DOI: 10.15514/ISPRAS-2021-33(4)-4

# Средства захвата и обработки высокоскоростного сетевого трафика

[1]Д.В. Ларин, ORCID: 0000-0001-8686-8916 <larin.dv@ispras.ru>
[2,3]А.И. Гетьман, ORCID: 0000-0002-6562-9008 <thorin@ispras.ru>

[1]Московский физико-технический институт,
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[2]Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[3]Национальный исследовательский университет «Высшая школа экономики»,
101978, Россия, г. Москва, ул. Мясницкая, д. 20

**Аннотация.** В данной работе дается обзор научных исследований в области захвата и обработки высокоскоростного трафика, а также рассматриваются конкретные программно-аппаратные решения. В работе выделены основные направления развития технологий перехвата трафика и описано их взаимодействие между собой. На основе обзора выделены основные проблемы сетевого стека операционных систем и способы их решения. Рассматриваются реализованные в программно-аппаратных средствах алгоритмы и структуры, а также их архитектура. Для каждого из решений описывается процесс получения пакетов с сетевого интерфейса. Кроме того, проведено сравнение их производительности, общий анализ реализаций, а также приведены рекомендации по области применимости.

**Ключевые слова:** перехват высокоскоростного трафика; программная обработка трафика; Анализ производительности; libpcap; PF_RING; DPDK; Netmap; eBPF; XDP; AF_XDP

**Для цитирования:** Ларин Д.В., Гетьман А.И. Средства захвата и обработки высокоскоростного сетевого трафика. Труды ИСП РАН, том 33, вып. 4, 2021 г., стр. 49-68. DOI: 10.15514/ISPRAS–2021–33(4)–4

## High-speed network traffic capturing and processing tools

[1]D.V. Larin, ORCID: 0000-0001-8686-8916 <larin.dv@ispras.ru>
[2,3]A.I. Getman, ORCID: 0000-0002-6562-9008 <thorin@ispras.ru>

[1]Moscow Institute of Physics and Technology,
9, Institutskiy per., Dolgoprudniy, 141701, Russia
[2]Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[3]National Research University Higher School of Economics,
20, Myasnitskaya Ulitsa, Moscow, 101978, Russia

**Abstract.** Network stacks currently implemented in operating systems can no longer cope with the packet rates offered by 10 Gbit Ethernet. Thus, frameworks were developed claiming to offer a faster alternative for this demand. These frameworks enable arbitrary packet processing systems to be built from commodity hardware handling a traffic rate of several 10 Gbit interfaces, entering a domain previously only available to custom-built hardware. In this paper, we survey various frameworks for high-performance packet IO and their interaction with a modular frameworks and specialized virtual network functions software for high-speed packet processing. We introduce a model to estimate and assess the performance of these packet processing





frameworks. Moreover, we analyze the performance of the most prominent frameworks based on representative measurements in packet capturing scenarios. Therefore, we provide a comparison between them and select the area of applicability.

**Keywords:** high-speed traffic capturing; software packet processing; performance measurements; libpcap; PF_RING; DPDK; Netmap; eBPF; XDP; AF_XDP

**For citation:** Larin D.V., Getman A.I., Hight-speed network traffic capturing and processing tools. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 4, 2021. pp. 49-68 (in Russian). DOI: 10.15514/ISPRAS–2021–33(4)–4

## 1. Введение

С развитием сети Интернет и увеличением доступности операторы связи и интернет-провайдеры вынуждены расширять каналы передачи данных и применять более современные с технологической точки зрения решения для обработки и анализа сетевого трафика на скоростях 100 Гбит/с и более. Такие задачи, как балансировка и фильтрация трафика, обнаружение и предотвращение различных сетевых угроз, высокопроизводительные коммутация и маршрутизация потребовали разработки новых подходов, способных обеспечивать хорошую производительность на высоконагруженных сетях.

Для решения возникающих проблем сетевые операторы начали использовать специализированные аппаратные решения (например, FPGA, ASIC, SmartNIC, архитектура P4). Такие устройства настраиваются под нужды какой-либо конкретной задачи, требующей очень высокой производительности – например, захват сетевого трафика без потерь, разбор пакетов, коммутация трафика. Однако, эти решения зачастую страдают от недостаточной гибкости и масштабируемости, что становится проблемой в условиях современных высокомасштабируемых систем.

Альтернативой стало использование возможностей стандартного сетевого оборудования и программной обработки трафика. Перед аппаратными реализациями такой подход обладает рядом преимуществ, среди которых возможность адаптации аппаратуры для решения различных задач, снижение операционных и капитальных расходов при построении больших систем для мониторинга и анализа сети, допустимость использования широкого класса программных решений с открытым исходным кодом. В то же время, данные решения требуют разработки эффективных методов взаимодействия с сетью и ставят перед исследователями новые задачи по созданию гибких и масштабируемых систем для перехвата, анализа и обработки высокоскоростного трафика.

Дальнейшая статья устроена следующим образом. В разд. 2 описаны особенности программной обработки трафика. В разд. 3 приведен подробный обзор существующих программных решений. Разд. 4 посвящен анализу и сравнению производительности описанных решений. В разд. 5 подведены итоги работы.

## 2. Программная обработка трафика

Программные решения по обработке сетевого трафика можно разделить на три основных категории [1-3]:

1) низкоуровневые решения, обеспечивающие ввод/вывод трафика: DPDK [4], PF_RING [5], XDP [6], netmap [7], libpcap [8];
2) специализированные программные реализации виртуальных сетевых функций: Open virtualSwitch (Open vSwitch) [9], ESwitch [10], PacketShader [11], DPDKStat [12];
3) модульные фреймворки для создания сетевых функций: Click [13], FastClick [14], VPP (Vector Packet Processing) [15], BESS [3], Snabb NFV [16], PacketMill [17].

Данная статья фокусируется на исследовании решений первой категории, т.е. низкоуровневых решений по перехвату и передаче трафика. Они могут быть использованы

для всех типов приложений, которым необходим доступ к сетевому трафику. Их можно разделить на два типа в зависимости от используемой архитектуры:

*Kernel-Bypass.* Основная особенность данной архитектуры заключается в передаче трафика от сетевой карты в пространство пользователя “в обход” ядра ОС, т.е. вся обработка пакетов производится не в сетевом стеке операционной системы, а в пользовательском пространстве. По этому принципу функционирует, например, прием и обработка пакетов с помощью libpcap. К частному случаю архитектуры Kernel-Bypass можно отнести схемы по перехвату трафика “без копирований” или “0-copy”. Последнее стало возможным, благодаря “эффективному” использованию механизма DMA (Direct Memory Access) – контроллер DMA записывает пришедшие на сетевую карту пакеты в предварительно выделенную область памяти ядра, которая, в свою очередь, отображена на область памяти в пространстве пользователя. Таким образом, пользовательские приложения могут напрямую обращаться к содержимому пакетов без необходимости его копирования. Поддержкой DMA снабжаются все современные сетевые карты, а на его основе сделаны такие решения, как PF_RING Zero-Copy (ZC) и DPDK (Data Plane Development Kit)

*In-Kernel FastPath.* Особенность данной архитектуры заключается в запуске первичной обработки трафика в пространстве ядра ОС до того, как пакет передается в сетевой стек ОС. Исторически программирование ядра было сложной задачей из-за необходимости написания кода на низкоуровневых языках программирования и строгих требований к безопасности кода – ошибка в программе могла привести к отказу ОС. С приходом технологий eBPF (extended Berkeley Packet Filter) и XDP (eXpress Data Path) исследователям удалось добиться загрузки и выполнения пользовательских программ, написанных на высокоуровневых языках программирования, в ядре ОС с сохранением всех требований по безопасности кода. Архитектура подразумевает использование возможностей ядра ОС для работы с входящим трафиком, без необходимости передавать трафик в сетевой стек или в пространство пользователя. Если последнее все же необходимо, то для этих целей используется сокет AF_XDP.

## 2.1 Receive Side Scaling (RSS)

С увеличением количества входящих сетевых пакетов в случае обработки в однопоточном режиме процессор может не справляться с обработкой всего поступающего трафика. Для решения этой проблемы можно воспользоваться преимуществами многопоточной обработки, одной из составляющих которой является механизм Receive Side Scaling (RSS).

