

**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технологический университет «СТАНКИН»**

**(ФГБОУ ВО «МГТУ «СТАНКИН»)**

|  |  |
| --- | --- |
| **Институт**  **информационных систем**  **и технологий** | **Кафедра**  **информационных систем** |

**КУРСОВАЯ РАБОТА**

по дисциплине «**Объектно-ориентированное программирование**»

Тема: «Сравнение реализации протокола TCP на языке C++ и Go»

|  |  |
| --- | --- |
| Студент  группы ИДБ-23-08 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_  **Борисенко В.Д.**  подпись |
| Руководитель  Ст. преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_  **Бельченко Ф.М.**  подпись |

**Москва**

**2025 г.**

[**КУРСОВАЯ РАБОТА** 1](#_Toc1)

[Обзор сетевых моделей: OSI и TCP/IP 4](#_Toc2)

[Протокол Transmission Control Protocol (TCP) 6](#_Toc3)

[Основные характеристики TCP 6](#_Toc4)

[Соединение (Connection-Oriented) 6](#_Toc5)

[Надежность (Reliability) 7](#_Toc6)

[Потоковость (Stream-Oriented) 8](#_Toc7)

[Дуплексность (Full-Duplex) 9](#_Toc8)

[Структура сегмента TCP 9](#_Toc9)

[Управление соединением: Трехстороннее и Четырехстороннее рукопожатие 12](#_Toc10)

[Трехстороннее рукопожатие (3WHS) для установления соединения 12](#_Toc11)

[Четырехстороннее рукопожатие для завершения соединения 13](#_Toc12)

[Механизмы надежности TCP: Подтверждения и Ретрансмиссия 15](#_Toc13)

[Положительное подтверждение с повторной передачей (PAR) 15](#_Toc14)

[Быстрая повторная передача (Fast Retransmit) 16](#_Toc15)

[Управление потоком: Скользящее окно 17](#_Toc16)

[Управление перегрузками: Алгоритмы и фазы 18](#_Toc17)

[Основные принципы и фазы 18](#_Toc18)

[Алгоритмы управления перегрузками 19](#_Toc19)

[Порядок байтов в сети 21](#_Toc20)

[Big-Endian и функции преобразования 21](#_Toc21)

[Программирование сокетов 22](#_Toc22)

[Основы сокетов: Потоковые и Дейтаграммные сокеты 23](#_Toc23)

[Базовые операции с TCP-сокетами 23](#_Toc24)

[Клиентская сторона 24](#_Toc25)

[Серверная сторона 24](#_Toc26)

[Модели ввода-вывода: Блокирующий и Неблокирующий 26](#_Toc27)

[Блокирующий ввод-вывод (Blocking I/O) 26](#_Toc28)

[Неблокирующий ввод-вывод (Non-Blocking I/O) 27](#_Toc29)

[Мультиплексирование ввода-вывода (I/O Multiplexing) 28](#_Toc30)

[Системный вызов select() 29](#_Toc31)

[Системный вызов poll() 29](#_Toc32)

[Системные вызовы epoll (специфично для Linux) 29](#_Toc33)

[Событийно-ориентированный ввод-вывод и паттерн Reactor 31](#_Toc34)

[Событийно-ориентированная архитектура 31](#_Toc35)

[Паттерн Reactor 32](#_Toc36)

[Преимущества асинхронного ввода-вывода 33](#_Toc37)

[Языки программирования для сетевых приложений 35](#_Toc38)

[C++ для сетевого программирования 35](#_Toc39)

[Особенности и применение 35](#_Toc40)

[Boost.Asio: Архитектура и операции 37](#_Toc41)

[Go для сетевого программирования 38](#_Toc42)

[Особенности, конкурентность и стандартная библиотека 38](#_Toc43)

[Пакеты net, io, bufio, encoding/binary, time, runtime 39](#_Toc44)

[Заключение 41](#_Toc45)

[Источники 42](#_Toc46)

### **Обзор сетевых моделей: OSI и TCP/IP**

Двумя наиболее известными и широко используемыми сетевыми моделями являются модель взаимодействия открытых систем (OSI) и модель Transmission Control Protocol/Internet Protocol (TCP/IP). Каждая из них предлагает свой подход к структурированию сетевых протоколов.

Модель OSI, разработанная Международной организацией по стандартизации (ISO), является семиуровневой концептуальной эталонной моделью, которая четко нумерует и называет свои уровни.1 Она служит универсальной основой для стандартизации и обеспечивает высокую степень независимости между уровнями, что означает, что каждый уровень выполняет одну, четко определенную функцию.1 Такая архитектура упрощает процесс устранения неполадок в сети, поскольку проблемы можно локализовать на конкретном уровне.1 Кроме того, модель OSI поддерживает как ориентированные на соединение, так и бессобытийные парадигмы трафика.1

В отличие от этого, модель TCP/IP представляет собой четырехуровневую модель, которая названа в честь своих ключевых протоколов и непосредственно отражает структуру стека сетевых протоколов, используемых в реальном мире.1 Каждый из четырех уровней модели TCP/IP виден в сетевом пакете TCP/IP.1 Например, транспортный уровень TCP/IP соответствует транспортному уровню OSI, отвечая за сквозное соединение, целостность данных и управление потоком. Интернет-уровень TCP/IP аналогичен сетевому уровню OSI, управляя маршрутизацией и пересылкой пакетов. Уровень канала передачи данных в модели TCP/IP охватывает функции уровней канала передачи данных и физического уровня OSI, отвечая за физические и логические соединения между устройствами.1 Основное преимущество модели TCP/IP заключается в ее практической применимости, поскольку она построена вокруг протоколов, фактически используемых для установления соединений между компьютерами.1

Несмотря на схожую цель — предоставление концептуальной основы для понимания сетевых протоколов — модели OSI и TCP/IP имеют существенные различия. Модель OSI, с ее семью уровнями, значительно сложнее, чем четырехслойная модель TCP/IP.2 Модель OSI является концептуальной и эталонной, не всегда напрямую сопоставляемой с реальными сетевыми протоколами, в то время как TCP/IP основана на конкретных, существующих протоколах.1 Это различие в применимости подчеркивает фундаментальный компромисс в системном проектировании: теоретическую полноту и стандартизацию против практической применимости и простоты реализации. Модель OSI, с ее детализированной и независимой структурой уровней, превосходна для стандартизации, академического изучения и отладки, поскольку четкое разделение функций позволяет легко выявлять и изолировать проблемы.1 Однако такой уровень детализации может быть чрезмерно предписывающим или неэффективным для фактической реализации, где некоторые функции могут быть объединены для повышения производительности или упрощения.1 В свою очередь, более простая и протоколозависимая структура TCP/IP отражает прагматичный подход к построению работающих сетей, даже если это жертвует некоторой теоретической чистотой модели OSI. Выбор между этими моделями часто зависит от предполагаемого сценария использования организации.1

Модель OSI является протокольно-независимой, что позволяет использовать ее с любым стеком протоколов, тогда как модель TCP/IP основана на конкретных протоколах, разработанных для достижения определенных целей.1 Кроме того, модель OSI является более общей и поддерживает как ориентированный на соединение, так и бессобытийный сетевой трафик, в то время как TCP/IP специально разработана для ориентированного на соединение TCP-трафика.1 Понимание этих моделей имеет решающее значение для сетевой безопасности, поскольку кибератаки часто нацелены на различные уровни этих моделей.1

**Таблица 1: Сравнение моделей OSI и TCP/IP**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Модель OSI** | **Модель TCP/IP** |
| **Количество уровней** | 7 уровней | 4 уровня |
| **Основное назначение** | Концептуальная, эталонная модель для стандартизации и абстракции 1 | Реальная модель, основанная на существующих протоколах 1 |
| **Сложность** | Более сложная 2 | Менее сложная 2 |
| **Гарантии надежности** | Транспортный уровень гарантирует надежную доставку 2 | Надежная доставка гарантируется только при использовании TCP 2 |
| **Зависимость от протокола** | Протокольно-независимая, может использоваться с любым стеком 1 | Основана на конкретных протоколах 1 |
| **Поддерживаемые типы трафика** | Поддерживает как ориентированный на соединение, так и бессобытийный трафик 1 | Разработана специально для ориентированного на соединение TCP-трафика 1 |
| **Ключевые преимущества** | Стандартизация, независимые уровни (одна функция на уровень), упрощенная отладка 1 | Описывает протоколы установления соединения, реальная применимость 1 |

## **Протокол Transmission Control Protocol (TCP)**

Протокол Transmission Control Protocol (TCP) является одним из основных протоколов набора интернет-протоколов, часто называемого TCP/IP из-за их взаимодополняющей роли с Internet Protocol (IP).3 TCP функционирует на транспортном уровне (уровне 4) модели TCP/IP, предоставляя услуги связи между прикладными программами и Интернет-протоколом.3 Он абстрагирует сложности передачи данных, такие как рукопожатия и детали передачи, представляя приложению абстрактное сетевое соединение, обычно через интерфейс сетевого сокета.3

