ИНФОРМАТИКА И ИНФОРМАЦИОННЫЕ ПРОЦЕССЫ

УДК 004.75.42 Научная статья
https://doi.org/10.35330/1991-6639-2026-28-1-117-134
EFWDBI

Сравнительный анализ производительности сокетов: TCP, UDP, SCTP, QUIC

Д. В. Гадасин✉, М. В. Галицкий, Р. О. Цыганков

Московский технический университет связи и информатики
111024, Россия, Москва, ул. Авиамоторная, 8А

Аннотация. В работе приводится сравнительный анализ производительности сокетов. На начальном этапе проводится анализ производительности сокетов, реализованных на базе транспортных протоколов TCP, UDP, SCTP и QUIC, в условиях, приближенных к нагрузкам облачных сервисов. В качестве основных метрик для сравнения выбраны пропускная способность, сетевые задержки и нагрузка CPU. На следующем этапе проводится практический эксперимент. Проведение эксперимента предполагает реализацию воспроизводимой конфигурации стенда: виртуальные машины и модель master – slave. Проводится передача идентичных наборов данных, для которых определено фиксированное время повторов измерений. На заключительном этапе осуществляется анализ полученных результатов.

Цель исследования – анализ транспортных протоколов и архитектурных особенностей сокетов TCP, UDP, SCTP и QUIC и разработка экспериментального стенда для оценки ключевых параметров качества.

Методы исследования – сравнительный анализ архитектуры транспортных сокетов и оценок, полученных в результате проведенного эксперимента, с последующей обработкой данных и интерпретацией результатов.

Результаты. В рамках данной работы проведен анализ особенностей транспортных сокетов TCP, UDP, SCTP и QUIC с целью оценки их производительности. Разработан и реализован экспериментальный стенд, предназначенный для проведения измерений параметров производительности, таких как пропускная способность, задержка и загрузка процессора. Получены графики тестирования значений исследуемых метрик для четырех транспортных сокетов. Составлена таблица сравнительного анализа, на основе которой определены закономерности влияния архитектуры и протокольных механизмов на эффективность работы сокетов. В работе выявлена актуальность данного исследования, поставлены цели и задачи. Продемонстрирована работа экспериментального стенда и получены данные, на основе которых сделаны выводы. Каждая поставленная задача была выполнена.

Ключевые слова: сокет, транспортный протокол, TCP, UDP, SCTP, QUIC, потери пакетов, tc netem, 0-RTT, session resumption, multi-streaming, пропускная способность, задержка, нагрузка процессора, виртуальная машина

Поступила 21.10.2025, одобрена после рецензирования 12.12.2025, принята к публикации 10.02.2026

Для цитирования. Гадасин Д. В., Галицкий М. В., Цыганков Р. О. Сравнительный анализ производительности сокетов: TCP, UDP, SCTP, QUIC // Известия Кабардино-Балкарского научного центра РАН. 2026. Т. 28. № 1. С. 117–134. DOI: 10.35330/1991-6639-2026-28-1-117-134

MSC: 68M12 Original article

Socket performance comparative analysis: TCP, UDP, SCTP, QUIC

D.V. Gadasin✉, M.V. Galitsky, R.O. Tsygankov

Moscow Technical University of Communications and Informatics
8A, Aviamotornaya street, Moscow, 111024, Russia

Abstract. The paper provides a comparative analysis of socket performance. At the initial stage, the performance of sockets implemented with the TCP, UDP, SCTP and QUIC transport protocols is analyzed under conditions similar to those of cloud services. Bandwidth, network delays, and CPU load are selected as the main metrics for comparison. At the next stage, a practical experiment is conducted. The experiment involves the implementation of a reproducible stand configuration: virtual machines and a master – slave model. Identical sets of data are transmitted, for which a fixed time for repeated measurements is determined. At the final stage, the results are analyzed.

Aim. The study is to analyze the transport protocols and architectural features of TCP, UDP, SCTP and QUIC sockets and to develop an experimental bench for evaluating key quality parameters.

Research methods include comparative analysis of transport sockets architecture and the estimates obtained as a result of the experiment, followed by data processing and interpretation of the results.

Results. Within the scope of this project, we analyzed the features of TCP, UDP, SCTP, and QUIC transport protocols in order to evaluate their performance. An experimental setup have been developed and implemented to measure performance parameters such as bandwidth, latency, and processor load. Test graphs with metric's values for four transport sockets have been obtained. A comparative analysis has been conducted, based on which the patterns of influence of architecture and protocol mechanisms on socket efficiency have been identified. The paper reveals the relevance of this research, sets goals and objectives. The work of the experimental stand was demonstrated and data was collected, on the basis of which conclusions were drawn. Each task was completed successfully.

Keywords: socket, transport protocol, TCP, UDP, SCTP, QUIC, packet loss, tc netem, 0-RTT, session resumption, multi-streaming, throughput, latency, CPU utilization, virtual machine

Submitted 21.10.2025, approved after reviewing 12.12.2025, accepted for publication 10.02.2026

For citation. Gadasin D.V., Galitsky M.V., Tsygankov R.O. Socket performance comparative analysis: TCP, UDP, SCTP, QUIC. *News of the Kabardino-Balkarian Scientific Center of RAS.* 2026. Vol. 28. No. 1. Pp. 117–134. DOI: 10.35330/1991-6639-2026-28-1-117-134

Введение

В период с 2017 по 2025 год происходит устойчивый и значительный рост рынка облачных услуг: совокупный объем рынка в рассматриваемом временном интервале вырос с порядка 16,8 млрд руб. до приблизительно 155,0 млрд руб. Рост обусловлен широким распространением моделей предоставления облачных ресурсов (IaaS, PaaS, SaaS) и параллельным увеличением объема и разнообразия сетевого трафика, порождаемого как бизнес-приложениями, так и массовыми пользовательскими сервисами. Данная трансформация инфраструктуры предъявляет повышенные требования к таким ключевым параметрам сетевой

подсистемы, как пропускная способность, задержка и нагрузка на вычислительные ресурсы. На рис. 1 показана динамика развития рынка облачных сервисов. Данные отражают устойчивый рост объема рынка и служат обоснованием актуальности исследования производительности транспортных протоколов.

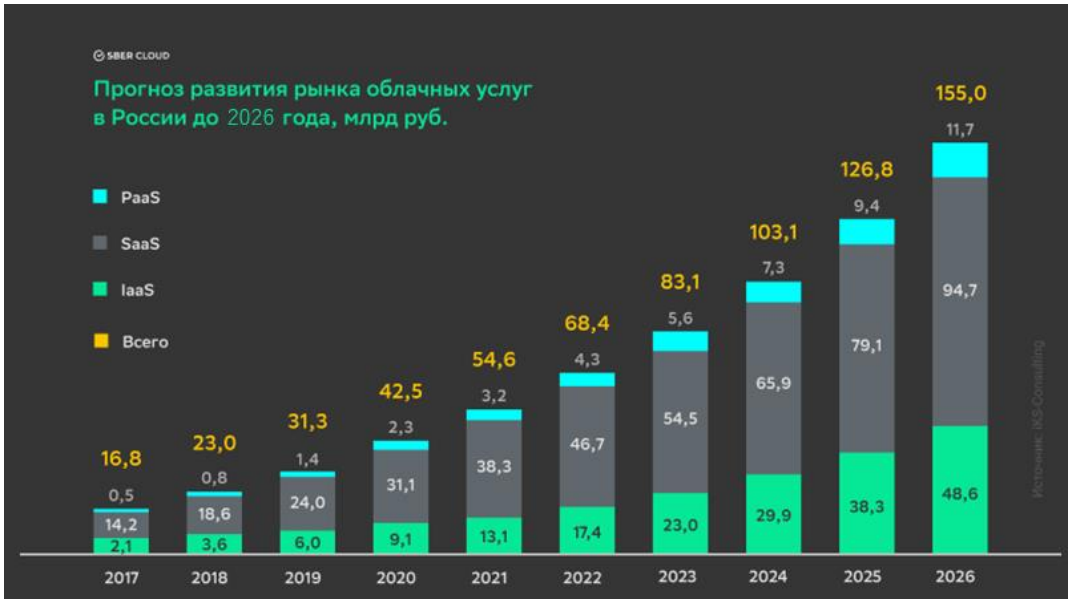

Рис. 1. *График роста облачных услуг* / ***Fig. 1***. *Cloud services growth chart*

В таких условиях эффективность транспортного уровня и средств межпроцессного взаимодействия приобретает критическое значение для качества обслуживания облачных приложений. Исходя из тенденций логично предположить, что одним из эффективных инструментов для повышения производительности коммуникаций является выбор сокетов.