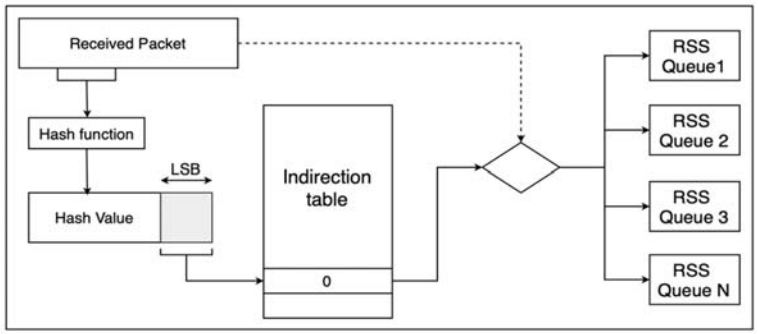


*Рис. 1. Механизм работы RSS [18]*
*Fig. 1. RSS architecture [18]*

RSS позволяет распределять полученные сетевой картой пакеты по нескольким входным очередям, каждой из которых операционная система назначает свое ядро процессора. Таким образом, нагрузка по обработке входящего трафика распределяется на несколько ядер





многоядерной системы, что позволяет избежать проблем, связанных с обработкой на одном процессорном ядре, и оптимизировать использование кэша. Говоря более детально, RSS распределяет трафик по входным очередям при помощи значений, полученных после применения хэш-функции к нескольким полям входящих пакетов, и таблицы косвенной адресации (indirection table).

Как изображено на рис. 1, наименее значащие биты (Least Significant Bits, LSB) хэша используются в качестве ключей для доступа к соответствующим позициям в таблице косвенной адресации. Такая таблица содержит в себе значения, служащие для распределения полученных данных на обработку конкретным ядром процессора. Для вычисления хэшей используется хэш-функция Тёплица (Toeplitz), на вход которой подается массив данных (т.н. 5-tuple или 5-ка полей пакета [19]), состоящий из IPv4/IPv6 адресов отправителя/получателя, TCP/UDP портов отправителя/получателя и опциональных расширенных заголовков IPv6, и секретный 40-байтовый ключ - как правило, битовая маска. Стандартный ключ распределяет трафик по очередям, поддерживая однонаправленную когерентность на уровне потока - пакеты, которые содержат одинаковые адреса и порты отправителя/получателя, будут направлены на обработку в одну и ту же очередь [18, 20].

## 2.2 Сетевой стек операционной системы

Современная сетевая аппаратура стремительно развивается для работы с высокоскоростным трафиком, однако программные решения зачастую не следуют этому тренду. Большинство существующих операционных систем предоставляют универсальный сетевой стек с простым пользовательским интерфейсом на основе сокетов для получения/отправки данных и поддержки большого числа сетевых протоколов и аппаратуры, в приоритете которого стоит совместимость, а не производительность.

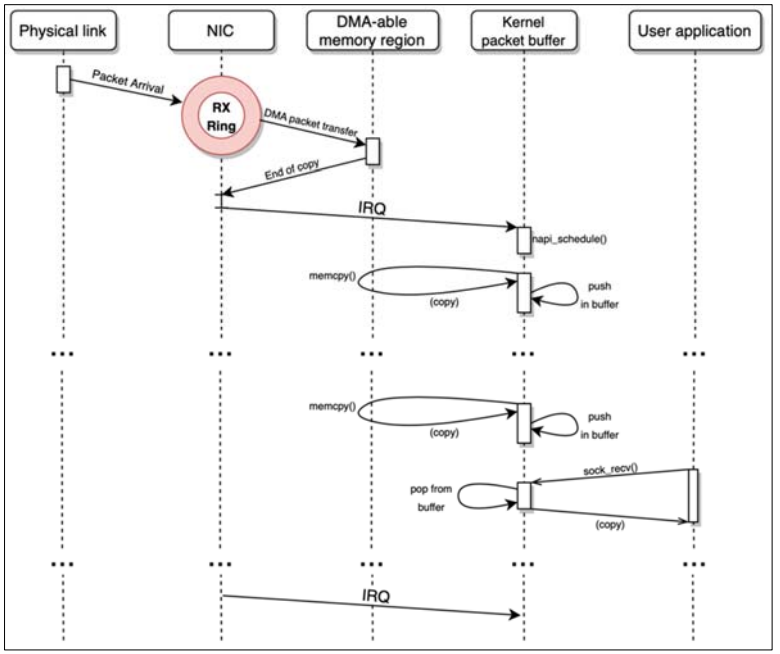


*Рис. 2. Сетевой стек GNU/Linux, механизм NAPI [20]*
*Fig. 2. Linux NAPI RX scheme [20]*

В данной статье рассматриваются существующие решения для GNU/Linux как для самого широко используемого семейства операционных систем с открытым исходным кодом, предоставляющего множество средств для анализа производительности системы. Для каждого решения будет отдельно указано, поддерживается ли оно на других ОС.

Большинство современных высокоскоростных драйверов, начиная с версии 2.6.0 ядра Linux (2003 г.), используют механизм NAPI для захвата пакетов с сетевой карты. NAPI позволяет существенно снизить количество прерываний по сравнению с предыдущими схемами. Более детально происходит это следующим образом: при получении первого прерывания драйвером запускается ответственная за прерывания NAPI функция, но, в отличие от традиционного подхода, она не приступает к копированию и помещению пакета в очередь, а планирует (shedule) выполнение функции poll() и отключает аналогичные прерывания в дальнейшем. Данная функция проверяет наличие новых пакетов, и, в случае их готовности, копирует и помещает в очередь сетевого стека GNU/Linux, без ожидания прерывания. В дальнейшем функция poll() планирует свой следующий запуск, продолжая до тех пор, пока на сетевую карту приходят пакеты. Алгоритм работы изображен на рис. 2.

Кроме того, если система не справляется с обработкой высокоскоростного трафика, механизм NAPI позволяет отбрасывать пакеты на уровне сетевого адаптера – т.е. до их прихода на уровень ядра. Стоит отметить, что механизм NAPI сильнее нагружает CPU при обработке медленного трафика по сравнению с традиционными механизмами. С увеличением скорости входящего трафика нагрузка снижается [20].

## 2.3 Проблемы сетевого стека

Среди проблем сетевого стека GNU/Linux и, в частности, механизмов NAPI и RSS, в соответствии с [20], можно выделить следующие.

### 2.3.1 Попакетное выделение и освобожение памяти

Каждый раз, когда на сетевую карту приходит новый пакет, для хранения информации о теле и заголовке пакета в памяти выделяется пакетный дескриптор. После завершения работы с пакетом этот дескриптор освобождается. Такой процесс работы с памятью приводит к значительным временным издержкам, что серьезно влияет на производительность при работе с высокоскоростным трафиком – 14.88 миллионов пакетов в секунду (Million packets per second, Mpps) для 10GbE. Кроме того, структура sk_buff, используемая для хранения всей информации о пакетах, обладает значительным размером, так как содержит множество информации о различных протоколах на нескольких уровнях. Как описано в [21], обращение и выделение sk_buff использует около 1200 тактов CPU на каждый пакет, а освобождение порядка 1100 тактов. А на операции по взаимодействию с sk_buff приходится порядка 63% нагрузки CPU при обработке пакетов Ethernet минимального размера, составляющего 64 байта [22].

Для решения этой проблемы предлагается использовать заранее выделенные участки памяти и переиспользовать их в дальнейшем. По этому принципу работают кольцевые буферы. Перед началом обработки, для нужного количества кольцевых буферов выделяется память, в которую записывается информация о пакетах. После обработки пакета, память не освобождается, а переиспользуется снова для новых пакетов. Таким образом, удается избежать множественных временных задержек. К минусам подхода можно отнести временные затраты на выделение памяти перед началом обработки и ее постоянную занятость во время обработки.

### 2.3.2 Сериализуемость доступа к трафику

Современные сетевые карты поддерживают получение пакетов с использованием аппаратных Receive Side Scaling (RSS) очередей. Они позволяют разделять входящий трафик





на несколько потоков, за счет применения хэш-функции, например, к 5-ке полей пакета, и параллелизовать процесс захвата пакетов, путем отображения каждой входной очереди на собственное ядро процессора (см. подраздел 2.1). Проблемы начинают появляться несколькими уровнями выше, когда сетевой стек GNU/Linux начинает совмещение пакетов из всех очередей в одну на сетевом и транспортных уровнях для их последующего анализа. В связи с этим возникает две проблемы. Во-первых, весь трафик собирается в одной точке, что ведет к снижению производительности. Во-вторых, пользовательский процесс не может получать данные из какой-то конкретной RSS очереди.