### **Основные характеристики TCP**

TCP разработан для точной доставки данных, а не для своевременной, что означает, что он может допускать задержки (до нескольких секунд) в ожидании неупорядоченных сообщений или повторных передач потерянных данных.3 Это делает его менее подходящим для приложений реального времени, таких как Voice over IP (VoIP), где предпочтительны протоколы, такие как Real-time Transport Protocol (RTP) поверх User Datagram Protocol (UDP).3

#### **Соединение (Connection-Oriented)**

TCP является протоколом, ориентированным на соединение. Это означает, что перед передачей каких-либо данных отправитель и получатель должны сначала установить соединение на основе согласованных параметров.3 Это установление соединения достигается посредством процедуры "трехстороннего рукопожатия" (3WHS).3 Сервер должен находиться в состоянии "прослушивания" (пассивное открытие), чтобы принимать запросы на соединение от клиентов.3

Ориентированная на соединение природа TCP является не просто процедурным шагом, а фундаментальным проектным решением, которое позволяет реализовать его функции надежности. Устанавливая соединение и синхронизируя порядковые номера *до* передачи данных, TCP создает контекст с сохранением состояния, который позволяет осуществлять упорядоченную доставку, подтверждения и повторные передачи. Без предварительно установленного, синхронизированного состояния такие функции, как нумерация последовательностей (для упорядочивания и обнаружения дубликатов) и подтверждения (для надежности), было бы значительно сложнее или невозможно эффективно реализовать. Если бы данные могли просто отправляться без такой настройки (как в UDP), не было бы согласованного контекста для обеспечения порядка или отслеживания потерянных пакетов. Таким образом, ориентированный на соединение дизайн является *предпосылкой* для надежных механизмов надежности TCP. Он создает необходимое состояние и общее понимание между конечными точками для эффективного управления потоком байтов, обеспечивая целостность и порядок данных, несмотря на ненадежный базовый IP-уровень.

#### **Надежность (Reliability)**

TCP обеспечивает надежную, упорядоченную и проверенную на ошибки доставку потока октетов (байтов).3 Это имеет решающее значение, поскольку базовые IP-сети ненадежны (пакеты могут быть потеряны, дублированы или доставлены не по порядку).3

Надежные механизмы TCP являются свидетельством многоуровневой архитектуры сетей. Он строит надежный сервис поверх изначально ненадежного IP-уровня, эффективно абстрагируя от приложений сложности потери пакетов и изменения порядка. Это прямое применение принципа многоуровневой модели. Уровень Интернета (IP) фокусируется на наилучшей, бессобытийной пересылке пакетов, отдавая приоритет скорости и простоте маршрутизации. Затем транспортный уровень (TCP) добавляет необходимую сложность (состояние, подтверждения, повторные передачи, управление потоком, управление перегрузками), чтобы преобразовать эту ненадежную доставку пакетов в надежный поток байтов для приложений. Такое разделение задач позволяет каждому уровню специализироваться, делая общую систему более модульной и управляемой. Если бы IP должен был быть надежным, он был бы намного медленнее и сложнее. Таким образом, надежность TCP является ярким примером того, как более высокие уровни в сетевом стеке компенсируют ограничения более низких уровней, демонстрируя мощь абстракции и модульного проектирования в сложных системах. Это позволяет приложениям предполагать надежное соединение без необходимости самостоятельно реализовывать эти сложные механизмы восстановления ошибок.

Механизмы надежности включают:

* **Порядковые номера:** Каждый байт данных идентифицируется порядковым номером, что позволяет хосту назначения переупорядочивать сегменты, даже если они приходят не по порядку.3
* **Положительное подтверждение с повторной передачей (PAR):** Получатель отправляет сообщение подтверждения (ACK) для полученных данных.3 Отправитель ведет учет отправленных пакетов и таймер; если ACK не получен до истечения таймера, пакет повторно передается.3
* **Доставка с проверкой ошибок:** Поврежденные пакеты рассматриваются как потерянные и повторно передаются.3 16-битное поле контрольной суммы включается в заголовок TCP для проверки ошибок заголовка, полезной нагрузки и псевдозаголовка IP.3
* **Повторная передача на основе дублирующих ACK:** Если сегмент потерян, получатель отправляет дублирующие подтверждения для последнего успешно полученного сегмента. После трех дублирующих ACK отправитель повторно передает неподтвержденный пакет.3
* **Повторная передача на основе тайм-аута:** Если таймер истекает до получения подтверждения, сегмент повторно передается с экспоненциально увеличивающимся порогом тайм-аута.3

#### **Потоковость (Stream-Oriented)**

TCP предоставляет поток октетов (байтов) между приложениями.3 Он принимает данные из потока данных, делит их на части, называемые сегментами TCP, и добавляет заголовок TCP.3 Эти сегменты затем инкапсулируются в IP-дейтаграммы для передачи.3 На принимающей стороне программное обеспечение TCP повторно собирает сегменты, обеспечивая правильный порядок и безошибочную доставку, а затем передает содержимое файла принимающему приложению.3

Поскольку TCP является потоково-ориентированным и не имеет концепции "сообщений" или границ записей на своем уровне, приложения, построенные на TCP, *должны* реализовывать свои собственные механизмы кадрирования для разграничения логических сообщений. Если приложение отправляет "Hello", а затем "World" через TCP, получатель может получить "HelloWorld" как одно чтение, или "Hell", затем "oWor", затем "ld". Без механизма более высокого уровня для определения того, где заканчивается "Hello" и начинается "World", приложение не сможет правильно разобрать данные. Это требует от разработчика приложения разработки и реализации "протокола кадрирования".9 Распространенные методы включают префикс длины 9, использование терминаторов 10 или структуры типа-длины-значения (TLV).9 Это критическое проектное решение для разработчиков сетевых приложений. Неспособность реализовать правильное кадрирование на прикладном уровне приведет к поврежденным или неверно интерпретированным данным, даже если TCP гарантирует надежную доставку байтов. Это перекладывает ответственность за разграничение сообщений с транспортного уровня на прикладной.

#### **Дуплексность (Full-Duplex)**

После установления TCP-соединения посредством трехстороннего рукопожатия устанавливается полнодуплексная связь. Это означает, что данные могут передаваться в обоих направлениях одновременно между клиентом и сервером.3 Однако фаза завершения соединения обрабатывается в симплексном режиме, то есть каждая сторона завершает работу независимо.4

### **Структура сегмента TCP**

TCP принимает данные из потока, делит их на части и добавляет заголовок TCP для создания сегмента TCP.3 Этот сегмент затем инкапсулируется в IP-дейтаграмму.3 Размер заголовка может варьироваться от 20 до 60 байтов, причем 40 байтов отводятся под опции.5 Если опций нет, заголовок составляет 20 байтов.5

Понимание структуры сегмента TCP определяет *синтаксис* TCP-связи, в то время как конечный автомат протокола и процедуры рукопожатия определяют его *семантику*. Оба аспекта одинаково важны для полного понимания TCP. Знание синтаксиса (какие поля существуют, их размер и расположение) необходимо для разбора и построения TCP-пакетов. Однако простое знание синтаксиса недостаточно для понимания *того, как* ведет себя TCP. Семантика (правила, состояния и процедуры, такие как рукопожатие, управление потоком и управление перегрузками) диктует, *когда* и *почему* устанавливаются определенные флаги, изменяются порядковые номера или сегменты повторно передаются. Например, знание того, что флаг SYN существует (синтаксис), менее полезно без понимания его роли в инициировании трехстороннего рукопожатия (семантика). Полное теоретическое понимание TCP требует интеграции как его синтаксического определения (структура сегмента), так и его семантического поведения (переходы состояний, алгоритмы). Это позволяет не только *описывать* TCP, но также *прогнозировать* и *отлаживать* его поведение в сети.