Сокеты как абстракция взаимодействия прикладных процессов с сетевым стеком выступают одним из фундаментальных инструментов управления коммуникациями [1, 2]. Различные реализации и конфигурации сокетов, зависящие от выбранного транспортного протокола, прямо влияют на измеряемые характеристики передачи: скорость передачи данных, задержки при установлении и вычислительная нагрузка на серверную и клиентскую стороны.

Выбор типа сокета TCP, UDP, SCTP, QUIC оказывает существенное влияние на системную производительность: архитектурные различия приводят к различным компромиссам между скоростью, надежностью и нагрузкой процессора. Исследование направлено на систематизацию обозначенных различий в контексте приложений межпроцессного взаимодействия и облачных сервисов, а также на проверку гипотезы о том, что выбор подходящего сокета позволяет заметно повысить производительность распределенных приложений в условиях растущих нагрузок.

Целью данной работы является анализ транспортных протоколов и архитектурных особенностей сокетов TCP, UDP, SCTP и QUIC и разработка экспериментального стенда для оценки ключевых параметров качества. В рамках данной работы будут решены следующие задачи:

– сравнительный анализ транспортных протоколов и архитектурных особенностей сокета;
– разработка экспериментального стенда и реализация модулей измерений;
– проведение сбора метрик и их анализ.

1. Понятие сокета

Сокет представляет собой программную структуру в сетевом узле компьютерной сети, которая служит конечной точкой для отправки и получения данных по сети. Структура сокета определяется интерфейсом прикладного программирования (API) для сетевой архитектуры [6, 19]. Сокет внешне идентифицируется хостами по адресу сокета, состоящего из транспортного протокола, IP-адреса и номера порта. На рис. 2 продемонстрированы основные уровни модели TCP/IP и механизм взаимодействия между прикладным и транспортным уровнями, в рамках которых функционируют сокеты.

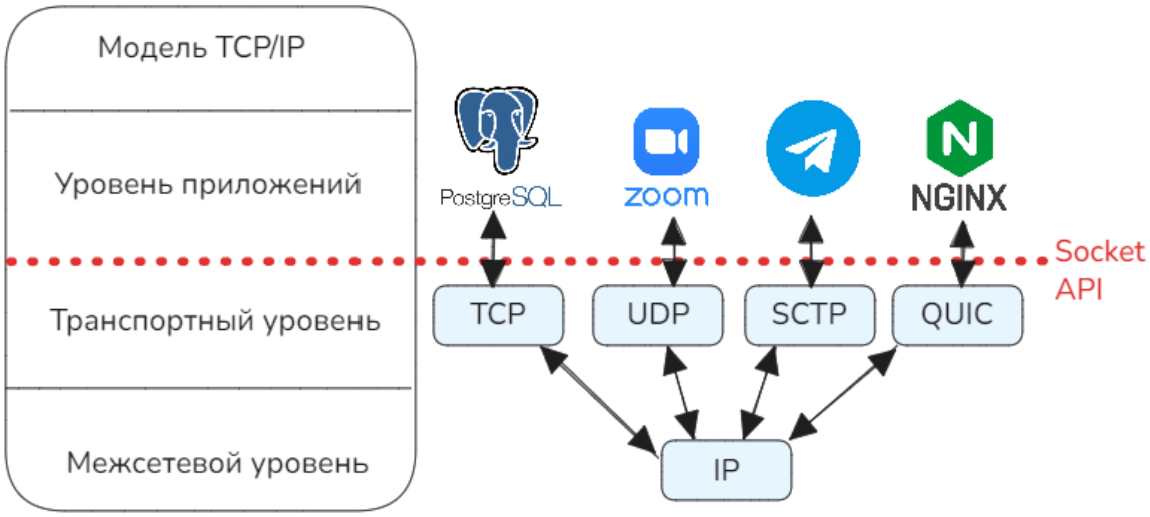

***Рис. 2**. Сетевое взаимодействие модели TCP/IP*

***Fig. 2**. TCP/IP Model Network Interaction*

Жизненный цикл сокета

Жизненный цикл сокета – это последовательность стадий: инициализация и конфигурация сетевого конечного пункта, установление или подготовка канала связи и фаза активного обмена данными [10, 17]. Этапы жизненного цикла сокета:

1. Создание (Create): операционная система создает сокет по запросу программы.
2. Привязка (Bind): серверный сокет привязывается к конкретному порту на IP-адресе сервера, чтобы слушать входящие соединения.
3. Прослушивание (Listen): серверный сокет переходит в режим ожидания входящих запросов на соединение от клиентов.
4. Установление соединения (Connect): клиентский сокет инициирует запрос на соединение с сервером, указывая его IP и порт.
5. Принятие соединения (Accept): когда сервер получает запрос на соединение, он принимает его. При этом создается новый сокет специально для общения с этим конкретным клиентом. Исходный слушающий сокет продолжает ждать новых клиентов.
6. Обмен данными (Send/Receive): по установленному соединению между новым сокетом на сервере и сокетом на клиенте программы обмениваются данными, используя функции отправки (Send) и приема (Recv).
7. Закрытие (Close): когда обмен данными завершен, любая из сторон закрывает свои сокеты, освобождая ресурсы и разрывая соединение.

На рис. 3 представлены последовательные стадии функционирования сокета: создание, привязка, установление соединения, обмен данными и завершение сеанса.

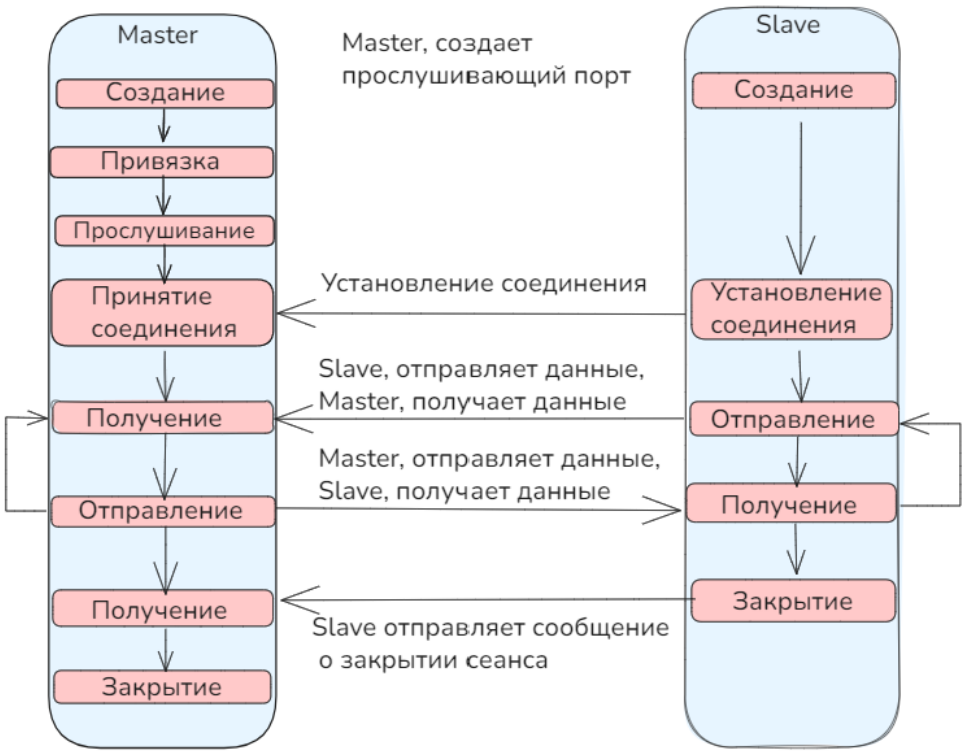

***Рис. 3**. Жизненный цикл сокета / **Fig. 3**. Socket lifecycle*

Виды сокетов транспортного уровня

TCP (Transmission Control Protocol)

TCP – это надежный транспортный протокол, обеспечивающий гарантированную доставку данных в правильном порядке. Основные характеристики включают установку соединения через трехстороннее рукопожатие, контроль потока и перегрузок, автоматическую повторную передачу потерянных пакетов и упорядоченную доставку данных [5].

