.
Определение изменений: 100% (104366/104366), готово.
```

```
$ du -hs emergoin
160M emergoin
$ cd emergoin
```

В файле emergoin/doc/dependencies.md (в некоторых скачанных версиях его может не быть, тогда в сетевом репозитории: <https://github.com/emercoin/emercoin/blob/master/doc/build-unix.md>) находим перечень зависимостей, которые нужно удовлетворить прежде сборки. Кратко:

```
Berkeley DB | [4.8.30]
Boost | [1.70.0]
```



```

Clang | [3.3+]
Expat | [2.2.7]
fontconfig | [2.12.1]
FreeType | [2.7.1]
GCC | [4.8+]
HarfBuzz-NG
libevent | [2.1.8-stable]
libpng
librsvg
MiniUPnPc | [2.0.20180203]
OpenSSL | [1.0.1k]
PCRE
protobuf | [2.6.1]
Python (tests) | | [3.5]
qrencode | [3.4.4]
Qt | [5.9.7]
XCB | (Linux only) |
xkbcommon (Linux only) |
ZeroMQ | [4.3.1]
zlib | [1.2.11]

```

Все пакеты по зависимостям кроме первого (Berkeley DB), если что из них ещё не установлено, устанавливаются типовым образом из пакетной системы вашего дистрибутива и никаких проблем не составляют.

А вот с Berkeley DB — это неприятность, которая требует пояснений. В вашей пакетной системе (дистрибутива Linux), скорее всего, присутствует версия Berkeley DB v5.3.28 или что-то около того. А практически **все** проекты криптовалют, и это идёт ещё от Bitcoin, требуют версию Berkeley DB 4.8 — это уже такая традиция проектов блокчейна. Можно собрать и с другой версией, задав в `./configure` опцию `--with-incompatible-bdb`. Но такая сборка будет **несовместима** и **непереносима** с другими сборками, другими компьютерами и другими операционными системами по **форматам баз данных**³⁹. А Berkeley DB 4.8 — это 2010 год, 13 лет назад, и найти это для установки, и даже собрать из исходников, дело непростое. Удалось это только из старого репозитория пакетов Ubuntu (<https://old-releases.ubuntu.com/ubuntu/pool/main/d/db4.8/>) скачать требуемые пакеты. Поэтому привожу прямые ссылки на 5 скачанных пакетов необходимых для установки: <https://linux-ru.ru/download/file.php?id=5971> , <https://linux-ru.ru/download/file.php?id=5972> , <https://linux-ru.ru/download/file.php?id=5973> , <https://linux-ru.ru/download/file.php?id=5974> , <https://linux-ru.ru/download/file.php?id=5975> . Вот этот набор пакетов:

```

$ ls -l *.deb
-rw-rw-r-- 1 olej olej 728074 июн 1 2010 libdb4.8++_4.8.30-1_amd64.deb
-rw-rw-r-- 1 olej olej 687878 июн 1 2010 libdb4.8_4.8.30-1_amd64.deb
-rw-rw-r-- 1 olej olej 843972 июн 1 2010 libdb4.8-dev_4.8.30-1_amd64.deb
-rw-rw-r-- 1 olej olej 1723614 июн 1 2010 libdb4.8+-dev_4.8.30-1_amd64.deb
-rw-rw-r-- 1 olej olej 6916 фев 24 02:23 multiarch-support_2.29-0ubuntu2_amd64.deb

```

Дальнейшая последовательность действий следующая:

- Удаляем поштучно пакеты версии 5.3 (они будут препятствовать установке), полученных по списку, и примерно командами вот такого вида:

```

$ aptitude search bdb | grep ^i | grep Berkeley
i libdb5.3 - динамические библиотеки базы данных Berkeley v5.3
i A libdb5.3++ - Berkeley v5.3 Database Libraries for C++ [runtime]
i libdb5.3+-dev - Berkeley v5.3 Database Libraries for C++ [development]
i A libdb5.3-dev - Berkeley v5.3 Database Libraries [development]

```

³⁹ В принципе, если вы не планируете переносимости своих данных (кошелька), намереваетесь работать с проектом автономно, то версия Berkeley вас не должна беспокоить, и можно собирать с опцией `--with-incompatible-bdb`; такая сборка идёт вполне успешно.

```

$ sudo dpkg -r libdb5.3+-dev
(Чтение базы данных ... на данный момент установлено 535974 файла и каталога.)
Удаляется libdb5.3+-dev (5.3.28+dfsg1-0.8ubuntu3) ...
...
- Устанавливаем 5 пакетов (пакет multiarch-support_2.29-0ubuntu2_amd64.deb требуется для
установки пакетов версии 4.8):
$ sudo dpkg -i multiarch-support_2.29-0ubuntu2_amd64.deb
Выбор ранее не выбранного пакета multiarch-support.
(Чтение базы данных ... на данный момент установлено 535963 файла и каталога.)
Подготовка к распаковке multiarch-support_2.29-0ubuntu2_amd64.deb ...
Распаковывается multiarch-support (2.29-0ubuntu2) ...
Настраивается пакет multiarch-support (2.29-0ubuntu2) ...
$ sudo dpkg -i libdb4.8_4.8.30-1_amd64.deb libdb4.8-dev_4.8.30-1_amd64.deb
Выбор ранее не выбранного пакета libdb4.8.
(Чтение базы данных ... на данный момент установлено 535952 файла и каталога.)
Подготовка к распаковке libdb4.8_4.8.30-1_amd64.deb ...
Распаковывается libdb4.8 (4.8.30-1) ...
Выбор ранее не выбранного пакета libdb4.8-dev.
Подготовка к распаковке libdb4.8-dev_4.8.30-1_amd64.deb ...
Распаковывается libdb4.8-dev (4.8.30-1) ...
Настраивается пакет libdb4.8 (4.8.30-1) ...
Настраивается пакет libdb4.8-dev (4.8.30-1) ...
Обрабатываются триггеры для libc-bin (2.35-0ubuntu3.1) ...
$ sudo dpkg -i libdb4.8+-dev_4.8.30-1_amd64.deb
...
$ sudo dpkg -i libdb4.8+-dev_4.8.30-1_amd64.deb
...

```

- Проверяем:

```

$ apt search libdb4.8 | grep ^i
i   libdb4.8                      - Berkeley v4.8 Database Libraries [runtime]
i   libdb4.8++                    - Berkeley v4.8 Database Libraries for C++ [runtime]
i   libdb4.8+-dev                 - Berkeley v4.8 Database Libraries for C++ [development]
i   libdb4.8-dev                  - Berkeley v4.8 Database Libraries [development]

```

Вот теперь всё готово для стандартной сборки EmerCoin! Генерация проекта под свою архитектуру (это типовая Autogen технология UNIX):

```

$ time ./autogen.sh
libtoolize: putting auxiliary files in AC_CONFIG_AUX_DIR, 'build-aux'.
libtoolize: copying file 'build-aux/ltmain.sh'
libtoolize: putting macros in AC_CONFIG_MACRO_DIRS, 'build-aux/m4'.
libtoolize: copying file 'build-aux/m4/libtool.m4'
libtoolize: copying file 'build-aux/m4/ltoptions.m4'
libtoolize: copying file 'build-aux/m4/ltugar.m4'
libtoolize: copying file 'build-aux/m4/ltversion.m4'
libtoolize: copying file 'build-aux/m4/lt-obsolete.m4'
configure.ac:45: installing 'build-aux/compile'
configure.ac:45: installing 'build-aux/config.guess'
configure.ac:45: installing 'build-aux/config.sub'
configure.ac:28: installing 'build-aux/install-sh'
configure.ac:28: installing 'build-aux/missing'
Makefile.am: installing 'build-aux/depcomp'
...
parallel-tests: installing 'build-aux/test-driver'

real 0m25,456s
user 0m21,176s
sys 0m1,525s

```

Конфигурирование:

```
$ time ./configure --disable-dependency-tracking --disable-tests --disable-util-tx \
  --disable-gui-tests --enable-bip70 --disable-hardening --disable-debug
...
checking for Berkeley DB C++ headers... default
checking for main in -ldb_cxx-4.8... yes
...
Options used to compile and link:
  with wallet      = yes
  with gui / qt    = yes
    qt version     = 5
    with qr        = auto
  with zmq         = yes
  with test        = no
  with bench       = yes
  with upnp        = auto
  debug enabled    = no
  werror           = no

  target os       = linux
  build os        =

  CC               = gcc
  CFLAGS           = -g -O2
  CPPFLAGS         = -DHAVE_BUILD_INFO -D__STDC_FORMAT_MACROS
  CXX              = g++ -std=c++11
  CXXFLAGS         = -g -O2 -Wall -Wextra -Wformat -Wvla -Wformat-security
                  -Wno-unused-parameter
  LDFLAGS          =

real 0m29,751s
user 0m23,408s
sys  0m6,423s
```

(Особое внимание обращаем на специально показанную строку «Berkeley DB C++ headers», контролируем, что она должна выглядеть именно так!)

И, наконец, сборка проекта:

```
$ time make -j
Making all in src
make[1]: вход в каталог «/home/olej/2023/EmerCoin/2/emercoin/src»
make[2]: вход в каталог «/home/olej/2023/EmerCoin/2/emercoin/src»
  CXX      emercoind-bitcoind.o
...
  CXXLD     emercoind
  CXXLD     emercoin-tx
  CXXLD     bench/bench_bitcoin
  OBJCXXLD qt/emercoin-qt
make[2]: выход из каталога «/home/olej/2023/EmerCoin/2/emercoin/src»
make[1]: выход из каталога «/home/olej/2023/EmerCoin/2/emercoin/src»
Making all in doc/man
make[1]: вход в каталог «/home/olej/2023/EmerCoin/2/emercoin/doc/man»
make[1]: Цель «all» не требует выполнения команд.
make[1]: выход из каталога «/home/olej/2023/EmerCoin/2/emercoin/doc/man»
make[1]: вход в каталог «/home/olej/2023/EmerCoin/2/emercoin»
make[1]: Цель «all-am» не требует выполнения команд.
make[1]: выход из каталога «/home/olej/2023/EmerCoin/2/emercoin»
```

```
real 1m51,629s
user 45m9,021s
sys 3m39,244s
```

Эта сборка, в показанном виде, потребовала 2 минуты, но это делалось на сервере в 40 ядер и 96Gb оперативной памяти, в более типовой архитектуре это может занять несколько больше времени.

Примечание: В проекте, независимо от сборки из исходных кодов, или инсталляции из бинарного DEB-пакета, присутствуют 2 клиента (в терминологии криптовалюты называемых кошельками) — графический GUI клиент Qt и консольный CLI клиент (их позиции при сборке показаны от корня GIT репозитория):

```
$ ls -l src/emercoin-cli
-rwxrwxr-x 1 olej olej 9922928 сен  9 14:18 src/emercoin-cli
$ ls -l src/qt/emercoin-qt
-rwxrwxr-x 1 olej olej 280259832 сен  9 14:19 src/qt/emercoin-qt
```

Их функциональность различается. Основные операции можно выполнять в графическом GUI клиенте, но некоторые тонкие операции с блокчейном доступны только в консольном CLI, но мы не будем его затрагивать в дальнейшем рассмотрении.

Запускаем приложение ... Работа его начинается с начальной синхронизации блокчейна из сети Интернет, и это может занять минут 45, или даже свыше часа. После завершения синхронизации вы будете иметь работающий кошелёк EmerCoin:

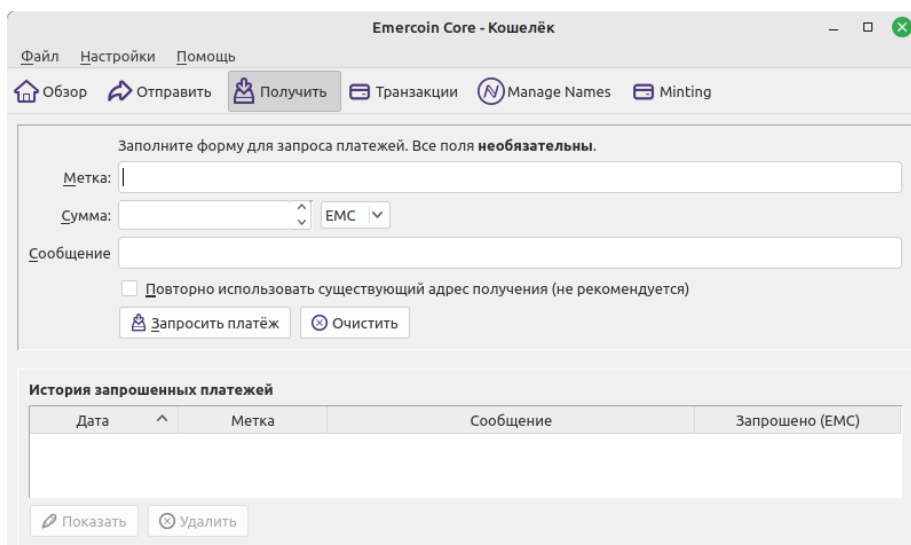


Рис. 5.12 Клиент (кошелёк) EmerCoin

Примечание: При каждом запуске приложения он будет начинать работу с синхронизации с текущим состоянием блокчейна. Но синхронизация будет происходить от времени вашего последнего старта, поэтому задержка запуска будет небольшой: он нескольких десятков секунд до нескольких минут — здесь всё зависит от того как давно вы производили предыдущий запуск!

Из публикаций:

После запуска программы кошелька, на него нужно откуда-то получить небольшое количество монет EmerCoin (EMC). Сверху есть кнопка «Получить». Нажав на неё, перейдёте в список с адресами. После первого старта, в кошельке будет присутствовать только один адрес наподобие:

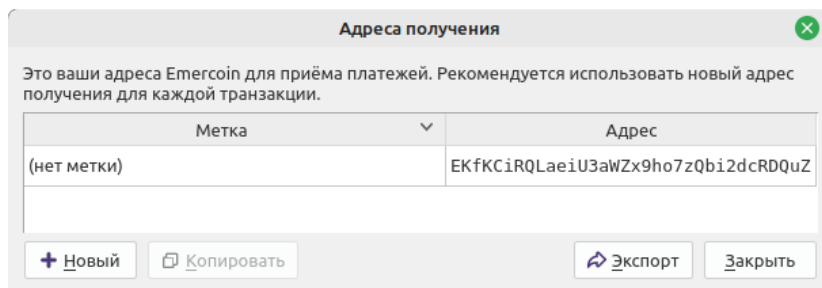


Рис. 5.13 Индивидуальный идентификатор (адрес) кошелька EMC

Далее, если в кошельке есть хотя бы один EMC — можно регистрировать домены. Справа вверху есть кнопка «Manage Names». Переходим в этот раздел...

Откуда взять минимальное число монет EmerCoin (0.5 — 1 уже более чем достаточно для любых манипуляций с DNS) для выполнения регистрационных действий? Опять же, из публикаций от авторов проекта EmerCoin:

Это, как видите, крайне недорого. Можно просто намайнить на пуле Pool.Emercoin.com или даже получить за помощь мировой науке. Также можно просто попросить у разработчиков по адресу team@emercoin.com — пару EMC на благое дело они пришлют. Цена регистрации одной записи DNS — примерно 0.2 EMC за год — то есть примерно 1/3 цента. Это существенно, на 3-4 порядка дешевле чем регистрация доменов у классических регистраторов.

Допустим, мы хотим зарегистрировать домен `linux-ru.lib` (в альтернативной доменной зоне) для этого заносим в поле Name значение `dns:linux-ru.lib` (без кавычек), а в поле Value заносим запись `A=90.156.230.27` (запись типа A — это адресная запись протокола IPv4):

