Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление подготовки: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Вычислительные машины, комплексы, системы и сети”

Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

К защите допустить:
Заведующий кафедрой
д. т. н., профессор _____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема: “Реализация протокола TCP пространства пользователя на основе сетевого стека ОС FreeBSD”

Студент _____ С. А. Репин

Руководитель
к. т. н, доцент _____ А. В. Тимофеев

Консультант от предприятия
начальник отдела разработки
АО «ИнфоТеКС» _____ Н. С. Чудов

Консультант по обеспечению качества
разработки к. э. н _____ О. С. Артамонова

Консультант от кафедры
к. т. н., доцент, с. н. с. _____ И. С. Зуев

Санкт-Петербург
2022 г.

Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление: 09.04.01 “Информатика и вычислительная техника”
Профиль: “ Вычислительные машины, комплексы, системы и сети”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2022 г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент С. А. Репин Группа № 8307

1. Тема Реализация протокола TCP пространства пользователя на основе сетевого стека ОС FreeBSD
(утверждена приказом № _____ от _____)
Место выполнения ВКР: АО «ИнфоТеКС»

2. Объект и предмет исследования: transmission control protocol (TCP), исходный код сетевого стека операционной системы FreeBSD.

3. Цель: адаптация кода операционной системы FreeBSD для получения реализации протокола TCP, работающей в пространстве пользователя и способную интегрироваться внутрь существующего продукта компании АО «ИнфоТеКС».

4. Исходные данные: исходный код операционной системы FreeBSD, стандарт TCP, архитектурная документация продукта, для которого разрабатывается реализация.

5. Содержание: описание протокола TCP, обзор архитектуры сетевого стека FreeBSD, разработка архитектуры реализации протокола с учетом особенностей продукта, разработка программного кода.

6. Технические требования:
– реализация должна быть написана на C++17 и C11 и использовать CMake в качестве системы сборки;
– должны быть внесены минимальные изменения в оригинальный код FreeBSD;

– разработка должна быть представлена в виде статической библиотеки с четким интерфейсом на C и C++, приближенным к интерфейсу сокетов Беркли;
– разработка должна производиться с учетом обеспечения максимальной производительности;
– реализованный протокол должен соответствовать стандарту RFC 793, остальные расширения TCP опциональны;
– должна быть возможность использования произвольных портов, IP-адресов и MAC-адресов;
– работа функций TCP должна поддерживать асинхронность;
– код должен быть покрыт модульными тестами.

7. **Дополнительные разделы:** обеспечение качества разработки, продукции, программного продукта.
8. **Результаты**: пояснительная записка, реферат, презентация, исходный код.

Дата выдачи задания
«_____» _____ 2022 г.

Дата представления ВКР к защите
«_____» _____ 2022 г.

Руководитель
к. т. н, доцент _____ А. В. Тимофеев

Студент _____ С. А. Репин

Санкт-Петербургский государственный электротехнический университет “ЛЭТИ” им. В. И. Ульянова (Ленина) (СПбГЭТУ “ЛЭТИ”)

Направление 09.04.01 “Информатика и вычислительная техника”
Профиль: “ Вычислительные машины, комплексы, системы и сети”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2022 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема Реализация протокола TCP пространства пользователя на основе сетевого стека ОС FreeBSD

Студент С. А. Репин Группа № 8307

№ этапа	Наименование работ	Срок выполнения
1	Изучение принципов работы TCP	24.03–31.03
2	Изучение архитектуры сетевого стека FreeBSD	31.03–06.04
3	Планирование работы, определение участков оригинального кода, в которые требуется вносить изменения	07.04–09.04
4	Разработка архитектуры программного решения	10.04–16.04
5	Разработка реализации протокола	17.04–30.4
6	Разработка интерфейса библиотеки	01.05–04.05
7	Написание модульных тестов	05.05–08.05
8	Тестирование	09.05
9	Оформление пояснительной записки	10.05–26.05
10	Предварительное рассмотрение работы	27.05–08.06
11	Представление работы к защите	09.06

Руководитель
к. т. н., доцент _____ А. В. Тимофеев

Студент _____ С. А. Репин

РЕФЕРАТ

Пояснительная записка содержит: 80 страниц, 12 рисунков, 9 таблиц, 1 приложение, 35 источников литературы.

Цель работы представляет собой реализацию Transmission Control Protocol (протокола TCP), которую можно использовать в приложениях пространства пользователя.

В основе разработки лежит исходный код операционной системы FreeBSD, в который вносятся различные изменения, позволяющие оформить код TCP в виде библиотеки и применять ее независимо от остальной части операционной системы. В работе рассматривается основание для такого применения, а также производится анализ существующих решений.

Представлен обзор протокола TCP, архитектуры FreeBSD, процесс разработки и тестирования реализации.

Результатом стала готовая к использованию библиотека с интерфейсом на языках C и C++, представляющая реализацию Transmission Control Protocol. Она многоядерная, работает в пространстве пользователя, обеспечивает высокую производительность (до 10 тысяч соединений в секунду на ядро процессора) и предоставляет прямой доступ к машине состояний TCP.

Эта библиотека уже используется внутри продукта компании АО «ИнфоТеКС».

ABSTRACT

The explanatory note contains: 80 pages, 12 figures, 9 tables, 1 appendix, 35 references.

The purpose of the work is an implementation of Transmission Control Protocol (TCP), which can be used in user space applications.

It is based on FreeBSD source code, with various modifications to make the TCP code a library and use it independently of the rest of the operating system. This paper discusses the basis for such an application and also analyzes the existing solutions.

It presents an overview of the TCP protocol, the FreeBSD architecture, and the process of developing and testing the implementation.

The result is a ready-to-use library with a C and C++ interface representing an implementation of the Transmission Control Protocol. It is multicore, works in userland, provides high throughput (up to 10,000 connections per second per processor core) and direct access to the TCP state machine.

This library is already used inside the product of InfoTeCS, JSC.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

API (Application Programming Interface) **–** программный интерфейс приложения (библиотеки).

DPDK (Data Plane Development Kit) – набор разработки передающего уровня.

HTTP(S) (HyperText Transfer Protocol Secure) – (безопасный) протокол передачи гипертекста.

IPv4 (Internet Protocol Version 4) – межсетевой протокол версии 4.

IPv6 (Internet Protocol Version 6) – межсетевой протокол версии 6.

MAC (Media Access Control) – управление доступом к среде.

MSS (Maximum Transmission Segment) – максимальный размер сегмента.

MTU (Maximum Transmission Unit) – максимальный размер Ethernet-фрейма.

RFC (Requrest For Comments) – технический или организационный документ в области Интернета.

TCP (Transmission Control Protocol) – протокол управления передачей.

UDP (User Datagram Protocol) – протокол пользовательских датаграмм.

ПО – программное обеспечение.

ОС – операционная система.

ВВЕДЕНИЕ

Тестирование – это одна из общепринятых и неотъемлемых стадий жизненного цикла разработки программного обеспечения (ПО) и аппаратных платформ [1]. Само по себе тестирование тоже состоит из множества этапов и видов. В частности, существует нагрузочное тестирование – это «тип тестирования уровня производительности, проводимого для оценки поведения элемента тестирования при ожидаемых условиях переменной нагрузки обычно для ожидаемых условий низкого, типичного и пикового использования» [2].

В контексте данной работы, представляет интерес нагрузочное тестирование узлов (как программных средств, так и аппаратных) и каналов компьютерных сетей, в особенности сети Интернет. Одним из средств выполнения такого тестирования служит программно-аппаратный комплекс LR100Gen, разработанный компанией АО «ИнфоТеКС». В отличие от других подобных продуктов LR100Gen, во-первых, нацелен на соответствие отечественным стандартам в области быстродействия ПО (например, [3, 4]), во-вторых, позиционирует себя как полноценное средство измерения, тем самым выставляя для себя высокие требования по точности и воспроизводимости результатов измерений.

До сих пор LR100Gen предлагал поддержку только UDP/IPv4 (User Datagram Protocol, протокол пользовательских датаграмм и Internet Protocol Version 4, межсетевой протокол версии 4) траффика. Это, конечно же, представляет ценность для потребителя, но не насколько сильно, как траффик TCP/IPv4 (Transmission Control Protocol, протокол управления передачей). Поверх TCP работают известные протоколы передачи гипертекста HTTP и HTTPS, которые сейчас лежат в основе Всемирной паутины и обеспечивают работу веб-сайтов. Поддержка этих протоколов будет ключом к тому, чтобы LR100Gen смог обеспечить тестирование через эмуляцию полноценной сети устройств, создавая трафик, наиболее приближенный к реальному. Наличие

такой функциональности даст продукту более широкий рынок и конкурентное преимущество на нем.

Создание протоколов TCP и IPv4 (тогда он назывался просто IP, но с появлением межсетевого протокола версии 6 IPv6 стоит их разделять) и переход на них сети ARPANET 1 января 1983 года ознаменовали появление на свет Интернета – той сети, которая стала основой множества важнейших элементов нашего современного мира и используется миллиардами человек [5]. Важнейшая реализация этого стека была представлена Калифорнийским университетом в Беркли в 1983-1994 годах в ОС 4.x BSD, сетевые стеки многих операционных систем берут свое начало именно от нее [6].

Как можно заметить, реализация сетевого стека обычно относится к области задач операционной системы. Но, к сожалению, применять реализации протоколов из ОС в LR100Gen не удается за счет того, что эти реализации имеют обобщенный характер, они нацелены на широкий круг задач для обычного обмена данными по сети, который связан с выполнением большего числа операций для каждого обрабатываемого пакета. Универсальное решение проигрывает оптимизированному. Задача нагрузочного тестирования не вписывается в типичные задачи, под которые разрабатывались операционные системы и их сетевые стеки. Она имеет, например, такие требования как:

– обеспечение максимально возможной производительности, чтобы гарантировать работу тестируемого объекта при всевозможных условиях. Например, для генерации упомянутого UDP трафика на максимально возможной скорости 14,88 миллионов пакетов в секунду для 10-гигабитного интерфейса минимальными по размеру пакетами требуется отправлять пакет за время, не превышающее порядка 100-200 тактов процессора (67 нс). Так, ядро Линукса испытывает с этим трудности [7];

– эмуляция работы многих тысяч и миллионов пользователей, чтобы выполнить проверку тестируемого объекта в реалистичных условиях. Операционные системы в первую очередь нацелены на стандартный сценарий, когда один узел сети имеет один IP-адрес и один MAC-адрес;

– сбора нетипичных метрик, чтобы предоставить разработчику тестируемого объекта всю полноту информации для анализа и отладки;

– наличия временных гарантий, чтобы лучше управлять распределением трафика во времени (например, контролировать межпакетные интервалы и не вызывать micro-bursting [8], а также точно измерять скорости и задержки);

– прямой доступ ко всем протоколам, чтобы генерировать реалистичный траффик, имитировать различные ошибки в сети и оптимизировать некоторые сценарии (например, создания большого количества параллельных соединений малыми затратами памяти).

Попытки решения перечисленных проблем средствами ОС в теории возможны, но практике приводят к серьезным проблемам с производительностью. Вместо этого применяется технология kernel bypass (обхода ядра ОС и выполнения всех затратных операций в пространстве пользователя) [9, 10].

По этому пути пошли LR100Gen и другие аналогичные продукты на рынке (например, TRex [11]), они предлагают собственный оптимизированный под задачи нагрузочного тестирования сетевой стек, соединенный напрямую с сетевым интерфейсом. Все это сделано посредством фреймворка Data Plane Development Kit (DPDK), специально разработанного компанией Intel для обеспечения высочайшего уровня производительности сетевых приложений [12].

Подробнее проблема производительности сетевых стеков операционных систем рассматривается в [10, 13, 14].

Все эти ограничения, а к тому же и необходимость интегрирования в существующий стек, относятся также к протоколу TCP, что вынуждает реализовывать его самостоятельно. Но делать это с нуля излишне – корректная реализация трудоемка в разработке, тестировании, отладке и поддержке – поэтому создавать протокол решено было на основе уже существующего кода, который написан крупными специалистами области и отлажен на протяже-

нии десятилетий эксплуатации в одной из крупнейших операционных систем – FreeBSD.

Цель работы – создание протокола TCP, работающего целиком в пространстве пользователя, без взаимодействия с операционной системой, на основе исходного кода FreeBSD.

Объект работы – протокол TCP, а предмет – его устройство во FreeBSD и вносимые в сетевой стек FreeBSD изменения.

В первой главе объясняются основные принципы работы протокола TCP. Во второй главе производится анализ существующих реализаций TCP пространства пользователя. В третьей главе приводится описание сетевого стека ОС FreeBSD, а также процесса разработки библиотеки, реализующей протокол. В четвертой главе описываются мероприятия по обеспечению качества созданной библиотеки. В заключении подводится итог проделанной работы, делаются выводы и предлагаются улучшения.

1 ОБЗОР ПРОТОКОЛА УПРАВЛЕНИЯ ПЕРЕДАЧЕЙ TCP

Концепция сетевой модели TCP/IP предлагает разделять протоколы на четыре уровня: прикладной, транспортный, сетевой и канальный. Протоколы TCP и UDP относятся к транспортному уровню, они определяют принципы непосредственно передачи данных по сети между двумя прикладными процессами, запущенными на узлах. Путь (маршрут) передачи данных устанавливается двумя нижележащими уровнями: сетевым и канальным. Сами данные поставляются прикладным уровнем (его еще именуют уровнем приложений или просто приложением) [15].

В этой главе внимание будет обращено протоколу TCP, будет выполнен обзор его структуры и функциональности. В качестве источника информации используется [5], если не указано иначе. Этот фундаментальный труд по устройству сетевой модели TCP/IP, широко цитируется внутри сетевого стека FreeBSD. Русский перевод терминов дается по [15]. Первичными источниками информации о Интернет-протоколах являются документы Request For Comments (RFC).

1.1 Принципы работы протокола управления передачей

Вообще, сетевого и канального уровня достаточно для организации обмена данными двумя хостами, даже не соединенных напрямую. Но с ростом сложности сетей и многообразия устройств, появляются требования к гарантиям доставки данных в том виде, в котором они были отправлены. Действительно, пакеты в сети по разным причинам могут пропасть, дублироваться, перемешаться (нарушить свой исходный порядок). Из-за ограниченности ресурсов и ширины каналов связи встает вопрос об обеспечении управления потоком пакетов, то есть управлением скоростью отправки данных. Кроме того, обычно на хостах запущено множество процессов, выполняющих сетевые функции, из-за этого должен быть способ мультиплексиро-

вания исходящих пакетов и демультиплексирования входящих – их переадресацию нужным процессам.

После многолетнего изучения этих вопросов Винтом Серфом и Бобом Каном в 1974 году в статье «Протокол связи для сети на основе пакетов» был описан протокол TCP, а затем окончательно сформирован в 1984 году в виде RFC 793 (Transmission Control Protocol) [16] и RFC 791 (Internet Protocol) [17]. Именно эти два протокола легли в основу модели TCP/IP и сети Интернет. Последнее десятилетие предпринимаются попытки смены TCP на более эффективный протокол QUIC [18], но до сих пор TCP, уже около 40 лет, остается доминирующим.

Итак, определим основные черты протокола [16]:

(П1) Обеспечивает надежную доставку данных: все данные будут получены ровно в том виде, в котором они отправлены. Потери, дублирования, нарушения порядка и целостности будут устранены.

(П2) Поддерживает параллельную отправку сразу множества пакетов через сеть. Он не простаивает в ожидании уведомления о приеме пакетов, а сразу отправляет все данные.

(П3) Контролирует уровень использования ширины канала. Отправляться будет ровно столько данных, сколько способен обработать приемник (управление потоком) и сколько может пропустить через себя канал (управление перегрузкой).

(П4) Выполняет двусторонний равноправный обмен данными. Любая сторона в любой момент времени может как отправлять, так и принимать.

TCP воспринимает пользовательские данные в виде непрерывного потока байтов (называемых точнее октетами, как это принято в RFC [19]). Этот набор байт протоколом разбивается на группы, называемых сегментами (иногда и просто пакетами). Сегменты могут иметь произвольную длину, TCP выбирает ее оптимальной, но чаще всего равной MSS – максимальный размер сегмента (Maximum Segment Size**)**. Он вычисляется на основе максимального размера фрейма (канальный уровень) и пакета (сетевой уровень). В

обычной Ethernet сети с IPv4 без использования дополнительных опций в заголовках IP и TCP равен 1460 октетам.

(П1) решается применением нумерации всех октетов порядковыми номерами (sequence number), контрольных сумм и пересылки данных. Если в течении некоторого времени отправитель не получает подтверждения принятия сегмента, он отправляет его повторно.

Без (П2) обмен данными будет очень медленным, придется дожидаться прихода подтверждения для предыдущего сегмента, прежде чем отправлять новый. Набор сегментов, которые в данный момент времени уже были отправлены, но прием которых еще не был подтвержден, называется *окном отправителя* и строго поддерживается отправителем. С помощью него выполняется учет данных, которые были успешно приняты, которые были только отправлены и которые еще не были переданы на отправку приложением.

Аналогичная сущность есть и у приемника – *окно приемника*. Оно нужно для учета принятых и подтвержденных сегментов, ожидаемых сегментов и сегментов, принять которые сейчас невозможно (например, недостаточно памяти).

Рисунок 1.1 – Окно отправителя

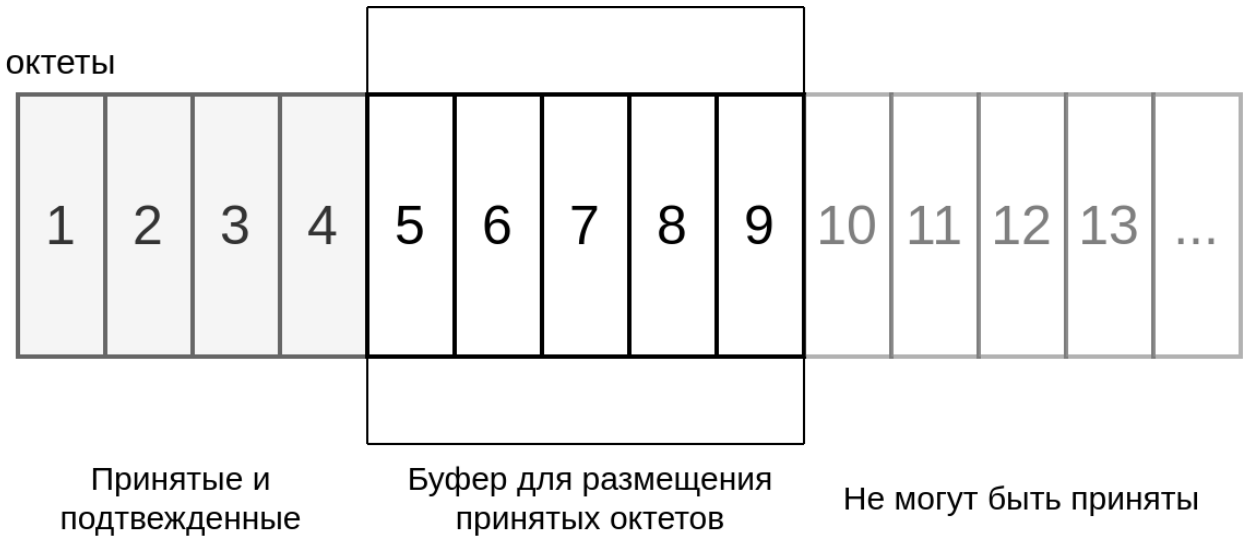


Рисунок 1.2 – Окно приемника

В процессе обмена по протоколу оба окна начинают «скользить», сдвигаться вправо.