Чтобы решить эти проблемы необходимо организовать прямой доступ к RSS очередям для пользовательских процессов. Максимальная производительность данной архитектуры достигается, когда один процесс выполняет сразу две задачи - прием пакетов из RSS очереди и их дальнейшую передачу в пространство пользователя. Кроме того, такой подход увеличивает масштабируемость системы - можно добавлять новые потоки обработки по мере того, как нарастает число задействованных ядер и RSS очередей. К минусам данного подхода можно отнести следующее.

1) Необходимость использовать сразу несколько ядер, которые можно использовать под другие задачи.
2) Использование хэш-функции для распределения пакетов по очередям. В том случае, если задача подразумевает работу со связанными пакетами, потоками или сессиями, использование хэш-функции для распределения трафика по входным очередям может привести к тому, что связанный трафик будет обрабатываться на разных очередях [23]. Эта проблема актуальна, например, для систем по мониторингу Voice over IP (VoIP), а именно для потоков SIP и RTP трафика, которые могут не разделять заголовки сетевого и транспортного уровней.

### 2.3.3 Множественные копирования пакетов на пути от сетевой карты до пространства пользователя

Пакеты, проходящие через сетевую подсистему Linux, несколько раз копируются до их получения пользовательским приложением. Как минимум дважды – один раз копирование происходит во время переноса пакета из DMA-области памяти в буфер на уровне ядра, второй раз из буфера в пространстве ядра в буфер пользователя. К примеру, на одно копирование требуется от 500 до 2000 тактов процессора в зависимости от размера пакета [21]. Вдобавок к этому, множественные копирования небольших по размеру пакетов приводят к значительной потере производительности.

Для решения этой проблемы предлагается использовать отображение DMA-участков памяти ОЗУ напрямую в пространство пользователя. Такой подход позволяет свести количество копирований данных пакетов к нулю (zero-copy), однако возможен только при поддержке со стороны сетевой карты. В качестве альтернативы, которая поддерживается на большем числе сетевых карт, возможно использование отображения буфера на уровне ядра в пространство пользователя, что является реализацией схемы с одним копированием (1-copy).

### 3.3.4 Переключения контекста между ядром и пространством пользователя

Пользовательское приложение должно осуществлять системный вызов для получения каждого пакета. Каждый такой вызов влечет за собой переключение контекста, между пространством пользователя и пространством ядра и наоборот. Это ведет к возрастанию использования ресурсов CPU. Каждый системный вызов и переключение контекста могут потребовать до 1000 тактов процессора на обработку одного пакета [21].

Решается данная проблема путем получения пакетов группами (batch-processing). Решение подразумевает сбор нескольких пакетов в буфер и копирование полученной группы в память пространства ядра или пользователя. С помощью данной методики удается уменьшить

количество системных вызовов и соответствующих им прерываний, а также снизить количество копирований. В соответствии с архитектурой NAPI, если трафик перехватывается при помощи механизма опроса, то можно запросить получение не одного пакета, а сразу нескольких. Если трафик принимается с помощью механизма прерываний, то можно использовать дополнительный буфер-посредник для хранения полученных пакетов, пока они не будут запрошены приложением. Основная проблема получения пакетов группами заключается в увеличении задержек и джиттере, а также в неточности выставления временных меток на полученные пакеты, так как пакетам необходимо ждать заполнения группы или истечения таймера [24].

#### 2.3.5 Неэффективное использование локализации памяти

Первичное обращение к DMA-области памяти вызывает принудительные промахи кэша (compulsory cache-misses), происходящие из-за инвалидации линий кэша процессора для консистентности памяти. Такие промахи кэша составляют 13.8% от общего числа тактов процессора, затраченных на получение одного 64-байтного пакета [22]. Кроме того, в системах, основанных на использовании NUMA (Non-Uniform Memory Access), задержки при обращении к памяти зависят от того, к какому узлу NUMA происходит обращение. Таким образом, неэффективное использование локализации памяти приводит к снижению производительности.

Архитектура NUMA подразумевает разделение доступной в системе памяти между различными процессорами, назначая каждому из них по своей области. Комбинация из процессора и области памяти называется NUMA-узлом. Для увеличения производительности и использования локализации памяти, процесс должен работать в области памяти, назначенной тому же процессору, на котором он исполняется. Такая техника называется закрепление памяти (memory affinity). Кроме того, существуют привязка к процессору (CPU affinity) и привязка прерываний (interrupt affinity). Первое подразумевает закрепление процесса или потока за каким-то конкретным процессором. Второе отвечает за привязку обработки прерываний к различным процессорам или ядрам (smp_affinity). Важность обработки прерываний и потоков входных данных на одном ядре заключается в повышении эффективности обращений к данным кэша и распределении нагрузки по ядрам системы. Как только потоку требуется получить доступ к полученным данным, он быстрее сможет найти их в локальном кэше процессора, если до этого эти данные были получены обработчиком прерываний, назначенным на то же ядро. Все необходимые настройки можно произвести при помощи утилиты *numactl*.

Помимо описанного, необходимо также обратить внимание на привязку потоков входных данных к узлам NUMA, назначенным на ту же шину PCIe, к которой подключена сетевая карта. Стоит упомянуть и средства аппаратной предвыборки (prefetching) данных следующего пакета и его дескриптора во время обработки текущего пакета. Задача предвыборки заключается в загрузке данных в кэш-память из оперативной памяти до того, как они потребуются.

## *3. Обзор программных решений*

### 3.1 libpcap

libpcap – библиотека, предоставляющая высокоуровневый интерфейс для систем по захвату трафика, а также поддерживающая возможности чтения пакетов в неразборчивом режиме (promiscuous mode) и их записи в файл [25]. С использованием libpcap разработаны такие приложения, как Wireshark, tcpdump, snort [26-28] и многие другие. Данная библиотека не предназначена для перехвата высокоскоростного трафика, поскольку в своей основе она использует драйвер уровня ядра, а именно Berkeley Packet Filter [8, 19]. Таким образом, она





переносит пакеты через сетевой стек Linux, производя их копирование на уровне ядра в структуру sk_buff, которая используется для хранения всей информации о пакетах. Такая схема сильно влияет на итоговую производительность по сравнению с реализованными без копирований архитектурами. Процесс захвата пакетов с использованием библиотеки libpcap принципиально не отличается от описанной ранее (см. рис. 2) схемы по устройству сетевого стека GNU Linux.

К преимуществам libpcap можно отнести поддержку всеми популярными операционными системами и сетевыми картами. Использование libpcap оправданно, если для решения необходима не производительность, а совместимость с оборудованием.

### 3.2 PF_RING

PF_RING — это фреймворк для захвата сетевого трафика на различных сетевых картах [5] (см. рис. 3). Существует несколько реализаций, которые применяются в зависимости от аппаратных возможностей и драйверов сетевых карт. А именно – не требующий специальной аппаратной поддержки Vanilla PF_RING и, наоборот, требующий такой поддержки PF_RING Zero-Copy (ZC). Помимо этого, вне зависимости от режима, с PF_RING можно взаимодействовать при помощи стандартного libpcap API, а также использовать собственный API, предоставляющий возможности написания BPF фильтров, приема пакетов из нескольких входных очередей, закрепления программ за выбранными ядрами и другой функционал, позволяющий применять PF_RING для решения различных задач.