**Таблица 2: Поля структуры сегмента TCP**

|  |  |  |
| --- | --- | --- |
| **Поле** | **Размер (биты)** | **Описание / Назначение** |
| **Порт источника** | 16 | Адрес порта отправляющего приложения. Используется для мультиплексирования.4 |
| **Порт назначения** | 16 | Адрес порта принимающего приложения. Используется для мультиплексирования.4 |
| **Порядковый номер** | 32 | Номер первого байта данных в сегменте. Используется для восстановления порядка и повторной сборки.3 |
| **Номер подтверждения** | 32 | Если установлен флаг ACK, это следующий порядковый номер, который отправитель сегмента ожидает получить (кумулятивное подтверждение).3 |
| **Смещение данных (Data Offset)** | 4 | Длина заголовка TCP в 32-битных словах, указывает, где начинаются данные.4 |
| **Зарезервировано** | 4 | Зарезервировано для будущего использования; должно быть нулевым и игнорироваться.4 |
| **Флаги управления** | 6 (1 бит каждый) | Управляют установлением, завершением, сбросом соединения, управлением потоком и режимом передачи.4 |
| URG | 1 | Указатель срочности действителен.4 |
| ACK | 1 | Поле номера подтверждения действительно.4 |
| PSH | 1 | Запрос на немедленную отправку данных (push).4 |
| RST | 1 | Сброс соединения.4 |
| SYN | 1 | Синхронизация порядковых номеров (установление соединения).4 |
| FIN | 1 | Нет больше данных от отправителя (завершение соединения).4 |
| **Окно** | 16 | Объем данных (в октетах), который отправитель данного сегмента готов принять, начиная с номера, указанного в поле подтверждения. Используется для управления потоком.4 |
| **Контрольная сумма** | 16 | 16-битная сумма для контроля ошибок. Обязательна в TCP.3 |
| **Указатель срочности** | 16 | Смещение от порядкового номера, указывающее на конец срочных данных. Действителен только если установлен флаг URG.4 |
| **Опции** | Переменная (до 40 байтов) | Дополнительные поля, такие как Maximum Segment Size (MSS).4 |
| **Данные** | Переменная | Фактические пользовательские данные.4 |

### **Управление соединением: Трехстороннее и Четырехстороннее рукопожатие**

TCP, будучи протоколом, ориентированным на соединение, требует процесса рукопожатия как для установления, так и для завершения надежной связи.3

#### **Трехстороннее рукопожатие (3WHS) для установления соединения**

Трехстороннее рукопожатие является фундаментальным процессом, который устанавливает надежное, полнодуплексное соединение путем синхронизации порядковых номеров между клиентом и сервером.3 Оно гарантирует, что обе стороны готовы к обмену данными.15

Процесс состоит из трех шагов:

1. **SYN (Synchronize Sequence Number):** Клиент инициирует соединение, отправляя сегмент с установленным флагом SYN, сообщая серверу о своем желании начать связь и свой начальный порядковый номер (ISN).3 Поле ACK в этом начальном сегменте SYN устанавливается в ноль.16 Клиент переходит в состояние SYN-SENT.17
2. **SYN-ACK (Synchronize-Acknowledge):** Сервер отвечает сегментом SYN-ACK, подтверждая SYN клиента (ACK = ISN клиента + 1) и отправляя свой собственный ISN.3 Сервер переходит в состояние SYN-RCVD.17
3. **ACK (Acknowledge):** Клиент отправляет сегмент ACK, подтверждая SYN сервера (ACK = ISN сервера + 1).3 Клиент переходит в состояние ESTABLISHED.17 Сервер также переходит в состояние ESTABLISHED после получения этого ACK.17

Это рукопожатие имеет большое значение, поскольку оно предотвращает нерациональное использование ресурсов из-за некорректных соединений.17 Оно также помогает смягчить атаки SYN-флуда, при которых злоумышленник отправляет волну SYN-пакетов, но никогда не завершает рукопожатие, истощая ресурсы сервера.15 Кроме того, трехстороннее рукопожатие закладывает основу для управления потоком, позволяя обеим сторонам согласовать объем данных, который может быть отправлен в любой момент.15 Это также полезно при устранении неполадок в сети, помогая определить, на каком этапе происходит сбой соединения.15

**Таблица 3: Этапы трехстороннего рукопожатия TCP**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Шаг** | **Отправитель** | **Получатель** | **Тип пакета (Флаги)** | **Описание** | **Пример порядкового номера (ISN/ACK)** |
| **1: SYN** | Клиент | Сервер | SYN | Клиент инициирует соединение, отправляя свой начальный порядковый номер (ISN).5 | seq = A, ack = 0 16 |
| **2: SYN-ACK** | Сервер | Клиент | SYN, ACK | Сервер подтверждает SYN клиента (ACK = A+1) и отправляет свой собственный ISN.5 | seq = B, ack = A+1 16 |
| **3: ACK** | Клиент | Сервер | ACK | Клиент подтверждает SYN сервера (ACK = B+1), завершая установление соединения.5 | seq = A+1, ack = B+1 16 |

#### **Четырехстороннее рукопожатие для завершения соединения**

Хотя трехстороннее рукопожатие требует передачи трех пакетов, завершение надежного TCP-соединения обычно требует передачи четырех пакетов.16 Это связано с тем, что TCP-соединение является полнодуплексным, что означает, что данные могут передаваться в каждом направлении независимо от другого, и, следовательно, каждое направление должно быть завершено независимо.16

Процесс состоит из следующих шагов:

1. **FIN (Клиент):** Клиент отправляет сегмент FIN (часто сопровождаемый ACK для ранее полученных данных), сообщая серверу, что у него больше нет данных для отправки.16 Клиент переходит в состояние FIN\_WAIT\_1.17
2. **ACK (Сервер):** Сервер получает FIN и отправляет ACK, подтверждая получение FIN клиента.16 Сервер переходит в состояние CLOSE\_WAIT.17 На этом этапе сервер может все еще иметь данные для отправки.17
3. **FIN (Сервер):** Как только приложение на сервере завершит отправку всех своих данных, сервер отправляет свой собственный сегмент FIN клиенту.16 Сервер переходит в состояние LAST\_ACK.17
4. **ACK (Клиент):** Клиент получает FIN сервера и отправляет окончательный ACK.16 Клиент переходит в состояние TIME\_WAIT.17 Сервер переходит в состояние CLOSE после получения этого ACK. Клиент переходит в состояние CLOSE после ожидания в течение 2MSL (Maximum Segment Lifetime), что гарантирует обработку любых задержанных пакетов и полное закрытие соединения с обоих концов.17

Разница в количестве шагов между установлением (3-стороннее) и завершением (4-стороннее) соединения подчеркивает асимметрию в управлении соединениями TCP. 4-стороннее рукопожатие является прямым следствием полнодуплексной природы TCP и необходимости для каждой конечной точки независимо сигнализировать и подтверждать прекращение передачи данных в своем соответствующем направлении. 3-стороннее рукопожатие объединяет SYN и ACK сервера в один пакет, потому что сервер *инициирует* синхронизацию своего порядкового номера *в ответ* на SYN клиента. Однако для завершения FIN клиента означает "у меня больше нет данных для отправки", но у сервера *все еще могут быть* данные. Следовательно, сервер должен подтвердить FIN клиента (ACK), продолжить отправку оставшихся данных, а *затем* отправить свой собственный FIN, когда он закончит.17 Это требует отдельных FIN от каждой стороны и отдельных ACK для этих FIN. Такая конструкция гарантирует, что данные не будут потеряны во время завершения работы, даже если одна сторона закончит отправку раньше другой.

**Таблица 4: Этапы четырехстороннего рукопожатия TCP для завершения соединения**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Шаг** | **Отправитель** | **Получатель** | **Тип пакета (Флаги)** | **Описание** |
| **1: FIN** | Клиент | Сервер | FIN (ACK) | Клиент сообщает, что у него больше нет данных для отправки.16 |
| **2: ACK** | Сервер | Клиент | ACK | Сервер подтверждает получение FIN клиента. Сервер может все еще отправлять данные.16 |
| **3: FIN** | Сервер | Клиент | FIN (ACK) | Сервер сообщает, что у него больше нет данных для отправки.16 |
| **4: ACK** | Клиент | Сервер | ACK | Клиент подтверждает получение FIN сервера. Соединение полностью закрывается после ожидания 2MSL.16 |

### **Механизмы надежности TCP: Подтверждения и Ретрансмиссия**

TCP обеспечивает надежную передачу данных по ненадежным сетям посредством комбинации стратегий подтверждения и повторной передачи.3

#### **Положительное подтверждение с повторной передачей (PAR)**

Основной механизм надежности TCP — это положительное подтверждение с повторной передачей. Когда получатель принимает пакеты данных, он отправляет отправителю подтверждение (ACK), указывающее на успешное получение данных.3 Отправитель ведет запись отправленных пакетов и устанавливает таймер.3 Если ACK не получен до истечения таймера (тайм-аут повторной передачи, RTO), предполагается, что пакет был потерян, и отправитель повторно передает этот пакет.3 Поврежденные пакеты также рассматриваются как потерянные и повторно передаются.3

#### **Быстрая повторная передача (Fast Retransmit)**

Быстрая повторная передача — это усовершенствование стандартного процесса повторной передачи, разработанное для ускорения восстановления после потери пакетов без ожидания истечения полного RTO.6 Этот механизм срабатывает, когда отправитель получает определенное количество дублирующих ACK для одного и того же пакета данных, обычно три дублирующих ACK.3 Это является сильным индикатором потери пакета.6 В отличие от стандартной повторной передачи, которая полагается на истечение тайм-аута, быстрая повторная передача реагирует немедленно после обнаружения нескольких дублирующих ACK.6 Такой проактивный подход значительно сокращает время ожидания и ускоряет исправление ошибок в процессе передачи данных.6

Быстрая повторная передача также играет важную роль в механизмах управления перегрузками. Она помогает быстро разрешать ситуации потери пакетов, которые могут быть вызваны перегрузкой сети, обеспечивая более быстрое восстановление и поддержание потока данных по сети.6 Преимущества включают снижение задержки, улучшенную пропускную способность и повышенную надежность.6