Размер этих окон (то есть число сегментов в них) непостоянный, а динамически меняется в процессе обмена информацией. Размер окна приемника напрямую зависит от размера доступного буфера для входящих сегментов. А размер окна отправителя может менять явно через посылку размера окна (*обновления окна*) от приемника или неявно с помощью специальных алгоритмов, анализирующих различные параметры сети и предсказывающих лучшею величину окна. Тем самым разрешая (П3).

Максимальный размер окон – 64 Кбайта, но он может быть увеличен с помощью опции TCP Windows Scale до 1 Гбайта.

(П4) же обеспечивается двухсторонним соединением.

1.2 Соединение

Отличительной чертой TCP является установка соединения. Соединение представляет собой некоторую информацию, которую должны хранить обе стороны и которая необходима им для обеспечения общения между собой. По этой причине протокол TCP называют stateful протоколом (а UDP, в

свою очередь, stateless). Установку соединения необходимо провести до начала всякого обмена данными, а в конце разорвать, причем так, чтобы гарантировать доставку всех данных обеими сторонами.

TCP предоставляет full duplex соединение, то есть такое соединение, которое одинаково функционирует в обе стороны, поэтому стороны содержат одинаковый набор данных, включающий порядковые номера, окно отправителя, окно приемника и другое. Обычно эти данные объединяются в одну структуру – управляющий блок TCP (TCP Control Block) [6].

Каждое соединение идентифицируется четырехместным кортежем: (IP-адрес отправителя, порт отправителя, IP-адрес получателя, порт получателя). Портом называется число в диапазоне от 1 до 2^{16}, однозначно указывающая на прикладной процесс, использующий TCP. Он нужен как раз для того, чтобы один хост (один IP-адрес) мог создавать более одного соединения.

Сервером называется сторона TCP, которая открыта для новых соединений. Другое название сервера – passive opener. С помощью портов один сервер, находящийся на определенном порте, может обслуживать множество входящих соединений, генерируя новый локальный порт для каждого нового соединения.

Клиентом называется сторона TCP, которая не ждет подключений к себе, а сама создает новое соединения, подключаясь к серверу. Именуются также active opener.

1.3 Заголовок и типы сегментов

Как и любой протокол, TCP требует, чтобы каждый сегмент содержал в начале определенным образом структурированную информацию о себе и отправляющей стороне. Эти данные называются TCP-заголовком сегмента. Они занимают от 20 до 60 байтов (зависит от набора дополнительных опций, идущих после обязательного заголовка).

Общий вид пакета (Ethernet, IPv4, TCP) [15], исходящего в канал, показан на рисунке 1.3.

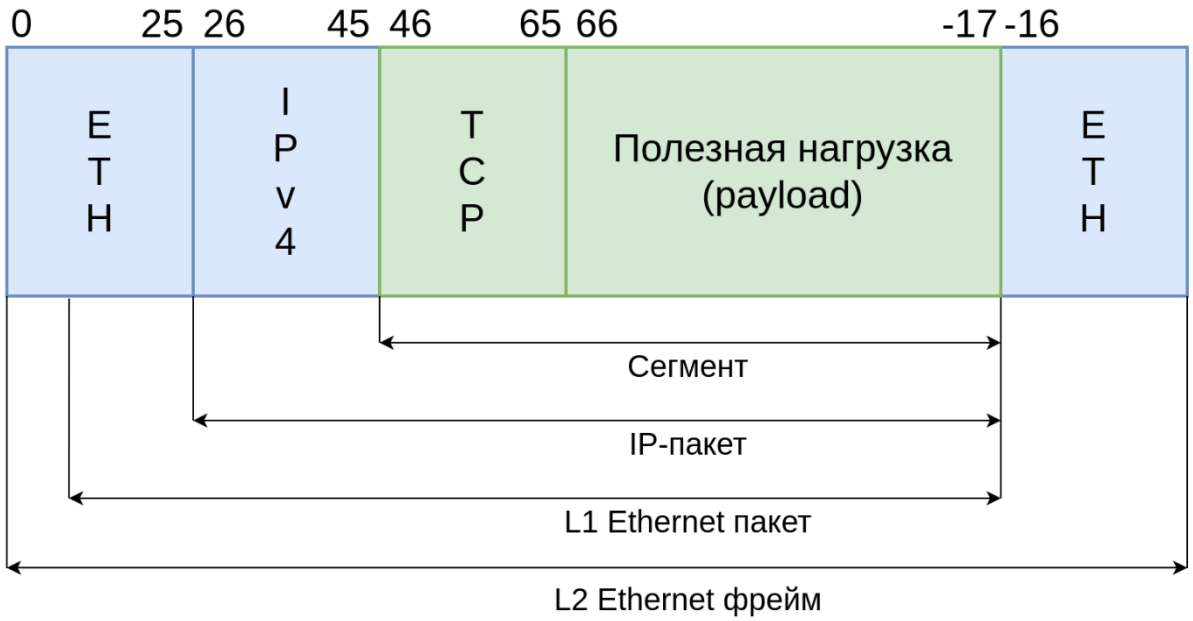


Рисунок 1.3 – Общий вид пакета, передающегося по витой паре

Формат заголовка сегмента представлен в таблице 1.1.

Таблица 1.1 – Заголовок сегмента

Бит	0–3	4–6	7–15	16–31
0	Порт источника		Порт назначения	
32	Порядковый номер			
64	Номер подтверждения			
96	Длина заголовка	Зарезервировано	Флаги	Размер окна
128	Контрольная сумма			Важность
160	Опции			
160–192	Данные			

Рассмотрим его подробнее.

Порт источника (Source Port) и *Порт назначения (Destination Port)*. Идентифицируют приложения на хостах. Комбинация IP-адреса и порта называется сокетом или конечной точкой (endpoint).

Порядковый номер (Sequence Number, SN). Порядковый номер первого октета в данном сегменте относительно всего потока октетов, которые отправитель пересылает получателю. Самый первый октет будет иметь номер ISN+1, где ISN – начальный номер (initial sequence number), а сложение делается из-за того, что сегмент, отправляемый для открытия соединения, тоже нумеруется.

Номер подтверждения (Acknowledgment Number, ACK SN). Порядковый номер октета, которые отправитель сегмента хочет получить у получателя. Считается, что все октеты до указанного уже удачно получены.

Длина заголовка (Data offset). Занимает 4 бита, указывает длину заголовка сегмента. Минимально 20 бит, максимально 60. Не постоянная величина, так как сегмент может иметь опции.

Флаги (Flags). Указывают на тип сегмента (см. ниже).

Размер окна (Window Size). Размер буфера, доступного для получения данных.

Контрольная сумма (Checksum). Для обеспечения целостности данных и заголовка, передаваемых внутри сегмента.

Указатель важности (Urgent Pointer). В данный момент не рекомендовано к использованию RFC 6093 [20], поэтому его описание опустим.

Опции (Options). Некоторые расширения протокола, позволяющие добавлять информацию в заголовок. Например, TCP Window Scale (увеличивает максимальный размер окна до 1 Гбайта) и TCP Timestamp (включение временных меток внутрь заголовка позволяет более точно высчитывать периоды для таймеров).

С помощью флагов все сегменты делятся на типы, причем типы могут совмещаться:

– SYN-сегмент. Производит запрос на создание нового соединения;

– ACK-сегмент. Подтверждает полученные данные, показывает, что все октеты до ACK SN получены удачно;

– FIN-сегмент. Инициирует закрытие соединения стороной, отправившей такой сегмент;

– RST-сегмент. Выполняет разрыв и сброс соединения. Обычно возникает из-за какой-то ошибки или приема сегмента, который не ожидается в данный момент;

– PSH-сегмент. Указывает принимающей стороне немедленно освободить свой буфер и передать накопленные данные на прикладной слой;

– и некоторые другие, менее важные.

1.4 Протокол управления передачей как автомат состояний

Состояние соединения удобно представлять в виде конечного автомата. И именно поддержка этого автомата вызывает достаточно сложности ввиду существовать большего количество возможных ситуаций. Диаграмма автомата достаточно сложная, поэтому здесь не приводится.

Рассмотрит состояния автомата:

– *CLOSED*. В этом состоянии TCP находится сразу после инициализации, а также после закрытия соединения;

– *LISTEN*. Ожидает получения SYN-сегмента (ждет подключения);

– *SYN_SENT*. TCP отправил SYN-сегмент, ждет ответа от стороны в состоянии LISTEN;

– *SYN_RCVD*. TCP получил SYN-сегмент, переходит сюда из состояния LISTEN;

– *ESTABLISHED*. Обе стороны удачно завершили рукопожатие и открыли соединение, в этом состоянии происходит обмен данными в обоих направлениях;

– *FIN_WAIT_1*. Сторона инициировала закрытие соединения, отправила FIN-сегмент;

– *FIN_WAIT_2*. Сторона в состоянии FIN_WAIT_1 получила ACK-сегмент;

– *CLOSING*. Обе стороны закрывают соединение одновременно.

– *CLOSE_WAIT*. Сторона получила FIN-сегмент;

– *LAST_ACK*. Сторона, получившая FIN-сегмент первой, ждет подтверждения получения FIN-сегмента другой стороной;

– *TIME_WAIT*. Сюда переходит из состояния FIN_WAIT_2 при получении FIN-сегмента. В нем сторона ожидает пока в сети гарантировано не будет пересылаемых пакетов.

На рисунке 1.4 пример переходов TCP в различные состояния в процессе работы.

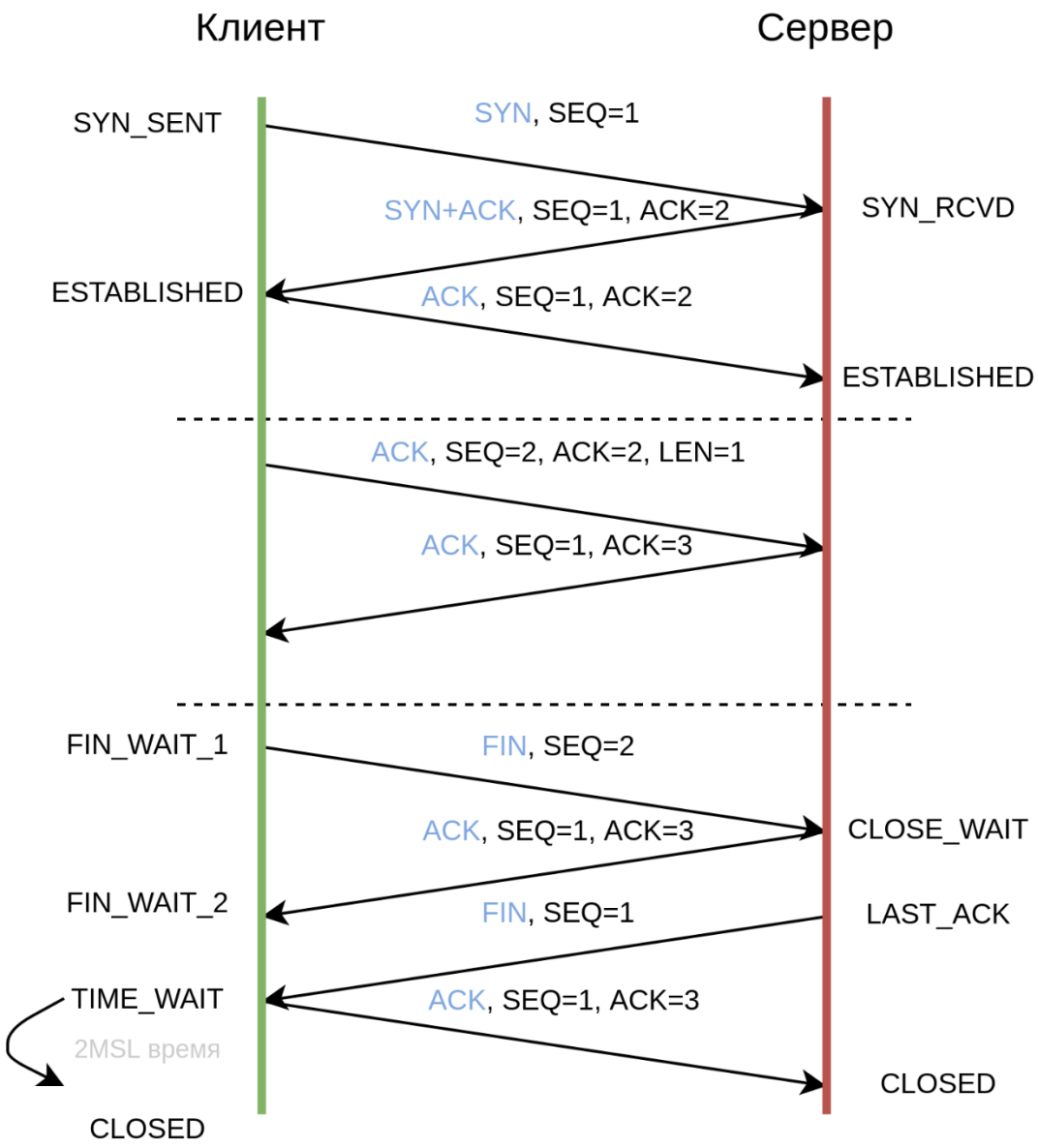


Рисунок 1.4 – Диаграмма взаимодействия по TCP

1.5 Таймеры

Во время работы TCP должен отмерять интервалы времени. Например, если в течении некоторого времени противоположная сторона не подтвердила получение пакета, то нужно произвести его повторную передачу, потому что другой возможности понять, был ли пакет действительно получен, нет. Таймер, занимающийся отсчетом времени ожидания подтверждения, называется таймером повторной передачи или Retransmission Timer.

Другой таймер – Keep Alive Timer, таймер проверки активности – занимается отслеживанием жизни другой стороны соединения. Если она внезапно пропадет (например, произойдет разрыв канала связи), то противоположная сторона об этом не узнает и соединение повиснет навечно. С помощью этого таймера TCP периодически отправляет другой стороне ACK-сегмент. Если же никакие сегменты не были получены в течении некоторого времени (обычно измеряемого часами), то соединение должно быть закрыто.

Persistent Timer (таймер настойчивости) нужен для борьбы с ситуацией, когда размер окна стал равен 0 (сторона соединения по какие-то причинам не может принимать данные, о чем уведомляет другую обновлением размера окна равным 0), а следующий сегмент, возвращающий окну ненулевой размер, был потерян. В такой ситуации обе стороны начинают ждать друг друга, и отправляющий данные никогда не сможет в действительности отправить данные, ведь окно равно 0. С помощью таймера с некоторой периодичностью выполняется запрос актуального размера окна.

Следующий таймер нужен только в состоянии TIME_WAIT, он соответственно называется Time Wait Timer (таймер 2MSL). В процессе закрытия соединения для того, чтобы гарантировать получение всех данных, нужно дождаться получения уже отправленных данных. Другая сторона точно не будет отправлять новые сегменты, но вот уверенности в том, что в сети отсутствуют отправленные в прошлом сегменты, нет. Для этих целей сторона,

первая решившая закрыть соединение, ждет некоторое время, прежде чем окончательно это сделать.

TCP должен подтверждать каждый полученный сегмент, но делать это именно таким образом (ответом на каждый пришедший сегмент) не очень эффективно. Поэтому обычно подтверждается сразу несколько сегментов скопом: TCP ждет некоторой время (порядка десятков миллисекунд) перед тем, как отправить ACK. Таймер для этого называется Delayed ACK Time (таймер отложенного подтверждения).

1.6 Пример взаимодействия клиента и сервера по протоколу

Для примера рассмотрим процесс получения клиентом (утилита curl) HTML-страницы по прикладному протоколу HTTP 1.1, работающего поверх TCP, от сервера (nginx) сайта lib.ru.

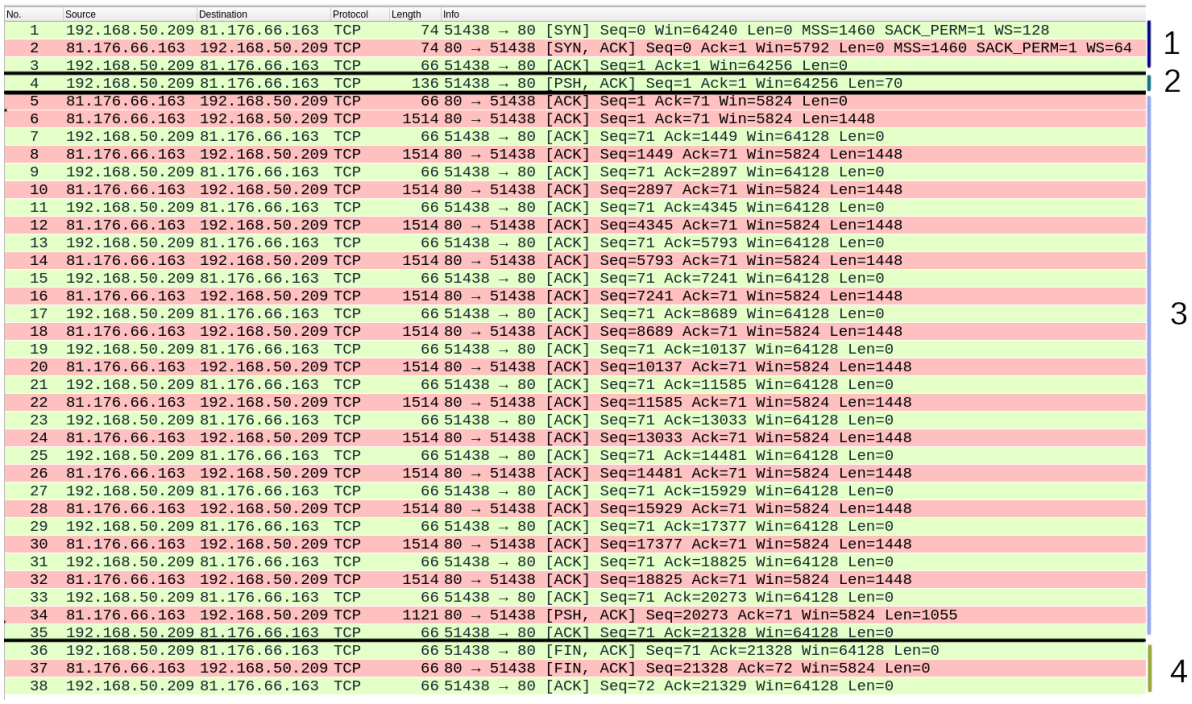
IP-адрес и порт клиента соответственно равны (192.168.50.209, 51438), а сервера – (81.176.66.163, 80). Скриншот программы Wireshark, содержащий таблицу сегментов, участвовавших в общении клиента и сервера, показан на рисунке 1.5.

Представленное взаимодействие полностью совпадает с описанием выше. Первая группа из 3 сегментов участвует в процессе рукопожатия и тем самым создает соединение. В начале размер окна клиента равен 64240 байта, а сервера – 5792 байта. Максимальный размер сегмента равен 1460 байтом.

Затем четвертый сегмент (порядковый номер 1 у клиента, вторая группа) отправляет HTTP-запрос (GET) с флагом PSH, занимающий 71 байт. После этого в течении еще 30 сегментов (судя по SEQ, около 20 Кбайт) сервер по частям отправляет клиенту запрошенную HTML-страницу.