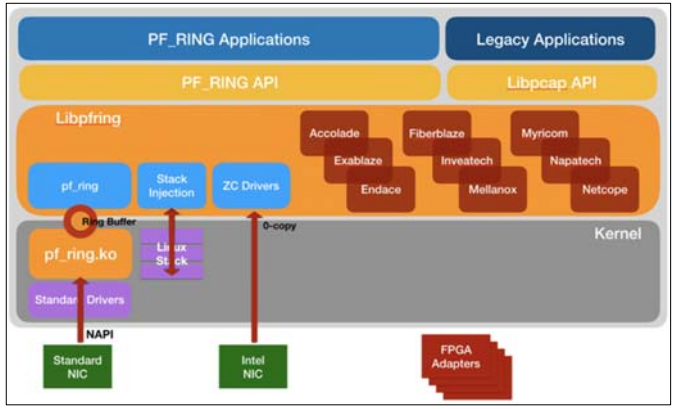


*Рис. 3. Общая схема PF_RING [5]*
*Fig. 3. PF_RING architecture [5]*

#### 3.2.1 Vanilla PF_RING

Vanilla PF_RING получает трафик с сетевой карты при помощи NAPI, копируя пришедшие на сетевую карту пакеты в кольцевой буфер, из которого пользовательское приложение может считать пакеты. При необходимости, поддерживается использование сразу нескольких кольцевых буферов. Такая схема позволяет избежать издержек, возникающих в классических механизмах 2-copy, тем не менее в ней все еще присутствует одно копирование. Проблемы данного подхода начинают проявляться, когда скорость входящего трафика возрастает и кольцевые буферы начинают быстро переполняться.

Vanilla PF_RING основан на использовании кольцевого буфера в пространстве ядра, в который копируются все входящие пакеты (см. рис. 4) [29]. Данный буфер выделяется при создании сокета PF_RING и освобождается при его удалении и может быть прикреплен к сетевому адаптеру при помощи системного вызова *bind()*. Как только сетевая карта получает

пакет, драйвер переносит его в пространство ядра при помощи механизма DMA. В случае использования сокета PF_RING каждый входящий пакет копируется и помещается в кольцевой буфер. В случае переполнения буфера пакет отбрасывается. Кольцевой буфер экспортируется в пространство пользователя при помощи *mmap()*.

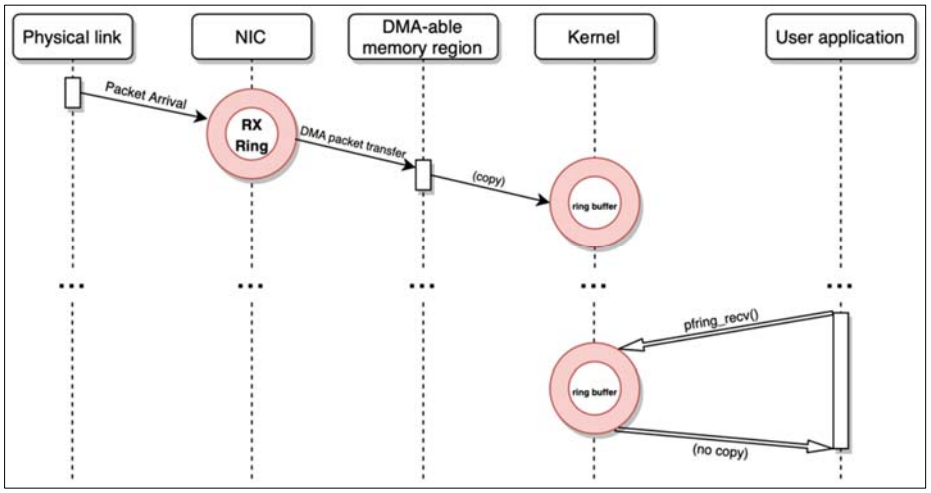


*Рис. 4. Процесс получения пакета Vanilla PF_RING [20]*
*Fig. 4. Vanilla PF_RING Rx process [20]*

Когда пользовательскому приложению необходимо получить доступ к сетевому трафику, оно вызывает *mmap()* для получения указателя на кольцевой буфер. Ядро копирует пакеты в кольцевой буфер и перемещает указатель на запись, пользовательское приложение производит аналогичную операцию для указателя на чтение. При этом новые пакеты перезаписывают уже прочитанные пользовательским приложением, то есть не происходит затратных с точки зрения потребления тактов процессора операций по выделению и освобождению памяти. К преимуществам данного подхода можно отнести следующее:

1) входящий трафик не отправляется в структуры сетевого стека ядра;
2) *mmap()* позволяет пользовательскому приложению получать доступ к трафику без необходимости осуществлять его копирование;
3) несколько приложений могут создавать собственные кольцевые буферы для чтения из нескольких входных RSS очередей.

Данный механизм осуществляет одно копирование на уровне ядра, в связи с чем пропускная способность модуля чтения с использованием Vanilla PF_RING заведомо ниже, чем с использованием механизма без копирований.

### 3.2.2 PF_RING Zero-Copy

PF_RING Zero-Copy основан на использовании модифицированных драйверов сетевых карт, которые позволяют осуществлять передачу пакетов пользовательским приложениям без копирований (рис. 5). PF_RING ZC отображает память пользовательского пространства в область памяти DMA, в связи с чем отпадает необходимость в использовании промежуточных буферов в пространстве ядра. Механизм предоставляет возможность чтения пакетов из нескольких входных очередей, что вкупе с отсутствием копирований дает значительный прирост производительности.





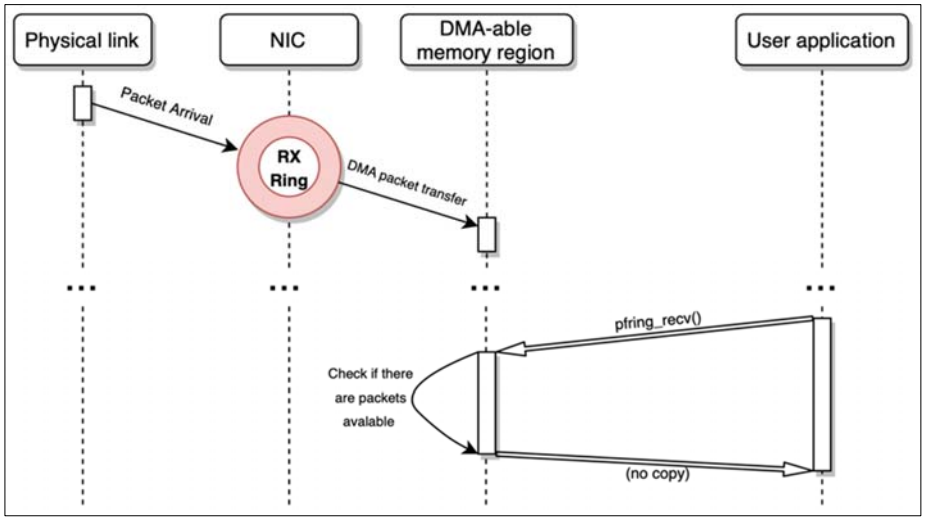


*Рис. 5. Процесс получения пакета PF_RING ZC [20]*
*Fig. 5. PF_RING ZC Rx process [20]*

Однако, данный подход обладает и очевидным недостатком – из-за необходимости использовать модифицированные драйверы для сетевых карт, количество оборудования на котором поддерживается PF_RING ZC ограничено (на момент написания работы поддерживаются только сетевые карты Intel).

Для всех перечисленных решений PF_RING предоставляет libpcap-like API, что сильно упрощает разработку приложений с использованием данного фреймворка. Кроме того, в PF_RING реализована возможность использовать XDP и сокет AF_XDP для захвата трафика. Интерфейс PF_RING можно использовать совместно с такими приложениями, как Snort, Suricata и Zeek (более известный как Bro).

## 3.3 DPDK

Data Plane Development Kit (DPDK) – это фреймворк, предоставляющий набор библиотек и драйверов для быстрой обработки трафика в пространстве пользователя, используя архитектуру Kernel Bypass (см. рис. 6). DPDK полностью замещает сетевой стек Linux - при подключении к сетевой карте все взаимодействие с ней будет происходить через компоненты DPDK и никакие другие приложения не смогут получить к ней доступ.

Стоит отметить, что сам по себе DPDK сетевым стеком не является – разбор пакетов необходимо реализовывать самостоятельно с использованием предоставленных библиотек. Совокупность библиотек DPDK образует EAL (Environment Abstraction Layer), который скрывает различия в аппаратной и программной частях систем и предоставляет API для взаимодействия с системой из пространства пользователя. Для перемещения сетевых пакетов DPDK использует буферы памяти (mbuf). Каждый mbuf состоит из трех секций: (i) структуры данных rte_mbuf, содержащей метаинформацию о пакете (например, номер VLAN или RSS очереди, ссылку на следующий пакет и тд), (ii) фиксированных по размеру областей памяти (headroom/tailroom) для добавления дополнительной информации и (iii) сегмента памяти для хранения всего пакета [4]. Каждая структура rte_mbuf использует только 2 блока кэша (cache line) для минимизации занимаемой памяти. Помимо этого, DPDK предоставляет драйверы для сетевых карт - т.н. Poll Mode Driver (PMD), которые позволяют приложениям напрямую взаимодействовать с сетевыми картами.