```
$ host linux-ru.lib
linux-ru.lib has address 90.156.230.27
```

В поле days выбираем побольше, чем дефолтные 30 дней — скажем, 5 или 10 лет (3650 days). Далее жмём кнопку Submit — и всё! Ждём примерно 10 минут (изредка доходит до 30) — и ваша запись внесена в блокчейн системы! Запись вносится в блокчейн, и её копия присутствует в каждом кошельке EmerCoin в мире. Теперь любой пользователь-клиент может сделать запрос в свой локальный кошелек, и обратиться к соответствующей записи.

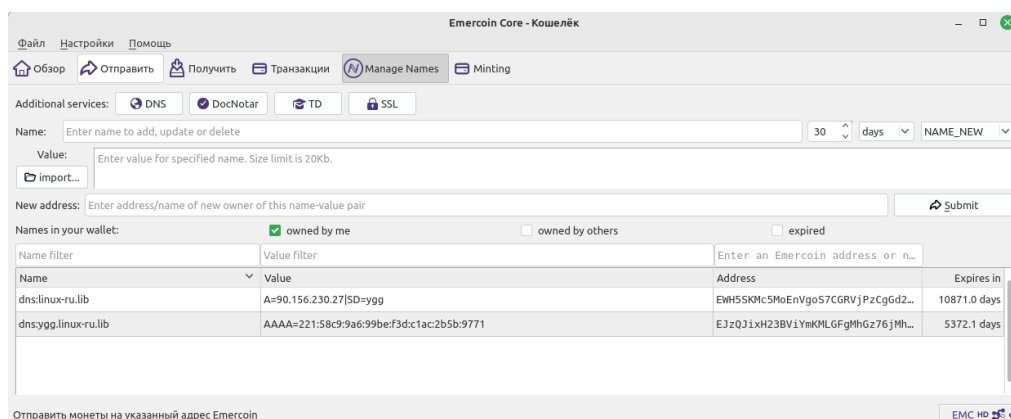


Рис. 5.14 DNS имена сохранённые в блокчейне

Для адресов IPv6 таким же образом в поле Value заносим записи типа AAAA. На рисунке выше показана в качестве примера запись для адреса того же ресурса в сети Yggdrasil (как описывалось выше):

```
$ host ygg.linear-ru.lib
```

ygg.linux-ru.lib has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771

Примечание: Здесь для хоста Yggdrasil зарегистрирована отдельная запись, потому, как уже отмечалось ранее, IPv6 сети Yggdrasil не являются «нативными» адресами IPv6. Но если несколько записей разных типов относятся к одному доменному имени (как это бывает и для Интернет обычных DNS), то они записываются в поле Value одной строкой с разделителем (символ | и без пробелов), например: A=185.31.209.8|AAAA=2a04:5340:1:1::3|MX=mx.ya.ru:10

Мы таким образом **создали** новые DNS записи. Но система (кошелек) EmerCoin имеет в своём составе и механизм для **разрешения** имён непосредственно запросами к кошельку, но по умолчанию этот механизм отключен. Активировать сервер DNS в стиле RFC1034 нужно в конфигурационном файле emercoin.conf, который обычно (может быть переопределено место при установке EmerCoin) в каталоге (часто это может быть ссылка на каталог, но это ничего не меняет):

```
$ grep ^emc $HOME/.emercoin/emercoin.conf
emcdns=1          # Запускать DNS сервер
emcdnsport=5335 # Порт для DNS, по умолчанию 5353
```

Первая необязательная строка здесь и разрешает работу DNS сервера. Вторая, необязательная, строка определяет прослушиваемый резолвером порт (если порт не указан явно, то по умолчанию используется 5335, более привычные для DNS порты 53 и 5353 обычно в Linux захвачены другими подсистемами DNS и могут порождать конфликты).

Перезапустим кошелек EmerCoin (здесь он выполняет роль сервера, в смысле рассматриваемых ранее сервисов):

```
$ ./emercoin-qt
...
```

И, после достаточно короткого старта, наблюдаем локальное разрешение DNS имён из альтернативных доменных зон EmerCoin:

```
$ sudo netstat -tunlp | grep emer
udp6      0      0 :::5335          :::*                248325/./emercoin-v
```

```
$ dig flibusta.lib @127.0.0.1 -p 5335
; <<>> DiG 9.18.12-0ubuntu0.22.04.1-Ubuntu <<>> flibusta.lib @127.0.0.1 -p 5335
...
;; ANSWER SECTION:
flibusta.lib.          3600    IN      A       179.43.150.83

;; Query time: 16 msec
;; SERVER: 127.0.0.1#5335(127.0.0.1) (UDP)
;; WHEN: Fri May 26 19:41:17 EEST 2023
;; MSG SIZE rcvd: 57
$ host -p 5335 linux-ru.lib 127.0.0.1
Using domain server:
Name: 127.0.0.1
Address: 127.0.0.1#5335
Aliases:
```

linux-ru.lib has address 90.156.230.27

Как понятно из рассказанного, точно такое же разрешение имён из альтернативных доменных зон EmerCoin может быть осуществлено через любой кошелек EmerCoin, запущенный в любой точке мира (запрос разрешается непосредственно из локальной копии распределённой и реплицируемой базы данных — блокчейна). Как следствие: показанные выше в примерах запросы для DNS имён могут быть направлены не только на петлевой локальный интерфейс 127.0.0.1, но и к любому другому работающему кошельку EmerCoin в Интернет.

Другим следствием будет то, что запрос разрешения к EmerDNS (так же как и обсуждаемому далее Alfis) за счёт локальности будет намного быстрее, чем запрос по сети к традиционным DNS серверам.

OpenNIC

Теперь любой пользователь-клиент может сделать запрос в свой локальный кошелёк для разрешения показанных выше имён. Но это только для пользователей имеющих запущенный кошелёк EmerCoin!

Для бесшовного объединения дерева альтернативных доменных имён от EmerCoin с обычными доменными именами от ICANN, со стороны клиента нужно иметь точку сопряжения, где происходит передача запроса в ту или иную систему. Для этого предложены разные варианты, одним из простейших в использовании является использование адресов DSN серверов сообщества OpenNIC (<https://www.opennic.org/>).

Emercoin поддерживает пиринговое соглашение с провайдером DNS OpenNIC, что означает, что домены, зарегистрированные в EmerDNS, доступны по умолчанию для всех пользователей DNS-серверов OpenNIC. OpenNIC - это простой и удобный способ беспрепятственного доступа ко всем веб-сайтам, зарегистрированным в блокчейне Emercoin, а также ко всем другим доменным зонам, которые поддерживает OpenNIC.

Эти IP адреса DNS вы можете прописать как сетевые DNS, например, в кэширующем сервере systemd, о котором речь шла ранее. Вот, например, как может быть прописана конфигурация кэширующего сервера для этого случая:

```
$ grep -v ^# /etc/systemd/resolved.conf | grep -v ^$
[Resolve]
DNS=94.16.114.254 94.247.43.254 194.36.144.87
FallbackDNS=192.168.1.3 1.1.1.1 8.8.4.4
LLMNR=yes
```

При этом:

```
$ resolvectl status
Global

    Protocols: +LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported
    resolv.conf mode: stub
    Current DNS Server: 94.16.114.254
    DNS Servers: 94.16.114.254 94.247.43.254 194.36.144.87
    Fallback DNS Servers: 192.168.1.3 1.1.1.1 8.8.4.4

Link 2 (eno1)
Current Scopes: LLMNR/IPv4 LLMNR/IPv6
    Protocols: +DefaultRoute +LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported

Link 3 (eno2)
Current Scopes: LLMNR/IPv4 LLMNR/IPv6
    Protocols: +DefaultRoute +LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported

Link 4 (tun0)
Current Scopes: LLMNR/IPv6
    Protocols: -DefaultRoute +LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported
```

И тогда у вас с адресами, разрешаемыми альтернативными DNS (EmerCoin), начинают работать привычные GNU утилиты без каких-либо других изменений в системе:

```
$ host linux-ru.lib
linux-ru.lib has address 90.156.230.27

$ host ygg.lib
ygg.lib has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771

$ nslookup linux-ru.lib
Server:          127.0.0.53
```

Address: 127.0.0.53#53

Non-authoritative answer:

Name: linux-ru.lib

Address: 90.156.230.27

Alfis

Ещё одной альтернативной системой (в чём-то аналогичной EmerDNS — мы же рассматриваем тенденции?) разрешения имён в Интернет является Alfis (Alternative Free Identity System)⁴⁰. Система также основывается на собственном блокчейне и поддерживает 10 собственных доменных зон: .anon, .btn, .conf, .index, .merch, .mirror, .mob, .screen, .srv, .ygg.

Коротко о самой системе со слов её автора:

Задачи ... сводились примерно к такому списку:

- Автономность и минимальный размер блокчейна
- Минимальная скорость роста блокчейна
- Общее удобство использования, кроссплатформенность

В процессе разработки пришлось пару раз изменить концепцию интерфейса ... в итоге выбрал библиотеку web-view, которая использует системный браузерный движок, и позволяет получить очень компактные программы.

Итак, что из себя представляет сам ALFIS?

- Блокчейн, хранимый в базе *sqlite*.
- Сетевая подсистема — *p2p*, написанная с нуля с использованием библиотеки *mio*, с шифрованием трафика и обменом пирами.
- Сервер DNS, который резолвит домены из блокчейна, перенаправляет незнакомые запросы на настроенные анстрим-серверы...

Как это обычно и бывает всегда в Linux, мы можем взять для установки готовые .deb пакеты для установки из репозитория <https://deb.revertron.com/pool/alfis/a/alfis/>. Но нас сейчас не интересует этот простой способ (который легко осуществить и без подсказок), а мы рассмотрим создание продукта из исходных кодов, раз в проекте заявлена открытость исходных кодов:

```
$ git clone https://github.com/Revertron/Alfis.git
```

```
Клонирование в «Alfis»...
```

```
remote: Enumerating objects: 4048, done.
```

```
remote: Counting objects: 100% (770/770), done.
```

```
remote: Compressing objects: 100% (110/110), done.
```

```
remote: Total 4048 (delta 690), reused 688 (delta 660), pack-reused 3278
```

```
Получение объектов: 100% (4048/4048), 2.29 МиБ | 2.22 МиБ/с, готово.
```

```
Определение изменений: 100% (2867/2867), готово.
```

Проект разработан на новом языке системного программирования Rust. Помимо прочего, он позволяет компилировать код в компактные запускаемые файлы под множество разных платформ (аппаратных и операционные системы). Мы уже использовали Rust ранее по тексту и отмечали, что если сама языковая система (менеджер cargo) Rust установлен полностью и правильно (а это делается несложным образом), то сборка приложений не составляет проблем. Характерным для проектов Rust является то, что система сама подгрузит все необходимые для проекта библиотечные компоненты (крейты в терминологии языка Rust).

⁴⁰ ALFIS, как видно из этой фразы — это аббревиатура, и должна бы писаться так, и в таком написании фигурирует в ряде описаний. Но сам автор в открытом GIT репозитории исходного кода называет проект Alfis. Поэтому мы будем использовать оба написания, не гадая какое из них должно быть правильным.

Но в данном проекте прежде, в некоторых дистрибутивах Linux (Debian, Ubuntu, Mint...), необходимо, по зависимостям, проверить и, при необходимости, установить пакет (которого часто нет в дефолтной установке):

```
$ sudo apt install libwebkit2gtk-4.0-dev
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Будут установлены следующие дополнительные пакеты:
  libjavascriptcoregtk-4.0-dev libpsl-dev libsoup2.4-dev libsqlite3-dev
Предлагаемые пакеты:
  libsoup2.4-doc sqlite3-doc
Следующие НОВЫЕ пакеты будут установлены:
  libjavascriptcoregtk-4.0-dev libpsl-dev libsoup2.4-dev libsqlite3-dev libwebkit2gtk-4.0-dev
...
```

И вот теперь можно переходить к сборке проекта Alfis:

```
$ cd Alfis/
$ cargo build --release
  Updating git repository `https://github.com/Boscop/web-view`
  Downloaded atk-sys v0.10.0
  Downloaded aead v0.3.2
  Downloaded block-buffer v0.9.0
  Downloaded convert_case v0.4.0
  Downloaded opaque-debug v0.3.0
  Downloaded cpufeatures v0.2.1
  ...
  Downloaded sqlite3-src v0.4.0
  Downloaded 140 crates (8.9 MB) in 50.92s (largest was `sqlite3-src` at 2.5 MB)
  Compiling proc-macro2 v1.0.51
  Compiling unicode-ident v1.0.6
  ...
  Compiling web-view v0.7.3 (https://github.com/Boscop/web-view#e87e08ca)
  Compiling toml v0.7.2
  Compiling alfis v0.8.4 (/home/olej/2023/Alfis/Alfis)
  Finished release [optimized] target(s) in 1m 22s
```

Всё достаточно бегло... В результате создаётся:

```
$ file target/release/alfis
target/release/alfis: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=7f87c74b8a991b9e9475af078f137916842eaa5d, for GNU/Linux 3.2.0, stripped
```

Это и есть наше итоговое приложение. Вот и всё относительно сборки. Менее понятно с его использованием. В описаниях можем найти: *Конфиг будет по адресу /etc/alfis.conf*. На самом деле это не совсем так ... точнее совсем не так: приложение вообще не создаёт нигде и никакого конфига, а его дефолтный конфиг мы находится в файле в каталоге сборки:

```
$ ls -l alfis.toml
-rw-rw-r-- 1 olej olej 1582 мая 13 19:48 alfis.toml
```

Но если уж авторы предлагают определённое место для конфига, то создадим его именно там (обратите внимание, что делаю это под `root`, но только из-за требований каталога `/etc`):

```
# pwd
/home/olej/2023/Alfis/Alfis
# ./target/release/alfis -g > /etc/alfis.conf
```

Дальше подправим конфигурационный файл под свои желания, например у меня это так (собственно, подправлены здесь поля `listen` и `forwarders`):

```
$ grep -v ^$ /etc/alfis.conf | grep -v ^#
origin = "0000001D2A77D63477172678502E51DE7F346061FF7EB188A2445ECA3FC0780E"
key_files = ["key1.toml", "key2.toml", "key3.toml", "key4.toml", "key5.toml"]
check_blocks = 8
[net]
peers = ["peer-v4.alfis.name:4244", "peer-v6.alfis.name:4244", "peer-ygg.alfis.name:4244"]
listen = "[::]:4244"
public = true
yggdrasil_only = false
[dns]
listen = "127.0.0.4:53"
threads = 10
forwarders = ["94.16.114.254:53", "94.247.43.254:53",
              "194.36.144.87:53", "1.1.1.1:53", "1.0.0.1:53"]
hosts = ["system", "adblock.txt"]
[mining]
threads = 0
lower = true
```

Принципиальным здесь, на начальном этапе, будет `listen = "127.0.0.4:53"` — порт на котором прослушиваются запросы DNS (здесь может быть, например, `"127.0.0.1:53"` или `"127.0.0.2:5353"`), и `forwarders = [...]` — это **массив** адресов DNS-серверов, которым транзитом пересылается запрос когда Alfis не может найти запрашиваемое имя в своём блокчейне.

Вот теперь мы можем запустить Alfis в качестве локального резолвера имён (мы явно указываем путь к конфигурационному файлу):

```
$ sudo ./target/release/alfis -c /etc/alfis.conf -n
18:47:44.842 [INFO ] (1) alfis::Main: Starting ALFIS 0.8.4
18:47:44.844 [INFO ] (1) alfis: Blocks count: 8720, domains count: 767, users count: 340
...
```

(Мы, пока, сознательно запускаем приложение с опцией `-n` — без GUI, только в качестве резолвера Интернет имён!)

```
$ sudo netstat -anp | grep 127.0.0 | grep 53
```

tcp	0	0	127.0.0.53:53	0.0.0.0:*	LISTEN	1104/systemd-resolv
tcp	0	0	127.0.0.4:53	0.0.0.0:*	LISTEN	22946/./target/rele
udp	0	0	127.0.0.4:53	0.0.0.0:*		22946/./target/rele
udp	0	0	127.0.0.53:53	0.0.0.0:*		1104/systemd-resolv
udp	0	0	127.0.0.1:5353	0.0.0.0:*		1313/python3

(Здесь PID=22946 — это новый резолвер Alfis, а PID=1104 — это кэширующий резолвер `systemd`, который мы рассматривали как-то ранее ... здесь важно то, что они работают **параллельно**, независимо, не препятствуя друг-другу, на разных адресах.)

И теперь, наконец, наблюдаем как происходит разрешение имён непосредственно из блокчейна:

```
$ nslookup myip.ygg 127.0.0.4
Server:          127.0.0.4
Address:         127.0.0.4#53

Name:myip.ygg
Address: 200:6223::d35b:1fd8:be0d:2841
$ host -p53 myip.ygg 127.0.0.4
Using domain server:
```

```
Name: 127.0.0.4
Address: 127.0.0.4#53
Aliases:
```

```
myip.ygg has IPv6 address 200:6223::d35b:1fd8:be0d:2841
```

И то как это происходит перенаправлением запросов к (традиционным) серверам IANA когда имя не находится в блокчейне:

```
$ host -p53 linux-ru.ru 127.0.0.4
```

```
Using domain server:
```

```
Name: 127.0.0.4
```

```
Address: 127.0.0.4#53
```

```
Aliases:
```

```
linux-ru.ru has address 90.156.230.27
```

```
linux-ru.ru mail is handled by 10 mx1.timeweb.ru.
```

```
linux-ru.ru mail is handled by 10 mx2.timeweb.ru.
```

Хорошо ... Последний вопрос: а как добавить свою доменную запись в блокчейн? И что для этого нужно? Систем названа в своей аббревиатуре как «Free», что означает, в том числе, и бесплатная. Но бесплатная она в денежном смысле. Сделать такую систему бесплатной в абсолютном смысле нельзя: её тут же всю «загадят» хакеры-мерзавцы или просто ничего не понимающие недоумки из Интернет. Но плата здесь имеет смысл в том, что ключ для записи DNS здесь нужно майнить, на манер как это делается в криптовалютах, хотя этот блокчейн и не имеет вообще никакого отношения к крипто-деньгам и какого-либо финансового наполнения. Плата здесь производится машинным временем!

Но такой майнинг здесь возможен только в варианте приложения с GUI интерфейсом, в консольном интерфейсе CLI, как мы его запускали раньше, это невозможно. Для этих целей запускаем приложение **без** опции **-n**.

Создание новой доменной записи в Alfis происходит в два этапа (логика авторов не совсем понятна, но она именно такая)...

1. Сначала мы должны майнить ключ (один или несколько последовательно) к блокчейну, вкладке Credentials. Ключ (или ключи), по мере их нахождения, нужно обязательно сохранить в файле (файлах) с произвольными именами. Если это не сделать, то работа по нахождению ключа будет безвозвратно потеряна. В файле сохраняются приватный и публичный ключи, и нужно это для подтверждения, впоследствии, ваших прав на владение записями в блокчейне. (Этим, пожалуй, и объясняется двухуровневая логика создания записей.)

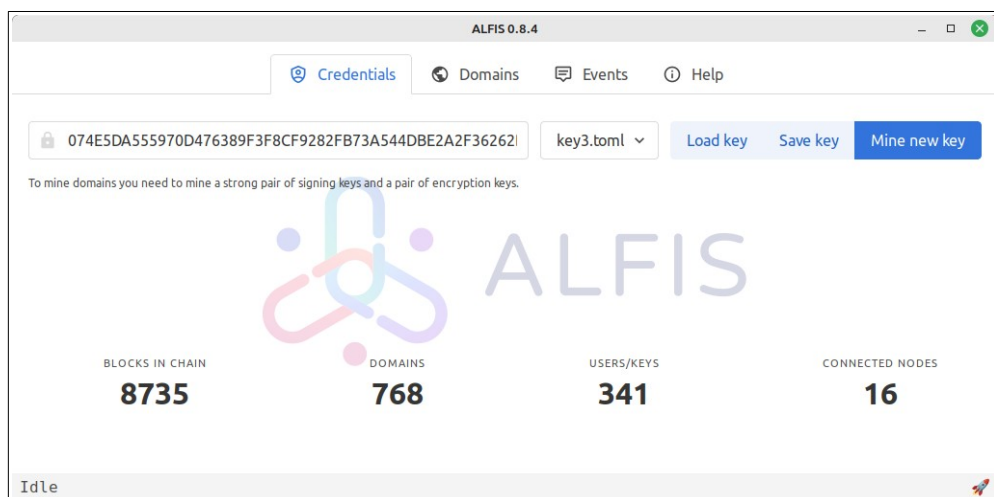


Рис. 5.15 Создание ключа

- И только затем, переходя во вкладку Domains, мы можем, выбрав соответствующий ключ из найденных (к которому и будут привязаны записи), создавать новые DNS записи. При этом мы определяем (в открывающихся окнах ввода) отдельные параметры доменной записи (имя корневого домена, имена субдоменов при необходимости, контакты владельца записи, адресные записи типов: A, AAAA, MX или другие нужные).

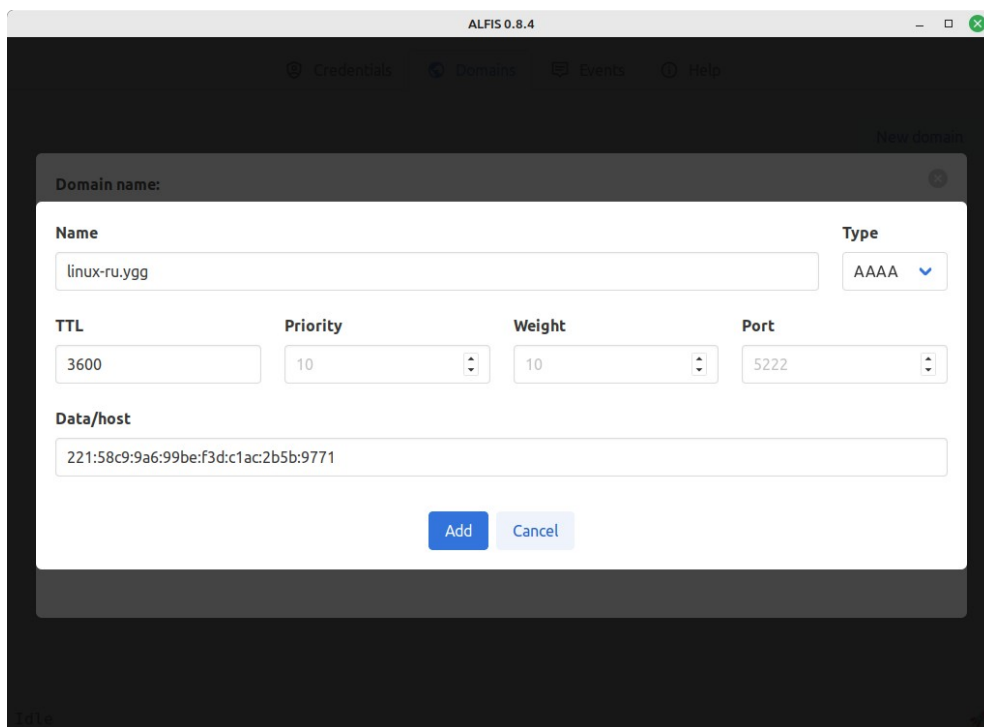


Рис. 5.16 Определение параметров нового доменного имени

Создание новой доменной записи (занесение записи в распределённый по сети блокчейн), заметьте, это всё такой же, весьма продолжительный процесс майнинга.

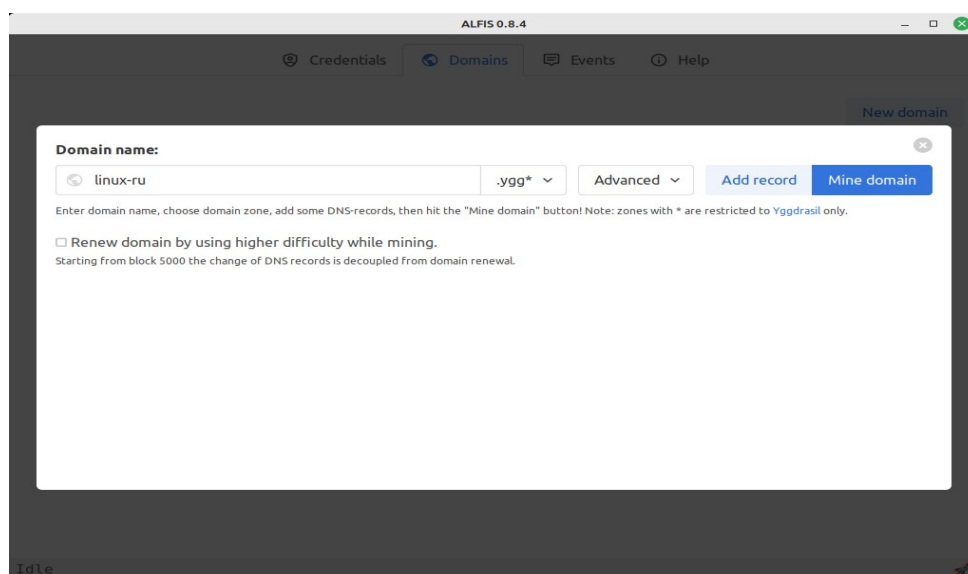


Рис. 5.17 Создание (майнинг) доменной записи

В это время в терминале, откуда было запущено само GUI приложение Alfis, мы можем наблюдать нечто подобное (по меткам времени видно, что даже на весьма производительном серверном оборудовании этот процесс занял больше часа времени):