В конце 36-ым сегментом четвертой группы (порядковый номер 71 у клиента) клиент инициирует завершение соединения. Сервер в ответ также отправлять FIN+ACK-сегмент, который затем клиент подтверждает.

Все сегменты, кроме самого первого, содержат флаг ACK и имеют максимальный размер, равный 1448 байтам (MSS без 12 байт, использующихся под опции в TCP-заголовке), кроме последнего. Размер окон в процессе взаимодействия не меняется.



No.	Source	Destination	Protocol	Length	Info
1	192.168.50.209	81.176.66.163	TCP	74	51438 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
2	81.176.66.163	192.168.50.209	TCP	74	80 → 51438 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 SACK_PERM=1 WS=64
3	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0
4	192.168.50.209	81.176.66.163	TCP	136	51438 → 80 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=70
5	81.176.66.163	192.168.50.209	TCP	66	80 → 51438 [ACK] Seq=1 Ack=71 Win=5824 Len=0
6	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=1 Ack=71 Win=5824 Len=1448
7	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=1449 Win=64128 Len=0
8	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=1449 Ack=71 Win=5824 Len=1448
9	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=2897 Win=64128 Len=0
10	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=2897 Ack=71 Win=5824 Len=1448
11	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=4345 Win=64128 Len=0
12	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=4345 Ack=71 Win=5824 Len=1448
13	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=5793 Win=64128 Len=0
14	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=5793 Ack=71 Win=5824 Len=1448
15	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=7241 Win=64128 Len=0
16	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=7241 Ack=71 Win=5824 Len=1448
17	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=8689 Win=64128 Len=0
18	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=8689 Ack=71 Win=5824 Len=1448
19	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=10137 Win=64128 Len=0
20	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=10137 Ack=71 Win=5824 Len=1448
21	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=11585 Win=64128 Len=0
22	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=11585 Ack=71 Win=5824 Len=1448
23	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=13033 Win=64128 Len=0
24	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=13033 Ack=71 Win=5824 Len=1448
25	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=14481 Win=64128 Len=0
26	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=14481 Ack=71 Win=5824 Len=1448
27	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=15929 Win=64128 Len=0
28	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=15929 Ack=71 Win=5824 Len=1448
29	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=17377 Win=64128 Len=0
30	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=17377 Ack=71 Win=5824 Len=1448
31	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=18825 Win=64128 Len=0
32	81.176.66.163	192.168.50.209	TCP	1514	80 → 51438 [ACK] Seq=18825 Ack=71 Win=5824 Len=1448
33	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=20273 Win=64128 Len=0
34	81.176.66.163	192.168.50.209	TCP	1121	80 → 51438 [PSH, ACK] Seq=20273 Ack=71 Win=5824 Len=1055
35	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=71 Ack=21328 Win=64128 Len=0
36	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [FIN, ACK] Seq=71 Ack=21328 Win=64128 Len=0
37	81.176.66.163	192.168.50.209	TCP	66	80 → 51438 [FIN, ACK] Seq=21328 Ack=72 Win=5824 Len=0
38	192.168.50.209	81.176.66.163	TCP	66	51438 → 80 [ACK] Seq=72 Ack=21329 Win=64128 Len=0

Рисунок 1.5 – Скриншот программы Wireshark, перехватившей пакеты соединения клиента (зеленый/белый) и сервера (красный/серый)

2 ОБЗОР СУЩЕСТВУЮЩИХ РЕАЛИЗАЦИЙ TCP ПРОСТРАНСТВА ПОЛЬЗОВАТЕЛЯ

Во введении к данной работе были определены причины, по которым реализации TCP, расположенные внутри операционных систем, не подходят. В этой главе речь пойдет о существующих реализациях TCP, способных работать независимо от ОС в пространстве пользователя, будут рассмотрены причины, из-за которых эти реализации не подходят и необходимо разрабатывать собственную.

Все рассматриваемые стеки работают поверх DPDK. И все предоставляют интерфейс сокетов (Berkley Sockets), подробнее о котором будет рассказано в следующей главе.

2.1 mTCP

Широко масштабируемый стек TCP уровня пользователя для многоядерных систем [21]. Распространяется по лицензии Modified BSD, имеет открытый исходный код. Кроме TCP предоставляет IP и ARP протоколы.

Может быть достаточно легко интегрирован в существующие приложения путем замены стандартных сокетов на сокеты из стека. Сокеты поддерживают epoll для асинхронной обработки. Не имеет описания архитектуры, из документации только справки по API.

Из важных недостатков можно выделить следующие:

– возможно поднять только один сетевой стек на сетевой интерфейс;

– ориентирован на использование на одном сетевом интерфейсе;

– ограничивает число сокетов – 100000, чего недостаточно для высокопроизводительных тестов;

– тестирован разработчиками только для сетевых интерфейсов, выпущенных компанией Intel;

– отсутствует поддержка алгоритмов управления перегрузкой;

– не обновлялся несколько лет;

– заточен на функциональность, а не производительность.

2.2 F-Stack

Набор разработки пространства пользователя с высокой производительностью, основанный на DPDK, FreeBSD TCP/IP стеке, и API сопрограмм (corutine API) [22]. Распространяется с открытым исходным кодом по лицензии BSD 2-Clause.

Пользователю предоставляет полноценное Posix API: сокеты, epoll, kqueue, select, poll. Кроме этого, также работает с корутинами и микропотоками.

Во многом схож с предыдущим стеком. Документация также слабая. Но предоставляет больше возможностей для асинхронной обработки. И также, как mTCP, не предоставляет доступа к деталям машины состояний TCP, создавая сложности решения задач с необычным использованием TCP. Может генерировать трафик с различными IP, но их количество ограничено 64.

2.3 ANS

Нативный сетевой стек, ускоренный с помощью DPDK. Имеет закрытый исходный код, распространяется в виде набора статических библиотек, скомпилированных под разные процессоры [23]. Имеет крайне скудную документацию в виде нескольких небольших руководств.

Стек работает в виде отдельного процесса, к которому обращается пользовательский код. По этой причине нет возможности использовать более одного стека. Как и у предыдущих стеков отсутствуют способы прямого взаимодействия с машиной состояний, что еще и усугубляется закрытыми исходниками.

2.4 VPP

Расширяемый фреймворк, предоставляющий из коробки функциональности маршрутизатора и коммутатора [24]. Разрабатывается компанией Cisco, исходные коды открыты под лицензией Apache.

По сравнению с предыдущими библиотеками, имеет внушительный функционал, сопоставимый с таковым из операционных систем: кроме обычных TCP, IP, ARP, работает с IPv6, IPSec, VLAN, VXLAN, GRE и много другое. Имеет серьезную документацию. Требует значительного времени на изучение, очень большой и сложный. Запускает в виде отдельной службы, к которому сторонние приложения обращаются по специальному API. Соответственно, нельзя запускать больше одного стека.

Объемность фреймворка и является недостатком – он имеет высокий порог входа, тянет большое число зависимостей и функций, которые в данной задаче просто не нужны.

2.5 OpenFastPath

Реализация высокопроизводительного TCP/IP стека с открытым исходным кодом (лицензия BSD 3-Clause) [25]. Хорошо и просто документирован. Работает на устаревшей версии DPDK 17.11. Как и первые 3 рассмотренных продукта, не предоставляет доступ к внутренностям TCP.

2.6 Итог рассмотрения существующих реализаций протокола

Рассмотренные сетевые стеки направлены на замену стандартного стека в серверных приложениях, требующих высокой производительности. То есть, как и стеки операционных систем, ориентируются на широкие и стандартные запросы обычных приложений. На них невозможно или очень сложно реализовать вещи, относящиеся к нестандартным применениям TCP, например, создания «виртуальных» соединений (которые только открывают-

ся, но не закрываются только на одной стороне, без уведомления второй), для тестирования максимального числа параллельных соединений, поддерживаемых сервером.

Но по сравнению с операционными система, эти стеки имеют большую проблему – им мало доверия. Это малопопулярные решения в узкой области, чаще всего редко и неохотно обновляемые. В отличие от стеков операционных систем, данные проекты могут легко содержать ошибки и неотлаженные места.

Попытки адаптации исходных кодов рассмотренных библиотек из-за часто поверхностной документации и априорной сложности таких проектов сами по себе займут много сил и времени.

По всем описанным причинам было решено не использовать и не переиспользовать существующие проекты, а написать собственную реализацию TCP, взяв за основу сетевой стек популярной, разрабатываемой многие годы и свободной (в том числе для коммерческого использования) ОС FreeBSD (решение похоже на F-Stack, но использует куда меньшую часть ОС). Описание архитектуры сетевого стека FreeBSD и процесс переноса ее в пространство пользователя представлены в следующих главах.

3 ОПИСАНИЕ ПРОЦЕССА РАЗРАБОТКИ РЕАЛИЗАЦИИ ПРОТОКОЛА УПРАВЛЕНИЯ ПЕРЕДАЧЕЙ TCP

Данная глава подробно останавливается на описании всего процесса выделения реализации TCP из сетевого стека FreeBSD в отдельную библиотеку. Здесь обсуждается архитектура FreeBSD и получившейся библиотеки, рассказывается о компонентах реализации TCP и вносимых в них изменениях, демонстрируются результаты модульного, функционального и нефункционального тестирования.

3.1 Обзор операционной системы FreeBSD

Итак, в качестве основы для реализации протокола TCP взят исходный код операционной системы FreeBSD (ветка releng/12.1 [26] —релизная ветка FreeBSD версии 12, к которой выходят со временем только исправления серьезных ошибок). Это современная стабильная UNIX-подобная операционная система, появившаяся в 1992 году как продолжатель ОС 4.4BSD-Lite. Она активно развивается и имеет поддержку всех современных функций (многопроцессорность, IPv6, файловая система ZFS, jails и многое другое) [27]. Ядро системы написано на языке C99.

Код ОС распространяется по лицензии BSD 2-Clause, важным отличием которой от другой, не менее популярной лицензии GNU General Public License, применяемой, к примеру, проектом ядра Linux, является возможность коммерческого использования кода [28].

3.2 Обзор фреймворка DPDK

Важной зависимостью библиотеки является фреймворк Data Plane Development Kit (DPDK) — набор связанных библиотек, позволяющих сильно повысить скорость обработки сетевых пакетов [12]. Основным назначением является обеспечение прямого доступа к сетевым интерфейсам в обход ядра

операционной системы (kernel bypass), а также различные высоко оптимизированные решения в области памяти, логических ядер процессора (и вообще параллельной обработки данных), межпотокового взаимодействия, управлением потоками трафика и другое. Полный список библиотек и возможностей доступен в официальном руководстве [29].

В данной реализации TCP активно используются некоторые возможности из DPDK. В первую очередь, это библиотека rte_mbuf и rte_mempool для работы с пакетами, а также функции для создания так называемых thread local переменных — глобальных переменных, имеющих одинаковое имя, но уникальное значения в каждом потоке. Также используется библиотека rte_malloc для выделения динамической памяти в куче, расположенной в больших страницах (hugepages). Подробнее аспекты применения DPDK будут изложены дальше.

3.3 Архитектура разрабатываемой библиотеки

Архитектура во многом повторяет таковую сетевого стека FreeBSD, при этом были оставлены лишь необходимые его части, в основном относящиеся к TCP (практически полностью, кроме дополнительных расширений, например, TCP Fast Open или специфичного для ОС логирования), сокетам и работе с пакетами. Получившаяся архитектура представлена на рисунке 3.1.

Проект, в который будет внедряться протокол, предполагает выполнение больших объемов процессорных вычислений и полное отсутствие блокирующих операций ввода/вывода. По этой причине архитектура проекта применяет такое многопоточное выполнение, когда один поток закрепляется за ровно одним логическим ядром многоядерного процессора. Кроме этого, важную роль играет понятие шардинга (sharding) [30], т.е. строгое разделение всех данных программы, используемых в многозадачном режиме, так, чтобы

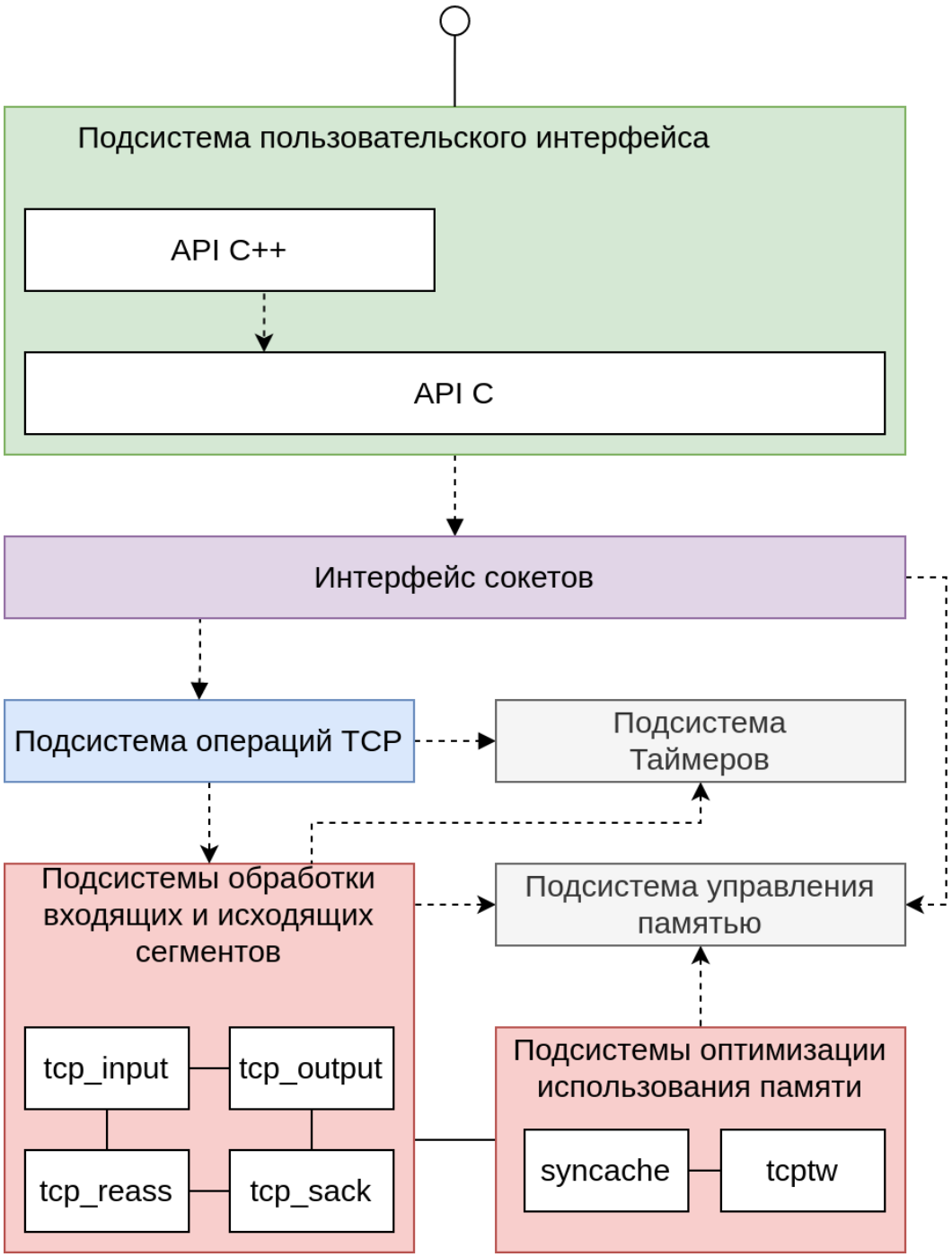


Рисунок 3.1 – UML-диаграмма компонентов библиотеки

отсутствовали критические секции — у потоков не было общих данных. Это делается либо непосредственно разделением данных на непересекающиеся части, либо копированием данных для каждого потока.

Этим же принципам следует и разработанная реализация TCP. На каждом потоке (логическом ядре) создается свой уникальный экземпляр глобальных данных протокола (например, хэш-таблицы) в thread local перемен-

ных. Таким образом, каждое TCP-соединение присваивается своему экземпляру и обрабатывается всегда одним и тем же логическим ядром. Это, во-первых, позволяет избавиться от необходимости использования множества блокировок, а их были десятки внутри оригинального кода, они составляли сложную и опасную иерархию. И, во-вторых, делает код дружественным к кэшу уровней L2 и L1, т.к. одни и те же данные обрабатываются на одном и том же ядре, что не приводит к вымыванию кэша процессора [31].

Внешние зависимости библиотеки инкапсулируются в структуру типа struct net_stack (таблица 3.1). Экземпляры этой структуры для каждого ядра создаются при инициализации библиотеки и наполняются пользователем.

Таблица 3.1 – Поля структуры struct net_stack

Название поля	Тип	Описание
small_mp	struct rte_mempool*	Указатель на пул памяти, через который выделяются буферы памяти для заголовков и indirect буферов памяти
ts	struct timeouts*	Хранилище таймеров, управляющая структура для создания новых и изменения старых таймеров
ip_output	int (*)(void*, struct rte_mbuf*)	Указатель на функцию, в которую передаются готовые сегменты для отправки
arg	void*	Дополнительный параметр (контекст) для функции ip_output

В следующих подразделах будут подробнее рассмотрены подсистемы, их задачи и структура, а также важные внесенные изменения в оригинальный код FreeBSD. Мелкие изменения, например, подмены макросов на пустые операции (для блокировок и логирования), удаление ненужных функций и т.п., опущены. Всего объем полученного кода составляет около 25 тысяч строк.

3.4 Подсистема интерфейса сокетов

Программный интерфейс сокетов является мощной абстракцией для межпроцессорного взаимодействия типа клиент-сервер, причем общающиеся

процессы могут быть расположены на разных компьютерах. Появившись впервые в ОС 4.2BSD в 1983 году [6] набрал большую популярность и теперь представлен во всех современных ОС. Сокетом называется одна из сторон взаимодействия.

Все сокеты должны реализовывать одинаковый набор функций, таким образом предоставляя пользователю единообразный интерфейс. Функций много и в библиотеку перенесены не все, а только самые необходимые. В таблице 3.2 представлены их описания.

Таблица 3.2 – Список функций интерфейса сокетов

Функция	Описание функции
socket()	Создает новый сокета
sobind()	Привязывает сокет к локальному адресу
solisten()	Превращает сокет в слушающий (готовый принимать входящие соединения)
soconnect()	Подключается по указанному адресу к слушающему сокету
soaccept()	Принимает входящие соединение, создает новый сокет, подключенный к клиенту
sowrite()	Отправляет данные
soread()	Получает данные
soshutdown()	Закрывает соединение
soclose()	Закрывает соединение и очищает ресурсы сокета

Существуют разные типы сокетов, но в данной работе представляют интерес интернет-сокеты (Internet sockets) и конкретно их реализация для TCP/IP (во FreeBSD сокеты реализованы для всех транспортных протоколов).

Внутри FreeBSD (как и в других ОС) сокеты сделаны на основе объектно-ориентированного подхода: для каждого стека протоколов создается экземпляр структуры типа struct protosw, который через структуру struct pr_usrreqs содержит в себе указатели на функции, реализующие интерфейс сокетов для конкретного протокола. Затем при создании экземпляра сокета (тип struct socket) его поле so_proto будет ссылаться на соответствующий экземпляр struct protosw, и через него все вызовы операций над сокетом будут переадресовываться нужным функциям. Названия переменных типа struct

protosw и struct pr_usrreqs для TPC/IPv4 соответственно inetsw и tcp_usrreqs (рисунок 3.2).