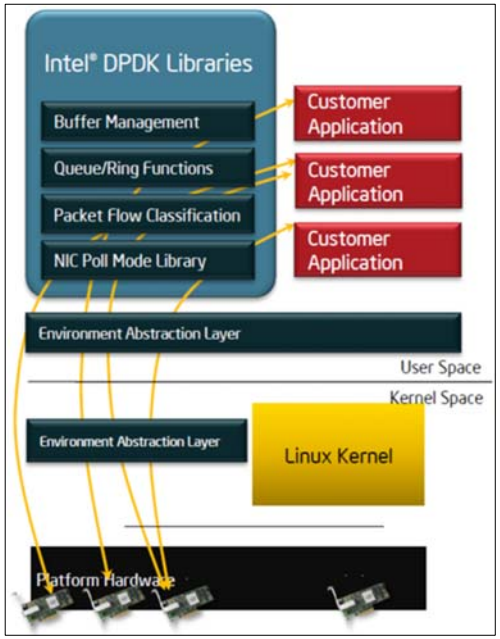


*Рис. 6. Общая схема DPDK [30]*
*Fig. 6. DPDK architecture [30]*

PMD использует mbuf для получения и передачи пакетов во время работы, а выделение памяти для них происходит на стадии инициализации приложения. Для захвата пакетов, PMD передает указатель на mbuf и его дескрипторы, зависящие от драйвера, сетевой карте, что позволяет ей при помощи DMA записывать полученные пакеты и их метаинформацию в необходимые разделы памяти. Затем, PMD при помощи опроса обнаруживает завершение операции DMA и копирует необходимую информацию из дескрипторов драйвера в структуру rte_mbuf. Все буферы памяти хранятся в объекте rte_mempool из которого при помощи кольцевых очередей (rte_ring) пользовательские приложения могут получать доступ к данным принятых пакетов [17, 30]. Схема взаимодействия DPDK с приложениями пользователя принципиально не отличается от аналогичной для PF_RING ZC, изображенной на рис. 6.

Помимо описанного, DPDK предоставляет и другие возможности для получения и обработки трафика. К ним можно отнести захват трафика при помощи сокета AF_XDP, библиотеку с реализацией алгоритма сопоставления максимального префикса (Longest Prefix Match, LPM) для IPv4 и IPv6 адресов, библиотеку таймеров для асинхронного выполнения функций, библиотеку хэширования и многие другие. Однако он обладает значительного размера документацией, которая сильно усложняет работу с данным фреймворком.

Ресурсы DPDK для захвата и передачи пакетов используют такие фреймворки, как VPP, PacketMill, FastClick и BESS, а также возможно его использование с Open vSwitch и для работы с виртуальными машинами (SR-IOV и VMDq режимы драйвера).

## 3.4 XDP/eBPF и сокет AF_XDP

eBPF (extended Berkeley Packet Fiter) и XDP (eXpress Data Path) необходимо рассматривать в совокупности из-за их тесной связи между собой (см. рис. 7). XDP – это фреймворк, который





выделяет ограниченную среду выполнения внутри виртуальной машины eBPF. Эта среда позволяет запускать различные программы непосредственно в контексте ядра, еще до того, как само ядро получит доступ к входным данным. Это дает возможность обработки трафика (в том числе и его перенаправление) на самой ранней из возможных стадий – сразу после получения пакетов от сетевой карты. Программа XDP запускается драйвером сетевой карты каждый раз, когда в систему приходит новый пакет. Запуск происходит внутри виртуальной машины eBPF, что позволяет добавить необходимую предобработку. Кроме того, eBPF предоставляет возможность использовать функции помощники ядра (kernel helpers) для доступа к структурам ядра и различным системным вызовам, а также разделяемую память - eBPF maps, которую можно применять как для хранения состояния между вызовами, так и для связи с другими eBPF программами и пространством пользователя (рис. 7). Помимо перечисленного, eBPF использует Verifier - верификатор пользовательских программ, который проверяет код на безопасность (например, на отсутствие бесконечных циклов, что гарантирует завершение программы) перед его загрузкой в пространство ядра [6].

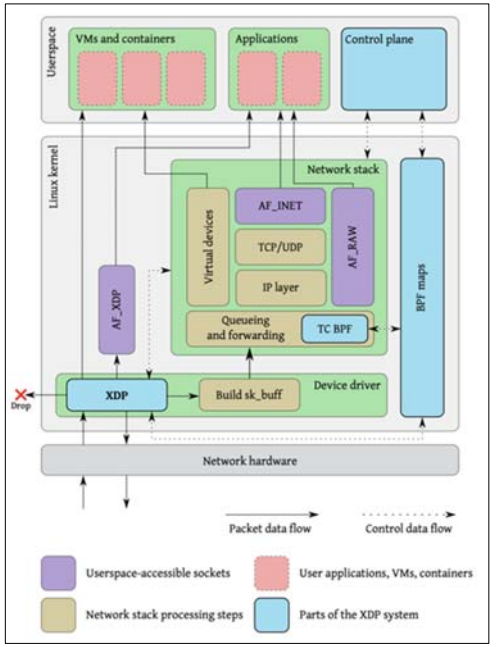


*Рис. 7. Общая схема XDP/eBPF [6]*
*Fig. 7. XDP/eBPF architecture [6]*

Действия, которые производит программа XDP можно разделить на отбрасывание пакета (XDP_DROP), его пропускание в сетевой стек Linux (XDP_PASS) или передачу пакета в тот же (XDP_TX) или в другой сетевой интерфейс (XDP_REDIRECT), как указано на рис. 8. Так как программа XDP исполняется на уровне драйвера, то выполнять задачи она может достаточно быстро. Но она ограничена в размере и может выполнять только ограниченное число инструкций, поэтому, в случае необходимости проведения более сложных операций, пакет может быть передан в сетевой стек Linux или в пространство пользователя, используя множество существующих сокетов, таких как AF_PACKET, стандартный для TCP/IP сокет AF_INET или AF_XDP. Рассмотрим работу последнего более подробно. AF_XDP – это тип

сокета, который позволяет передавать данные пакетов напрямую с сетевой карты в пространство пользователя используя XDP и избегая копирований. AF_XDP работает в трех режимах, начиная от самого медленного до самого быстрого:

1) skb режим, работающий на любой сетевой карте;
2) XDP-copy режим, который требует поддержки драйвером сетевой карты;
3) Режим без копирований, который работает только на драйверах с поддержкой XDP и расширением для zero-copy.

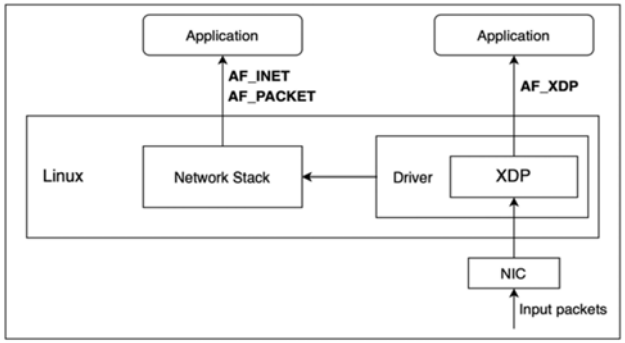


*Рис. 8. Поток трафика с использованием XDP [31]*
*Fig. 8. XDP traffic flow [31]*

Далее мы будем рассматривать только архитектуру zero-copy как самую производительную. С точки зрения приложения все пакеты находятся в выделенной области памяти, называющейся UMEM, как изображено на рис. 9. Эта область состоит из сегментов одного размера - пакетных буферов в которых находится информация о пакетах. Вместе с UMEM создаются два кольца: кольцо заполнения (fill ring) и кольцо завершения (completion ring). Кольцо заполнения используется для передачи управления пакетным буфером от пользовательского пространства в ядро. Кольцо завершения, наоборот, сигнализирует о передаче управления пакетным буфером от ядра в пространство пользователя. Приложение уведомляет о передаче управления записывая относительный адрес пакета в кольцо заполнения. Аналогичным образом поступает ядро, используя кольцо завершения. При помощи данных очередей мы можем передавать управление пакетными буферами без необходимости получать и отправлять данные. Для получения и отправки пакетов используются еще два кольца: кольцо получения (Rx ring) и кольцо отправки (Tx ring).