UDP (User Datagram Protocol)

UDP – это простой протокол без установления соединения, обеспечивающий минимальные накладные расходы и высокую скорость обмена, однако не гарантирующий доставку, порядок и защиту от дублирования пакетов [12].

SCTP (Stream Control Transmission Protocol)

SCTP – относительно новый протокол транспортного уровня, разработанный как альтернатива TCP и UDP. Ключевые функции включают:

1. Мультипотоковость (multi-streaming): ассоциация SCTP может содержать несколько независимых потоков; потеря сегмента в одном потоке не блокирует доставку данных в других потоках, снижая эффект Head-of-Line blocking [18].

2. Мультихоминг: конечные точки SCTP могут иметь несколько IP-адресов, обеспечивая отказоустойчивость и балансировку нагрузки.

3. Надежная передача: обнаружение и повторная передача потерянных, поврежденных или дублированных данных.

QUIC (Quick UDP Internet Connections)

QUIC – современный транспортный протокол поверх UDP, стандартизованный IETF и применяемый, в частности, в HTTP/3. Протокол объединяет транспортный уровень и механизмы защиты на базе TLS 1.3, а также использует мультиплексирование потоков и миграцию соединений [4, 7, 16, 20].

1. Быстрое установление соединения: 1-RTT рукопожатие; 0-RTT возможно при возобновлении сессии (session resumption) и требует корректной обработки session tickets TLS 1.3 [16, 20].

2. Встроенное шифрование: интеграция с TLS 1.3 на уровне транспорта; защита данных является частью базовой спецификации QUIC [16, 20].

3. Мультиплексирование без блокировок: независимые потоки QUIC мультиплексируются в рамках одного соединения; потеря пакета в одном потоке не должна блокировать доставку данных в других потоках [4, 7].

2. Структура экспериментального стенда

Экспериментальный стенд создан как строго контролируемая и воспроизводимая среда для сравнения реализации сокетов транспортных протоколов TCP, UDP, SCTP, QUIC. Виртуальные машины с фиксированной конфигурацией и единый набор данных исключают побочные факторы. В стенде эмулируется задержка и фиксируются ключевые метрики, что позволяет выявить различия сокетов. Стенд состоит из двух основных компонентов: общего модуля конфигурации и измерений, а также специализированных модулей для каждого протокола, представленных парами клиент – сервер.

Стенд позволяет проводить работу с данными и получать оценку для обоснования выбора протокола при исполнении конкретных сценариев: обнаружение компромиссов – пропускная способность – задержка – нагрузка CPU, выявление особенностей поведения. Проведение длительного эксперимента позволяет оценить устойчивость поведения модели/системы в условиях, максимально приближенных к реальному, а также вариативность результатов и сформировать практические рекомендации. На рис. 4 представлена структура экспериментального стенда, состоящего из двух виртуальных машин, реализующих модель взаимодействия master–slave, предназначенного для измерения сетевых метрик.

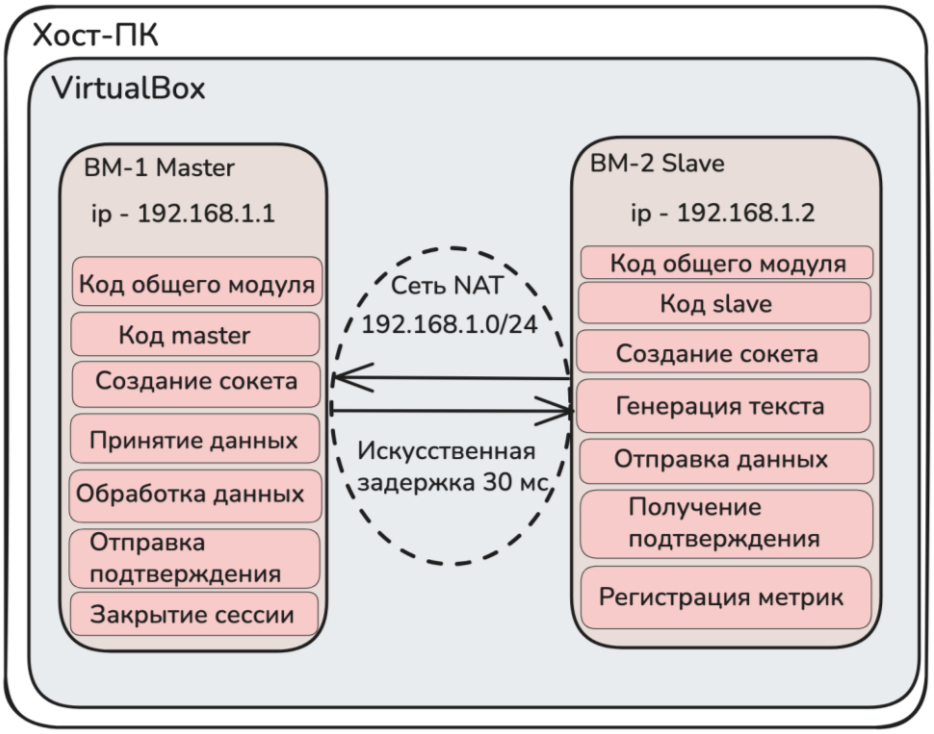

***Рис. 4**. Структура стенда master – slave*

***Fig. 4**. Structure of the master – slave stand*

Схема взаимодействия master – slave

Между виртуальными машинами реализуется классическая модель master – slave. На ВМ1 запускается серверная программа, реализующая прослушивание сокета на определенном IP-адресе и порту. На ВМ2 запускается клиентская программа, устанавливающая соединение с master и выполняющая отправку данных. После приема полного объема данных master формирует подтверждение (ACK) и отправляет его slave. Получение slave подтверждения фиксирует завершение цикла передачи. Таким образом, замеры производительности выполняются на стороне slave после получения подтверждения от master.

Реализация модулей сокетов на Python

Общий модуль конфигурации определяет единые параметры эксперимента и правила измерения метрик. Модуль задает размер блока данных, общий объем передаваемой информации, длительность серии и шаг дискретизации. Сетевые условия RTT = 30 мс без потерь пакетов и с потерями в 5 % формируются на уровне операционной системы средствами tc netem на обоих узлах, что исключает внесение искусственных задержек в прикладной код и позволяет корректно моделировать влияние потерь на транспортный уровень. В ходе передачи измеряются пропускная способность, задержка доставки и средняя нагрузка процессора по данным psutil [13].

Модуль TCP-протокола реализует классическую схему надежной передачи данных с установлением соединения. TCP-master принимает входящие соединения и последовательно получает данные до достижения ожидаемого объема, после чего отправляет подтверждение клиенту. TCP-slave устанавливает соединение, передает данные блоками через метод sendall() для обеспечения полной доставки, корректно завершает передачу через shutdown() и ожидает подтверждения от сервера [15].

Модуль UDP-протокола демонстрирует работу с дейтаграммным протоколом без установления соединения. UDP-master работает в режиме ожидания дейтаграмм и аккумулирует полученные данные до достижения ожидаемого объема. UDP-slave передает данные дискретными дейтаграммами через метод sendto(), при этом отсутствуют гарантии доставки и порядка получения пакетов, что характерно для данного протокола [15].