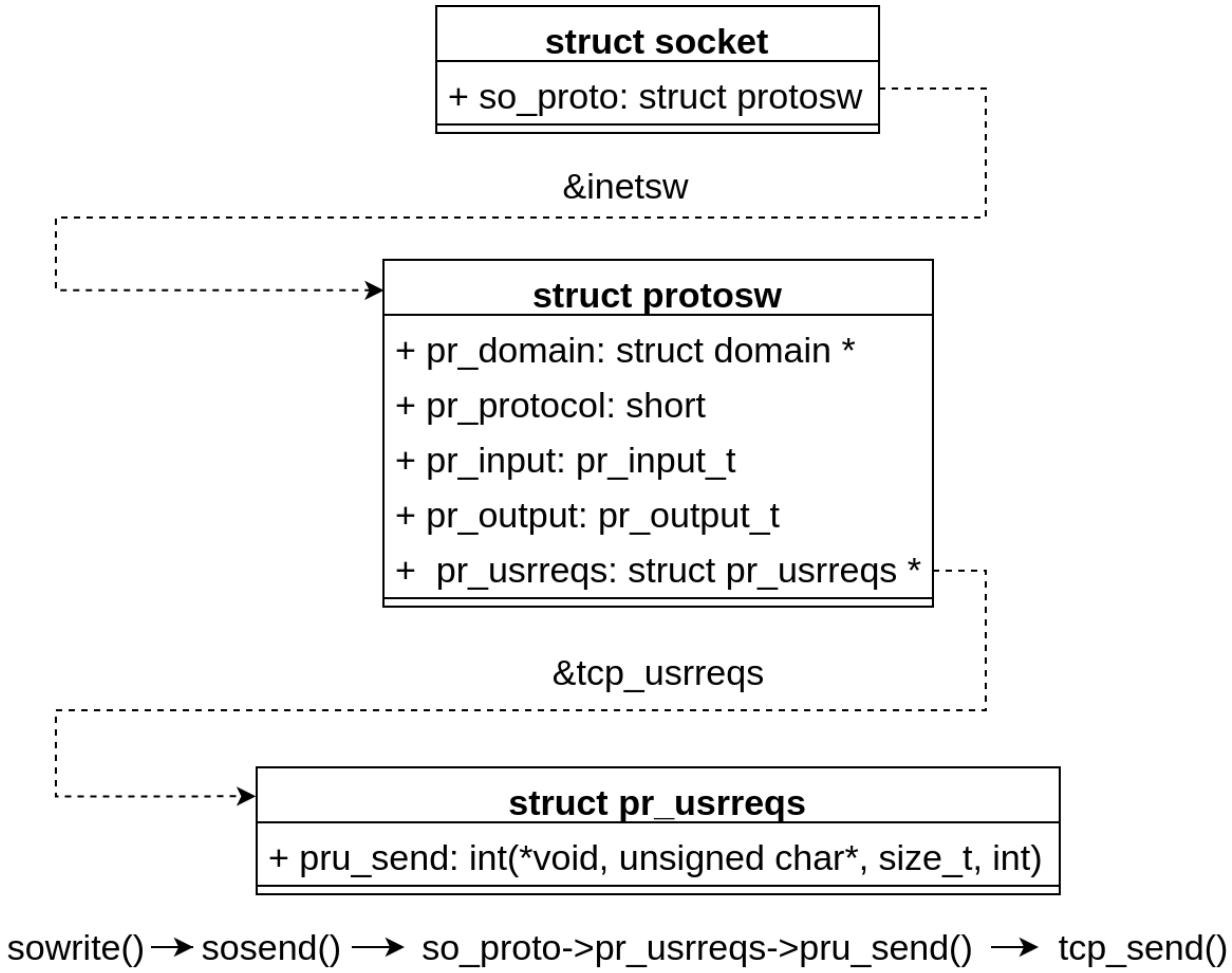


Рисунок 3.2 – UML-диаграмма структур, содержащих операции сокетов, и пример последовательности вызовов функции отправки данных через сокет

Существует два вида сокетов обычный (regular socket) и слушающий (listening socket). На самом деле они сильно отличаются. Первый представляет собой два буфера (struct sockbuf), по сути являющихся реализацией окон приемника и отправителя, и нужен для приема/отправки данных по созданному соединению или подключению к слушающему сокету. Слушающий сокет используется со стороны сервера и отвечает за создание новых соединений, он хранит в себе очередь клиентов, которые хотят подключиться. При

вызове операции soaccept() на слушающем сокете создается соответствующий обычный сокет для взаимодействия с клиентом со стороны сервера.

Поскольку работа с протоколом в библиотеке предполагается однопоточная, то операции выполнены не в блокирующем (синхронном) варианте, а неблокирующем (асинхронном). Это означает, что все операции над сокетом выполняются мгновенно, а если они не могут быть выполнены тут же (к примеру, при отправке данные необходимо дождаться получения подтверждения), то возвращают код EBLOCKING, не приводя к приостановке работы потока. Вместо ожидания блокировки пользователь передает указатель на функцию, которую библиотека должна будет в будущем вызвать (сделать обратный вызов), чтобы уведомить пользователя, когда эта операция для сокета станет доступна. Например, запись в сокет выглядит так:

```
void on_written_someting(struct socket* so) {
   // С этого места гарантируется, данные удачно отправлены
   // и подтверждены.
}

void write_something(struct socket* so, struct mbuf* m) {
    bsd_tcp_async_write(so, m, on_written_something);
}
```

вместо:

```
void write_something(struct socket* so, struct mbuf* m) {
    bsd_tcp_write(so, m);
    // С этого места гарантируется, данные удачно
    // отправлены и подтверждены.
}
```

Такой подход к построению асинхронных систем называется шаблоном проектирования «Реактор»: пользователь регистрирует обработчики событий о возможности совершения неблокирующей операции над ресурсом, а система через обработчики уведомляет пользователя о наступлении этих событий.

FreeBSD уже имеет такую встроенную поддержку обратных вызовов, она называется upcall (обратный вызов) и используется только внутри ядра. К

сожалению, в данной версии FreeBSD есть особенность, связанная с работой машины состояний при обработки входящего сегмента, которая приводит к тому, что обратный вызов происходит в тот момент, когда управляющий блок TCP только частично перешел в новое состояние. Это приводит к серьезным ошибкам при попытки вызвать новые операции над сокетом в upcall. Поэтому пришлось воспользоваться набором изменений (патчем) [32] из FreeBSD 13, устраняющим эту проблему.

Другой способ построения асинхронных операций — kqueue — использоваться не стал из-за излишней его сложности, он сам по себе является целой отдельной подсистемой. Более простых обратных вызовов вполне достаточно.

В библиотеке функции интерфейса сокетов представлены в файлах bsd_kern/uipc_socket.c и bsd_kern/uipc_sockbuf.c. Напрямую функции не выполняют никаких сложных операций, а производят проверки внутреннего состояния сокета и его буферов на предмет возможности выполнения функции (например, нельзя записать в сокет, если его буфер заполнен), выделяют память под сокет, при отсутствии ссылок на сокет очищают память, управляют буферами памяти (добавляют и удаляют их них данные).

Все блокирующие функции удалены, а все вызовы функций пробуждения потоков, ожидающих сокет, перенесены в файлы bsd_netinet/tcp_output.c и bsd_netinet/tcp_input.c и больше не взаимодействуют с планировщиком ОС, выполняя только обратные вызовы (upcall).

Непрямые вызовы по указателю функций из структуры struct protosw заменены прямыми обращениями к операциям TCP.

3.5 Подсистема операций протокола управления передачей TCP

Пользователь взаимодействует с реализаций протокола TCP посредством набора специальных операций. Интерфейс сокетов из таблицы 3.2 полностью соответствует этим операциям один к одному.

Все функции, реализующие операции TCP, находятся в файле bsd_netinet/tcp_usrreq.c. Также в файле bsd_netinet/tcp_subr.c расположены некоторые внутренние (приватные) функции, например создания и удаления управляющего блока, расчета максимального размера фрейма (maximum transmission unit, MTU), расчета MSS.

Основные изменения в этих файлах связаны с удалением поддержки нескольких реализаций TCP (были добавлены в оригинальный код для алгоритмов управления перегрузками BBR и RACK, которые в библиотеке не нужны), а также упрощением некоторых функций, например, теперь не генерируется случайный безопасный ISN, а MTU считается константным и не вычисляется на основе параметров сетевого интерфейса.

Управляющий блок TCP представлен структурой struct tcpcb. Она имеет размер около 900 байтов, содержит всю информацию о соединении: размеры и положение окон, флаги состояний, данные таймеров (включая сами таймеры) и т. п. К тому же, он включает в себя часть управляющего блока IP, который хранит информацию об адресе сокета, и, что важнее, является элементом хэш-таблицы, которая таким образом хранит в себе все созданные сокеты. Ключом этой таблицы является кортеж (локальный порт, локальный адрес, удаленный порт, удаленный адрес). С помощью таблицы для каждого входящего сегмента определяется его управляющий блок. Все три структуры: struct socket, struct tcpcb и struct inpcb — ссылаются через указатели друг на друга. На рисунке 3.3 показаны связи между основными структурами библиотеки.

3.6 Подсистема таймеров

FreeBSD имеет крупную и сложную подсистему, называющуюся callouts и позволяющую планировать вызов некоторой функции через некоторый промежуток времени. С помощью callouts реализуются и таймеры

TCP: каждый управляющий блок содержит структуру с четырьмя struct callout, соответствующих каждому таймеру (см. раздел 1.4).

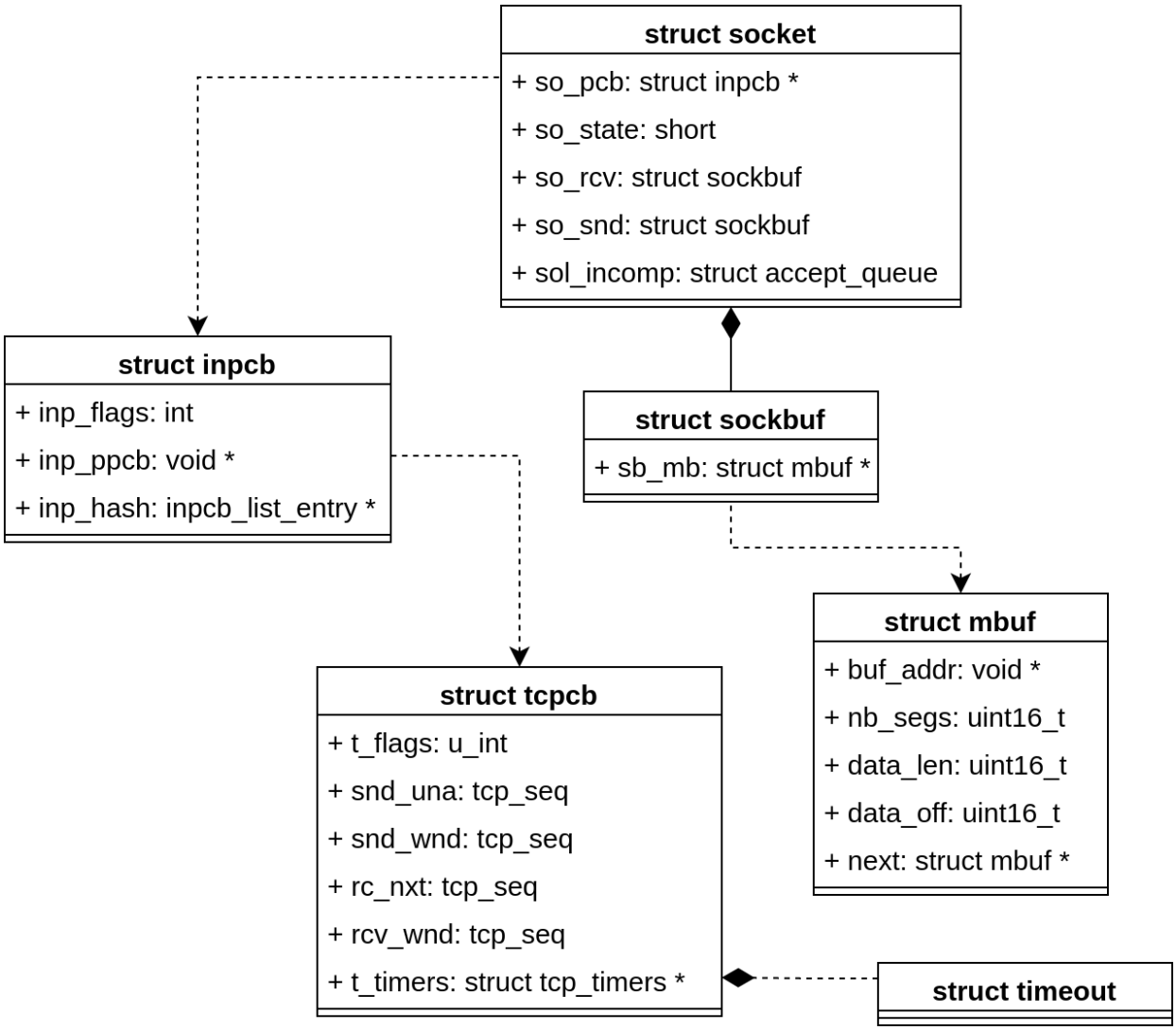


Рисунок 3.3 – UML-диаграмма основных структур. Показаны только некоторые поля

В библиотеке таймеры реализованы на основе структуры данных «циклическое расписание» (timer wheel) из библиотеки timeouts [33] на основе публикации [34]. Эта структура данных предоставляет все операции над таймерами (добавление, удаление, обработка) за O(1) в худшем случае.

Суть идеи заключается в поддержании специальной таблицы, каждая ячейка которой содержит указатель на список таймеров, которые должны сработать в одно время. Каждой ячейке соответствует смещение в тактах от-

носительно текущего времени, когда таймеры в этой ячейке должны сработать. С каждым новым тактом указатель переходит к следующей ячейке и таймеры в ней обрабатываются. Поэтому таблица напоминает колесо (wheel), которое с каждым тактом проворачивается на одну ячейку относительно указателя текущего времени. Разные вариации идеи помогают уменьшить сложность перебора, создания, удаления и обработки таймеров.

Функции всех пяти таймеров TCP находятся в файле bsd_netinet/tcp_timer.c, здесь же находится функция tcp_fasttime(), вызываемая каждые 500мс, и увеличивающая на один такт текущего время. Каждая функция выполняет нужные действия, в том числе высчитывает время своего следующего запуска, и устанавливает таймер заново. В этом же файле присутствуют общие утилитарные функции для запуска, приостановки и остановки конкретных таймеров по имени.

Функции таймеров могут выполнять закрытие соединения и удаление управляющего блока, а вместе с ним и удаление обрабатываемого таймера из списка. Это приводит к инвалидации итератора истекших таймеров (тех таймеров, функции которых должны быть выполнены на данном тике). Дальнейшее использование инвалидированного итератора приведет к доступу к недействительной памяти.

Для решения этой проблемы, в функции таймеров добавлен возврат целого числа — ненулевой код означает, что итератор был инвалидирован. Исходный код обработки таймеров приведен ниже

```
// @param ts Указатель на хранилище таймеров.
// @param step Число тактов, на которые изменилось время.
static inline int
callout_process_all(struct timeouts *ts, timeout_t step)
{
    int ec;
    callout *c;
    // Обновление времени для всех таймеров.
    timeouts_step(ts, step);

    // Обход всех таймеров, которые наступили к данному
    // моменту.
```

```
    while (1) {
        TIMEOUTS_FOREACH(c, ts, TIMEOUTS_EXPIRED) {
            if (c->callback.fn != NULL) {
                // Если вернулся не 0, то были внесены
                // изменения в Timer Wheel и текущий
                // итератор инвалидирован.
                ec = c->callback.fn(c->callback.arg);
                if (ec != 0) {
                   continue;
                }
            }
        }
        break;
    }

    return 0;
}
```

По сравнению со стандартной реализацией был сокращен период 2MSL-таймера, используемого в состоянии TIME_WAIT после закрытия соединения, до 1 с (минимальное возможное время). Сделано это с целью минимизации используемой памяти, потому что в ином случае после завершения соединения сокет еще долгое время (больше минуты) занимал в память место — это приводило к тому, что на долгих тестах у генератора заканчивалась память. Надо заметить, конкретное значение (2 минуты) указано в стандарте TCP с оговоркой, что может быть на практике изменено.

3.7 Подсистема управления памятью

Краеугольным камнем сетевого стека является механизм управления памятью. Память интенсивно используется для управляющих структур (управляющие блоки проколов, сокеты, таймеры, хэш-таблицы).

DPDK предлагает применять в основном большие страницы (hugepages). Стандартная страница в 64-битной версии ядра Linux — 4 Кбайта. Большие же страницы имеют размер 4 Мбайта или 1 Гбайт. Они нужны для того, чтобы снизить нагрузку на буфер ассоциативной трансляции (translation lookaside buffer, TLB) — кэш процессора, позволяющего ускорить трансляцию виртуальных адресов. Чем больше размер страницы, тем боль-

шее пространство виртуальной памяти покрывает буфер. А также такие страницы не помещаются в своп [35]. Вся динамически аллоцируемая память выделяется в больших страницах с помощью функции rte_malloc() (аналог malloc() из DPDK).

Также DPDK предоставляет специальный механизм для буферов памяти под пакеты. Эти буферы представляются структурой rte_mbuf (от message buffer — буфер сообщения). Они хранятся в пулах памяти (rte_mempool) — предвыделенных участках памяти, обеспечивающих быстрое создание и удаления блоков памяти фиксированного размера. mbuf хранит в себе метаданные (тип, размер, указатель и смещение на начало данных и т.д.) и участок памяти фиксированного размера (2048 байт), который можно свободно использовать для данных. Кроме того, mbuf могут быть объединены в цепочки для представления произвольных по размеру пакетов. Указатель на rte_mempool, с помощью которого должны создаваться mbuf, содержится в экземпляре struct net_stack, хранящемся в глобальной переменной.

Буферы сообщений бывают двух типов: прямые (direct) и непрямые (indirect). Прямые буферы полностью владеют своей памятью, а непрямые буферы лишь ссылаются на память, которой владеют прямые буферы. Таким образом, можно создавать разные буферы, которые ссылаются на одни и те же данные. С помощью этого механизма можно реализовывать подход отсутствия копирования (zero-copy) — когда сетевой стек не выполняет внутри себя копирования данных, переданных пользователем.

Дизайн rte_mbuf вдохновлен похожей структурой struct mbuf из сетевого стека FreeBSD. Это положительно сказалось на процессе адаптирования стека к новой структуре. Для некоторых полей получилось (например, размера) сделать псевдонимы с помощью макросов (#define). Из mbuf убрано сохранение информации о заголовках при пересечении границ слоев (остался только размер заголовков для каждого слоя, это нужно DPDK). Убрана возможность ссылаться на другие пакеты (поле pktnext).

В таблице 3.3 указан список функций над mbuf. В них внесены большие изменения из-за разницы в struct rte_mbuf и struct mbuf.