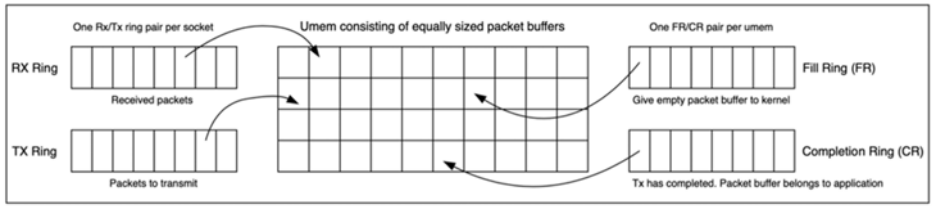


*Рис. 9. 4 кольца AF_XDP и структура UMEM для хранения данных о пакетах [31]*
*Fig. 9. Four AF_XDP rings and UMEM structure [31]*

При получении пакета ядро записывает дескриптор пакета в кольцо получения, указывая, что пакетный буфер содержит данные пакета и задает его относительный адрес и длину. Проверяя кольцо получения, приложение определяет, что пакет получен и считывает его.

Аналогичным образом приложение поступает при отправке пакета, используя очередь отправки. Стоит отметить, что очереди отправки и получения создаются для одного сокета,





и каждый сокет привязывается к одной структуре UMEM, которая имеет лишь одну пару колец заполнения и завершения. Однако, к структуре UMEM можно привязать несколько сокетов, в этом случае буферов получения и отправки будет несколько.

Процесс получения пакета можно описать следующим образом (см. рис. 10). Пакет приходит на сетевую карту и забирается драйвером, выполняющим программу XDP, которая принимает решение об отправке пакета в какой-то конкретный сокет AF_XDP. Так как сокет AF_XDP работает в режиме zero-copy, то сетевая карта уже записала данные пакета в пакетный буфер структуры UMEM, поэтому все, что остается сделать ядру — это записать в кольцо получения дескриптор пакета, чтобы уведомить приложение о нахождении пакета в памяти. Затем приложение проверяет кольцо получения на наличие пакетов. Как только оно заканчивает обработку пакета, дескриптор пакета возвращается в ядро при помощи кольца заполнения, таким образом новый пакет может быть записан в тот же пакетный буфер.

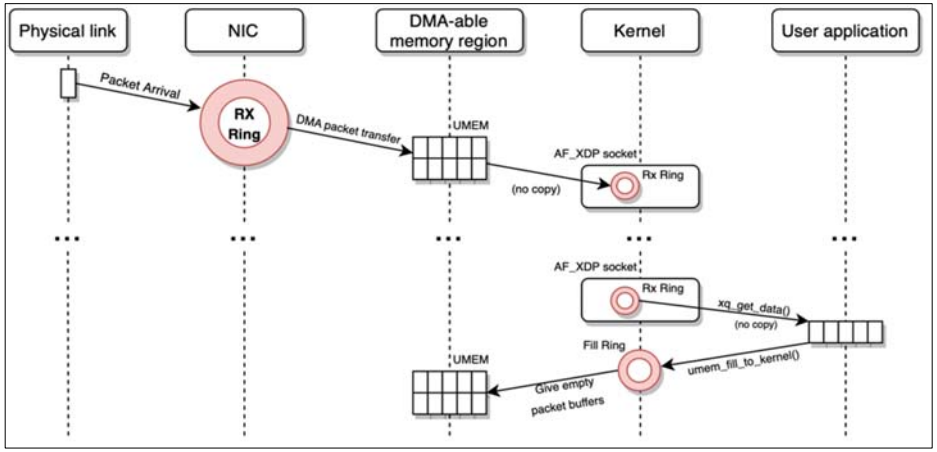


*Рис. 10. Процесс получения пакета AF_XDP*
*Fig. 10. AF_XDP Rx process*

Интерфейс AF_XDP для захвата и передачи пакетов может быть использован с такими фреймворками, как VPP, FastClick и BESS, а также с Open vSwitch.

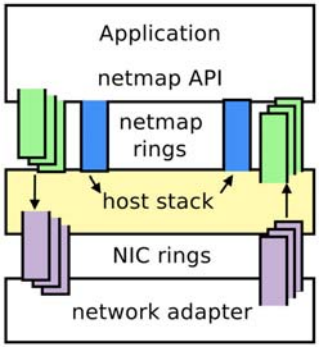


*Рис. 11. Общая схема Netmap [7]*
*Fig. 11. Netmap architecture [7]*

## 3.5 Netmap

Netmap – это фреймворк для работы с высокоскоростным сетевым трафиком на стандартном сетевом оборудовании (рис. 11). В соответствии с [7] решение использует предвыделение памяти на стадии инициализации, буферы фиксированного размера (2048 байт), обработку пакетов группами, а также предоставляет возможность параллельной обработки трафика. Кроме того, netmap использует отображение памяти для обеспечения пользовательских приложений прямым доступом к пакетным буферам с простым и оптимизированным представлением метаданных - кольцом памяти netmap (netmap memory ring). Оно содержит информацию о размере кольца памяти, указатель на текущую позицию буфера, количество полученных буфером пакетов или число свободных позиций в нем в зависимости от работы в режиме приема или передачи трафика соответственно, флаги о текущем статусе, отступ памяти пакетного буфера и массив с метаинформацией. Также на каждый пакет выделена память для хранения размера пакета, его индекса в пакетном буфере и некоторых флагов.

Кольца памяти netmap создаются по одному на каждую входную RSS очередь, что позволяет обрабатывать трафик в многопоточном режиме. Процесс захвата пакета изображен на рис. 12.

Аналогично XDP, интерфейс Netmap для захвата и передачи пакетов может быть использован с VPP, FastClick, BESS и Open vSwitch.

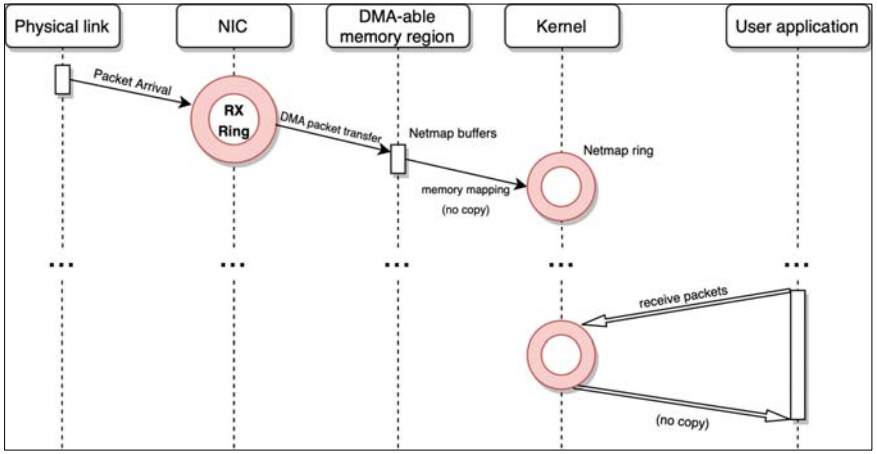


*Рис. 12. Процесс получения пакета Netmap*
*Fig. 12. Netmap Rx process*

## *4. Анализ программных решений*

Далее приводится сравнительный анализ описанных решений: Vanilla PF_RING, PF_RING Zero-Copy, DPDK, Netmap и XDP/eBPF с AF_XDP.