Надежность TCP не является статической; она использует адаптивные алгоритмы (такие как динамический RTO и быстрая повторная передача) для реагирования на меняющиеся сетевые условия, демонстрируя сложный подход к поддержанию производительности при различных нагрузках и скоростях потерь. Стандартный RTO является запасным вариантом; он ждет потенциально долгое время, если пакет действительно потерян. Быстрая повторная передача, запускаемая дублирующими ACK, предоставляет *более ранний сигнал* о потере. Это крайне важно, поскольку ожидание RTO может значительно задержать передачу данных.6 Однако быстрая повторная передача полагается на то, что *несколько* пакетов находятся в пути, и последующие пакеты прибывают для генерации дублирующих ACK. Если отправляется только один пакет или если последующие пакеты *также* теряются, быстрая повторная передача может не сработать, и RTO становится необходимым последним средством. Динамический расчет RTO 19 дополнительно демонстрирует адаптацию к изменяющейся задержке сети. Таким образом, комбинация динамического RTO и быстрой повторной передачи (наряду с SACK) иллюстрирует адаптивный и многогранный подход TCP к надежности. Он отдает приоритет быстрому восстановлению (быстрая повторная передача), когда это возможно, но сохраняет надежный, хотя и более медленный, запасной вариант (RTO) для более серьезных или изолированных событий потери. Такая адаптивная конструкция необходима для эффективности TCP в различных и динамичных условиях Интернета.

Для дальнейшей оптимизации стратегий повторной передачи используются адаптивные механизмы тайм-аута, выборочное подтверждение (SACK), мониторинг сетевых условий и балансирование между быстрой повторной передачей и управлением перегрузками.6 SACK позволяет получателю информировать отправителя о конкретно полученных пакетах, уменьшая ненужные повторные передачи.21

### **Управление потоком: Скользящее окно**

Управление потоком TCP предотвращает перегрузку получателя отправителем слишком большим объемом данных, чем он может обработать.8 Это своего рода процесс синхронизации скорости между отправителем и получателем.14

TCP использует механизм скользящего окна для управления потоком.8 Получатель объявляет отправителю размер своего "окна приема" (RWND), который представляет собой доступное пространство в его буфере приема.8 Это значение является частью заголовка сегмента TCP.5 Отправитель может отправлять данные только в объеме, не превышающем размер объявленного окна приема.13 Если буфер приема заполнен, получатель объявляет нулевой размер окна, что приостанавливает отправителя.13 По мере того, как принимающее приложение считывает данные из буфера, пространство становится доступным, и объявляется новый, больший размер окна, позволяя отправителю возобновить передачу.13

Управление потоком TCP с использованием скользящего окна является прямой реализацией стратегии управления ресурсами, гарантирующей, что ограниченная емкость буфера получателя не будет перегружена, тем самым предотвращая потерю пакетов на стороне получателя из-за переполнения буфера. Без управления потоком быстрый отправитель мог бы непрерывно передавать данные со скоростью, превышающей скорость обработки или буферизации получателем. Это неизбежно привело бы к переполнению буфера получателя, вынуждая его отбрасывать входящие пакеты. Отброшенные пакеты затем вызывают повторные передачи, которые расходуют пропускную способность сети и увеличивают задержку, в конечном итоге ухудшая общую производительность и надежность. Механизм скользящего окна обеспечивает динамический, основанный на обратной связи способ для получателя сигнализировать о своей текущей емкости, позволяя отправителю упреждающе регулировать свою скорость. Таким образом, управление потоком является важнейшим превентивным механизмом против потери пакетов на стороне получателя и неэффективных повторных передач. Оно обеспечивает эффективное использование ресурсов получателя и напрямую способствует общей надежности и производительности TCP, поддерживая сбалансированный поток данных.

Взаимодействие между буферами отправки и приема имеет следующие последствия: максимальное количество неподтвержденных байтов, которое может отправить система, является меньшим из двух чисел: размера буфера отправки на отправляющей системе или размера окна приема, объявленного принимающей системой.13 Плавный поток данных происходит, когда принимающее приложение считывает данные так же быстро, как их отправляет отправляющая система, поддерживая окно приема открытым.13 Размер окна TCP (16 бит) может достигать 65 535 байтов.14 Его можно настраивать (например, параметры tcp\_sendspace и tcp\_recvspace в AIX).13

Процессы скользящего окна включают: открытие окна (позволяет отправлять больше данных), закрытие окна (отправленные данные подтверждены) и уменьшение окна (уменьшение размера, не рекомендуется).14 Чтобы предотвратить сценарии, когда объявляется нулевое окно, а подтверждение для последующего ненулевого окна теряется, что может привести к тупику, TCP использует таймер сохранения, который периодически отправляет небольшие пакеты получателю для проверки ненулевого окна.14

### **Управление перегрузками: Алгоритмы и фазы**

Управление перегрузками ограничивает поток пакетов в сеть, чтобы предотвратить перегрузку, которая может привести к замедлению времени отклика и снижению производительности.14 Оно отличается от управления потоком тем, что касается трафика в масштабах всей сети, а не только сквозного соединения.14

#### **Основные принципы и фазы**

Управление перегрузками TCP динамически регулирует поток данных в зависимости от текущего состояния сети.21 Оно включает три основные фазы:

* **Медленный старт (Slow Start):** На этой начальной фазе размер окна перегрузки (CWND) экспоненциально увеличивается после каждого кругового времени задержки (RTT).14 Оно начинается с небольшого cwnd (например, 1 MSS или 10 сегментов в Linux).14 Цель состоит в том, чтобы исследовать доступную пропускную способность.21 Фаза медленного старта завершается, когда cwnd достигает ssthresh (порога медленного старта) или обнаруживается потеря пакета.14
* **Избегание перегрузок (Congestion Avoidance):** Эта фаза начинается после достижения ssthresh. CWND увеличивается аддитивно (линейно) после каждого RTT.14 Цель состоит в том, чтобы поддерживать справедливую долю пропускной способности без чрезмерной перегрузки.21
* **Обнаружение перегрузок (Congestion Detection):** Запускается при потере пакетов.
  + **Тайм-аут:** Если ACK не получен в течение расчетного времени, ssthresh уменьшается вдвое, cwnd сбрасывается до 1, и медленный старт перезапускается.21 Это указывает на высокую степень перегрузки.21
  + **Три дублирующих ACK (быстрая повторная передача/быстрое восстановление):** Указывает на меньшую вероятность перегрузки. ssthresh уменьшается вдвое, cwnd устанавливается в ssthresh, и процесс продолжается с избеганием перегрузок.21 Быстрое восстановление ускоряет процесс восстановления, избегая полного возврата к медленному старту.21

#### **Алгоритмы управления перегрузками**

Разнообразие алгоритмов управления перегрузками TCP (Tahoe, Reno, New Reno, BIC, CUBIC и т.д.) демонстрирует непрерывную адаптацию протокола к растущему разнообразию и масштабу сетевых сред (например, высокоскоростные, беспроводные, спутниковые, дальние сети). Большинство этих алгоритмов являются усовершенствованиями стека TCP, разработанными с учетом различных сетевых сред.22 Это указывает на то, что ни один алгоритм не является оптимальным для всех сетевых условий. Например, консервативный подход Tahoe/Reno может недоиспользовать пропускную способность в высокоскоростных средах с низкими потерями, что привело к разработке более агрессивных алгоритмов, таких как CUBIC. Беспроводные сети вводят случайную потерю пакетов, не обязательно из-за перегрузки, требуя алгоритмов, таких как Veno или Westwood+, которые могут различать типы потерь. Таким образом, разнообразный ландшафт алгоритмов управления перегрузками TCP отражает непрерывные усилия по оптимизации производительности сети в широком спектре сетевых характеристик. Эта эволюция является динамическим ответом на меняющуюся природу Интернета, где различные среды требуют индивидуальных стратегий для максимизации пропускной способности и минимизации задержки при предотвращении коллапса. Это подчеркивает, что сетевые протоколы не статичны, а постоянно совершенствуются для решения новых задач.