Таблица 3.3 – Описание функций над mbuf в библиотеке

Функция	Описание функции
void m_free(struct mbuf *m)	Освобождает mbuf. Возвращает его в пул памяти
struct mbuf *m_copym(struct mbuf *m, int off0, int len)	Копирует указанный диапазон памяти в новый буфер
void m_adj(struct mbuf *m, int req_len)	Выделяет память в начале буфера
void m_freem(struct mbuf *m)	Удаляет первый mbuf из цепочки
void m_cat(struct mbuf *m, struct mbuf *n)	Соединяет две цепочки
struct mbuf *m_gethdr(struct mempool *mp, int how, int type)	Создает новый mbuf для хранения заголовка протокола

Для управления памятью, отведенной для TCP-окон обоих типов, существует структура struct sockbuf. Она хранит цепочку mbuf и позволяет получать из нее участок памяти по смещению и размеру. Каждый sockbuf хранит указатель на функцию, используемую для обратного вызова. В таблице 3.4 показаны методы struct sockbuf. Их пришлось сильно переписать из-за изменений, связанных с mbuf, а также удалением ненужных связей с другими системами ядра (например, контролем ресурсов процесса).

Таблица 3.4 – Описание функций для struct sockbuf

Функция	Описание функции
uint sbavail(struct sockbuf *sb)	Возвращает число доступных байт
uing sbused(struct sockbuf *sb)	Возвращает число всего находящихся в буфере байт
long sbspace(struct sockbuf *sb)	Возвращает число байт доступного пространства в буфере
void sballoc(struct sockbuf *sb, struct mbuf *m)	Добавляет новый mbuf
void sbfree(struct sockbuf *sb, struct mbuf *m)	Удаляет mbuf
void sbrelease(struct sockbuf *sb, struct socket *so)	Очищает буфер и зарезервированное пространство
void sbdrop(struct sockbuf *sb, int len)	Удаляет указанное число байт из начала буфера
void sbflush(struct sockbuf *sb)	Очищает буфер (освобождает все mbuf)
void sbreserve_locked(struct sockbuf *sb, u_long cc, struct socket *so)	Резервирует указанное число байт в буфере

Продолжение таблицы 3.4

Функция	Описание
struct mbuf *sbcut_locked(struct sockbuf *sb, int len)	Удаляет указанное число байт из начала буфера и возвращает их в виде mbuf
void sbcompress(struct sockbuf *sb, struct mbuf *m, struct mbuf *n)	Добавляет цепочку m после цепочки n в буфере. Если n равно NULL, считается, что буфер пуст
void sbappend(struct sockbuf *sb, struct mbuf *m, int flags)	Добавляет цепочку m в конец буфера

3.8 Подсистемы обработки входящих и исходящих сегментов

Разбор и обработка входящих сегментов выполняется в файле tcp_input.c одноименной функцией. Основной задачей является разбор заголовка TCP и соответствующее ему изменение состояния конечного автомата TCP и данных в управляющем блоке, а также отправка при необходимости новых сегментов.

Патчем [32] для сокета были добавлены три флага TF2_WAKESOL, TF2_WAKESOR, TF2_WAKESOW, которые соответственно означают, что стала доступна неблокирующая операция открытия соединения на слушающем сокете, чтение данных сокета или записи данных в сокет. Установка этих флагов происходит в процессе обработки ACK-сегмента в функции tcp_do_segment(). В конце обработки сегмента вызывается новая функция tcp_handle_wakeup(), которая в зависимости от установленного флага выполняет нужный обратный вызов (upcall).

В файле tcp_output.c функцией tcp_output() выполняется создание нового сегмента и отправка данных. Эта функция вызывается косвенно через таймеры или операции TCP. Она формирует заголовок сегмента и добавляет в него данные.

Данные для отправки находятся в специальном буфере struct sockbuf. При вызове tcp_output() отправляется некоторый участок памяти из этого буфера, причем оригинальный буфер должен в итоге остаться без изменений, так как может потребоваться отправить данные несколько раз (например, если возникнут потери). Обычно в таких случаях делают копирование, что не

очень эффективно. Поэтому в библиотеке используются indirect mbuf: для отправляемого участка данных аллоцируется цепочка таких mbuf, которые ссылаются на данные без владения. Кроме того, к этой цепочки в начало добавляется обычный mbuf небольшого размера — под память для будущего заголовка сегмента. Была отдельно написана специальная функция pktmbuf_clone_span(), которая выделяет необходимое количество indirect mbuf, ссылающихся на указанный участок оригинальной цепочки mbuf (текст функции приведен в приложении).

3.9 Подсистемы оптимизации использования памяти в состояниях SYN_RCVD и TIME_WAIT

Как уже было сказано, одно TCP-соединение занимает достаточно много памяти. Это создает возможность атаки типа отказ от обслуживания на сервер, которая получила название SYN-флуд атака. Злоумышленник может без труда (не занимая у себя память) генерировать большое число SYN-сегментов, которые при получении на сервере приводят к необходимости выделения на нем значительного объема памяти под эти псевдосоединения. В итоге, злоумышленник легко добивается отказа в работе сервера. Подсистема syncache призвана сильно сократить (до десятков байтов) размер памяти, необходимой при совершении рукопожатия.

Для конкретно данной реализации TCP эта атака не имеет смысла (окружение абсолютно контролируется), но syncache слишком сильно интегрирован внутрь реализации в FreeBSD, поэтому в библиотеки также оставлен. А вот другой способ борьбы с этой атакой — известный как syncookies — убран.

Аналогично используется подсистема tcptw, которая резко уменьшает использование памяти для сокета в состоянии TIME_WAIT. В библиотеки эта подсистема отсутствует за ненадобностью, т. к. MSL сокращено.

3.10 Интерфейс библиотеки на C и C++

Взаимодействие пользователя с библиотекой происходит через интерфейс, представленный файлом bsd_tcp.h. Интересной особенностью является сокрытие внутреннего устройства библиотеки — bsd_tcp.h не включает (#include) никакой заголовочный файл, связанный с реализацией TCP. Это необходимо для того, чтобы не загрязнять пространство имен у пользователя лишними символами, создать чистый и четкий интерфейс, повысить скорость компиляции. Поэтому пользователь не работает напрямую с struct socket. Ему предоставляется структура struct socketd (от socket descriptor), которая хранит в себе единственное поле — указатель типа void* на struct socket. Подобный метод представлен в Win32 API в виде дескриптора HANDLE, указывающего на системный ресурс.

Структуры, псевдонимы типов (typedef) и функции, доступные пользователю, показаны в таблице 3.5.

Таблица 3.5 – Описание интерфейса библиотеки

Тип элемента	Название	Описание
Структура	bsd_socketd	Дескриптор сокета
Структура	bsd_instance_info	Информация о системе TCP (число сокетов, периоды таймеров и т.д.)
Структуры	bsd_tcp_hdr_info	Информация о сегменте (тип, размер и т.д.)
Псевдоним	bsd_tcp_ip_out_cb_fn	Указатель на функцию, принимающую сегменты для дальнейшей отправки
Псевдоним	bsd_tcp_connect_cb_fn	Указатель на функцию обратного вызова для функции bsd_tcp_async_connect()
Псевдоним	bsd_tcp_listen_cb_fn	Указатель на функцию обратного вызова для функции bsd_tcp_async_listen()
Псевдоним	bsd_tcp_write_cb_fn	Указатель на функцию обратного вызова для функции bsd_tcp_async_write()
Псевдоним	bsd_tcp_read_cb_fn	Указатель на функцию обратного вызова для функции bsd_tcp_async_read()
Псевдоним	bsd_tcp_close_cb_fn	Указатель на функцию обратного вызова для функции bsd_tcp_async_close()
Псевдоним	bsd_tcp_shutdown_cn_fn	Указатель на функцию обратного вызова для функции bsd_tcp_async_shutdown()
Функция	bsd_tcp_init_lcore	Инициализирует протокол на текущем ядре
Функция	bsd_tcp_set_mempool_lcore	Устанавливает пул памяти для текущего ядра

Продолжение таблицы 3.5

Тип элемента	Название	Описание
Функция	bsd_tcp_get_mempool_lcore	Возвращает пул памяти текущего ядра
Функция	bsd_tcp_set_ip_output_func	Устанавливает функцию дальнейшей обработки сегментов на отправку
Функция	bsd_tcp_fasttick	Увеличивает время таймеров на указанное число тиков. Выполняет обработку таймеров
Функция	bsd_tcp_input	Передает сегмент на обработку в TCP
Функция	bsd_tcp_socket	Создает новый сокет
Функция	bsd_tcp_bind	Привязывает сокет к локальному адресу
Функция	bsd_tcp_connect	Подключает сокет к указанному слушающему сокету по адресу
Функция	bsd_tcp_listen	Переводит сокет в состояние слушающего
Функция	bsd_tcp_accept	Принимает запрос на соединение
Функция	bsd_tcp_write	Передает данные другой стороне соединения
Функция	bsd_tcp_read	Принимает данные от другой стороны соединения
Функция	bsd_tcp_close	Закрывает соединение и очищает память сокета
Функция	bsd_tcp_shutdown	Закрывает соединение
Функция	bsd_tcp_ async_connect	Асинхронная версия bsd_tcp_connect
Функция	bsd_tcp_async_listen	Асинхронная версия bsd_tcp_listen
Функция	bsd_tcp_ async_write	Асинхронная версия bsd_tcp_write
Функция	bsd_tcp_ async_read	Асинхронная версия bsd_tcp_read
Функция	bsd_tcp_ async_close	Асинхронная версия bsd_tcp_close
Функция	bsd_tcp_ async_shutdown	Асинхронная версия bsd_tcp_shutdown
Функция	bsd_tcp_getsockname	Возвращает пару (порт, локальный адрес)
Функция	bsd_tcp_get_instance_info	Возвращает информацию о состоянии TCP на текущем ядре
Функция	bsd_tcp_get_tcp_packet_info	Выполняет разбор mbuf и возвращает информацию о сегменте в нем

Для всех этих функций и структур данных созданы обертки на языке C++, обеспечивающие удобный объектно-ориентированных дизайн. По сути, для каждой структуры создан соответствующий класс, унаследованный от нее. Для каждой функции, принимающей эту структуру, создан метод в классе, который вызывает функцию и передает в нее указатель this вместо указателя на структуру. Для указателей на функции создан специальный класс CallbackBuilder, который позволяет использовать методы произвольного объекта в качестве функции обратного вызова. Исходный код этого класса находится в приложении В. Пример его использования ниже:

```
// Функция, которая принимает функцию обратного вызова (f),
// контекст (arg) и параметр (p).
int
func_with_callback(void* arg, void (f)(void *arg), int p));

class Foo
{
    // Обработчик обратного вызова.
    void Cb(int param);

    // Функция, которая вызывает func_with_callback и
    // передает в нее Cb() как функцию обратного вызова.
    int FuncWithCallback(int param)
    {
        CallbackBuilder<Foo, &Foo::Cb> cbk(this);
        return func_with_callback(cbk.CreateArg(),
                    cbk.CreateFunc(), param);
    }
};
```

3.11 Модульное тестирование библиотеки

В соответствии с принятыми нормами разработки качественного программного обеспечения [2] для библиотеки были созданы модульные тесты, автоматически проверяющие работу протокола для некоторых ситуаций (например, выполнение рукопожатия, записи и чтения данных). Хотя разработка и велась по принципам разработки через тестирование, когда сначала создается тест, а потом только код, приводящий к успешному выполнению этого теста, но покрыть весь код на 100% чрезвычайно сложно (большая его часть уже написана) и цели такой не стояло. Тестами покрыты основные сценарии.

Для разработки использовался фреймворк Google Test Framework. В тестах имитировалась работа сервера и клиента, как двух сокетов, подключенных друг к другу. Для этого данные, исходящие из библиотеки (через указатель на функцию в net_stack::ip_output), тут же подавались библиотеки на вход (через функцию bsd_tcp_input()). В тесте же анализировались флаги сегментов, состояние сокетов, их количество в таблице сокетов. Существу-

ющие тесты показаны в таблице 3.6. Кроме того, все тесты еще и выводят на экран краткое описание всех переданных сегментов для простоты отладки.

Таблица 3.6 – Список модульных тестов

Название теста	Описание
GetName	Проверяет, что сокет корректно возвращает адрес и порт, к которому он привязан (через sobind())
Connect_SendFirstHandshakePacket	Проверяет сегмент, первый отправленный при открытии соединения (с обеих сторон)
RexmitSyn	Проверяет, что SYN-сегмент пересылается, если ответ не пришел в установленное время
ListenConnect_FullHandshake	Проверяет корректность выполнения рукопожатия
Connect_ResetOnUnknownIncomingConnection	Проверяет, что при попытке подключения к несуществующему сокету в ответ отправляется RST-сегмент
Shutdown_ConnectionGreacefulTermination	Проверяет корректность завершения соединения
AsyncAccept	Проверяет асинхронное создание соединения со стороны сервера
AsyncConnect	Проверяет асинхронное подключение к серверу
WriteRead	Проверяет корректность записи и чтения данных в сокет
WriteRead_CheckMbufAllocation	Проверяет, что во время чтения и записи не происходят лишние копирования данных и отсутствуют утечки mbuf
MultipleWriteSameMbuf	Проверяет, что корректно выполняется запись одного и того же mbuf (zero copy ничего не ломает)
AsyncWriteAsyncRead	Аналогично WriteRead, но делает все операции асинхронно

На рисунке 3.4 показан результат исполнения тестов.

3.12 Функциональное и нефункциональное тестирование

Кроме модульных тестов проводились также функциональные и нефункциональные тесты. Первые направлены на проверку корректности работы протокола в соответствии с его стандартом. Это выполнено двумя способами:

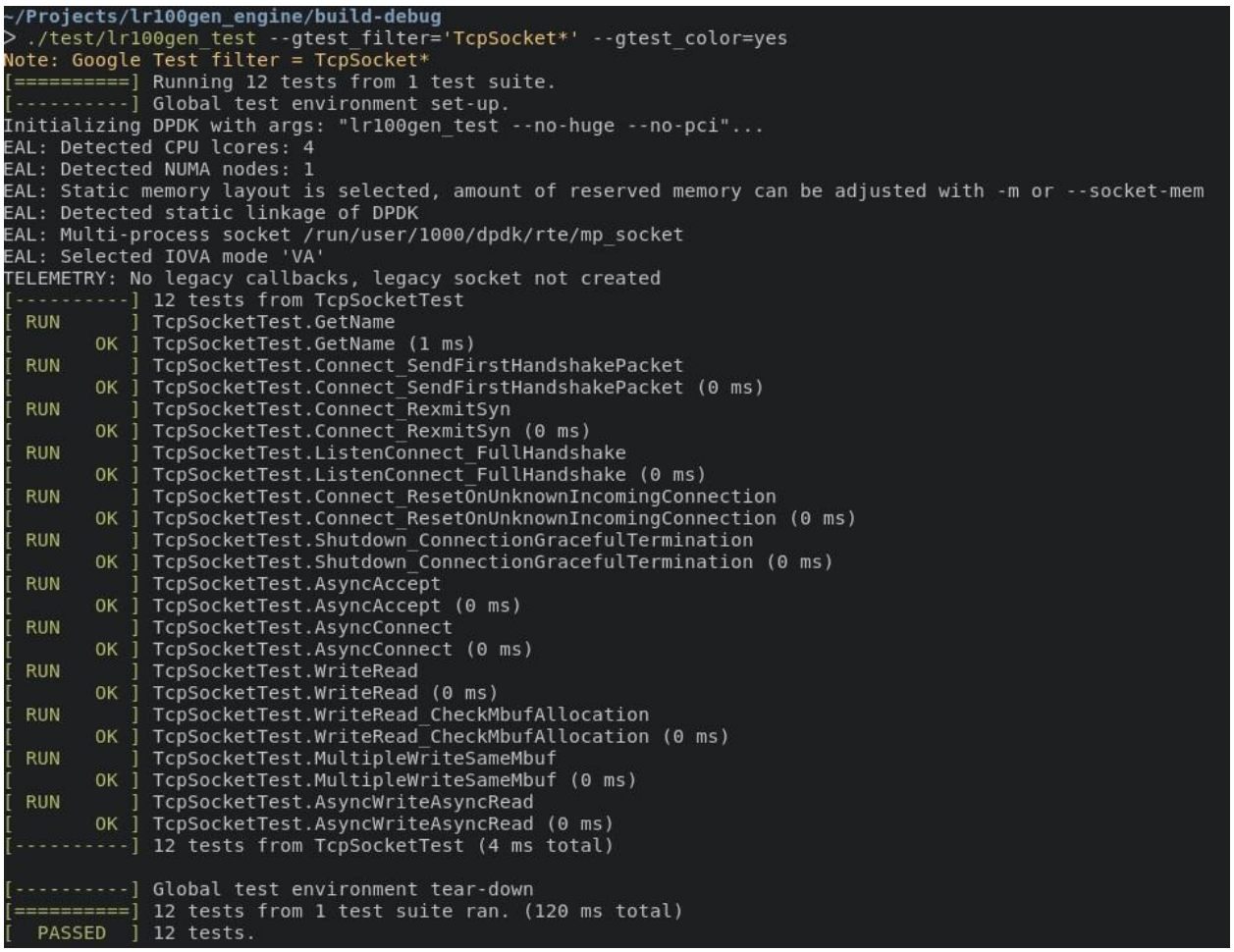


Рисунок 3.4 – Результат выполнения тестов

– подключение утилиты curl 7.74.0 к HTTP-серверу, работающего поверх библиотеки;

– подключение HTTP-клиента, работающего поверх библиотеки, к серверу nginx 1.18.

Программы curl и nginx работают на сетевом стеке ОС Debian Linux 11 и считаются эталонными. Проверяется работоспособность клиента и сервера в паре с этими программами (успешно передается веб-страница в 1 Кбайт клиенту в виде curl, либо от сервера в виде nginx). Также в ручном режиме с помощью утилиты Wireshark проверяются последовательность и содержимое всех пакетов, участвующих во взаимодействии.

Результаты тестирования на рисунках 3.5 и 3.6. Можно убедиться, что все сегменты соответствуют требованиям [16]. В случае работы с утилитой

curl видны также ARP-запросы (так как изначально MAC-адреса сервера неизвестен), а также пересылка SYN-сегмента по этой же причине.