*Табл. 1. Сравнение решений (D - драйвер, K - ядро, K-U - интерфейс ядро/пользователь)*
*Table 1. Comparison of solutions (D - driver, K - kernel, K-U - kernel / user interface)*

| Характеристики / решения | Vanilla PF_RING | PF_RING ZC | DPDK | XDP/eBPF AF_XDP | Netmap |
|---|---|---|---|---|---|
| Бесплатность | ✓ | X | ✓ | ✓ | ✓ |





| | | | | | |
|---|---|---|---|---|---|
| Open-source | ✓ | ✓ | ✓ | ✓ | ✓ |
| Переиспользование и предвыделение памяти | ✓ | ✓ | ✓ | ✓ | ✓ |
| Поддержка RSS и исполнения в несколько потоков | ✓ | ✓ | ✓ | ✓ | ✓ |
| Отображение памяти | ✓ | ✓ | ✓ | ✓ | ✓ |
| Zero-Copy | X | ✓ | ✓ | ✓ | ✓ |
| Обработка пакетов группами | ✓ | ✓ | ✓ | ✓ | ✓ |
| Закрепление CPU и прерываний | ✓ | ✓ | ✓ | ✓ | ✓ |
| Закрепление памяти | ✓ | ✓ | ✓ | ✓ | X |
| Обработка на уровне ядра ОС | X | X | X | ✓ | X |
| Обработка в пространстве пользователя | ✓ | ✓ | ✓ | ✓ | ✓ |
| Уровни модификаций | D, K, K-U | D, K, K-U | D, K-U | D, K, K-U | D, K, K-U |
| API | libpcap-like | libpcap-like | Custom | Custom | Standard libc |
| Архитектура | Kernel-Bypass | Kernel-Bypass | Kernel-Bypass | In-Kernel Fastpath | Kernel-Bypass |
| Поддерживаемые ОС | GNU/Linux | GNU/Linux | GNU/Linux, Windows | GNU/Linux, Windows | GNU/Linux, FreeBSD |
| Поддерживаемые сетевые карты | Любые* | Intel | Любые* | Любые* | Intel, Mellanox, Nvidia, Realtek |
| • необходима поддержка драйверами сетевых карт | | | | | |

## 4.1 Анализ производительности

Данный подраздел содержит результаты тестирования Vanilla PF_RING, PF_RING Zero-Copy, XDP/eBPF с AF_XDP и библиотеки libpcap на тестовом стенде, построенном на базе двух серверов с ОС Ubuntu 18.04 (128 Gb RAM, процессор: AMD Ryzen Threadripper 3960X с тактовой частотой 3.8 GHz и 24 физическими ядрами) на каждом из которых установлены

сетевые карты Intel i40e XL710 с двумя портами 40G QSFP+. Сервера соединены напрямую, один из них выступает в роли генератора трафика, на втором запущен анализатор трафика и происходит перехват.

В работе использовался программный генератор трафика PF_RING (pfsend), позволяющий в режиме zero-copy отправлять трафик на 40 Gb сетевых картах на скорости от 30 Гбит/сек для пакетов минимального размера (64 байта), достигающей 40 Гбит/сек для пакетов, размер которых превышает 100 байт. Отправка происходит в несколько потоков с использованием RSS очередей. Для достижения максимальной производительности в большинстве случаев достаточно одновременно запустить 3 генератора. Дальнейшее увеличение числа генераторов к улучшению производительности не приводит. Кроме того, генератор PF_RING позволяет балансировать трафик для передачи его в разные RSS очереди на приемной стороне. Реализовано это за счет выставления и изменения адресов и портов отправителя в сгенерированных пакетах.

На рис. 13 и 14 приведены результаты тестирования PF_RING (Vanilla и Zero-Copy), XDP/eBPF с AF_XDP и библиотеки libpcap на стенде с 40 Gb сетевыми картами. На графиках изображена скорость вхождения пакетов в систему в зависимости от их размера. Измерение скорости осуществляется при помощи специально разработанной программы, которая получает пакеты с сетевого интерфейса при помощи одного из фреймворков и передает трафик в выделенный кольцевой буфер. Таким образом, программа производит одно копирование и предоставляет возможность дальнейшей обработки пакетов в пространстве пользователя.

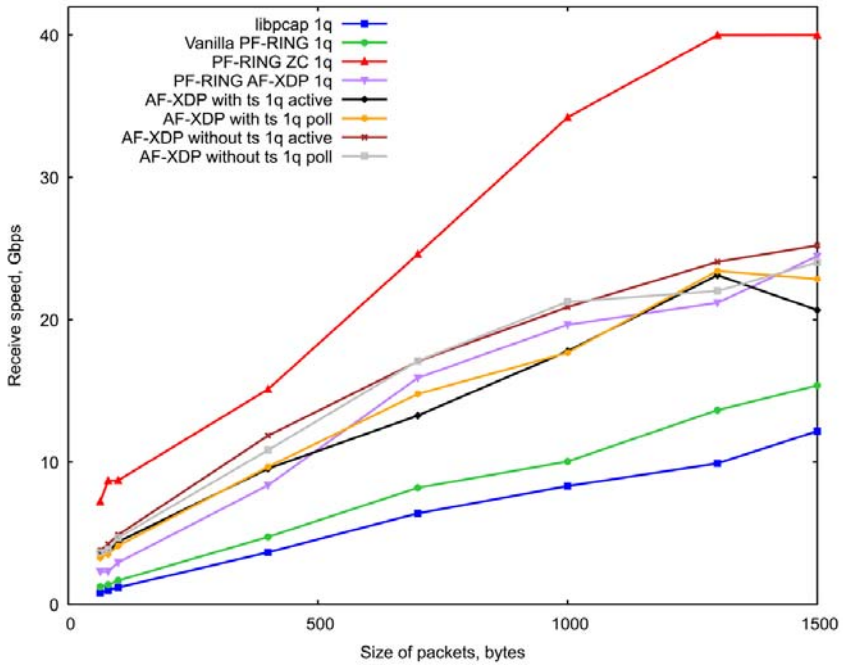


*Рис. 13. Сравнение производительности решений на одной входной очереди*
*Fig. 13. Performance comparison on one input queue*

На рис. 13 сравниваются производительности программных решений, работающих в режиме захвата пакетов из одной входной очереди. По всем приведенным графикам можно заметить, что использование libpcap и Vanilla PF_RING закономерно приводит к наименьшим





пропускным способностям. Кроме того, из всех решений необходимо выделить PF_RING ZC, который способен принимать пакеты на скорости 40 Гбит/сек, используя только одну входную очередь. Наиболее близок к нему AF_XDP, работающий в режиме активного ожидания без выставления временных меток. Кроме того, каждое из программных решений тестировалось на приеме пакетов из одной, шести и двенадцати входных RSS очередей в режиме активного ожидания (см. рис. 14).