Некоторые из известных алгоритмов включают:

* **TCP Tahoe/Reno:** Считаются классическими моделями управления перегрузками.21 Они демонстрируют типичный "медленный старт" передач, где пропускная способность постепенно увеличивается до стабилизации.22 Пропускная способность уменьшается при возникновении перегрузки, а затем скорость снова медленно возрастает.22 TCP Reno использует алгоритм мультипликативного уменьшения для снижения размера окна.21 TCP Reno является наиболее широко используемым алгоритмом.22
* **New Reno:** Усовершенствование алгоритма Reno, направленное на улучшение производительности TCP в сценариях, связанных с потерей пакетов.21 Особенности включают быструю повторную передачу, улучшенное быстрое восстановление, поддержку SACK (выборочное подтверждение) и обработку частичных подтверждений.21 Цель состоит в снижении частоты ненужных тайм-аутов.21
* **BIC (Binary Increase Congestion control):** Использует метод бинарного поиска для определения оптимальной скорости отправки.21 Был стандартным алгоритмом в ядре Linux.21
* **CUBIC:** Менее агрессивный вариант BIC, разработанный для высокоскоростных и дальних сетей.21 Использует кубическую функцию для роста окна, обеспечивая более быструю сходимость и повышенную отзывчивость к динамике сети.21 Рекомендуется в качестве алгоритма по умолчанию во многих системах.22
* **Другие варианты:** Включают High-Speed TCP, H-TCP, Scalable TCP (для соединений с большой пропускной способностью и RTT), TCP Hybla (для спутниковых каналов), TCP Vegas (использует измерение RTT для оценки качества канала), TCP Veno (оптимизирован для беспроводных сетей, лучше справляется со случайной потерей пакетов) и TCP Westwood+ (модификация TCP Reno для больших значений пропускной способности/RTT и случайной потери пакетов).22

**Таблица 5: Сравнение алгоритмов управления перегрузками TCP**

|  |  |  |  |
| --- | --- | --- | --- |
| **Алгоритм** | **Ключевые особенности** | **Преимущества** | **Применимость / Недостатки** |
| **TCP Tahoe/Reno** | Медленный старт, избегание перегрузок, мультипликативное уменьшение (Reno) 21 | Простой и надежный, эффективное использование пропускной способности 21 | Классические модели, широко распространены. Могут быть менее реактивными к изменениям сети.21 |
| **New Reno** | Улучшенная быстрая повторная передача и быстрое восстановление, поддержка SACK, обработка частичных подтверждений 21 | Улучшенное восстановление после потери пакетов, эффективная обработка дублирующих ACK, снижение тайм-аутов 21 | Более сложный, зависит от поддержки получателем (SACK), потенциал для "всплесков" повторных передач.21 |
| **BIC** | Бинарный поиск для роста окна перегрузки 21 | Эффективен для высокоскоростных сетей 21 | Был стандартным в ядре Linux. Может быть агрессивным.21 |
| **CUBIC** | Кубическая функция для роста окна, быстрая сходимость 21 | Агрессивный и эффективный рост окна, подходит для высокоскоростных и дальних сетей 21 | Менее агрессивен, чем BIC. Может переоценивать доступную пропускную способность.21 |

## **Порядок байтов в сети**

В сетевой связи крайне важно обеспечить правильную интерпретацию данных при их обмене между различными компьютерными системами. Различные архитектуры компьютеров используют разные внутренние порядки байтов для многобайтовых целых чисел, что может привести к неверной интерпретации данных при обмене между системами с разным порядком байтов.23

### **Big-Endian и функции преобразования**

Существуют два основных метода порядка байтов на хосте:

* **Little-endian (прямой порядок):** Наименее значимый байт располагается первым. Этот метод используется, например, в микропроцессорах Intel.23
* **Big-endian (обратный порядок):** Наиболее значимый байт располагается первым. Этот метод используется, например, в мейнфреймах IBM z/Architecture и S/390, а также в микропроцессорах Motorola.23

Чтобы решить проблему несовместимости порядка байтов, для сетевой связи был установлен стандартный порядок байтов, который всегда определяется как **big-endian**.23 Эта стандартизация гарантирует, что адресная информация и многобайтовые данные правильно интерпретируются принимающими машинами, независимо от их собственного порядка байтов хоста.23

Установление сетевого порядка байтов (Big-Endian) и предоставление функций преобразования (таких как htonl, ntohl) имеют решающее значение для достижения совместимости в гетерогенных сетевых средах, предотвращая тонкие, но значительные проблемы с повреждением данных. Без универсального стандарта для сетевой связи и абстракций, предоставляемых функциями преобразования, каждое приложение должно было бы реализовывать сложную, подверженную ошибкам логику для обнаружения и преобразования порядка байтов, или просто не смогло бы правильно взаимодействовать между различными архитектурами.24 Это не просто деталь программирования, а фундаментальное требование для способности Интернета соединять разнообразные вычислительные системы. Это подчеркивает, как стандартизация на самых низких уровнях представления данных необходима для построения глобально совместимой сети. Существование этих функций абстрагирует архитектурные различия, позволяя разработчикам сосредоточиться на логике приложений более высокого уровня.

Для облегчения этого обмена предоставляются стандартные функции C, которые позволяют программам легко переключать числа между порядком байтов хоста и сетевым порядком байтов без необходимости знать, какой метод используется для порядка байтов хоста.23 Крайне важно всегда использовать эти функции в коде, чтобы обеспечить правильную работу на различных машинных архитектурах.24

* htonl(uint32\_t hostlong): Преобразует 32-битное беззнаковое целое число из порядка байтов хоста в сетевой порядок байтов. Буква 'h' означает "host", 'to' — "to", 'n' — "network", а 'l' — "long" (32-битное).23
* htons(uint16\_t hostshort): Преобразует 16-битное беззнаковое целое число из порядка байтов хоста в сетевой порядок байтов.23
* ntohl(uint32\_t netlong): Преобразует 32-битное беззнаковое целое число из сетевого порядка байтов в порядок байтов хоста. Буква 'n' означает "network", 'to' — "to", 'h' — "host", а 'l' — "long" (32-битное).23
* ntohs(uint16\_t netshort): Преобразует 16-битное беззнаковое целое число из сетевого порядка байтов в порядок байтов хоста.23

## **Программирование сокетов**

Программирование сокетов является основным интерфейсом для сетевой связи на прикладном уровне. Оно позволяет процессам, работающим на одной или разных машинах, взаимодействовать друг с другом.25 Сокеты служат конечными точками связи, представляя собой комбинацию IP-адреса и номера порта (например, 192.168.1.1:8080).25

### **Основы сокетов: Потоковые и Дейтаграммные сокеты**

Выбор между различными типами сокетов напрямую отражает фундаментальный компромисс между надежностью/порядком и скоростью/накладными расходами в сетевой связи, заставляя разработчиков выбирать соответствующий транспортный протокол на основе конкретных требований их приложения.

* **Потоковые сокеты (TCP):**
  + Обеспечивают надежную, ориентированную на соединение связь.25
  + Данные поступают в том порядке, в котором они были отправлены.25
  + Используют протокол TCP.25
  + Если приложению требуется гарантированная доставка и упорядоченные данные (например, передача файлов, просмотр веб-страниц), потоковые сокеты TCP необходимы, даже с их накладными расходами.
* **Дейтаграммные сокеты (UDP):**
  + Обеспечивают бессобытийную и ненадежную связь.25
  + Быстрее из-за меньших накладных расходов.26
  + Используют протокол UDP.25
  + Если приложение может допустить некоторую потерю или изменение порядка ради скорости (например, аудио/видео в реальном времени, онлайн-игры, где старые данные быстро становятся неактуальными), предпочтительны дейтаграммные сокеты UDP. В этом случае приложение должно самостоятельно обрабатывать надежность, упорядочивание и управление потоком, если это необходимо.
* **Сырые сокеты (Raw Sockets):**
  + Обходят встроенную поддержку стандартных протоколов, таких как TCP/UDP.25
  + Используются для разработки пользовательских низкоуровневых протоколов.25

Выбор типа сокета является критически важным архитектурным решением в сетевом программировании, напрямую влияющим на производительность, надежность приложения и сложность его реализации. Это подчеркивает, что сетевые протоколы не являются универсальными решениями, а разработаны для конкретных сценариев использования и компромиссов.

### **Базовые операции с TCP-сокетами**

Программирование сокетов TCP следует модели клиент-сервер, где один сокет прослушивает (сервер), а другой подключается (клиент).26 Отдельная последовательность операций для клиентских и серверных сокетов (bind/listen/accept для сервера против connect для клиента) отражает фундаментальную асимметрию в управлении сетевыми ресурсами и инициировании соединений.

Сервер должен быть постоянно доступен по *известному адресу и порту* для приема входящих запросов от потенциально множества клиентов. Это требует от него bind (привязки) к определенному адресу, listen (прослушивания) соединений и accept (принятия) их по одному (или одновременно). Если сервер не привязывается к фиксированному порту, клиенты не будут знать, куда подключаться. Если он не прослушивает и не принимает соединения, он не сможет обрабатывать несколько входящих запросов. Клиент, с другой стороны, *инициирует* соединение с известным сервером. Ему не нужно привязываться к определенному порту (ОС может назначить временный), потому что он является активной стороной. Эта асимметрия гарантирует, что несколько клиентов могут подключаться к одному серверу, что является фундаментальным для клиент-серверной архитектуры Интернета. Таким образом, эта операционная асимметрия не является произвольной, а является архитектурной необходимостью для масштабируемых и обнаруживаемых сетевых служб. Она четко определяет роли и обязанности для установления соединения, позволяя эффективно распределять и управлять ресурсами в многоклиентской среде.