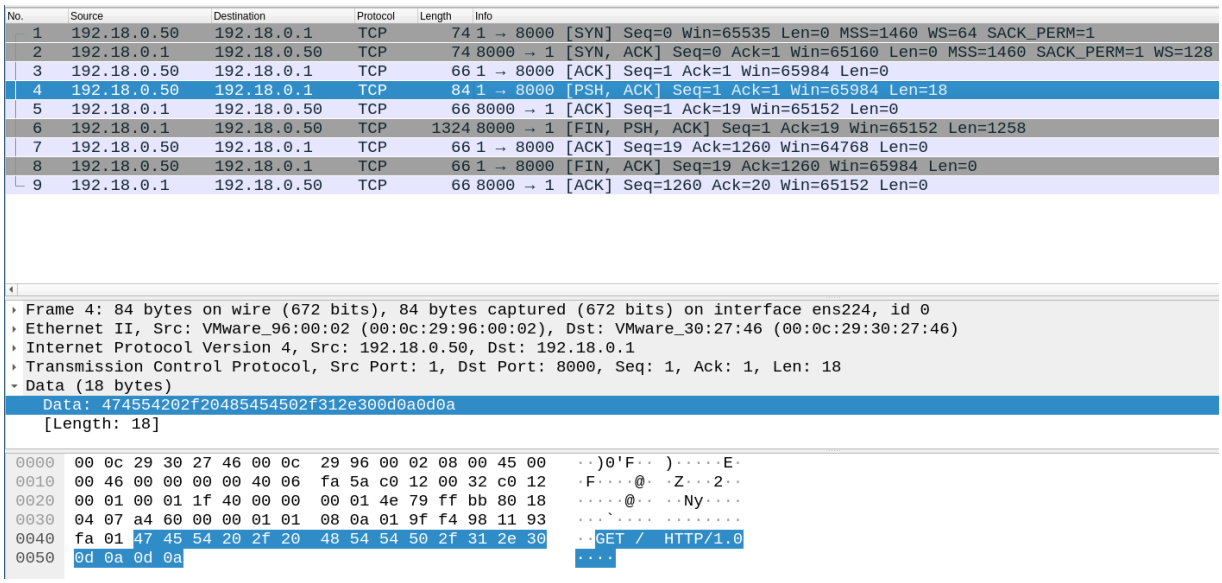


Рисунок 3.5 – Выполнение GET-запроса (HTTP 1.0) к серверу nginx

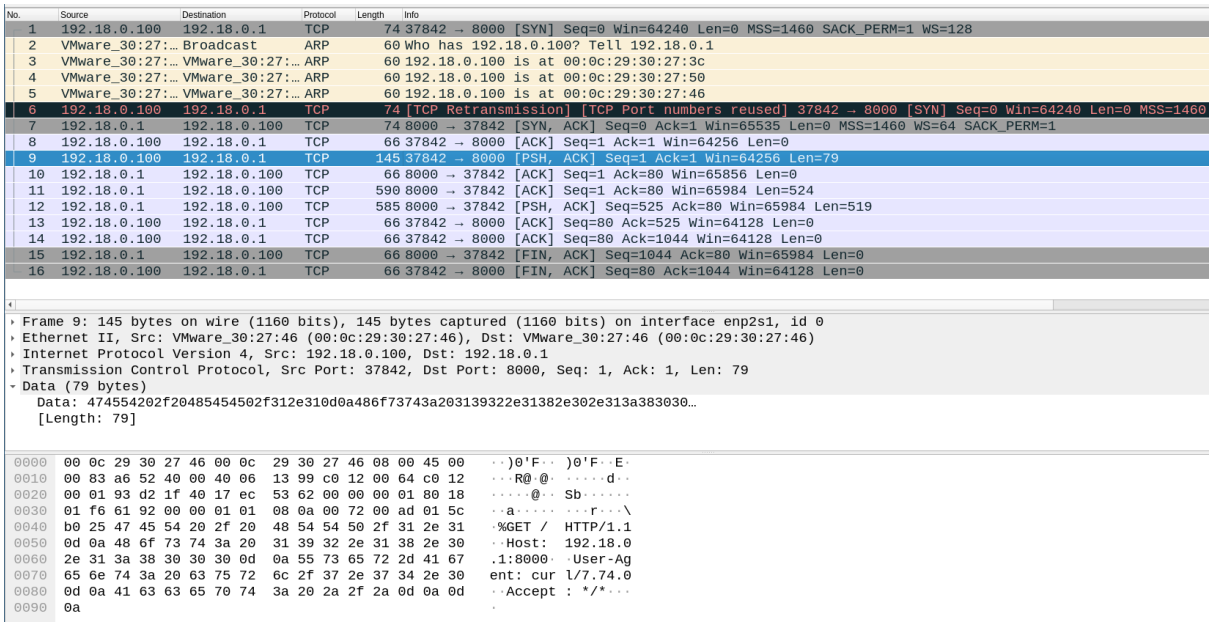


Рисунок 3.6 – Выполнение GET-запроса (HTTP 1.1) curl к HTTP-серверу

Второй вид тестирования – нефункциональные тесты – необходим для замеров производительности получившегося решения. Для этого на машине запускался HTTP клиент и сервер, которые через отдельные сетевые 10-гигабитные интерфейсы, включенные напрямую в петлю, общались друг с

другом (только запрос–ответ без лишних данных в теле). Таблица 3.7 содержит характеристики компьютера, на котором производилось тестирование. На рисунке 3.7 показан график зависимости количества соединений в секунду от числа используемых ядер (не учитывая управляющее ядро) клиентом (у сервера столько же).

Таблица 3.7 – Характеристики компьютера, на котором производилось тестирование

Характеристика	Значение
Название платформы	HW2000Q4
Операционная система	Debian GNU/Linux 11 (ядро 5.10.0)
Процессор	2 x Intel Xeon E5-2609v3 (12 ядер)
Сетевой интерфейс	2 x Intel 82599ES 10G SFP+

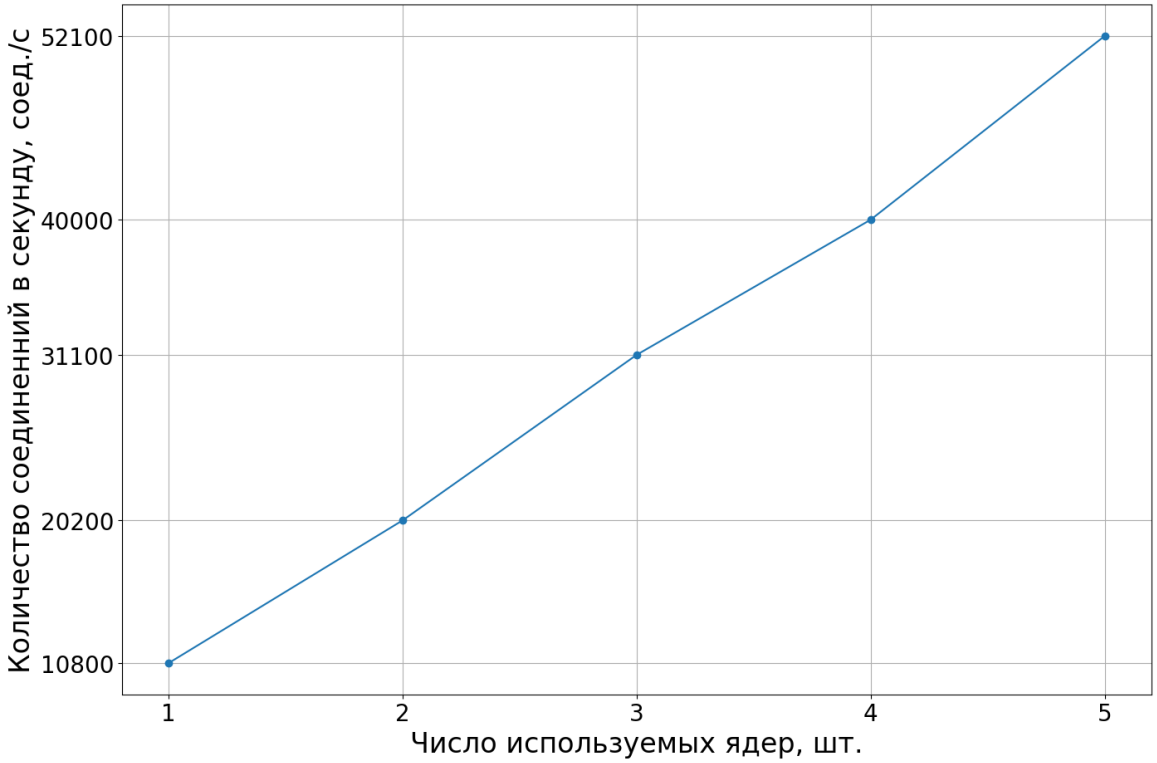


Рисунок 3.7 – График производительности реализации TCP в зависимости от числа используемых ядер

4 ОБЕСПЕЧЕНИЕ КАЧЕСТВА РАЗРАБОТКИ, ПРОДУКЦИИ ПРОГРАММНОГО, ПРОДУКТА

Это глава посвящена качеству разработанной библиотеки. В ней определяются лица-потребители разработки, способы выявления у них требований и производится строгое формулирование этих требований.

4.1 Лица или группы лиц, являющиеся потребителями разработки

Разрабатываемый компонент предназначен для использования внутри проекта, создаваемого в первую очередь для внутренних нужд компании – тестирования собственных продуктов. Таким образом, прямыми потребителями будут команды этих продуктов, а значит для выявления требований необходимо взаимодействие с владельцами и тестировщиками продуктов. В то же время, создается компонент для существующего проекта, над которым также трудятся другие люди. Поэтому обязательно какие-то требования исходят и от них. Для формулирования этих требований достаточно общения с системным аналитиком и программистами команды.

4.2 Примеры методов выявления требований

Существует довольно много способов получения требований от конкретных потребителей: интервью, устный опрос, анкетирование, исследование фокус-групп, ролевые игры, мозговой штурм и другие.

Главные области применения разрабатываемого продукта, как говорилось выше, известны – это внутренние продукты компании. Это дает возможность ограничить круг потребителей до определенных лиц, что в свою очередь позволяет их всех опросить и согласовать их требования. Соответственно для таких целей лучше всего подойдут интервьюирование, опрос и анкетирование; остальные перечисленные методы более трудоемки, их стоит применять для случаев, когда отсутствует непосредственная связь с потреби-

телей, либо этих лиц слишком много (неизвестный рынок, необходимость изобретения новшества).

С другой стороны, предметная область достаточно широка и сложна, и тогда анкетирование может не позволить изучить желания потребителей наиболее глубоко. А вот комбинация интервью, проводимого с заготовленными вопросами, и устного опроса, более свободного, дадут все возможности для получения требований, которые предельно близко отражают действительность.

4.3 Формулирование требований потребителей

В итоге, проведя интервью и устный опрос с владельцами тестируемых продуктов и их тестировщиками, а также с системным аналитиком и программистами, работающими над проектом, компонент которого разрабатывается в данной работе, были выявлены требования, которые можно разделить на группы в соответствии с ГОСТ 25010: корректность, производительность, тестируемость и совместимость. Операционные определения указанных требований представлены в таблице 4.1.

Таблица 4.1 – Операционные определения требований

Требование (критерий)		Измерение и/или испытание (тест)			Анализ (решение)	
Название критерия	Формулирование требования (критерия)	Характеристики качества	Ед. изм.	Процедура измерения и/или испытания характеристики	Целевое значение	Процедура анализа и принятия решения о соответствии
Корректность	Реализация протокола не должна отличаться от положений RFC 793	Число положений, в которых реализация не соответствует RFC 793	шт.	Ручной анализ текста RFC и исходного кода реализации	0	Если после анализа получается, что реализация не отличается от RFC 793 – критерий выполнен

Продолжение таблицы 4.1

	Утилита curl должна получать всю страницу от сервера на 100% корректно	Процентное соотношение числа корректно загруженных байт к общему размеру тестовой страницы	%	1. Тестовый TCP сервер предоставляет страницу размером 1 КБ по протоколу HTTP 1.0 2. С помощью утилиты curl производится загрузка этой страницы 3. Подсчитывается число байт, равных байтам в странице	100	Если полученная curl страница соответствует исходной побайтно на 100%, то критерий выполнен
Корректность	Клиент должен получать всю страницу от nginx на 100% корректно	Процентное соотношение числа корректно загруженных байт к общему размеру тестовой страницы	%	1. Nginx сервер предоставляет HTML-страницу размером 10 КБ по протоколу HTTP 1.0 2. Тестовый TCP клиент общается к nginx серверу и загружает страницу 3. Подсчитывается число байт, равных байтам в исходной странице	100	Если полученная клиентом страница соответствует исходной побайтно на 100%, то критерий выполнен
Тестируемость	Тесты, имеющие зависимости от внешнего мира, дожны отсутствовать	Число тестов, имеющих зависимость от внешнего мира (сети, файловой системы, глобальных переменных и т.д.)	шт.	1. Разрабатывается дизайн системы тестирования реализации протокола 2. Выполняется написание тестов для основных функций TCP	0	Если присутствует хотя бы один тест, зависимый от внешнего мира, то критерий не выполнен

Продолжение таблица 4.1

	Реализованные операции TCP дожны быть покрыты тестами более, чем на 90%	Отношение числа операций TCP, покрытых тестами, к общему числу операций	%	Покрыты должны быть функции создания и закрытия соединения, отправки и приема данных, выполнения таймеров. Должны быть протестирован синхронные и асинхронные варианты функций	> 90	Если покрытие тестами выше 90%, то критерий выполнен
Производительность	Среднее значение количества соединений в секунду (CPS) должно быть не ниже 50.000	Среднее за тест количество соединений в секунду	шт / с	Две ЭВМ соединяются физически порт в порт и на них соответственно запускаются тестовые TCP сервер и TCP клиент. Клиент постоянно в течение 10 минут выполняет обращение к серверу для получения тестовых данных в размере 1 байта. Подсчитывается среднее число удачных соединений в секунду.	≥ 50К	Если полученное значение CPS при выполнении 3 одинаковых тестов выше или равно 50.0000, то критерий выполнен

Продолжение таблицы 4.1

	Максимальное число параллельных соединений (MAC CC) должно быть не менее 100.000.000	Максимальное за тест количество параллельных соединений	шт.	Две ЭВМ соединяются физически порт в порт и на них соответственно запускаются тестовые TCP сервер и TCP клиент. Клиент постоянно в течение 10 минут выполняет обращение к серверу на создание соединения. Закрытие соединения не выполняется. Подсчитывается число открытых соединений.	≥ 100 M	Если полученное значение MAX CC при выполнении 3 одинаковых тестов выше или равно 100.000.000, то критерий выполнен
Совместимость	Число добавленных и/или измененных пользовательских функций и классов должно быть менее или равно 10	Число добавленных или измененных функций и классов во внешнем коде	шт.	Во время интеграции проекта постоянно проводится контроль изменений внешнего кода — подсчет измененных и добавленных функций и/или классов	≤ 10	Количество добавленных или измененных классов во внешнем коде меньше или равно 10, тогда критерий выполнен
	Программный интерфейс реализации должен соответствовать интерфейсу сокетов Unix на минимум 90%	Процентное соотношение числа публичных функций, семантически соответствующих таковым из интерфейса сокетов семейства операционных систем Unix.	%	После создания дизайна публичной части реализации протокола, выполняется его сравнение с существующим слоем сокетов в ОС FreeBSD и Linux	≥ 90	Критерий выполнен, если соотношение больше или равно целевому значению

Одной из причин использования в основе разработки исходного кода FreeBSD были как раз требования по корректности. Как обсуждалось в главе 3, в алгоритмы работы основных функций реализации, контролирующих поведение TCP, никакие изменения не вносились. Кроме изменения длительности 2MSL и, соответственно, длительности таймера TIME_WAIT. Но, как и было замечено, такое изменение не противоречит RFC 793 [16]. По этим причинам реализация TCP полностью удовлетворяет указанному RFC.

Соответствие требованию корректности по другим пунктам подтверждается выполненными в конце главы 3 тестами, где утилита curl и сервер nginx использовались в связке с HTTP сервером и клиентом, построенными поверх созданной реализации TCP.

Результаты нефункционального тестирования из главы 3 (таблица 3.7) показывают, что система отвечает требованию по метрике CPS. Для обеспечения требования по метрике MAX CC в реализации предусмотрена возможность добавления функции разрыва соединения без корректного его завершения и переиспользования памяти сокета в дальнейшем. Это позволит сильно оптимизировать память, требуемую для генерации 100.000.000 соединений (в обычной реализации нужно будет 114 ГБайт оперативной памяти).

Разработка не имеет тестов с внешними зависимостями (можно тестировать TCP в отрыве от остальной программы), и, как видно по таблице 3.6, тестами покрыты все функции сокетов. Это удовлетворяет требованию тестируемости.

Соответствие требованию совместимости проверялось параллельно разработке и оказалось подтвержденным.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы была создана библиотека, содержащая реализацию протокола TCP на основе кода сетевого стека операционной системы FreeBSD, способная функционировать в пространстве пользователя. Библиотека имеет интерфейс на C и C++, приближенный к оригинальному интерфейсу сокетов. Этому предшествовал анализ существующих реализаций, который показал, что они имеют существенные ограничения.

Тот факт, что в основе разработки лежит FreeBSD, позволяет говорить о высокой надежности и соответствии кода стандарту TCP. Выполненное модульное и функциональное тестирование также свидетельствуют об этом. Тест производительности показал достаточную по требованиям производительность (10.000 соединений в секунде на логическое ядро процессора).

В дальнейшем библиотека была интегрирована внутрь продукта компании АО «ИнфоТеКС» и позволила расшить его функционал поддержкой нагрузочного тестирования такими протоколами, как HTTP и HTTPS.

Но на самом деле, работа над протоколом не завершена. Существует большое количество дополнений к TCP, разработанных в поздние годы и повышающих его производительность и безопасность. Например, алгоритмы управления перегрузки (congestion control), случайная генерация начального номера сегментов (random ISN), логирование, поддержка IPv6 и многие другие. Немаловажно произвести профилирование и устранение узких по производительности мест в реализации. Именно в этих направлениях может вестись работа над улучшениями.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. ГОСТ Р ИСО/МЭК 12207-2010. Процессы жизненного цикла программных средств. – М.: Стандартинформ, 2011. – 105 с.

2. ГОСТ Р 56920-2016. Тестирование программного обеспечения. Часть 1. Понятия и определения. – М.: Стандартинформ, 2016. – 53 с.

3. ГОСТ Р 8.654-2015. Требования к программному обеспечению средств измерений. Основные положения. – М.: Стандартинформ. 2015. – 12 с.

4. Об утверждении Требований к организационно-техническому обеспечению устойчивого функционирования сети связи общего пользования [Текст]: Приказ Министерства информационных технологий и связи РФ от 27 сентября 2007 г. № 113 // Официальный интернет-портал правовой информации. – http://pravo.gov.ru/proxy/ips/?docbody=&link_id=0&nd=102118732&intelsearch=&firstDoc=1

5. Stevens, W. R. TCP/IP Illustrated, Volume 1: The Protocols. – США: Addison-Wesley Professional, 1994. – 576 с.

6. Stevens, W. R. TCP/IP Illustrated, Volume 2: The Implementation / W. R. Stevens, G. R. Wright. – США: Addison-Wesley Professional, 1995. – 1174 с.

7. Corbet, J. Improving Linux networking performance [Электронный ресурс] // Linux Conf Au. – 2015. – https://lwn.net/Articles/629155/

8. Компания Huawei. What Is a Microburst? How to Detect a Microburst? [Электронный ресурс] – https://support.huawei.com/enterprise/ru/doc/EDOC1100086962.

9. MoonGen: A Scriptable High-Speed Packet Generator / P. Emmerich, S. Gallenmüller, D. Raumer [и др.] // Proceedings of the 2015 Internet Measurement Conference. – Нью-Йорк, 2015. – С. 275–287.

10. Rizzo, L. Netmap: A Novel Framework for Fast Packet I/O // USENIX ATC 12. – Бостон, 2012. – С. 101–112.

11. Компания Cisco. Руководство по TRex [Электронный ресурс]. – https://trex-tgn.cisco.com/trex/doc/trex_book.pdf

12. Linux Foundation. Data Plane Development Kit [Электронный ресурс]. – https://www.dpdk.org/.

13. Enberg, P. On Kernel-Bypass Networking and Programmable Packet Processing [Электронный ресурс]. – https://medium.com/@penberg/on-kernel-bypass-networking-and-programmable-packet-processing-799609b06898

14. Botta A. Do You Trust Your Software-Based Traffic Generator? / A. Botta, A. Dainotti, A. Pescapé // IEEE Communications Magazine. – Нью-Йорк: IEEE, 2010. – Том 48, вып. 9. – С. 158–165.

15. Таненбаум Э. Компьютерные сети. 5-е изд. / Э. Таненбаум, Д. Уэзеролл. — СПб.: Питер, 2012. — 960 с.