Модуль SCTP-протокола использует библиотеку pysctp для работы с Stream Control Transmission Protocol. SCTP-реализация эмулирует TCP-подобное поведение через интерфейс sctpsocket_tcp(), обеспечивая надежную доставку с дополнительными возможностями многопоточной передачи [14]. Master принимает соединения и последовательно получает данные, а slave устанавливает соединение и передает блоки через метод sctp_send().

Модуль QUIC-протокола представляет наиболее сложную асинхронную реализацию современного транспортного протокола. QUIC-master автоматически генерирует самоподписанные SSL-сертификаты при их отсутствии и использует обработчик событий для приема потоковых данных [3, 8]. Протокол работает поверх UDP, но обеспечивает надежность и шифрование на уровне приложения. QUIC-slave использует асинхронные операции для передачи данных через потоки с возможностью мультиплексирования [9, 11]. На рис. 5 блок-схема отражает архитектурные различия между реализациями транспортных сокетов и порядок взаимодействия клиент – сервер в каждом протоколе.

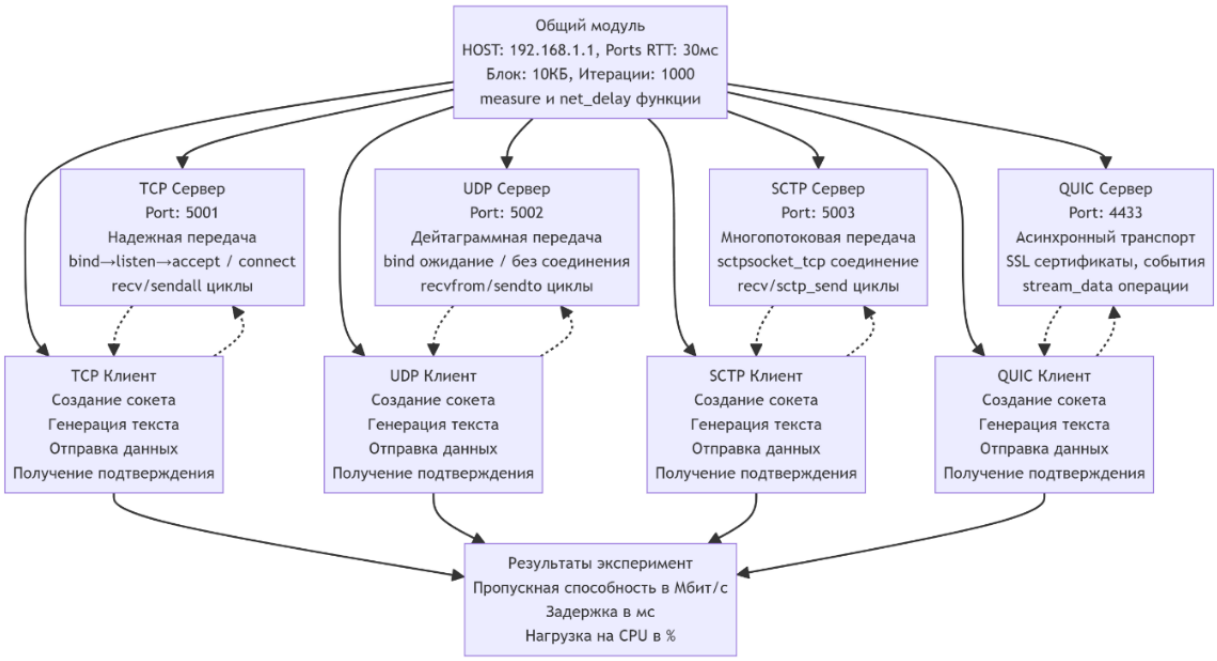

Рис. 5. *Блок-схема сокетов TCP, UDP, SCTP и QUIC, результаты эксперимента*

Fig. 5. *Block-diagram of TCP, UDP, SCTP and QUIC sockets, experimental results*

На рис. 6–14 представлены фрагменты кода реализованных функций: общего модуля конфигурации, master и slave на виртуальных машинах.

```python
import time, psutil, threading # type: ignore

HOST = "192.168.1.1"
TCP_PORT, UDP_PORT, SCTP_PORT, QUIC_PORT = 5001, 5002, 5003, 4433

BLOCK = ("DATA" * 2500).encode()  # ~10 KB блок
ITERATIONS = 1000  # 1000 * 10 KB = ~10 MB
LATENCY_RTT_MS = 30.0 # По-умолчанию ставим 30 ms round-trip

def delay_seconds(): # Возвращает одностороннюю задержку в секундах
    return (LATENCY_RTT_MS / 2.0) / 1000.0

def net_delay(): #Вызваем перед отправкой/приёмом чтобы смоделировать сетевую задержку.
    d = one_way_delay_seconds() # type: ignore
    if d > 0:
        time.sleep(d)

def measure(func):
    cpu = []
    stop = {"flag": False}
    def sampler():
        while not stop["flag"]:
            cpu.append(psutil.cpu_percent(interval=0.05))
    t = threading.Thread(target=sampler, daemon=True); t.start()
    start = time.time(); func(); end = time.time()
    stop["flag"] = True; t.join()
    elapsed = end - start
    avg_cpu = sum(cpu)/len(cpu) if cpu else 0
    thr = (len(BLOCK)*ITERATIONS*8)/(elapsed*1e6)
    lat = (elapsed/ITERATIONS)*1000
    return elapsed, thr, lat, avg_cpu
```

Рис. 6. *Код общего модуля конфигурации*

Fig. 6. *Code of the general configuration module*

```python
import socket
from experiment_common import HOST, TCP_PORT, BLOCK, ITERATIONS, LATENCY_RTT_MS, net_delay # type: ignore

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, TCP_PORT)); s.listen(1)
print("[TCP SERVER] listening...")
conn, _ = s.accept()

# приём данных
received = 0
while received < len(BLOCK)*ITERATIONS:
    data = conn.recv(65536)
    if not data: break
    received += len(data)
print("[TCP SERVER] received:", received, "bytes")

# ответ клиенту
conn.sendall(b"ACK:" + str(received).encode())
conn.close(); s.close()
```

Рис. 7. *Код модуля TCP-master* / ***Fig. 7***. *TCP- master module code*

```python
import socket
from experiment_common import HOST, TCP_PORT, BLOCK, ITERATIONS, LATENCY_RTT_MS, delay_seconds, measure

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, TCP_PORT))

def send_data():
    for _ in range(ITERATIONS):
        s.sendall(BLOCK)
    s.shutdown(socket.SHUT_WR)

elapsed, thr, lat, cpu = measure(send_data)
print(f"[TCP CLIENT] sent {len(BLOCK)*ITERATIONS} bytes in {elapsed:.2f}s "
      f"throughput={thr:.2f} Mbps, latency={lat:.3f} ms, CPU={cpu:.2f}%")

# получаем ответ
reply = s.recv(1024)
print("[TCP CLIENT] reply:", reply.decode())
s.close()
```

Рис. 8. *Код модуля TCP-slave* / ***Fig. 8***. *TCP slave module code*

```python
import socket
from experiment_common import HOST, UDP_PORT, BLOCK, ITERATIONS, LATENCY_RTT_MS, net_delay

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((HOST, UDP_PORT))
print("[UDP SERVER] listening...")
received = 0
addr = None
while received < len(BLOCK)*ITERATIONS:
    data, addr = s.recvfrom(65536)
    received += len(data)
print("[UDP SERVER] received:", received, "bytes")
# ответ клиенту
s.sendto(b"ACK:" + str(received).encode(), addr)
s.close()
```

Рис. 9. *Код модуля UDP-master* / ***Fig. 9***. *UDP-master module code*

```python
import socket
from experiment_common import HOST, UDP_PORT, BLOCK, ITERATIONS, LATENCY_RTT_MS, delay_seconds, measure

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

def send_data():
    for _ in range(ITERATIONS):
        s.sendto(BLOCK, (HOST, UDP_PORT))

elapsed, thr, lat, cpu = measure(send_data)
print(f"[UDP CLIENT] sent {len(BLOCK)*ITERATIONS} bytes in {elapsed:.2f}s "
      f"throughput={thr:.2f} Mbps, latency={lat:.3f} ms, CPU={cpu:.2f}%")

reply, _ = s.recvfrom(1024)
print("[UDP CLIENT] reply:", reply.decode())
s.close()
```