```
...
18:37:56.385 [INFO ] (1) alfis::web_ui: Parsed domain data:
DomainData {
  encrypted: ,
  zone: "ygg",
  info: "",
  records: [
    AAAA {
      domain: "linux-ru.ygg",
      addr: 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771,
      ttl: TransientTtl(
        3600,
      ),
    },
  ],
  contacts: [],
}
19:38:37.337 [INFO ] (249) alfis::miner: Mined good block!
```

И в итоге мы получаем новую доменную запись, сразу же доступную всем пользователям, в мире, кто использует систему разрешения доменных имён Alfis:

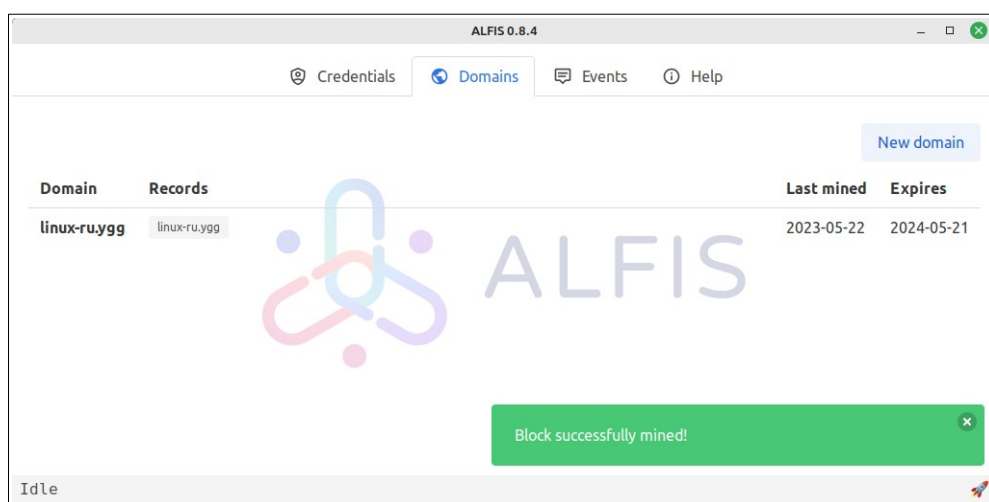


Рис. 5.18 Новая доменная запись со своим сроком действительности

И вот как происходит теперь разрешение нового DNS имени, только-что нами созданного, в альтернативной доменной зоне .ygg :

```
$ host linux-ru.ygg 127.0.0.4
Using domain server:
Name: 127.0.0.4
Address: 127.0.0.4#53
Aliases:
```

```
linux-ru.ygg has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
```

Здесь наблюдается прямая аналогия с рассмотренным выше разрешением имени через локальный кошелек EmerCoin — разрешение (сетевых) имён производится через локальную копию реплицируемой по сети базисной базы данных — блокчейна. Но репликация

содержимого и актуальность содержания блокчейна обеспечивается программными средствами самого блокчейна и находится ... «вне наших ощущений».

Альтернативное разрешение имён в домене .ygg

Корневой домен .ygg задекларирован как один из доменов в списке из 10 доменов, поддерживаемых системой разрешения доменных имён. Но разработчики придают ему особый смысл — это имена, которые предназначены для ресурсов сети Yggdrasil, и которые должны разрешаться в IPv6 адреса вида 200::/7.

Именно для этого домена (но не всех остальных десяти) разработчики предложили (оставили) альтернативный способ разрешения этих имён: нужно к .ygg URI дописать суффикс .at (это получается некоторый «бэкдор», потайная дверца, но оно очень необходимо при работах с Yggdrasil ... и поэтому способ этот нельзя не отметить).

Вот как это выглядит в системе где не установлена (или не запущена сейчас) система разрешения имён Alfis:

```
$ host myip.ygg
Host myip.ygg not found: 3(NXDOMAIN)
```

Понятно и естественно, что такой URI не разрешается стандартной DNS системой Интернет. Но:

```
$ host myip.ygg.at
myip.ygg.at has IPv6 address 200:6223::d35b:1fd8:be0d:2841
```

Или ещё один пример:

```
$ host linux-ru.ygg
Host linux-ru.ygg not found: 3(NXDOMAIN)

$ ping -c3 linux-ru.ygg.at
PING linux-ru.ygg.at(221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
(221:58c9:9a6:99be:f3d:c1ac:2b5b:9771)) 56 data bytes
 64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 (221:58c9:9a6:99be:f3d:c1ac:2b5b:9771):
icmp_seq=1 ttl=64 time=159 ms
 64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771 (221:58c9:9a6:99be:f3d:c1ac:2b5b:9771):
icmp_seq=2 ttl=64 time=160 ms
 64 bytes from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771: icmp_seq=3 ttl=64 time=167 ms

--- linux-ru.ygg.at ping statistics ---
 3 packets transmitted, 3 received, 0% packet loss, time 20337ms
 rtt min/avg/max/mdev = 159.354/162.117/166.626/3.215 ms
```

Польза (и немалая) такого альтернативного механизма разрешения имён .ygg, который, очевидно, и реализовали сами авторы проекта Alfis, состоит в том, что зарегистрировав некоторое имя xxx.ygg в блокчейне Alfis, вы можете разрешать его в IPv6 через имя xxx.ygg.at с любого хоста Интернет, стандартными DNS механизмами, без привлечения никакого вообще альтернативного механизма разрешения. Например, (пробуйте с любого хоста!) - для усложнения задачи делаю это из FreeBSD:

```
$ host linux-ru.ygg.at
linux-ru.ygg.at has IPv6 address 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771
```

Доступ к хостам локальной сети

Компьютеры в локальных сетях IPv4 (которых на сегодня абсолютное большинство) работают под частными (приватными) IP, не имеют прямого доступа в глобальные сети, и получают доступ к ресурсам Интернет посредством NAT-трансляции осуществляемой провайдером.

Общеизвестной истиной является то, что, симметричный доступ снаружи к таким локальным хостам LAN невозможен. Но таких локальных, закрытых для доступа извне, компьютеров числом на 2-3 порядка больше, чем «настоящих» компьютеров с «белыми» IP, маршрутизируемыми в WAN. А если считать и мобильные гаджеты под Android (который тот же Linux кем бы он не прикидывался), то и на все 6 порядков...

В некоторых ситуациях такое ограничение оказывается слишком жёстким и недопустимым (например при необходимости элементарного администрирования удалённого хоста по SSH). Для преодоления его давно предложено не сколько искусственных обходных путей, известных как: проброс портов, реверсивные туннели SSH и им подобные. Но большинство (все?) из таких способов имеют одно общее ограничение (помимо своих собственных индивидуальных): они требуют для своей реализации использование (наличие) некоторого промежуточного, транзитного, опорного хоста в Интернет с «белым» IPv4, доступным из любой точки сети.

Но ограничение на доступность хостов с «серым» IP **снимается** при использовании **всех** альтернативных способов адресации, показанных в этой последней части книги. На иллюстрации таких возможностей мы и сконцентрируемся в завершении этой последней части⁴¹.

Для осуществления удалённых доступов я буду использовать целевой компьютер в своей LAN:

```
$ hostname
R420
$ ip a s | grep "inet "
    inet 127.0.0.1/8 scope host lo
    inet 192.168.1.14/24 brd 192.168.1.255 scope global noprefixroute eno1
    inet 192.168.1.13/24 brd 192.168.1.255 scope global noprefixroute eno2
```

Как легко видеть, у этого компьютера есть **только** два IPv4 интерфейса частной сети 192.168.1.0/24, которые во внешней Интернет сети (куда этот хост не имеет прямого доступа), стараниями моего провайдера (его NAT) видятся под таким вот IP:

```
$ curl ifconfig.co
193.28.177.123
```

Это исходные условия экспериментов, которые воспроизводят и ситуацию у 99% читателей этого текста, и обеспечивающие адекватность таких экспериментов.

Первым вариантом достижимости этого хоста извне будет сеть Yggdrasil. На интересующем меня (локальном) хосте:

```
$ ip a s dev tun0
4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 53049 qdisc fq_codel state UNKNOWN
group default qlen 500
    link/none
    inet6 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f/7 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::255b:2ca5:b8ca:5c2a/64 scope link stable-privacy
        valid_lft forever preferred_lft forever
```

Организуем доступ из сервера с «белым» IPv4 и удалённого от тестируемого хоста на 4000 километров:

```
$ hostname
277938.local
$ ip a s | grep "inet "
    inet 127.0.0.1/8 scope host lo
    inet 90.156.230.27/24 brd 90.156.230.255 scope global dynamic eth0
```

⁴¹ Об этом можно было бы и не писать отдельной главой — эти возможности неявно вытекают из обсуждения выше, хоть это там и не акцентировалось. Но это настолько актуальная потребность (для удалённого администрирования), что это стоит обсудить отдельно!

SSH подключение к тестовому локальному хосту:

```
$ ssh -l olej 21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f
The authenticity of host '21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f
(21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f)' can't be established.
ECDSA key fingerprint is SHA256:j8BavG07g5rSIryljdB2DojGWk1cWJ23TQNIQI134fc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f' (ECDSA) to the list of
known hosts.
olej@21d:8a7c:aafa:f346:8115:14aa:9ca4:cd7f's password:

Last login: Mon Jul 31 17:36:46 2023 from 127.0.0.1

$ hostname
R420
```

Следующим, 2-м вариантом достижимости этого хоста извне будет использование .onion адресации проекта TOR. На тестируемом хосте:

```
$ sudo cat /var/lib/tor/onion.service/hostname
jfdbieogcloucayp6o4twh337fryoc6phg5yzpuv6phontov6ijm7id.onion
```

И подключение к нему с того же удалённого сервера:

```
$ torsocks ssh -l olej jfdbieogcloucayp6o4twh337fryoc6phg5yzpuv6phontov6ijm7id.onion
olej@jfdbieogcloucayp6o4twh337fryoc6phg5yzpuv6phontov6ijm7id.onion's password:

Last login: Thu Aug 3 12:09:13 2023 from 221:58c9:9a6:99be:f3d:c1ac:2b5b:9771

$ hostname
R420
```

И 3-й вариант — использование транспорта сети I2P. В этом варианте я устанавливаю SSH сессию между 2-мя хостами разных LAN, от разных провайдеров с их собственными NAT (.i2p тестируемого хоста был показан ранее на его WEB-консоли управления I2P). У клиента прописан туннель доступа I2P (как обсуждалось ранее):

```
$ sudo grep -v \# ~/.i2pd/tunnels.conf | grep -v ^$
[ssh-R420]
type = client
address = 127.0.0.1
port = 2222
destinationport = 22
keys = ssh-R420.dat
destination = esa5arjhtpyupdbl7677vfw7vh3q3wqqmvehipfow3gdqsn3eq.b32.i2p
```

И SSH подключение отсюда:

```
$ ssh olej@127.0.0.1 -p 2222
olej@127.0.0.1's password:

Last login: Thu Aug 3 12:16:59 2023 from 127.0.0.1

$ hostname
R420
```

Итого, мы имеем одновременно 3 одновременно открытые SSH сессии к локальному хосту. И вот как сам тестовый хост видит установленные к нему 3 подключения извне, использующих разные техники:

```
$ who
```



```
olej      tty7      2023-08-03 10:40 (:0)
olej      pts/7      2023-08-03 12:09 (221:58c9:9a6:99be:f3d:c1ac:2b5b:9771)
olej      pts/16     2023-08-03 12:16 (127.0.0.1)
olej      pts/29     2023-08-03 13:13 (127.109.217.232)
```

И в каждом из 3-х вариантов мы имеем 100% возможности и полномочия (посредством `sudo ...` или `sudo -i`, если потребуется) для административного управления всеми хостами удалённых LAN.

Но сказанное, естественно, относится не только к протоколу SSH, подобным образом вы можете организовать **любой** сервис (сервер, службу, протокол) на хосте локальной сети закрытой NAT, который будет доступен **из любой точки мира**. Для иллюстрации этого мы (на 5 минут работы) организуем локальный WEB-сервер, который будет общедоступным используя .onion URL (но точно так же может использоваться любой из рассматриваемых трёх способа доступа).

1. Установить WEB-сервер nginx (как простейший ... или любой другой, на свой вкус):

```
$ sudo apt install nginx
```

```
Чтение списков пакетов... Готово
```

```
Построение дерева зависимостей... Готово
```

```
Чтение информации о состоянии... Готово
```

```
Будут установлены следующие дополнительные пакеты:
```

```
libnginx-mod-http-geoip2 libnginx-mod-http-image-filter libnginx-mod-http-xslt-
filter libnginx-mod-mail libnginx-mod-stream libnginx-mod-stream-geoip2 nginx-common
nginx-core
```

```
...
```

Как и принято у многих серверов Linux, сервис сам стартует после инсталляции, в чём мы и убеждаемся (проверяем), подключившись любым браузером к 127.0.0.1:80 :



Рис. 5.19 Запуск nginx по умолчанию

2. Проверить или прописать заново в конфигурации TOR переадресацию портов:

```
$ grep -v ^# /etc/tor/torrc | grep -v ^$
HiddenServiceDir /var/lib/tor/onion.service/
HiddenServicePort 8080 127.0.0.1:80
HiddenServicePort 22 127.0.0.1:22
```

Я выбрал здесь TCP порт 8080 для переадресации к стандартному HTTP порту 80 локального WEB-сервера, но может быть выбран любой порт (в том числе, и даже,

возможно, желательно, порт 80 — порты .onion URL и стандартных URL Интернета не пересекаются).

3. Подключаемся любым браузером, понимающим .onion URL в адресной строке (Tor Browser, Brave Web Browser, ...) с любого хоста из любой точки мира:

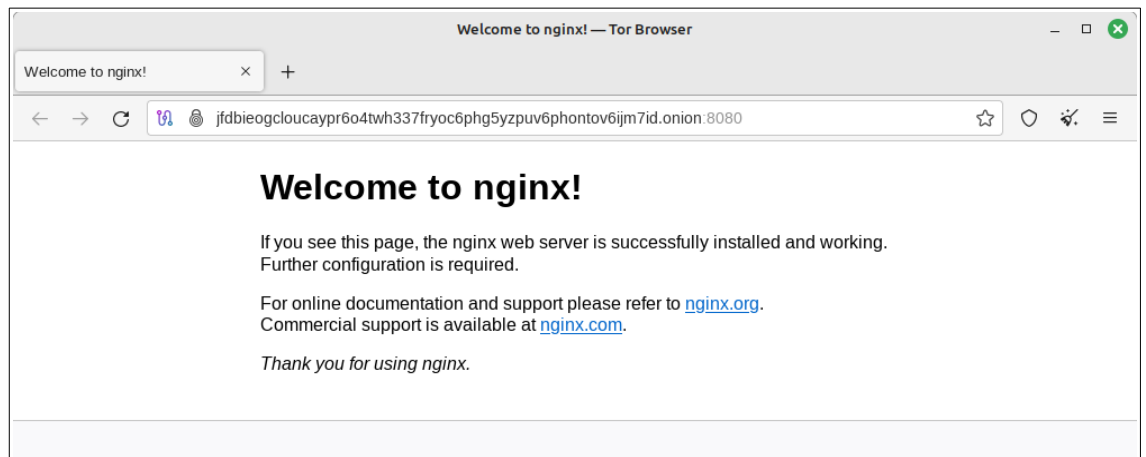


Рис. 5.20 Работа общедоступного .onion WEB ресурса

Вы получили здесь общедоступный WWW-сервер, работающий на локальном хосте LAN закрытом NAT провайдера, но доступный из любой точки мира.

Точно так же (и даже проще) реализует доступ и в других упоминаемых технологиях. Вот как выглядит всё тот же самый, свежее созданный, WEB ресурс при обращениях в терминологии Yggdrasil IPv6 адресов:



Рис. 5.21 WEB ресурс в сети Yggdrasil

Источники использованной информации

- [1] Что такое VPN, Proxy, Tor? Разбор — <https://habr.com/ru/companies/droider/articles/549212/>
- [2] Децентрализация как ответ беспределу: система доменных имен, которые невозможно делегировать — <https://roskomsvoboda.org/13508/>
- [3] О проекте Tor — <https://support.torproject.org/ru/about/>
- [4] Как работают Onion-ресурсы? — <https://community.torproject.org/ru/onion-services/overview>

- [5] Организация onion-сервера — https://howto.yggno.de/unix:onion_service_setup
- [6] Настроить свой Onion-ресурс — <https://community.torproject.org/ru/onion-services/setup/>
- [7] Tor ControlPort — https://wiki.archlinux.org/title/Tor_%28Русский%29#Пароль_Tor_Control
- [8] Управляем Тор из консоли с помощью Nux —
<https://imatvey.xyz/software/linux/upravljajem-tor-iz-konsoli-s-pomoshhju-nux/>
- [9] Tor specification (самый полный документ, 599 стр. в PDF формате, по состоянию на конец 2023г.) — <https://spec.torproject.org/index.html>
- [10] Wi-Fi Mesh сети для самых маленьких — <https://habr.com/ru/articles/196562/>
- [11] Yggdrasil Network: Заря бытовых меш-сетей, или Интернет будущего, 2021 —
<https://habr.com/ru/articles/547250/>
- [12] Добро пожаловать на wiki пользователей сети Yggdrasil — <https://howto.yggno.de/>
- [13] Yggdrasil Network — <https://yggdrasil-network.github.io/>
- [14] Релизы пакетов Yggdrasil для разных архитектур, для загрузок —
<https://github.com/yggdrasil-network/yggdrasil-go/releases>
- [15] Armbian. Linux for ARM development boards — <https://www.armbian.com/>
- [16] GIT репозиторий Yggdrasil — для сборки из исходных кодов —
<https://github.com/yggdrasil-network/yggdrasil-go.git>
- [17] Олег Цилюрик, «Linux и Go. Эффективное низкоуровневое программирование», BHV-СПб, декабрь 2022, 302 стр., ISBN: 978-5-9775-1741-6 — об установке GoLang —
<https://bhv.ru/product/linux-i-go-effektivnoe-nizkourovnevoe-programmirovanie/>
- [18] public peers (v0.4) — обновляемый список пиров Yggdrasil версии 0.4.X —
<https://publicpeers.neilalexander.dev/v0.4>
- [19] public peers (v0.5) — обновляемый список пиров Yggdrasil версии 0.5.X —
<https://publicpeers.neilalexander.dev/v0.5>
- [20] Multicast Peer Discovery — <https://howto.yggno.de/yggdrasil:mpd>
- [21] Как подключиться к Yggdrasil, не устанавливая его клиент на устройство —
https://howto.yggno.de/yggdrasil:network_connection_variants
- [22] Новый виток развития Yggdrasil, 2022 — <https://habr.com/ru/articles/701200/>
- [23] I2P — <https://ru.wikipedia.org/wiki/I2P>
- [24] Релизы пакетов I2P — <https://github.com/PurpleI2P/i2pd/releases>
- [25] GIT репозиторий I2P для сборки из исходных кодов — <https://github.com/PurpleI2P/i2pd>
- [26] i2pd documentation — <https://i2pd.readthedocs.io/en/latest/>
- [27] Децентрализация как ответ беспределу: система доменных имен, которые невозможно делегировать — <https://roskomsvoboda.org/13508/>

- [28] Meshname – DNS судного дня — <https://habr.com/ru/articles/550688/>
- [29] Блокчейн против блокировок — <https://dzen.ru/media/olegarch/blokchein-protiv-blokirovok-61367c945a15184a849007e8>
- [30] EmerCoin: децентрализованный альтернативный DNS на основе криптовалюты — <https://roskomsvoboda.org/12118/>
- [31] Emercoin, 2023г. (рус.), документация — <https://rtfm.emercoin.com/>
- [32] Релизы программ кошелька EmerCoin для скачивания — <https://github.com/emercoin/emercoin/releases/> или <https://sourceforge.net/projects/emercoin/files/>
- [33] EmerDNS — https://emercoin.com/ru/EMCDNS_and_NVS
- [34] Сообщество OpenNIC альтернативных DNS — <https://www.opennicproject.org/>
- [35] Wireguard VPN, Yggdrasil, ALFIS DNS и AdGuard, 2021 — <https://habr.com/ru/articles/595485/>
- [36] ALFIS DNS: перемещение точки доверия, 2021 — <https://habr.com/ru/articles/597207/>
- [37] GIT репозиторий Alfis — <https://github.com/Revertron/Alfis>
- [38] Мотивация (от автора Alfis) — https://github.com/Revertron/Alfis/blob/master/docs/about_ru.md

Итоги и перспективы

*«Будда был прав, где огонь, там прах и зола
На чужих пепелищах не грейся
Нет грехов, нет добра и нет зла
Есть прямые последствия действий»
Музыкальная группа «Рекорд Оркестр»*

Мы завершили рассмотрение сетевых средств Linux в самых разных их аспектах.

В 1-й части мы вспомнили основные принципы организации сети TCP/IP.

В части 2-й мы прошли по большинству распространённых сетевых сервисов (и их протоколам), функционирующих в сети Linux/UNIX (практически всё то же самое будет иметь место в любой операционной системе принадлежащей к классу UNIX: FreeBSD и OpenBSD, Solaris, Minix 3 ... и даже, в значительной степени, в MacOS).

Две следующих части, 3-я и 4-я, представляют краткий взгляд, с позиций реализующего программного кода, на то как сетевой трафик проходит сквозь сетевой стек операционной системы: от того момента когда пользователь в браузере набирает текстовую строку, и до того момента когда эта строка, за тысячи километров от пользователя попадает «в лапы» сервера на обработку. И только так, через программный код, можно увидеть цельную картину как IP пакеты из физического кабеля Ethernet превращаются в сокетные буфера в ядре, и далее отображаются в сокеты на уровне приложений пользовательского пространства (без рассмотрения программных кодов всему, рассказываемому относительно сети в Linux ... можно только «верить»).

В последней, 5-й части, мы, только мельком, «по касательной», по верхам, прикоснулись к проблематике и тенденциям, которые прорабатываются в современных новых проектах сети.

Из всего этого хотелось бы донести ощущение, что мир сетевых приложений и возможностей невообразимо широк, куда многообразнее чем можно уместить в одну, даже самую большую книгу.

И то, что этот мир вовсе не завершён и заостенел, а нас в области сетевых возможностей и их применений ещё ожидают большие неожиданности и сюрпризы... Например, это будет грядущий тотальный переход от привычной IPv4 к IPv6, когда рухнут понятия внутренних хостов локальных сетей и мнимое ощущение их защищённости за счёт локальности (защищённости NAT на шлюзовых маршрутизаторах). Или когда над привычным и понятным администрируемым Интернет может возникнуть и широко использоваться, публично доступная, «надструктура», не подчиняющаяся многолетнему организационному регламенту.

Сеть многогранна и многолика ... Давайте с интересом наблюдать за изменениями которые с ней происходят.

Об авторе

Ещё когда-то, в первой половине 80-х годов, когда компьютерными сетями интересовались только эстеты-энтузиасты и писатели фантасты, проректор по науке Харьковского Института Радио-Электроники предложил автору, занимающемуся зачатками этих технологий, провести для его аспирантов вводный курс по технике компьютерных сетей. С того и пошло...

В самые последние годы СССР автор был назначен руководителем вновь организуемого отдела КБ Полупроводникового Машиностроения (г.Москва) ... можно бы было и не упоминать о столь малозначимом событии, если бы не тот факт, что отдел этот в официальном документообороте значился как «искусственного интеллекта», впервые в канцелярской практике СССР. И, хотя до мирового бума искусственного интеллекта оставалось ещё лет 30, этим отделом были созданы 2 системы медицинской диагностики, в области кардиологии и энцефалографии, которые ещё больше 10 лет поставлялись в крупные клиники различных городов уже не существующей к тому времени страны...

Вот с тех времён и по сегодняшний день автор — постоянный участник разработок системотехнических проектов, крупных и не очень, но все их объединяет общее свойство, состоящее в том, что все они включают, в разной степени, сетевые компоненты. Так что без особо значительной доли шутки можно сказать, что автор книги «самый старый сетевик СССР», или наоборот «самый сетевой из корифеев отечественной IT индустрии».

За эти, более чем 30 лет, участия в разработках, отдельные артефакты сетевой инфраструктуры вызывали порой восхищение, порой удивление автора, и откладывались как памятки в «копилку» наблюдений — наброски первых глав книги начали складываться (и публиковались в качестве журнальных статей) лет 10-15 назад. Вот таким образом и сложилась настоящая книга: от вопросов самых классических ТСП/IP, до только ожидающих широкого внедрения IPv6, и далее до экспериментальных и поисковых проектов в развитие сетей завтрашнего дня, которые интересны пока как тенденции...