16. RFC 793. Transmission Control Protocol [Электронный ресурс]. – https://datatracker.ietf.org/doc/html/rfc793.

17. RFC 791. Internet Protocol [Электронный ресурс]. – https://datatracker.ietf.org/doc/html/rfc791.

18. RFC 9000. QUIC: A UDP-Based Multiplexed and Secure Transport [Электронный ресурс]. – https://datatracker.ietf.org/doc/html/rfc9000.

19. Kozierok, C. M. The TCP/IP Guide. – Сан-Франциско: No Starch Press, 2005. – 1616 с.

20. RFC 6093. On the Implementation of the TCP Urgent Mechanism [Электронный ресурс]. – https://datatracker.ietf.org/doc/html/rfc6093.

21. Jeong, E. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems / E. Jeong, S. Wood and M. Jamshed [и др.] // USENIX NSDI 14. – Seattle, 2014. – С. 489–502.

22. Компания Tencent. Официальный сайт проекта F-Stack [Электронный ресурс]. – http://www.f-stack.org/.

23. Компания Ansyun. Репозиторий проекта ANS [Электронный ресурс]. – https://github.com/ansyun/dpdk-ans/.

24. Компания Cisco. What is VPP? [Электронный ресурс]. – https://wiki.fd.io/view/VPP/What_is_VPP%3F.

25. Фонд OpenFastPath. Технический обзор проекта OpenFastPath [Электронный ресурс]. – https://openfastpath.org/index.php/services/technical-overview.

26. Проект FreeBSD. Официальный Git-репозиторий. Ветка releng/12.1 [Электронный ресурс]. – https://cgit.freebsd.org/src/log/?h=releng/12.1

27. Проект FreeBSD. Руководство FreeBSD [Электронный ресурс]. – https://docs.freebsd.org/doc/12.1-RELEASE/usr/local/share/doc/freebsd/ru/books/handbook.

28. Столмен, Р. Проблема лицензии BSD [Электронный ресурс]. – https://www.gnu.org/licenses/bsd.ru.html.

29. Linux Foundation. Руководство программиста DPDK 21.11.1 [Электронный ресурс]. – https://doc.dpdk.org/guides-21.11/prog_guide/index.html.

30. Компания ScyllaDB. Документация к проекту Seastart. Shared-nothing Design [Электронный ресурс]. – http://seastar.io/shared-nothing.

31. Drepper, U. What every programmer should know about memory? Part 5: What programmers can do [Электронный ресурс]. – https://lwn.net/Articles/255364.

32. Проект FreeBSD. Патч r367492 [Электронный ресурс]. – https://reviews.freebsd.org/D29690.

33. Ahern, W. Библиотека timeout.c [Электронный ресурс]. -- https://25thandclement.com/~william/projects/timeout.c.html

34. Varghese, G. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility / G. Varghese, A. Lauck // IEEE/ACM Transaction on Networking – Нью-Йорк: IEEE, 1997. – Том 5, вып. 6. – С. 824-834.

35. Компания IBM. Benefits of Huge Pages [Электронный ресурс]. – https://www.ibm.com/support/pages/benefits-huge-pages.

ПРИЛОЖЕНИЕ А

Фрагменты исходного кода

Файл utils/callback.hpp

```
#ifndef LR100GEN_ENGINE_UTILS_CALLBACK_HPP
#define LR100GEN_ENGINE_UTILS_CALLBACK_HPP

#include <system_error>

namespace lr100gen::utils
{

// Позволяет удобно хранить member function pointer в виде объекта с
// возможностью получения пары (указатель на функцию, объект-контекст).
//
// Нужен для передачи member function pointer в C API, ожидающее указатель на
// функцию.
//
// Хранит в себе указатель на функцию-обертку, передающуюся в C API, и
// указатель на объект-контекст, который передается в обертку как первый
// аргумент.
template< typename >
struct Callback;

// Специализация шаблона только для методов.
template< typename R, typename... TArgs > struct Callback< R( TArgs... ) >
{
     // Тип функции-обертки, которую можно передавать в C API.
     using TFunc = R( void*, TArgs... );
     // Объект к которому принадлежит исходная функция (передается в обертку
     // первым параметром).
     void* obj = nullptr;
     // Указатель на функцию.
     TFunc* f_p = nullptr;

     // Выполняет вызов функции.
     template< typename... TCallArgs >
     void ExecuteSafe( TCallArgs&&... args )
     {
          if ( f_p != nullptr )
          {
               ( *f_p )( obj, std::forward< TCallArgs >( args )... );
          }
     }

     void Clear()
     {
          obj = nullptr;
          f_p = nullptr;
     }
};

// Deduction guide для создания Callback из указателя на функцию и указателя
// на stateless-лямбду (см. CreateFunc())
template< typename R, typename... TArgs >
```

```
Callback( void*, R( * )( TArgs... ) ) -> Callback< R( TArgs... ) >;

template< typename R, typename... TArgs >
Callback( void*, R( * )( void*, TArgs... ) ) -> Callback< R( TArgs... ) >;

// Самый простой декоратор, выполняющий обычный вызов функции и передающий в
// нее все аргументы без изменений.
//
// Декораторы нужны для преобразования аргументов C API, переданных в обертку,
// в объекты, которые будут передаваться в исходный метод.
struct CallbackPassDecorator
{
     template< typename T, auto F, typename... TArgs >
     void operator()( T& obj, TArgs&&... args ) const
     {
          ( obj.*F )( std::forward< TArgs >( args )... );
     }

     template< typename F, typename... TArgs >
     void operator()( const F& fn, TArgs&&... args ) const
     {
          fn( std::forward< TArgs >( args )... );
     }
};

template< typename, auto... >
class CallbackBuilder;

// Билдер колбэков для member function poitner (методов).
template< typename T, auto F >
class CallbackBuilder< T, F >
{
public:
     explicit CallbackBuilder( T& obj )
          : obj_( &obj )
     { }

     // Возвращает объект-контекст.
     void* CreateArg() { return obj_; }

     // Создает указатель на функцию, которую можно передать в C API вместе с
     // контекстом.
     template< typename TDecorator, typename... TArgs >
     auto CreateFunc()
     {
          return
               // Обетка -- stateless лямда. Контекст -- объект, которому
               // принадлежит исходный метод.
               +[]( void* arg, TArgs... args )
               {
                    auto* obj = static_cast< T* >( arg );
                    TDecorator d; // Выполняет вызов исходного метода.
                    return d.template operator()< T, F >( *obj, args... );
               };
     }

     template< typename TDecorator, typename... TArgs >
     auto CreateCallback()
     {
          return Callback {
               CreateArg(), CreateFunc< TDecorator, TArgs... >()
          };
     }
```

```
private:
     T* obj_;
};

// Билдер колбэков для лямбд.
template<typename F >
class CallbackBuilder< F >
{
public:
     explicit CallbackBuilder( const F& f )
          : f_( f )
     { }

     void* CreateArg()
     {
          // Предполагаем, что `arg` не изменяется.
          return const_cast< void * >( static_cast< const void * >( &f_ ) );
     }

     auto CreateFunc()
     {
          return
               +[]( void *arg )
               {
                    auto* fn = static_cast< F* >( arg );
                    return ( *fn )();
               };
     }

     template< typename TDecorator, typename... TArgs >
     auto CreateFunc()
     {
          return
               +[]( void *arg, TArgs... args )
               {
                    TDecorator d;
                    auto* fn = static_cast< F* >( arg );
                    return d.template operator()< F >(
                          *fn, std::move( args )... );
               };
     }

     template< typename TDecorator, typename... TArgs >
     auto CreateCallback()
     {
          return Callback
          {
               CreateArg(), CreateFunc(),
          };
     }

private:
     const F& f_;
};

} // namespace lr100gen::utils

#endif // !LR100GEN_ENGINE_UTILS_CALLBACK_HPP
```

Файл dpdk/utils_rte_mbuf.h

```
#ifndef BSD_DPDK_UTILS_RTE_MBUF_H
#define BSD_DPDK_UTILS_RTE_MBUF_H

#include <stdint.h>
#include <rte_mbuf.h>

/*
 * Аналог `rte_pktmbuf_clone()`, но клонирует не все сегменты, а только
 * начиная с `off` и так, чтобы общая длина пакета была не больше `len`.
 *
 * См. lib/librte_mbuf/rte_mbuf.c:520 (rte_pktmbuf_clone())
 */
static inline struct rte_mbuf *
pktmbuf_clone_span(struct rte_mbuf *md, struct rte_mempool *mp, uint32_t off,
      uint32_t len)
{
      struct rte_mbuf *mc, *mi, **prev;
      uint32_t pktlen;
      uint16_t nseg;
      uint16_t off_seg;
      uint32_t trim;

      // Определяем rte_mbuf, с которого нужно будет начинать
      off_seg = off / md->data_len;
      off = off % md->data_len;
      while (off_seg--) {
            md = md->next;
      }

      if (unlikely(md == NULL)) {
            return NULL;
      }

    // Создаем первый буфер для цепочки.
      mc = rte_pktmbuf_alloc(mp);
      if (unlikely(mc == NULL))
            return NULL;

      mi = mc;
      prev = &mi->next;
      pktlen = 0;
      nseg = 0;

      // Создаем цепочку indirect-mbuf для выделенной выше цепочки.
      do {
            nseg++;
            // Делаем indirect ссылку на следующий буфер.
            rte_pktmbuf_attach(mi, md);
            *prev = mi;
            prev = &mi->next;
            pktlen += md->data_len;
      } while ((md = md->next) != NULL &&
             (pktlen < len) &&
             (mi = rte_pktmbuf_alloc(mp)) != NULL);

      *prev = NULL;
      mc->nb_segs = nseg;
      mc->pkt_len = len;

    // Очищаем всю память, если была ошибка.
```

```
	if (unlikely(mi == NULL)) {
		rte_pktmbuf_free(mc);
		return NULL;
	}

	// Задаем нужные смещения внутри первого и последнего indirect-mbuf.
	trim = pktlen - len;
	if (off > 0)
		rte_pktmbuf_adj(mc, off);
	if (trim > 0)
		rte_pktmbuf_trim(mi, trim);

	__rte_mbuf_sanity_check(mc, 1);
	return mc;
}

#endif // !BSD_DPDK_UTILS_RTE_MBUF_H
```

Файл bsd_tcp.c

```
#include <bsd_tcp.h>

#include <rte_log.h>

#include <bsd_glue.h>
#include <bsd_tcp_net_stack.h>

#include <bsd_sys/param.h>
#include <bsd_sys/types.h>
#include <bsd_sys/socket.h>
#include <bsd_sys/socketvar.h>

#include <bsd_netinet/in.h>
#include <bsd_netinet/in_pcb.h>
#include <bsd_netinet/ip.h>
#include <bsd_netinet/tcp_var.h>
#include <bsd_netinet/tcp_timer.h>

#include <bsd_tcp_net_stack.h>

#define RTE_LOGTYPE_APP RTE_LOGTYPE_USER1

#ifdef __cplusplus
#error "FreeBSD TCP should be built with C compiler"
#endif


// Макросы для быстрого доступа к управляющим
// структурам из struct socketd
#define SODTOSO(sod)    ((struct socket*)(sod->mem))
#define SODTOINPCB(sod) (sotoinpcb(SODTOSO((sod))))
#define SODTOTCPCB(sod) (sototcpcb(SODTOSO((sod))))

// Инициализирует TCP на текущем ядре.
void
bsd_tcp_init_lcore()
{
	tcp_init();
}
```

```
// Возвращает размер памяти, используемой под
// одно соединение.
size_t
bsd_tcp_get_sod_mem_size()
{
      return sizeof(struct tcp_mem);
}

// Возвращает размеры основных структур.
// @param[out] so_sz Содержит размеры.
void
bsd_tcp_get_cb_size(struct bsd_tcp_socket_cb_size *so_sz)
{
      so_sz->ns = sizeof(struct net_stack);
      so_sz->so = sizeof(struct socket);
      so_sz->inp = sizeof(struct inpcb);
      so_sz->tp = sizeof(struct tcpcb);
      so_sz->tt = sizeof(struct tcp_timer);
}

// Устанавливает mp как пул для выделения памяти под
// пакеты на текущем ядре.
void
bsd_tcp_set_mempool_lcore(struct rte_mempool *mp)
{
      V_tcbinfo.ns.small_mp = mp;
}

// Возвращает пул памяти текущего ядра.
struct rte_mempool *bsd_tcp_get_mempool_lcore()
{
      return V_tcbinfo.ns.small_mp;
}

// Устанавливает функцию передачи пакетов на слой IP.
void
bsd_tcp_set_ip_output_func_lcore(bsd_tcp_ip_out_cb_fn ip_out, void *arg)
{
      V_tcbinfo.ns.ip_output = ip_out;
      V_tcbinfo.ns.arg = arg;
}

// Создает новый сокет
// @param[out] sod Дескриптор созданного сокета.
int
bsd_tcp_socket(struct bsd_socketd *sod)
{
      struct socket *so;
      int err;
      err = socreate(&so);
      if (err != 0) {
            return err;
      }
      // Дескриптор только содержит указатель на саму
      // структуру сокета.
      sod->mem = so;
      return 0;
}

// Синхронно закрывает сокет.
int
bsd_tcp_close(struct bsd_socketd *sod)
{
      struct socket *so = SODTOSO(sod);
```

```
	return soclose(so);
}

// Асинхронно закрывает сокет
//
// Вызывает функцию fn, когда произойдет завершение соединения.
int
bsd_tcp_async_close(struct bsd_socketd *sod, void *arg, bsd_tcp_close_cb_fn
fn)
{
	struct socket *so = SODTOSO(sod);
	sbasyncopcbk_set(&so->so_snd, arg, fn);
	return bsd_tcp_close(sod);
}

// Сихронно завершает соединение, но не удаляет память под сокетом.
// @param how Тип закрытия: только чтения, только записи или оба.
int
bsd_tcp_shutdown(struct bsd_socketd *sod, int how)
{
	struct socket *so = SODTOSO(sod);
	return soshutdown(so, how);
}

// Асинхронно завершает соединение, но не удаляет память под сокетом.
//
// Вызывает функцию fn, когда произойдет завершение соединения.
// @param how Тип закрытия: только чтения, только записи или оба.
int
bsd_tcp_async_shutdown(struct bsd_socketd *sod, int how, void *arg,
                       bsd_tcp_shutdown_cb_fn fn)
{
     struct socket *so = SODTOSO(sod);
     sbasyncopcbk_set(&so->so_snd, arg, fn);
     return bsd_tcp_shutdown(sod, how);
}

// Увеличивает текущее время на tcks. Обрабатывает истекшие таймеры.
void bsd_tcp_fasttick(uint64_t tcks)
{
	tcp_fasttimo(tcks);
}

// Создает структуру, содержащую адрес сокета.
// @param[out] sin Результат
// @param addr IP-адрес
// @param port Порт
static void
make_sockaddr_in(struct sockaddr_in *sin, uint32_t addr, uint16_t port)
{
	sin->sin_len = sizeof(struct sockaddr_in);
	sin->sin_family = AF_INET;
	sin->sin_addr.s_addr = addr;
	sin->sin_port = port;
}

// Синхронно подключает сокет к удаленному слушающему сокету.
int
bsd_tcp_connect(struct bsd_socketd *sod, uint32_t ip_addr, uint16_t port)
{
	struct socket *so = SODTOSO(sod);
	struct sockaddr_in sin;
	make_sockaddr_in(&sin, ip_addr, port);
	// struct sockaddr обобщенно хранит IPv4 или IPv6 адрес.
```

```
	return tcp_usr_connect(so, (struct sockaddr *)&sin, NULL);
}

// Асинхронно подключает сокет к удаленному слушающему сокету.
//
// Вызывает функию fn, когда подключение выполнилось.
int
bsd_tcp_async_connect(struct bsd_socketd *sod, uint32_t ip_addr,
    uint16_t port, void *arg, bsd_tcp_connect_cb_fn fn)
{
	struct socket *so = SODTOSO(sod);
	sbasyncopcbk_set(&so->so_snd, arg, fn);
	return bsd_tcp_connect(sod, ip_addr, port);
}

// Привязывает сокет к адресу.
int
bsd_tcp_bind(struct bsd_socketd *sod, uint32_t ip_addr, uint16_t port)
{
	struct inpcb *inp = SODTOINPCB(sod);
	struct sockaddr_in sin;
	make_sockaddr_in(&sin, ip_addr, port);
	// TODO: Для красоты можно вызывать через tcp_bind().
	return in_pcbbind(inp, (struct sockaddr *)&sin, NULL);
}

// Переводит сокет в состояние готовности принимать
// новые подключения (слушающий сокет).
int
bsd_tcp_listen(struct bsd_socketd *sod, int backlog)
{
	struct socket *so = SODTOSO(sod);
	return tcp_usr_listen(so, backlog, NULL);
}

// Переводит сокет в состояние готовности принимать
// новые подключения (слушающий сокет).
//
// Вызывает fn, когда появляется новое соединение.
int
bsd_tcp_async_listen(struct bsd_socketd *sod, int backlog, void *arg,
    bsd_tcp_listen_cb_fn fn)
{
	int err;
	struct socket *so = SODTOSO(sod);
	err = bsd_tcp_listen(sod, backlog);
	if (err != 0) {
		return err;
	}
	solasyncopcbk_set(so, arg, fn);
	return 0;
}

// Принимает следующий в очереди запрос на подключение.
// @param headd Слушающий сокет
// @param[out] sod Сокет, отвечающий за локальную сторону нового соединения.
// @param[out] ip_addr IP-адрес удаленного сокета.
// @param[out] port Порт удаленного сокета.
int
bsd_tcp_accept(struct bsd_socketd *headd, struct bsd_socketd *sod, uint32_t
	*ip_addr, uint16_t *port)
{
	int err;
	struct socket *head, *so;
```

```
	struct sockaddr_in sin;

	head = SODTOSO(headd);

	err = soaccept(head, &so, (struct sockaddr *)&sin);
	if (err != 0) {
		return err;
	}

	sod->mem = so;

	if (ip_addr != NULL) {
		*ip_addr = sin.sin_addr.s_addr;
	}
	if (port != NULL) {
		*port = sin.sin_port;
	}
	return 0;
}

// Отправляет буфер m через сокет.
int
bsd_tcp_write(struct bsd_socketd *sod, struct rte_mbuf *m)
{
	struct socket *so = SODTOSO(sod);
	return sosend(so, NULL, m, NULL, 0);
}

// Отправляет буфер m через сокет.
//
// Вызывает fn, когда буфер будет удачно отправлен.
int
bsd_tcp_async_write(struct bsd_socketd *sod, struct rte_mbuf *m, void *arg,
bsd_tcp_write_cb_fn fn)
{
	struct socket *so = SODTOSO(sod);
	sbasyncopcbk_set(&so->so_snd, arg, fn);
	return bsd_tcp_write(sod, m);
}

// Читает уже принятые данные из сокета.
int
bsd_tcp_read(struct bsd_socketd *sod, struct rte_mbuf **m)
{
	struct socket *so = SODTOSO(sod);
	return soreceive(so, NULL, m, NULL, 0);
}

// Асинхронно читает данные из сокета.
//
// Вызывает fn, когда сокет примят данные (получить их можно через
// bsd_tcp_read())
void
bsd_tcp_async_read(struct bsd_socketd *sod, void *arg, bsd_tcp_read_cb_fn fn)
{
	int err, flags;
	struct socket *so = SODTOSO(sod);
	sbasyncopcbk_set(&so->so_rcv, arg, fn);
	// Если сокет в состоянии получать данные и его буфер прием пуст,
	// то ждем новых входящих сегментов. Иначе тут же вызываем колбэк.
	flags = MSG_PEEK;
	err = soreceive(so, NULL, NULL, NULL, &flags);
	if (err == EAGAIN) {
		return;
```

```
	}
	// Если в сокете уже есть принятые данные, то вызывает колбэк.
	sorwakeup(so);
}

// Возвращает адрес и порт сокета.
void bsd_tcp_getsockname(struct bsd_socketd *sod, uint32_t *addr, uint16_t
*port)
{
     struct socket *so = SODTOSO(sod);
     struct sockaddr_in sa;
     struct sockaddr *sap = (struct sockaddr *)&sa;
     in_getsockaddr(so, &sap);
     *addr = sa.sin_addr.s_addr;
     *port = sa.sin_port;
}

// Возвращает частоту таймеров (число тактов в одной секунде).
uint64_t bsd_tcp_gettimohz()
{
	return timeouts_hz(V_tcbinfo.ns.ts);
}

// Передает пакет на обработку TCP.
int
bsd_tcp_input(struct rte_mbuf *m)
{
	int ip_len = sizeof(struct ip);
	int ec = tcp_input(&m, &ip_len, 0);
	return ec == IPPROTO_DONE ? 0 : ec;
}

// Добавляет некоторую информацию (тэг) сокету (нужно для отладки).
void
bsd_tcp_tag(struct bsd_socketd *sod, int tag)
{
	SODTOSO(sod)->so_tag = tag;
}

// Получает информацию о TCP (число активных сокетов, величину MSL,
// длительность Delayed Ack).
void
bsd_tcp_get_instance_info(struct bsd_tcp_instance_info *tii)
{
	tii->socket_count = V_tcbinfo.ipi_count;
	tii->ti.msl = TCPTV_MSL;
	tii->ti.delack = TCPTV_DELACK;
}

// Выполняет разбор сегмента m и сохраняет в ti некоторую информацию о
// сегменте.
void
bsd_tcp_util_get_tcp_packet_info(struct bsd_tcp_hdr_info *ti, struct
                                 rte_mbuf* m )
{
	struct ip *ip = rte_pktmbuf_mtod(m, struct ip *);
	struct tcphdr *th =
	    rte_pktmbuf_mtod_offset(m, struct tcphdr *, sizeof(struct ip));
	struct tcpopt to;
	char *optp;
	int optlen;

	ti->payload_len = m->pkt_len - sizeof( *ip ) - th->th_off * 4;
	if (ti->payload_len < 0) {
```

```
            // Не TCP/IP пакет.
            return;
        }

        ti->sport = ntohs(th->th_sport);
        ti->dport = ntohs(th->th_dport);
        ti->flags = th->th_flags;
        ti->ack = ntohl(th->th_ack);
        ti->seq = ntohl(th->th_seq);
        ti->win = ntohs(th->th_win);

        optp = (char *)(th + 1);
        optlen = th->th_off * 4 - sizeof(struct tcphdr);
        // Разбираем опциональные данные сегмента.
        tcp_dooptions(&to, optp, optlen, TO_SYN);
        ti->mss = to.to_mss;
        ti->wscale = to.to_wscale;
}
```