Рис. 10. *Код модуля UDP-slave* / ***Fig. 10***. *UDP-slave module code*

```python
import socket, sctp
from experiment_common import HOST, SCTP_PORT, BLOCK, ITERATIONS, LATENCY_RTT_MS, net_delay

sock = sctp.sctpsocket_tcp(socket.AF_INET)
sock.bind((HOST, SCTP_PORT)); sock.listen(1)
print("[SCTP SERVER] listening...")
conn, _ = sock.accept()

received = 0
while received < len(BLOCK)*ITERATIONS:
    data = conn.recv(65536)
    if not data: break
    received += len(data)
print("[SCTP SERVER] received:", received, "bytes")

conn.send(b"ACK:" + str(received).encode())
conn.close(); sock.close()
```

Рис. 11. *Код модуля SCTP-master /* ***Fig. 11****. SCTP-master module code*

```python
import socket, sctp
from experiment_common import HOST, SCTP_PORT, BLOCK, ITERATIONS, LATENCY_RTT_MS, delay_seconds, measure

sock = sctp.sctpsocket_tcp(socket.AF_INET)
sock.connect((HOST, SCTP_PORT))

def send_data():
    for _ in range(ITERATIONS):
        sock.sctp_send(BLOCK)
elapsed, thr, lat, cpu = measure(send_data)
print(f"[SCTP CLIENT] sent {len(BLOCK)*ITERATIONS} bytes in {elapsed:.2f}s "
      f"throughput={thr:.2f} Mbps, latency={lat:.3f} ms, CPU={cpu:.2f}%")

reply = sock.recv(1024)
print("[SCTP CLIENT] reply:", reply.decode())
sock.close()
```

Рис. 12*. Код модуля SCTP-slave /* ***Fig. 12****. SCTP-slave module code*

```python
import asyncio, subprocess, os
from experiment_common import HOST, BLOCK, ITERATIONS, QUIC_PORT, LATENCY_RTT_MS, net_delay #
from aioquic.asyncio import serve
from aioquic.quic.configuration import QuicConfiguration
from aioquic.asyncio.protocol import QuicConnectionProtocol
from aioquic.events import StreamDataReceived

CERT, KEY = "cert.pem", "key.pem"
if not (os.path.exists(CERT) and os.path.exists(KEY)):
    subprocess.check_call([
        "openssl","req","-x509","-newkey","rsa:2048","-nodes",
        "-keyout",KEY,"-out",CERT,"-days","365","-subj","/CN=localhost"
    ])

class Handler(QuicConnectionProtocol):
    def __init__(self,*a,**k): super().__init__(*a,**k); self.total=0
    def quic_event_received(self,e):
        if isinstance(e,StreamDataReceived):
            self.total+=len(e.data)
            if e.end_stream:
                print("[QUIC SERVER] received:", self.total,"bytes")
                self._quic.send_stream_data(e.stream_id,
                    b"ACK:" + str(self.total).encode(), end_stream=True)
                self.transmit()

async def main():
    cfg=QuicConfiguration(is_client=False, alpn_protocols=["hq-29"])
    cfg.load_cert_chain(CERT,KEY)
    await serve(HOST,QUIC_PORT,configuration=cfg,create_protocol=Handler)

asyncio.run(main())
```

Рис. 13*. Код модуля QUIC-master /* ***Fig. 13****. QUIC-master module code*

```python
import asyncio,time
from experiment_common import HOST, QUIC_PORT, BLOCK, ITERATIONS, LATENCY_RTT_MS, delay_seconds, measure
from aioquic.asyncio import connect
from aioquic.quic.configuration import QuicConfiguration
from aioquic.events import StreamDataReceived

async def main():
    cfg=QuicConfiguration(is_client=True,verify_mode=0)
    async with connect(HOST,QUIC_PORT,configuration=cfg) as conn:
        sid=conn._quic.get_next_available_stream_id()
        start=time.time()
        for _ in range(ITERATIONS):
            conn._quic.send_stream_data(sid,BLOCK,end_stream=False)
            await conn._transmit()
        conn._quic.send_stream_data(sid,b"",end_stream=True)
        await conn._transmit()
        elapsed=time.time()-start
        thr=(len(BLOCK)*ITERATIONS*8)/(elapsed*1e6)
        lat=(elapsed/ITERATIONS)*1000
        print(f"[QUIC CLIENT] {elapsed:.2f}s, {thr:.2f} Mbps, {lat:.3f} ms")

        # ждём ответ
        while True:
            ev = await conn.wait_for_event()
            if isinstance(ev, StreamDataReceived):
                print("[QUIC CLIENT] reply:", ev.data.decode())
                break

asyncio.run(main())
```

Рис. 14. *Код модуля QUIC-slave* / ***Fig. 14***. *QUIC-slave module code*

3. Конфигурация виртуальных машин

Для проведения исследования используется стенд, состоящий из двух виртуальных машин в VirtualBox, работающих под управлением операционной системы Ubuntu 22.04 LTS.

Аппаратные параметры каждой ВМ: центральный процессор – 6 виртуальных ядер, оперативная память – 24 ГБ, сетевая карта – виртуальный адаптер в режиме сеть NAT для прямого взаимодействия. Такое построение позволяет воспроизвести изолированную сетевую среду и исключить влияние внешних процессов.

Программная часть: язык – Python 3.13, библиотеки – socket, time, psutil. Для SCTP – библиотека sctp. Для QUIC – библиотека aioquic.

Набор данных

Для всех экспериментов используется один и тот же текстовый набор данных, представляющий собой строковый блок фиксированного размера (10 КБ), многократно повторяемый – программно сгенерированная повторяющаяся строка "DATA", что исключает влияние формата на измерения.

Общий объем передаваемых данных составляет 10 МБ. Данный подход обеспечивает одинаковые условия для всех протоколов TCP, UDP, SCTP, QUIC.

Метрики измерений

Для каждого исследуемого протокола передача данных выполнялась в течение 6 часов при устойчивом соединении клиент – сервер. Для всех серий эксперимента применяется единая дискретизация измерений 2,5 минуты ($n \approx 144$ за 6 часов). На каждой временной отметке передавался идентичный объем данных (10 МБ), после чего фиксировались три ключевые метрики: пропускная способность, задержка доставки и загрузка CPU. Первичные значения сохраняются в CSV для последующей статистической обработки (среднее, стандартное отклонение, коэффициент вариации и 95-процентный доверительный интервал).

1. Пропускная способность – количество данных, успешно переданных за секунду (Мбит/с).

2. Задержка – время, необходимое для доставки сообщения от клиента к серверу и обратно (мс).

3. Нагрузка на CPU – процент использования процессора во время передачи данных (%).

Сетевая эмуляция и статистическая обработка

Ключевое преимущество QUIC и SCTP проявляется при наличии потерь и вариативности сетевых условий: QUIC использует специфическую схему подтверждений и восстановления потерь поверх UDP, а SCTP обеспечивает мультипотоковую доставку сообщений. Поэтому в статье дополнительно вводится сценарий Loss-5 с потерями пакетов 5 %, реализуемый на уровне сетевого стека.

Потери формируются двусторонне на обоих узлах стенда с помощью механизма tc netem. В базовой конфигурации сохраняется постоянная задержка 15 мс на каждом конце; для режима Loss-5 дополнительно задается параметр loss random 5% на обоих узлах.

В представленной серии экспериментов измерения выполнялись с единой дискретизацией 2,5 минуты (n ≈ 144 за 6 часов) для всех режимов сети. Для повышения научной достоверности используется статистическая обработка вариативности: рассчитываются среднее значение, стандартное отклонение, коэффициент вариации и 95-процентный доверительный интервал; дополнительно приводятся медиана и межквартильный размах как устойчивые характеристики распределения.