#### **Клиентская сторона**

1. **socket():** Создает дескриптор сокета (целое число, похожее на файловый дескриптор).25 При этом указывается домен связи (например, AF\_INET для IPv4), тип (SOCK\_STREAM для TCP) и протокол (обычно 0).26
2. **connect():** Устанавливает соединение с адресом и портом сервера.25 Эта функция инициирует трехстороннее рукопожатие TCP.27 Ядро автоматически выбирает IP-адрес источника клиента и временный порт, если они не были явно привязаны.27
3. **send() / recv() (или read() / write()):** Используются для передачи и приема данных по установленному соединению.25
4. **close():** Закрывает сокет и освобождает системные ресурсы.25

#### **Серверная сторона**

1. **socket():** Создает сокет, аналогично клиентской стороне.25
2. **setsockopt() (Необязательно):** Позволяет манипулировать опциями сокета, например, для повторного использования адресов/портов, чтобы предотвратить ошибки типа "адрес уже используется".26
3. **bind():** Присваивает локальный IP-адрес и номер порта сокету.25 INADDR\_ANY может использоваться для привязки ко всем доступным интерфейсам.26
4. **listen():** Преобразует сокет в пассивный прослушивающий сокет, указывая ядру, что оно должно принимать входящие запросы на соединение.25 Параметр backlog определяет размер очереди для ожидающих соединений.26
5. **accept():** Извлекает первый запрос на соединение из очереди, создает *новый* подключенный сокет для этого клиента и возвращает его файловый дескриптор.25 Исходный прослушивающий сокет остается открытым.27 Этот вызов обычно блокируется до тех пор, пока клиент не подключится.27
6. **send() / recv() (или read() / write()):** Используются для передачи и приема данных с подключенным клиентом.25
7. **close():** Закрывает сокет.25

**Таблица 6: Базовые операции с TCP-сокетами**

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Описание** | **Использование на клиентской стороне** | **Использование на серверной стороне** |
| **socket()** | Создает дескриптор сокета, указывая домен, тип и протокол связи.25 | **1.** Создание сокета.25 | **1.** Создание сокета.25 |
| **setsockopt()** | Манипулирует опциями сокета (необязательно).26 | Не используется | **2.** Настройка опций сокета (например, для повторного использования адреса).26 |
| **bind()** | Присваивает локальный IP-адрес и номер порта сокету.25 | Обычно не используется (ядро выбирает адрес).27 | **3.** Привязка сокета к адресу и порту.25 |
| **listen()** | Преобразует сокет в пассивный прослушивающий, ожидающий входящих соединений.25 | Не используется | **4.** Прослушивание входящих соединений.25 |
| **connect()** | Устанавливает соединение с адресом и портом сервера.25 | **2.** Подключение к серверу.25 | Не используется |
| **accept()** | Извлекает запрос на соединение из очереди и создает новый сокет для клиента.25 | Не используется | **5.** Принятие клиентского соединения.25 |
| **send() / recv()** | Передача и прием данных по установленному соединению.25 | **3.** Обмен данными.25 | **6.** Обмен данными.25 |
| **close()** | Закрывает сокет и освобождает системные ресурсы.25 | **4.** Закрытие сокета.25 | **7.** Закрытие сокета.25 |

### **Модели ввода-вывода: Блокирующий и Неблокирующий**

Операции ввода-вывода (I/O) могут значительно влиять на отзывчивость и производительность приложения, особенно в сетевом программировании.29

#### **Блокирующий ввод-вывод (Blocking I/O)**

Принцип блокирующего ввода-вывода заключается в том, что операции ввода-вывода приостанавливают выполнение программы до их завершения.29 Когда программа пытается читать или записывать данные из блокирующего сокета, она приостанавливает свой текущий процесс и ожидает завершения передачи данных или появления данных для чтения.29

Характеристики блокирующих сокетов включают блокирующее поведение, синхронную работу и простоту реализации.29 Однако эта простота имеет существенный недостаток: если блокирующий сокет используется для управления связью с несколькими клиентами одновременно, и операция одного клиента занимает необычно много времени, это может заблокировать всю программу, что приведет к значительным задержкам в обслуживании других клиентов.29 Пользователь не может выполнять другие действия, ожидая завершения системного вызова.30

#### **Неблокирующий ввод-вывод (Non-Blocking I/O)**

В отличие от блокирующих сокетов, неблокирующие сокеты функционируют асинхронно. Когда операция ввода-вывыода инициируется на неблокирующем сокете, программа немедленно продолжает свое выполнение, независимо от того, успешно ли завершилась операция.29 Программа продолжает выполнение, пока операция находится в процессе.29

Характеристики неблокирующих сокетов включают неблокирующее поведение, асинхронную работу и повышенную сложность.29 Преимущества неблокирующих сокетов включают улучшенную отзывчивость и лучшее использование ресурсов.29 Они позволяют продолжать выполнять другие задачи, пока операции ввода-вывода ожидают завершения, что особенно полезно для параллельных программ.30 Неблокирующий ввод-вывод более мощный и гибкий, поскольку позволяет выполнять несколько операций ввода-вывода одновременно и в любом порядке.30 Однако он требует тщательной обработки асинхронных событий, часто требуя от разработчиков реализации механизмов, таких как циклы событий или методы мультиплексирования (например, select(), poll()), для эффективного управления этими операциями.29

Выбор между блокирующим и неблокирующим вводом-выводом является фундаментальным проектным решением, которое обменивает простоту программирования на масштабируемость и отзывчивость приложения, что особенно важно в параллельных сетевых серверах. В сервере, обрабатывающем множество параллельных клиентов, если каждое клиентское соединение использует блокирующий ввод-вывод, один медленный клиент (например, ожидающий данных, задержка сети) может привести к блокировке потока/процесса сервера, выделенного для него. Если сервер использует модель "один поток на клиента" 31, это может привести к чрезмерному созданию потоков, накладным расходам на переключение контекста и исчерпанию ресурсов. Неблокирующий ввод-вывод, хотя и более сложен в кодировании (требует циклов событий или мультиплексирования), позволяет одному потоку управлять несколькими соединениями одновременно без блокировки, значительно улучшая масштабируемость и эффективность использования ресурсов. Это жизненно важно для высокопроизводительных серверов. Таким образом, решение между блокирующим и неблокирующим вводом-выводом напрямую определяет способность сервера эффективно обрабатывать параллельные соединения. Это классический пример того, как низкоуровневые решения по проектированию ввода-вывода имеют глубокие архитектурные последствия для масштабируемости и отзывчивости всего приложения.

**Таблица 7: Сравнение блокирующего и неблокирующего ввода-вывода**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Блокирующий ввод-вывод** | **Неблокирующий ввод-вывод** |
| **Поведение** | Операции ввода-вывода приостанавливают выполнение программы до завершения.29 | Операции ввода-вывода немедленно возвращают управление, программа продолжает выполнение.29 |
| **Тип операции** | Синхронный.29 | Асинхронный.29 |
| **Сложность** | Простой в понимании и реализации.29 | Повышенная сложность логики программы.29 |
| **Преимущества** | Простота.29 | Улучшенная отзывчивость, лучшее использование ресурсов, позволяет выполнять другие задачи.29 |
| **Недостатки** | Может заблокировать всю программу, вызывая задержки.29 Пользователь не может выполнять другие действия.30 | Требует тщательной обработки асинхронных событий, механизмов мультиплексирования.29 |

### **Мультиплексирование ввода-вывода (I/O Multiplexing)**

Мультиплексирование ввода-вывода — это механизм ядра, который позволяет одному потоку эффективно управлять несколькими источниками ввода-вывода (например, сетевыми сокетами).33 Это решение для обработки множества клиентов без создания отдельного потока для каждого клиента, что неэффективно для большого числа клиентов.33

Развитие от select() к poll() и далее к epoll (и аналогичным ОС-специфичным механизмам, таким как kqueue, IOCP) демонстрирует четкую эволюционную тенденцию в API мультиплексирования ввода-вывода, обусловленную необходимостью все более эффективной и масштабируемой обработки параллельных соединений, особенно в высокопроизводительных серверных приложениях.