Файл bsd_kern/uipc_socket.c (фрагмент)

```
// Выделяет память под сокет и все управляющие структуры.
struct socket *
soalloc()
{
    struct tcp_mem *tm;
    struct socket *so;
    struct inpcb *inp;
    struct tcpcb *tp;
    tm = rte_zmalloc("tcp_socket", sizeof(struct tcp_mem), 0);

    // Устанавливает внутренние ссылки между различными структурами.
    so = &tm->so;
    inp = &tm->inp;
    tp = &tm->tp;

    so->so_pcb = inp;
    inp->inp_socket = so;
    inp->inp_ppcb = tp;
    tp->t_inpcb = inp;
    tp->t_timers = &tm->tt;

    // Все API у нас всегда неблокирующее.
    so->so_state |= SS_NBIO;

    return so;
}

// Освобождает память под сокет (частично, сама память сокета удаляет в
// soderef()).
void
sodealloc(struct socket *so)
{
    struct tcp_mem *tm;
    KASSERT(so->so_count == 0, ("sodealloc(): so_count %d", so->so_count));
    KASSERT(so->so_pcb == NULL, ("sodealloc(): so_pcb != NULL"));

    if (!SOLISTENING(so)) {
        if (so->so_rcv.sb_hiwat)
            (void)chgsbsize(&so->so_rcv.sb_hiwat, 0, RLIM_INFINITY);
```

```
        if (so->so_snd.sb_hiwat)
            (void)chgsbsize(&so->so_snd.sb_hiwat, 0, RLIM_INFINITY);
    }

    tm = __containerof(so, struct tcp_mem, so);
    rte_free(tm);
}

// Создает новое соединение для слушающего сокета.
struct socket *
sonewconn(struct socket *head, int connstatus)
{
    struct socket *so;
    u_int over;

    // Проверяет лимит на число соединений.
    over = (head->sol_qlen > 3 * head->sol_qlimit / 2);
    if (over) {
        return (NULL);
    }
    // Создает и наполняет информацией новый сокет, представляющий локальную
    // сторону соединения.
    so = soalloc();
    if (so == NULL) {
        return (NULL);
    }
    so->so_listen = head;
    so->so_linger = head->so_linger;
    so->so_state = head->so_state | SS_NOFDREF;
    if (soreserve(so, head->sol_sbsnd_hiwat, head->sol_sbrcv_hiwat)) {
        sodealloc(so);
        return (NULL);
    }
    if (tcp_usr_attach(so, 0, NULL)) {
        sodealloc(so);
        return (NULL);
    }
    so->so_rcv.sb_lowat = head->sol_sbrcv_lowat;
    so->so_snd.sb_lowat = head->sol_sbsnd_lowat;
    so->so_rcv.sb_flags |= head->sol_sbrcv_flags & SB_AUTOSIZE;
    so->so_snd.sb_flags |= head->sol_sbsnd_flags & SB_AUTOSIZE;

    so->so_state |= connstatus;
    so->so_options = head->so_options & ~SO_ACCEPTCONN;
    soref(head);
    // Обновляет очередь запросов на соединение слушающего сокета.
    if (connstatus) {
        TAILQ_INSERT_TAIL(&head->sol_comp, so, so_list);
        so->so_qstate = SQ_COMP;
        head->sol_qlen++;
        //-solisten_wakeup(head);
    } else {
        while (head->sol_incqlen > head->sol_qlimit) {
            struct socket *sp;

            sp = TAILQ_FIRST(&head->sol_incomp);
            TAILQ_REMOVE(&head->sol_incomp, sp, so_list);
            head->sol_incqlen--;
            SOCK_LOCK(sp);
            sp->so_qstate = SQ_NONE;
            sp->so_listen = NULL;
            sorele(head);
            soabort(sp);
        }
```

```
        TAILQ_INSERT_TAIL(&head->sol_incomp, so, so_list);
        so->so_qstate = SQ_INCOMP;
        head->sol_incqlen++;
    }
    return (so);
}

// Выполняет обратный вызов, показывающий, что есть новый запрос на
// соединение.
void
solisten_wakeup(struct socket *sol)
{
    struct bsd_socketd sod;
    if (sol->sol_listen_cbk.fn != NULL) {
        sod.mem = sol;
        sol->sol_listen_cbk.fn(sol->sol_listen_cbk.arg, &sod);
        // Удаляет указатель на функцию, чтобы не делать один и тот же
        // обратный вызов несколько раз.
        solasyncopcbk_clear(sol);
    }
}

// Инициирует закрытие соединения. Затем очищает память сокета.
int
soclose(struct socket *so)
{
    struct accept_queue lqueue;
    bool listening;
    int error = 0;

    KASSERT(!(so->so_state & SS_NOFDREF), ("soclose: SS_NOFDREF on enter"));

    CURVNET_SET(so->so_vnet);
    //-funsetown(&so->so_sigio);
    if (so->so_state & SS_ISCONNECTED) {
        if ((so->so_state & SS_ISDISCONNECTING) == 0) {
            error = sodisconnect(so);
            if (error) {
                if (error == ENOTCONN)
                    error = 0;
                goto drop;
            }
        }
        //-if (so->so_options & SO_LINGER) {
        //- if ((so->so_state & SS_ISDISCONNECTING) &&
        //-     (so->so_state & SS_NBIO))
        //-     goto drop;
        //- while (so->so_state & SS_ISCONNECTED) {
        //-     error = tsleep(&so->so_timeo,
        //-         PSOCK | PCATCH, "soclos",
        //-         so->so_linger * hz);
        //-     if (error)
        //-         break;
        //- }
        //-}
    }

drop:
    //-if (so->so_proto->pr_usrreqs->pru_close != NULL)
    //- (*so->so_proto->pr_usrreqs->pru_close)(so);
    tcp_usr_close(so);

    SOCK_LOCK(so);
    if ((listening = (so->so_options & SO_ACCEPTCONN))) {
```

```
        // Очищает ресурсы слушающего сокета.
        struct socket *sp;

        TAILQ_INIT(&lqueue);
        TAILQ_SWAP(&lqueue, &so->sol_incomp, socket, so_list);
        TAILQ_CONCAT(&lqueue, &so->sol_comp, so_list);

        so->sol_qlen = so->sol_incqlen = 0;

        TAILQ_FOREACH(sp, &lqueue, so_list) {
            SOCK_LOCK(sp);
            sp->so_qstate = SQ_NONE;
            sp->so_listen = NULL;
            SOCK_UNLOCK(sp);
            /* Guaranteed not to be the last. */
            //-refcount_release(&so->so_count);
            so->so_count--;
        }
    }
    KASSERT((so->so_state & SS_NOFDREF) == 0, ("soclose: NOFDREF"));
    so->so_state |= SS_NOFDREF;
    // Декременитрует число ссылок на сокет.
    sorele(so);
    if (listening) {
        struct socket *sp, *tsp;

        TAILQ_FOREACH_SAFE(sp, &lqueue, so_list, tsp) {
            SOCK_LOCK(sp);
            if (sp->so_count == 0) {
                SOCK_UNLOCK(sp);
                soabort(sp);
            } else
                /* sp is now in sofree() */
                SOCK_UNLOCK(sp);
        }
    }
    CURVNET_RESTORE();
    return (error);
}

// Выполняет разрыв соединения.
int
sodisconnect(struct socket *so)
{
    int error;

    if ((so->so_state & SS_ISCONNECTED) == 0)
        return (ENOTCONN);
    if (so->so_state & SS_ISDISCONNECTING)
        return (EALREADY);
    //-VNET_SO_ASSERT(so);
    //-error = (*so->so_proto->pr_usrreqs->pru_disconnect)(so);
    error = tcp_usr_disconnect(so);
    return (error);
}

int
soreserve(struct socket *so, u_long sndcc, u_long rcvcc)
{
    //-struct thread *td = curthread;

    SOCKBUF_LOCK(&so->so_snd);
    SOCKBUF_LOCK(&so->so_rcv);
    if (sbreserve_locked(&so->so_snd, sndcc, so, /*-td*/ NULL) == 0)
```

```
        goto bad;
    if (sbreserve_locked(&so->so_rcv, rcvcc, so, /*-td*/ NULL) == 0)
        goto bad2;
    if (so->so_rcv.sb_lowat == 0)
        so->so_rcv.sb_lowat = 1;
    if (so->so_snd.sb_lowat == 0)
        so->so_snd.sb_lowat = MCLBYTES;
    if (so->so_snd.sb_lowat > so->so_snd.sb_hiwat)
        so->so_snd.sb_lowat = so->so_snd.sb_hiwat;
    SOCKBUF_UNLOCK(&so->so_rcv);
    SOCKBUF_UNLOCK(&so->so_snd);
    return (0);
bad2:
    sbrelease_locked(&so->so_snd, so);
bad:
    SOCKBUF_UNLOCK(&so->so_rcv);
    SOCKBUF_UNLOCK(&so->so_snd);
    return (ENOBUFS);
}

// Устанавливает указатель на функцию обратного вызова для слушающего сокета.
void
solasyncopcbk_set(struct socket *so, void* arg, async_op_callback_fn fn)
{
    so->sol_listen_cbk.arg = arg;
    so->sol_listen_cbk.fn = fn;
}

// Удаляет указатель на функцию обратного вызова для слушающего сокета.
void
solasyncopcbk_clear(struct socket *so)
{
    so->sol_listen_cbk.arg = NULL;
    so->sol_listen_cbk.fn = NULL;
}

// Отправляет данные через сокет.
int
sosend(struct socket *so, struct sockaddr *addr, /*struct uio *uio,*/
    struct mbuf *top, struct mbuf *control, int flags/*, struct thread *td*/)
{
    int error;

    CURVNET_SET(so->so_vnet);
    if (!SOLISTENING(so))
        error = sosend_generic(so, addr, top, control, flags);
    else {
        // Через слушающий сокет отправлять нельзя.
        m_freem(top); m_freem(control);
        error = ENOTCONN;
    }
    CURVNET_RESTORE();
    return (error);
}

int
sosend_generic(struct socket *so, struct sockaddr *addr, /*struct uio *uio,*/
    struct mbuf *top, struct mbuf *control, int flags/*, struct thread *td*/)
{
    long space;
    ssize_t resid;
    int clen = 0, error, dontroute;
    int atomic = sosendallatonce(so) || top;
```

```
    //-if (uio != NULL)
    //- resid = uio->uio_resid;
    //-else
    //- resid = top->m_pkthdr.len;
    resid = top->pkt_len;
    if (resid < 0 || (/*-so->so_type == SOCK_STREAM*/1 && (flags & MSG_EOR)))
    {
        error = EINVAL;
        goto out;
    }

    dontroute = 1;
        //-(flags & MSG_DONTROUTE) && (so->so_options & SO_DONTROUTE) == 0 &&
        //-(so->so_proto->pr_flags & PR_ATOMIC);
    //-if (td != NULL)
    //- td->td_ru.ru_msgsnd++;
    if (control != NULL)
        clen = control->m_len;

    //-error = sblock(&so->so_snd, SBLOCKWAIT(flags));
    //-if (error)
    //- goto out;

//-restart:
    do {
        SOCKBUF_LOCK(&so->so_snd);
        if (so->so_snd.sb_state & SBS_CANTSENDMORE) {
            SOCKBUF_UNLOCK(&so->so_snd);
            error = EPIPE;
            goto release;
        }
        if (so->so_error) {
            error = so->so_error;
            so->so_error = 0;
            SOCKBUF_UNLOCK(&so->so_snd);
            goto release;
        }
        if ((so->so_state & SS_ISCONNECTED) == 0) {
            /*
             * `sendto' and `sendmsg' is allowed on a connection-
             * based socket if it supports implied connect.
             * Return ENOTCONN if not connected and no address is
             * supplied.
             */
            if (0) {
            //-if ((so->so_proto->pr_flags & PR_CONNREQUIRED) &&
            //-    (so->so_proto->pr_flags & PR_IMPLOPCL) == 0) {
            //- if ((so->so_state & SS_ISCONFIRMING) == 0 &&
            //-     !(resid == 0 && clen != 0)) {
            //-     SOCKBUF_UNLOCK(&so->so_snd);
            //-     error = ENOTCONN;
            //-     goto release;
            //- }
            } else if (addr == NULL) {
                SOCKBUF_UNLOCK(&so->so_snd);
                //-if (so->so_proto->pr_flags & PR_CONNREQUIRED)
                    error = ENOTCONN;
                //-else
                //- error = EDESTADDRREQ;
                goto release;
            }
        }
        space = sbspace(&so->so_snd);
        if (flags & MSG_OOB)
```

```
    space += 1024;
if ((atomic && resid > so->so_snd.sb_hiwat) ||
    clen > so->so_snd.sb_hiwat) {
    SOCKBUF_UNLOCK(&so->so_snd);
    error = EMSGSIZE;
    goto release;
}
if (space < resid + clen &&
    (atomic || space < so->so_snd.sb_lowat || space < clen)) {
    if ((so->so_state & SS_NBIO) ||
        (flags & (MSG_NBIO | MSG_DONTWAIT)) != 0) {
        SOCKBUF_UNLOCK(&so->so_snd);
        error = EWOULDBLOCK;
        goto release;
    }
    KASSERT(false, ("sosend should be always non-blocking"));
    //-error = sbwait(&so->so_snd);
    //-SOCKBUF_UNLOCK(&so->so_snd);
    //-if (error)
    //- goto release;
    //-goto restart;
}
SOCKBUF_UNLOCK(&so->so_snd);
space -= clen;
do {
    //-if (uio == NULL) {
        resid = 0;
        //-if (flags & MSG_EOR)
        //- top->m_flags |= M_EOR;
    //-} else {
    //- /*
    //-  * Copy the data from userland into a mbuf
    //-  * chain.  If resid is 0, which can happen
    //-  * only if we have control to send, then
    //-  * a single empty mbuf is returned.  This
    //-  * is a workaround to prevent protocol send
    //-  * methods to panic.
    //-  */
    //- top = m_uiotombuf(uio, M_WAITOK, space,
    //-     (atomic ? max_hdr : 0),
    //-     (atomic ? M_PKTHDR : 0) |
    //-     ((flags & MSG_EOR) ? M_EOR : 0));
    //- if (top == NULL) {
    //-     error = EFAULT; /* only possible error */
    //-     goto release;
    //- }
    //- space -= resid - uio->uio_resid;
    //- resid = uio->uio_resid;
    //-}
    if (dontroute) {
        SOCK_LOCK(so);
        so->so_options |= SO_DONTROUTE;
        SOCK_UNLOCK(so);
    }
    /*
     * XXX all the SBS_CANTSENDMORE checks previously
     * done could be out of date.  We could have received
     * a reset packet in an interrupt or maybe we slept
     * while doing page faults in uiomove() etc.  We
     * could probably recheck again inside the locking
     * protection here, but there are probably other
     * places that this also happens.  We must rethink
     * this.
     */
```

```
            //-VNET_SO_ASSERT(so);
            //-error = (*so->so_proto->pr_usrreqs->pru_send)(so,
            //-    (flags & MSG_OOB) ? PRUS_OOB :
            //-/*
            //- * If the user set MSG_EOF, the protocol understands
            //- * this flag and nothing left to send then use
            //- * PRU_SEND_EOF instead of PRU_SEND.
            //- */
            //-    ((flags & MSG_EOF) &&
            //-     (so->so_proto->pr_flags & PR_IMPLOPCL) &&
            //-     (resid <= 0)) ?
            //- PRUS_EOF :
            //-/* If there is more to send set PRUS_MORETOCOME. */
            //-    (flags & MSG_MORETOCOME) ||
            //-    (resid > 0 && space > 0) ? PRUS_MORETOCOME : 0,
            //-    top, addr, control, td);
            error = tcp_usr_send(so, 0, top, addr, control, NULL);
            if (dontroute) {
                SOCK_LOCK(so);
                so->so_options &= ~SO_DONTROUTE;
                SOCK_UNLOCK(so);
            }
            clen = 0;
            control = NULL;
            top = NULL;
            if (error)
                goto release;
        } while (resid && space > 0);
    } while (resid);

release:
    //-sbunlock(&so->so_snd);
out:
    if (top != NULL)
        m_freem(top);
    if (control != NULL)
        m_freem(control);
    return (error);
}
```