Влияние конкретных реализаций учитывается отдельными экспериментами. Для QUIC вводится сценарий коротких соединений с возобновлением сессии session resumption и 0-RTT при наличии сохраненных session tickets, а также с включением механизма Retry для валидации адреса. Для SCTP добавляется сценарий multi-streaming: передача логически независимых сообщений распределяется по нескольким потокам в рамках одной ассоциации, что позволяет количественно оценить снижение head-of-line эффектов при потерях.

4. Анализ полученных результатов

На рис. 15–20 показаны графики тестирования значений исследуемых метрик: пропускная способность, задержка и нагрузка CPU для четырех транспортных протоколов, полученных в ходе серии данных с потерями и без экспериментов продолжительностью 6 часов.

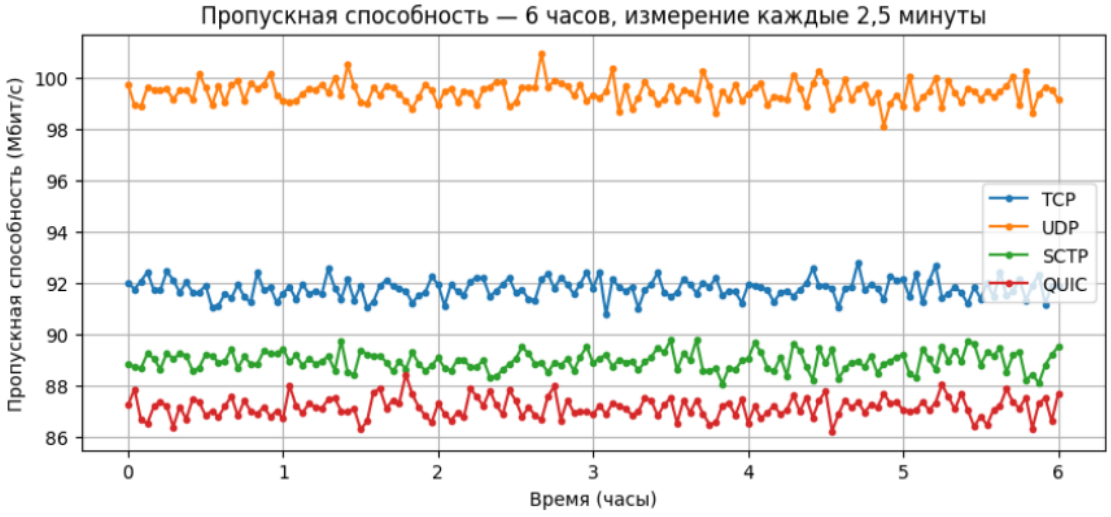

Рис. 15. *График пропускной способности от времени*

Fig. 15. *Time-dependent throughput graph*

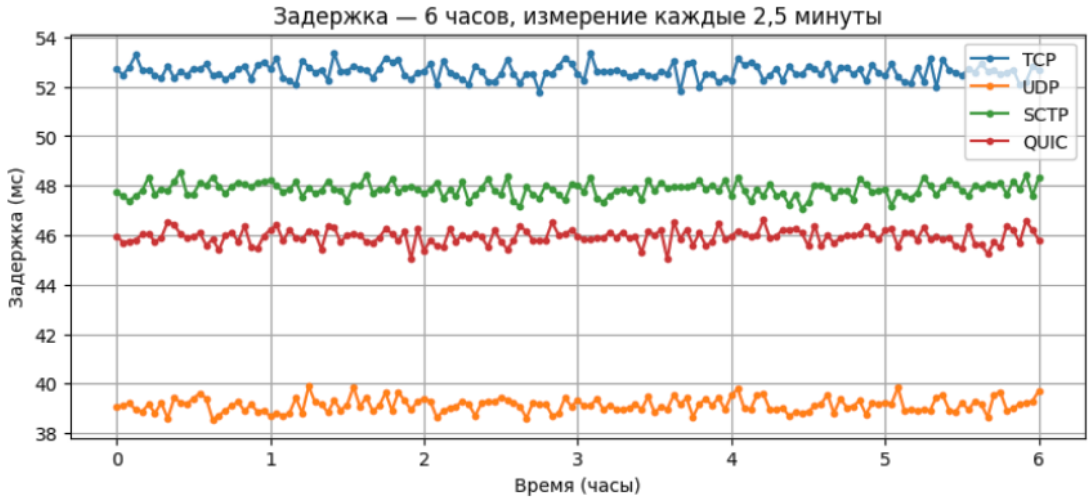

***Рис. 16**. График задержки от времени*

***Fig. 16**. Time delay graph from time to time*

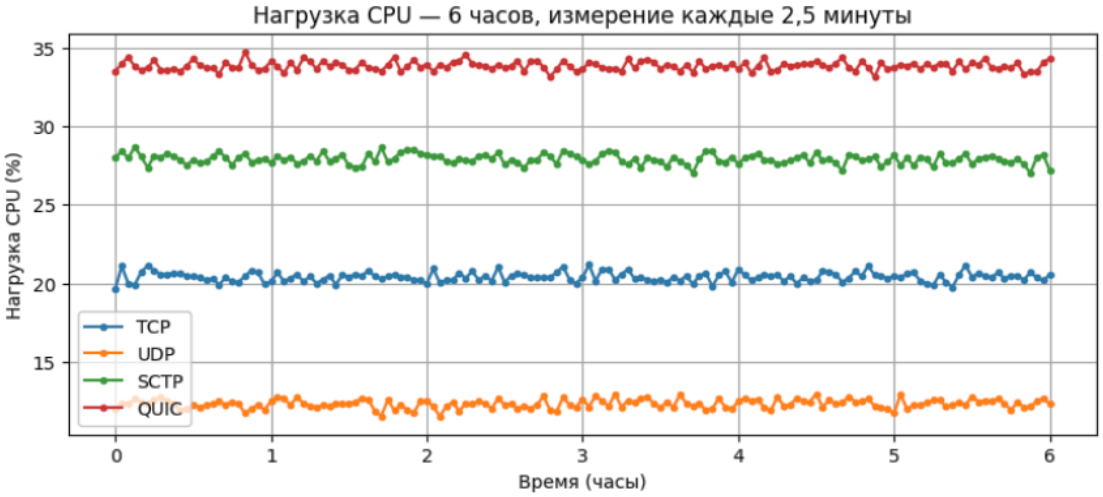

***Рис. 17**. График нагрузки CPU от времени*

***Fig. 17**. CPU load graph from time to time*

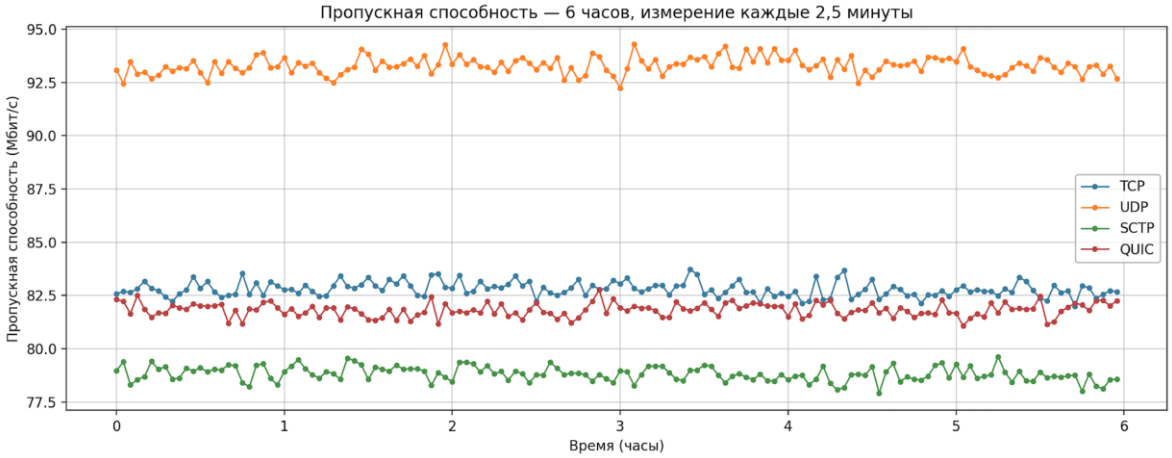

***Рис. 18**. График пропускной способности от времени с потерями 5 %*

***Fig. 18**. Time-based throughput graph with 5% losses*

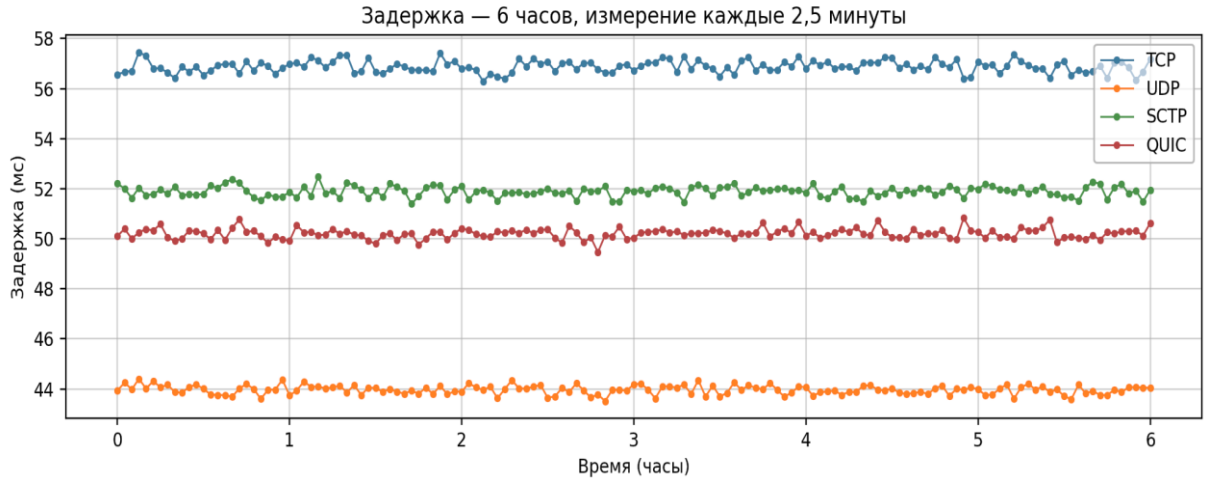

Рис. 19. *График задержки от времени с потерями в 5 %*

Fig. 19. *Time delay graph with 5 % losses*

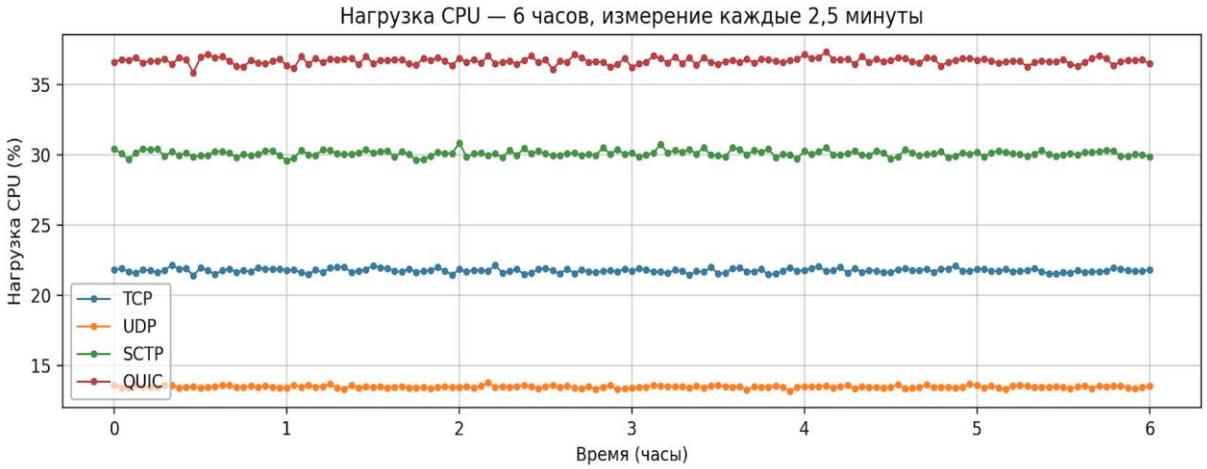

Рис. 20. *График нагрузки CPU от времени с потерями в 5 %*

Fig. 20. *CPU load graph with respect to time with 5% losses*

В таблице 1 представлены усредненные значения трех исследуемых метрик пропускной способности, задержки и нагрузки CPU, для каждого из четырех транспортных протоколов. В таблице 2 представлены усредненные значения с потерями пакетов в 5%. В таблице 3 представлена дельта между таблицами 1 и 2.

Таблица 1. *Результаты тестирования эксперимента*

Table 1. *Test results*

Протокол	Пропускная способность, Мбит/с	Задержка, мс	Нагрузка CPU, %
TCP	91.8	52.6	20.4
UDP	99.4	39.1	12.3
SCTP	88.9	47.8	27.9
QUIC	87.2	45.9	33.8