#### **Системный вызов select()**

select() предоставляет механизм синхронного мультиплексирования ввода-вывода. Он блокируется до тех пор, пока указанные файловые дескрипторы не будут готовы к вводу-выводу или не истечет дополнительный тайм-аут.33 Он отслеживает три набора файловых дескрипторов: readfds (для чтения), writefds (для записи) и exceptfds (для исключений).33

Ограничения select() включают необходимость перестройки fd\_set перед каждым вызовом.33 Производительность составляет O(n), где n — наибольший номер файлового дескриптора, поскольку он проверяет каждый бит до этого номера.33 После возврата необходимо итерировать по всем файловым дескрипторам, чтобы проверить их готовность.33 Основное преимущество select() — его высокая переносимость между Unix-подобными операционными системами.33

#### **Системный вызов poll()**

В отличие от select(), poll() использует один массив структур pollfd, что упрощает его прототип.33

Преимущества poll() по сравнению с select():

* Не требует вычисления highest\_fd + 1.33
* Более эффективен для файловых дескрипторов с большими значениями.33
* Массив pollfd может быть повторно использован без изменений (входные/выходные события разделены).33

Однако poll() менее переносим, чем select(), поскольку некоторые Unix-системы могут его не поддерживать.33

#### **Системные вызовы epoll (специфично для Linux)**

epoll управляет контекстом ввода-вывода непосредственно в ядре, уменьшая накладные расходы на передачу наборов между пользовательским и ядерным пространством при каждом вызове.33 Процесс состоит из трех шагов: epoll\_create (создает контекст), epoll\_ctl (добавляет/удаляет файловые дескрипторы) и epoll\_wait (ожидает событий).33

Преимущества epoll по сравнению с select()/poll():

* Файловые дескрипторы могут быть добавлены/удалены динамически во время ожидания.33
* epoll\_wait возвращает *только* готовые файловые дескрипторы, что устраняет необходимость итерирования.33
* Лучшая производительность: O(1) вместо O(n).33
* Поддерживает поведение, управляемое уровнем и фронтом.33

Основное ограничение epoll заключается в его специфичности для Linux, что делает его непереносимым на другие операционные системы.33

Прогресс от select() к poll() и далее к epoll (и аналогичным ОС-специфичным механизмам, таким как kqueue, IOCP) демонстрирует четкую эволюционную тенденцию в API мультиплексирования ввода-вывода, обусловленную необходимостью все более эффективной и масштабируемой обработки параллельных соединений, особенно в высокопроизводительных серверных приложениях. Производительность select() (O(N), где N — наибольший номер файлового дескриптора) и необходимость перестраивать fd\_set при каждом вызове становятся непомерными для серверов, управляющих тысячами соединений (проблема "C10k"). poll() улучшает select() за счет использования более гибкой структуры данных и отсутствия требования highest\_fd + 1, но все еще требует итерирования по всем отслеживаемым файловым дескрипторам. epoll принципиально меняет это, заставляя ядро управлять списком интересов и *возвращая только* готовые файловые дескрипторы, достигая производительности O(1). Эта эволюция напрямую решает узкое место масштабируемости при обработке большого числа параллельных, неактивных соединений, что характерно для современных веб-серверов и распределенных систем. Таким образом, разработка более совершенных механизмов мультиплексирования ввода-вывода, таких как epoll, является прямым ответом на растущие требования к масштабируемости серверов. Это иллюстрирует, как узкие места производительности на уровне системных вызовов стимулируют инновации в API операционных систем и, следовательно, в проектировании сетевых приложений. Компромисс часто заключается в переносимости ради чистой производительности.

**Таблица 8: Сравнение механизмов мультиплексирования ввода-вывода**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Механизм** | **Принцип** | **Преимущества** | **Недостатки** | **Производительность** | **Переносимость** |
| **select()** | Синхронное мультиплексирование, блокируется до готовности FDs.33 | Высокая переносимость между Unix-подобными ОС.33 | Требует перестройки fd\_set перед каждым вызовом, итерирования по всем FDs после возврата.33 | O(n).33 | Высокая.33 |
| **poll()** | Использует массив pollfd структур, блокируется до готовности FDs.33 | Не требует highest\_fd + 1, более эффективен для больших FDs, массив pollfd может быть повторно использован.33 | Менее переносим, чем select().33 | O(n).33 | Средняя.33 |
| **epoll** | Управляет контекстом I/O в ядре, возвращает только готовые FDs.33 | Динамическое добавление/удаление FDs, возвращает только готовые FDs, поддержка триггеров.33 | Непереносим (специфичен для Linux).33 | O(1).33 | Низкая (только Linux).33 |

### **Событийно-ориентированный ввод-вывод и паттерн Reactor**

Событийно-ориентированная архитектура — это парадигма проектирования, в которой поток программы определяется событиями.35 Это распространенный подход для эффективной обработки параллельных запросов, связанных с вводом-выводом.36

#### **Событийно-ориентированная архитектура**

В событийно-ориентированной архитектуре программа регистрирует обратные вызовы для событий и продолжает выполнение, вместо того чтобы блокироваться на вводе-выводе. Когда происходит событие (например, данные готовы на сокете), вызывается связанный обратный вызов.35

Основным компонентом является **цикл событий (Event Loop)**, работающий в одном потоке, который отслеживает файловые дескрипторы на готовность и отправляет события обработчикам.35 Он использует механизмы операционной системы, такие как epoll (Linux), kqueue (OSX), event ports (Solaris) или IOCP (Windows), для мониторинга этих дескрипторов.35 Когда операционная система указывает, что один из этих файловых дескрипторов готов, цикл событий преобразует это в соответствующее событие и вызывает связанные обратные вызовы.35 Тяжелые задачи, которые могут заблокировать цикл событий (например, файловый ввод-вывод, ресурсоемкие вычисления), обрабатываются **пулом рабочих потоков (Worker Pool)**.35

#### **Паттерн Reactor**

Паттерн Reactor — это стратегия обработки событий, которая может одновременно отвечать на множество потенциальных запросов на обслуживание.36 Он полагается на событийные механизмы, а не на блокирующий ввод-вывод или многопоточность, что позволяет ему обрабатывать множество параллельных запросов, связанных с вводом-выводом, с минимальной задержкой.36

Компоненты паттерна Reactor включают:

* **Дескриптор (Handle):** Идентификатор и интерфейс для конкретного запроса, включая ввод-вывод и данные (например, сокет или файловый дескриптор).36
* **Демплексор (Demultiplexer) / Уведомитель событий (Event Notifier):** Эффективно отслеживает состояние дескриптора и уведомляет другие подсистемы об изменении статуса (например, готовности дескриптора ввода-вывода к чтению). Исторически эту роль выполнял select(), но современные примеры включают epoll, kqueue и IOCP.36
* **Диспетчер (Dispatcher):** Цикл событий реактивного приложения, который поддерживает реестр обработчиков событий и вызывает соответствующий обработчик при возникновении события.36
* **Обработчик событий (Event Handler) / Обработчик запросов (Request Handler):** Конкретная логика для обработки типа запроса на обслуживание, регистрируемая как обратные вызовы.36

Преимущества паттерна Reactor включают возможность обработки множества параллельных запросов ввода-вывода с минимальной задержкой.36 Он также позволяет легко изменять или расширять процедуры обработчиков запросов.36 Однако к недостаткам относятся повышенная сложность программы из-за разделения инициирования и завершения операций, а также инвертированный поток управления, что может затруднить отладку.37

Событийно-ориентированные архитектуры и паттерн Reactor абстрагируют сложности низкоуровневого мультиплексирования ввода-вывода (такого как epoll или kqueue), чтобы предоставить более высокоуровневую, управляемую парадигму для создания высокопроизводительных и масштабируемых сетевых приложений с минимальными накладными расходами на потоки. Хотя epoll обеспечивает преимущества в производительности, прямое программирование с использованием epoll все еще может быть сложным. Паттерн Reactor (и событийно-ориентированная архитектура) предоставляет *структурированный способ* использования этих базовых примитивов ОС. Он определяет роли (демультиплексор, диспетчер, обработчик) и поток (регистрация обратного вызова, ожидание события, отправка обратного вызова), что упрощает разработку параллельных приложений. Цикл событий эффективно *организует* использование epoll/kqueue для мониторинга множества файловых дескрипторов, а затем *отправляет* правильную логику на прикладном уровне (обратные вызовы). Это позволяет разработчикам сосредоточиться на логике приложения, а не на непосредственном управлении низкоуровневыми наборами файловых дескрипторов и системными вызовами, при этом достигая высокой параллельности с небольшим количеством потоков.35 Таким образом, паттерн Reactor является важнейшим архитектурным паттерном, который использует эффективные механизмы ввода-вывода ОС для создания масштабируемых, неблокирующих и параллельных приложений. Он обеспечивает более высокий уровень абстракции, делая сложный асинхронный ввод-вывод более управляемым для разработчиков, и является краеугольным камнем многих современных высокопроизводительных сетевых фреймворков.