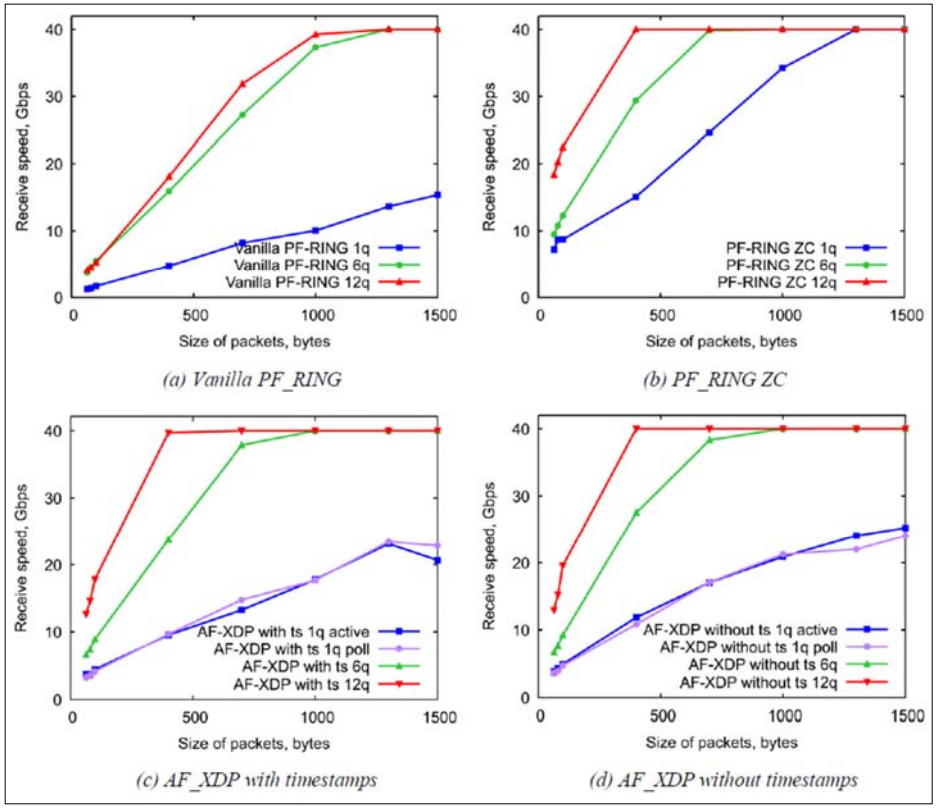


*Рис. 14. Измерения производительности*
*Fig. 14. Performance measurements*

Для XDP/eBPF с AF_XDP снята производительность захвата пакетов в режиме опроса (polling), что отражено на рис. 14 (c, d). Помимо этого, поскольку в стандартном режиме AF_XDP не поддерживает выставления временных меток, была реализована eBPF программа, которая записывает временные метки в метаданные пакета и позволяет пользовательским приложениям их считывать. Результаты тестирования данного режима изображены на рис. 14 (c), однако использование дополнительных системных вызовов приводит к ухудшению производительности. С увеличением количества входных очередей все решения показывают прирост производительности. Минусом данного подхода является сильная нагрузка на CPU из-*за* параллельной работы нескольких процессов.

Полученные результаты необходимо интерпретировать как оценочные, поскольку в зависимости от архитектуры пользовательского приложения, решаемой задачи и конфигурации аппаратуры производительность программных решений может сильно отличаться.

## *5. Заключение*

В рамках данной работы был проведен анализ решений по перехвату и обработке высокоскоростного трафика на стандартных сетевых картах. Для каждого из решений выделена область применимости. Libpcap позволяет осуществлять прием пакетов на любых сетевых картах и операционных системах. PF_RING реализует перехват трафика на операционной системе Linux, а при поддержке драйвером сетевой карты может работать в режиме 0-copy. DPDK предоставляет наибольшую совместимость с решениями по программной обработке трафика. AF_XDP позволяет осуществлять прием пакетов без копирований на поддерживающих XDP сетевых картах, кроме того, все больше решений по обработке трафика реализуют его поддержку. Netmap – достаточно легковесный инструмент для перехвата трафика с возможностями расширения функционала.

Полученные при тестировании программных решений результаты отражают возможности современных программно-аппаратных средств в задачах перехвата сетевого трафика, поступающего на скорости порядка 40 Гбит/сек. В качестве дальнейшего развития работы предлагается разработка и тестирование интерфейсов захвата высокоскоростного сетевого трафика с применением фреймворков DPDK и Netmap.

## Список литературы / References


[1] D. Cerović, V. Del Piccolo et al. Fast Packet Processing: A Survey. IEEE Communications Surveys & Tutorials, vol. 20, no. 4, 2018, pp. 3645-3676.

[2] L. Linguaglossa, S. Lange et al. Survey of Performance Acceleration Techniques for Network Function Virtualization. Proceedings of the IEEE, vol. 107, no. 4, 2019, pp. 746-764.

[3] Sangjin Han, Keon Jang et al. Berkeley Extensible Software Switch (BESS). Technical Report No. UCB/EECS-2015-155, University of California at Berkeley, 2015, 17 p.

[4] DPDK (Data Plane Development Kit). URL: https://core.dpdk.org.

[5] ntop. URL: https://www.ntop.org/.

[6] T. Høiland-Jørgensen, J. D. Brouer et al. The eXpress data path: fast programmable packet processing in the operating system kernel. In Proc. of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18), 2018, pp. 54-66.

[7] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In Proc. of the USENIX Annual Technical Conference (USENIX ATC 12), 2012, pp. 101-112.

[8] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In Proc. of the Winter USENIX Conference (USENIX'93), 1993, 11 p.

[9] Ben Pfaff, Justin Pettit et al. The Design and Implementation of Open vSwitch. In Proc. of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015, pp. 117-130.

[10] László Molnár, Gergely Pongrácz et al. Dataplane Specialization for High-Performance OpenFlow Software Switching. In Proc. of the ACM SIGCOMM Conference (SIGCOMM '16), 2016, pp. 539-552.

[11] Sangjin Han, Keon Jang et al. PacketShader: A GPU-Accelerated Software Router. ACM SIGCOMM Computer Communication Review, vol. 40, issue 4, 2010, pp. 195–206.

[12] M. Trevisan, A. Finamore et al. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. IEEE Communications Magazine, vol. 55, no. 3, 2017, pp. 163-169.

[13] Robert Morris, Eddie Kohler et al. The Click Modular Router. In Proc. of the Seventeenth ACM Symposium on Operating Systems Principles, 1999, pp. 217-231.

[14] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In Proc. of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2015, pp. 5-16.

[15] FD.io. Vector Packet Processing - One Terabit Software Router on Intel Xeon Scalable Processor Family Server. White Paper. Cisco, Intel Corporation, FD.io, 2017.

[16] M. Paolino, N. Nikolaev et al. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In Proc. of the IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), 2015, pp. 86-92.

[17] A. Farshin, T. Barbette et al. PacketMill: Toward per-core 100-GBPS Networking. In Proc. of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021), 2021, pp. 1-17.







[18] Intel: 82599 10 GbE controller datasheet, 2019. http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[19] А.И. Гетьман, Е.Ф. Евстропов, Ю.В. Маркин. Анализ сетевого трафика в режиме реального времени: обзор прикладных задач, подходов и решений. Препринт ИСП РАН, вып. 28, 2015, стр. 1-52 / A.I. Getman, E.F. Evstropov, Yu.V. Markin. Analysis of network traffic in real time: an overview of applications, approaches and solutions. ISP RAS Preprint, no. 28, 2015, pp. 1-52 (in Russian).

[20] V. Moreno, J. Ramos et al. Commodity packet capture engines: tutorial cookbook and applicability. IEEE Communications Surveys & Tutorials, vol. 17, no. 3, 2015, pp. 1364-1390.

[21] G. Liao, X. Znu, L. Bnuyan. A new server I/O architecture for high speed networks. In Proc. of the Symposium on High-Performance Computer Architecture, 2011, pp. 255-265.

[22] S. Han, K. Jang, K.S. Park, S. Moon. PacketShader: a GPU-accelerated software router. ACM SIGCOMM Computer Communication Review, vol. 40, issue 4, 2010, pp. 195–206.

[23] W. Wu, P. DeMar, M. Crawford. Why can some advanced Ethernet NICs cause packet reordering? IEEE Communications Letters, vol. 15, issue 2, 2011, pp. 253-255.

[24] V. Moreno, P.M. Santiago del Río, J. Ramos, J.J. Garnica, J.L. García-Dorado. Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines. IEEE Communications Letters vol. 16, issue 11, 2012, pp. 1888–1891.

[25] pcap(3pcap). URL: https://www.tcpdump.org/manpages/pcap.3pcap.html.

[26] Wireshark. URL: https://www.wireshark.org/.

[27] Tcpdump. URL: https://www.tcpdump.org.

[28] Snort. URL: https://www.snort.org.

[29] L. Deri. Improving passive packet capture: Beyond device polling. In Proc. of the 4th International System Administration and Network Engineering Conference, 2004, 9 p.

[30] D.Scholz. A look at Intel's Dataplane Development Kit. In Proc. of the Seminars on Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), 2014, pp. 115-122.

[31] M. Karlsson and B. Töpel. The Path to DPDK Speeds for AF_XDP. In Proc. of the Linux Plumbers Conference, 2018, 9 p.


## Информация об авторах / Information about authors


Дмитрий Викторович ЛАРИН является специалистом кафедры системного программирования МФТИ. Его научные интересы включают телекоммуникационные сети, распределенные системы и программную обработку сетевого трафика.

Dmitry Victorovich LARIN is a specialist of the Department of system programming of Moscow Institute of Physics and Technology. His research interests include telecommunication networks, distributed systems and software processing of network traffic.

Александр Игоревич ГЕТЬМАН – старший научный сотрудник, кандидат физико-математических наук. Сфера научных интересов: анализ бинарного кода, восстановление форматов данных, анализ и классификация сетевого трафика.

Aleksandr Igorevich GETMAN – senior researcher, PhD in physical and mathematical sciences. Research interests: binary code analysis, data format recovery, network traffic analysis and classification.