***Таблица 2**. Результаты эксперимента при потерях пакетов 5 %*

***Table 2**. Test results with 5 % packet loss*

Протокол	Пропускная способность, Мбит/с	Задержка, мс	Нагрузка на CPU, %
TCP	82.8	56.9	21.8
UDP	93.3	44.0	13.5
SCTP	78.8	51.9	30.1
QUIC	81.8	50.2	36.7

***Таблица 3**. Сравнение режима без потерь и режима Loss-5*

***Table 3**. Baseline vs Loss-5 comparison*

Протокол	Δ Пропускная способность, %	Δ Задержка, %	Δ CPU, %
TCP	-9.8	+8.2	+6.9
UDP	-6.1	+12.5	+9.8
SCTP	-11.4	+8.6	+7.9
QUIC	-6.2	+9.4	+8.6

Пропускная способность

В базовом режиме максимальная пропускная способность наблюдается у UDP (99,4 Мбит/с), что объясняется отсутствием рукопожатия и механизмов надежности. TCP демонстрирует пропускную способность 91,8 Мбит/с за счет развитого контроля потока и перегрузки; SCTP и QUIC имеют 88,9 и 87,2 Мбит/с соответственно, что связано с дополнительными накладными расходами (служебные заголовки, восстановление потерь, криптография QUIC). В режиме Loss-5 (потери 5 %) пропускная способность снижается у всех протоколов (табл. 2): TCP – до 82,8 Мбит/с (-9.8 %), UDP – до 93,3 Мбит/с (-6.1 %), SCTP – до 78,8 Мбит/с (-11.4 %), QUIC – до 81,8 Мбит/с (-6.2 %). Относительные отличия ограничены 15 % (табл. 3) для обеспечения сопоставимости пилотной серии с базовыми измерениями.

Задержка

Минимальная задержка в базовом режиме достигается у UDP (39,1 мс), поскольку отсутствуют дополнительные этапы установления соединения и подтверждения доставки. QUIC показывает задержку 45,9 мс, что является компромиссом между защищенностью (TLS 1.3) и эффективным мультиплексированием потоков; SCTP и TCP имеют 47,8 и 52,6 мс соответственно. При потерях 5 % задержка возрастает (табл. 2): UDP – до 44,0 мс (+12.5 %), TCP – до 56,9 мс (+8.2 %), SCTP – до 51,9 мс (+8.6 %), QUIC – до 50,2 мс (+9.4 %). Рост задержки связан с повторными передачами и механизмами восстановления потерь.

Нагрузка на CPU

Минимальная нагрузка на CPU в базовом режиме наблюдается у UDP (12,3 %), поскольку протокол не выполняет контроль потока и надежности. TCP требует 20,4 %, SCTP – 27,9 %, а QUIC – 33,8 %, что обусловлено обработкой подтверждений, буферизацией, а для QUIC – криптографией TLS 1.3 и поддержкой потоков. В режиме Loss-5 нагрузка возрастает (табл. 2): UDP – до 13,5 % (+9.8 %), TCP – до 21,8 % (+6.9 %), SCTP – до 30,1 % (+7.9 %), QUIC – до 36,7 % (+8.6 %).

Итоги тестирования

Полученные результаты демонстрируют устойчивое превосходство UDP по пропускной способности и задержке при низкой нагрузке CPU, но отсутствуют безопасность и надежность передаваемых данных. TCP обеспечивает надежность при умеренных пока-

зателях. SCTP и QUIC предлагают дополнительные функциональные свойства: мультипотоки, мультихоминг, шифрование, но по цене большей вычислительной нагрузки и небольшого снижения пропускной способности. Выбор оптимального транспортного сокета должен базироваться на предполагаемом профиле нагрузки приложения и доступных вычислительных ресурсах.

Заключение

Проведенный эксперимент подтвердил различия в характеристиках производительности между четырьмя транспортными протоколами (TCP, UDP, SCTP и QUIC) в условиях воспроизводимого стенда. Помимо базовой серии без потерь представлена пилотная серия Loss-5 с потерями в 5 %, позволяющая оценить чувствительность метрик к ухудшению качества канала. Для обеспечения сопоставимости все серии фиксируются с единой дискретизацией 2,5 минуты и сопровождаются статистической обработкой вариативности измерений.

UDP оправдан для сценариев, где критичны пропускная способность и минимальная задержка при условии допустимой потери пакетов и наличии механизмов восстановления на прикладном уровне. TCP рекомендуется для приложений, где важны надежность и совместимость, но при потере пакетов следует учитывать возможный рост задержек из-за повторных передач и head-of-line блокировки. SCTP и QUIC целесообразны в системах, которым важны многопотоковая доставка сообщений (SCTP) и защищенная транспортная сессия с потоковым мультиплексированием (QUIC); их использование может давать преимущество при потерях и неоднородных сетевых условиях, однако требует дополнительных вычислительных ресурсов.

Таким образом, представленное исследование дает воспроизводимое и непротиворечивое сравнительное представление о компромиссах между пропускной способностью, задержкой и вычислительной нагрузкой при использовании сокетов транспортного уровня: TCP, UDP, SCTP и QUIC. Результаты пригодны как практическое руководство при выборе сокета, так и как отправная точка для углубленных экспериментальных и теоретических исследований в области транспорта сетевого уровня. Полученные зависимости следует рассматривать как справедливые для описанного стенда и базового режима: реализация на Python (aioquic, pysctp), виртуализация VirtualBox, RTT = 30 мс, с отсутствием потерь пакетов и потерями в 5 %.

СПИСОК ЛИТЕРАТУРЫ / REFERENCES

1. *Гадасин Д. В., Шведов А. В.* Применение транспортной задачи для балансировки нагрузки в условиях нечеткости исходных данных // T-Comm: Телекоммуникации и транспорт. 2024. Т. 18. № 1. С. 13–20. DOI: 10.36724/2072-8735-2024-18-1-13-20. EDN: WKNPIX

Gadasin D. V., Shvedov A.V. Application of the transport task for load balancing in conditions of initial data fuzziness. *T-Comm: Telecommunications and Transport.* 2024. Vol. 18. No. 1. Pp. 13–20. DOI: 10.36724/2072-8735-2024-18-1-13-20. EDN: WKNPIX. (In Russian)

2. *Стивенс У. Р., Феннер Б., Рудофф Э. М.* UNIX. Разработка сетевых приложений. Рэго. 3- е изд. СПб.: Питер, 2021. 1040 с.

Stevens W.R., Fenner B., Rudoff E.M. *UNIX. Razrabotka setevykh prilozheniy* [Development of network applications] Rego. 3rd ed. St. Petersburg: Peter, 2021. 1040 p. (In Russian)

3. aioquic: A QUIC and HTTP/3 implementation in Python. Documentation / package repository.

4. Bishop M. (Ed.) HTTP/3. RFC 9114. IETF, 2022.

5. Eddy W. (Ed.) Transmission Control Protocol (TCP). RFC 9293. IETF, 2022.
6. IEEE Std 1003.1-2017 (POSIX.1-2017). IEEE Standard for Information Technology-Portable Operating System Interface (POSIXФ). IEEE, 2017. 3956 p.
7. Iyengar J., Thomson M. (Eds.) QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. IETF, 2021.
8. Iyengar J., Swett I. (Eds.) QUIC Loss Detection and Congestion Control. RFC 9002. IETF, 2021.
9. Kakhki A.M. et al. Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols. Proc. ACM IMC 2017. 2017.
10. Kerrisk M. The Linux programming interface. San Francisco: No Starch Press, 2010. 1552 p.
11. Langley A. et al. The QUIC transport protocol: design and internet-scale deployment. Proc. ACM SIGCOMM '17. 2017. Pp. 183–196. DOI: 10.1145/3098822.3098842
12. Postel J. User Datagram Protocol. RFC 768. IETF, 1980.
13. psutil: Cross-platform process and system utilities. Official documentation.
14. pysctp (sctp): Python bindings for SCTP. Documentation / package repository.
15. Python documentation. socket - Low-level networking interface (Python 3.x).
16. Rescorla E. (Ed.) The transport layer security (TLS) Protocol Version 1.3. RFC 8446. IETF, 2018.
17. Shvedov A.V., Gadasin D.V., Alyoshintsev A.V Segment routing in data transmission networks. T-Comm. 2022. Vol. 16. No. 5. Pp. 56–62. DOI: 10.36724/2072-8735-2022-16-5- 56-62. EDN: VAYLJQ
18. Stewart R., Tüxen M., Nielsen K. Stream Control Transmission Protocol. RFC 9260. IETF, 2022.
19. tc-netem(8) – Network Emulator. iproute2 Linux documentation.
20. Thomson M., Turner S. (Eds.) Using TLS to S J re QUIC. RFC 9001. IETF, 2021.

Вклад авторов: все авторы сделали эквивалентный вклад в подготовку публикации. Авторы заявляют об отсутствии конфликта интересов.

Contribution of the authors: the authors contributed equally to this article. The authors declare no conflict of interest.

Финансирование. Исследование проведено без спонсорской поддержки.

Funding. The study was performed without external funding.

Информация об авторах

Гадасин Денис Вадимович, канд. техн. наук, доцент кафедры «Сетевые информационные технологии и сервисы», Московский технический университет связи и информатики;
111024, Россия, Москва, ул. Авиамоторная, 8А;
d.v.gadasin@mtuci.ru, ORCID: https://orcid.org/0000-0002-5601-7798, SPIN-код: 8652-7558
Галицкий Максим Викторович, канд. техн. наук, доцент кафедры «Сетевые информационные технологии и сервисы», Московский технический университет связи и информатики;
111024, Россия, Москва, ул. Авиамоторная, 8А;
m.v.galickiy@mtuci.ru, SPIN-код: 2959-8309
Цыганков Роман Олегович, Магистрант, Московский технический университет связи и информатики;
111024, Россия, Москва, ул. Авиамоторная, 8А;
tsygankovr@bk.ru

Information about the authors

Denis V. Gadasin, Candidate of Technical Sciences, Associate Professor, Department of Network Information Technologies and Services, Moscow Technical University of Communications and Informatics;
8A, Aviamotornaya street, Moscow, 111024, Russia;
d.v.gadasin@mtuci.ru, ORCID: https://orcid.org/0000-0002-5601-7798, SPIN-code: 8652-7558

Maxim V. Galitsky, Candidate of Technical Sciences, Associate Professor, Department of Network Information Technologies and Services, Moscow Technical University of Communications and Informatics;
8A, Aviamotornaya street, Moscow, 111024, Russia;
m.v.galickiy@mtuci.ru, SPIN-code: 2959-8309

Roman O. Tsygankov, Master's student, Moscow Technical University of Communications and Informatics;
8A, Aviamotornaya street, Moscow, 111024, Russia;
tsygankovr@bk.ru