### **Преимущества асинхронного ввода-вывода**

Асинхронный ввод-вывод (AIO) — это метод, используемый в многопоточных моделях клиент-сервер для достижения высокопараллельных и эффективных по памяти операций ввода-вывода.31 Он решает проблемы традиционных моделей (например, "один поток на клиента" или select() для больших наборов), возвращая управление пользовательскому приложению, пока операции ввода-вывода ожидают завершения.31

Основное преимущество асинхронного ввода-вывода заключается в его способности обеспечивать *перекрывающуюся обработку*, позволяя центральному процессору выполнять другие задачи, пока операции ввода-вывода ожидают завершения. Это фундаментально меняет модель с "ожидания и блокировки" на "инициирование и уведомление", что приводит к превосходному использованию ресурсов и масштабируемости в сетевых приложениях. Хотя мультиплексирование ввода-вывода (такое как select/poll/epoll) позволяет одному потоку *отслеживать* несколько источников ввода-вывода без блокировки на каком-либо *одном* из них, поток все еще *блокируется* в ожидании готовности *любого* из них. Как только событие обнаружено, поток все еще выполняет операцию ввода-вывода (например, чтение данных) синхронно. Асинхронный ввод-вывод идет дальше: он *инициирует* операцию ввода-вывода и немедленно возвращает управление приложению. Затем операция ввода-вывода продолжается в фоновом режиме (часто обрабатывается ядром ОС или выделенными потоками ввода-вывода). Приложение *уведомляется* только по завершении (например, через порт завершения или обратный вызов). Это означает, что поток приложения *никогда* не блокируется в ожидании ввода-вывода, что максимально увеличивает использование ЦП для других задач. Эта "перекрывающаяся обработка" особенно полезна в многопроцессорных системах, поскольку ЦП может быть занят вычислениями, пока ввод-вывод обрабатывается параллельно другими частями системы. Таким образом, асинхронный ввод-вывод представляет собой сдвиг парадигмы в обработке ввода-вывода, переходя от простого избегания блокировки к активному перекрытию вычислений с вводом-выводом. Это наиболее продвинутая и эффективная модель для создания высокопараллельных, масштабируемых и отзывчивых сетевых приложений, особенно тех, которые имеют дело с большими объемами передачи данных или большим количеством соединений.

Преимущества асинхронного ввода-вывода включают:

* **Эффективное использование системных ресурсов:** Копирование данных между пользовательскими буферами и системой происходит асинхронно по отношению к инициирующему приложению, что обеспечивает перекрывающуюся обработку и эффективное использование нескольких процессоров.31 Системные буферы освобождаются для повторного использования.31
* **Минимизация времени ожидания процессов/потоков:** Сокращает время ожидания процессов и потоков.31
* **Немедленное обслуживание клиентских запросов:** Обеспечивает быстрое обслуживание клиентских запросов.31
* **Снижение накладных расходов:** Снижает затраты на "сон" и "пробуждение" потоков.31
* **Эффективная обработка "всплесков" трафика:** Эффективно обрабатывает приложения с "всплесками" активности.31
* **Улучшенная масштабируемость:** Обеспечивает лучшую масштабируемость.31
* **Эффективная передача больших объемов данных:** Является наиболее эффективным методом для передачи больших объемов данных.31
* **Минимизация количества потоков:** Сокращает количество необходимых потоков.31
* **Дополнительные таймеры:** Может использовать таймеры для ограничения времени выполнения операций, позволяя серверам закрывать неактивные соединения.31
* **Асинхронное инициирование защищенных сеансов:** Инициирует защищенные сеансы асинхронно.31

Асинхронный ввод-вывод работает путем создания приложением портов завершения ввода-вывода (например, с помощью API QsoCreateIOCompletionPort()), которые возвращают дескрипторы для планирования и ожидания завершения.31 Рабочие потоки обрабатывают завершенные асинхронные операции, управляя множеством клиентских запросов.31

## **Языки программирования для сетевых приложений**

Выбор языка программирования для сетевых приложений является критически важным решением, которое зависит от конкретных требований к производительности, масштабируемости, простоте разработки и доступности библиотек. В данном разделе будут рассмотрены C++ и Go, два популярных языка, широко используемых в сетевом программировании, с акцентом на их сильные и слабые стороны.

### **C++ для сетевого программирования**

C++ — это мощный, универсальный и высокопроизводительный язык программирования, появившийся в 1979 году.38 Он широко используется в различных отраслях, включая игровую индустрию, финансы и системное программирование.39

#### **Особенности и применение**

C++ предлагает беспрецедентный низкоуровневый контроль и чистую производительность для сетевого программирования, особенно через такие библиотеки, как Boost.Asio. Однако эта мощь сопряжена со значительной сложностью в управлении памятью, асинхронными парадигмами программирования и потенциальными тонкими ошибками, требующими высокой квалификации разработчиков. Философия проектирования C++ предоставляет максимальный контроль разработчику, что позволяет достигать пиковой производительности и тонкой настройки использования ресурсов. Это критически важно для приложений, где каждая микросекунда имеет значение (например, высокочастотная торговля 40). Однако этот контроль означает, что разработчик несет прямую ответственность за сложные задачи, такие как управление памятью (избегание утечек 39) и синхронизация (предотвращение состояний гонки 41). Boost.Asio пытается упростить асинхронный ввод-вывод с помощью паттернов, таких как Proactor, но он все же имеет свою собственную кривую обучения и может иметь накладные расходы на производительность по сравнению с низкоуровневыми подходами.43 "Инвертированный поток управления" 37 в асинхронном программировании, основанном на обратных вызовах, затрудняет отладку. Таким образом, C++ — это язык для высокооптимизированных сетевых приложений, но он требует от разработчиков высокого уровня квалификации для управления его неотъемлемыми сложностями и избегания распространенных ошибок. Преимущества производительности напрямую связаны со способностью разработчика тщательно контролировать низкоуровневые детали, что делает его мощным, но требовательным выбором.

Ключевые особенности и области применения C++:

* **Производительность:** C++ известен своей выдающейся производительностью и высокой скоростью выполнения задач. Он тесно связан с машинным языком, обеспечивая прямой контроль над системными ресурсами, что позволяет разработчикам писать эффективный код.38 Он является предпочтительным выбором для систем с ультранизкой задержкой.40
* **Контроль над ресурсами:** C++ предоставляет прямой доступ к памяти через указатели и возможность манипулировать ресурсами операционной системы на базовом уровне. Это очень эффективно, особенно для системного программирования и встроенных систем.38
* **Мультипарадигменность:** C++ поддерживает несколько парадигм программирования, включая процедурное, объектно-ориентированное (ООП) и обобщенное программирование.38
* **Библиотеки:** Обладает обширной стандартной библиотекой (контейнеры, алгоритмы, ввод-вывод) и многочисленными сторонними библиотеками (GUI, сетевые, научные).38
* **Переносимость:** Код, написанный на C++, может быть скомпилирован и запущен на нескольких операционных системах с минимальными изменениями.38

Несмотря на свои сильные стороны, C++ имеет и недостатки:

* **Сложность для начинающих:** Из-за сложного синтаксиса и ориентации на низкоуровневые манипуляции с данными C++ может быть сложен для изучения начинающими.39
* **Отсутствие сборки мусора:** Разработчики должны вручную управлять выделением и освобождением памяти, что может привести к более сложному коду и потенциальным утечкам памяти, если это не сделано должным образом.39
* **Больше ручного кодирования:** C++ часто требует больше ручного кодирования по сравнению с другими языками, такими как Java или Python, что может увеличить время разработки.39
* **Потенциальные переполнения буфера:** Неосторожные разработчики могут легко вызвать переполнения буфера, что является известной проблемой в C++.44
* **Более медленное время компиляции:** По сравнению с Go, C++ компилируется значительно дольше.44

#### **Boost.Asio: Архитектура и операции**

Boost.Asio — это кроссплатформенная библиотека C++ для сетевого и низкоуровневого программирования ввода-вывода.45 Она предоставляет согласованную асинхронную модель, основанную на паттерне проектирования Proactor.37

Основные концепции Boost.Asio:

* **io\_context:** Ядро цикла событий Asio, управляющее операциями ввода-вывода и планированием асинхронных задач.46 Оно отвечает за диспетчеризацию обработчиков завершения.46
* **tcp::socket:** Представляет TCP-сокет, поддерживающий как синхронные, так и асинхронные операции.45
* **Асинхронные операции:** Инициируются приложением, выполняются внутренними службами и ставят события в очередь завершения.37 Они не блокируют потоки.37
* **Обработчики завершения (Completion Handlers):** Функции обратного вызова, вызываемые по завершении асинхронных операций.47 Они обеспечивают неблокирующий ввод-вывод, разделяя инициирование операции и обработку результата.47 Гарантируется, что обработчики будут выполнены ровно один раз в цепочке вызовов run().46
* **Управление жизненным циклом объектов:** Критически важно для асинхронных операций, чтобы предотвратить висячие ссылки. shared\_ptr и enable\_shared\_from\_this обычно используются для управления владением объектами, гарантируя, что объекты существуют до завершения всех асинхронных операций.41
* **strand:** При использовании нескольких потоков ввода-вывода strands в Asio сериализуют выполнение обработчиков для предотвращения состояний гонки, предоставляя легковесную альтернативу мьютексам путем постановки в очередь параллельных обработчиков.41 Они гарантируют, что обработчики выполняются непоследовательно, устраняя состояния гонки при сохранении пропускной способности.42

Преимущества Boost.Asio: переносимость, производительность, масштабируемость (позволяет обрабатывать тысячи параллельных соединений), упрощенная синхронизация приложений (обработчики могут быть написаны так, как если бы они существовали в однопоточной среде), композиция функций.37 Минимизирует количество потоков ОС и переключение контекста.37

Недостатки Boost.Asio: повышенная сложность программы (разделение инициирования/завершения, инвертированный поток управления), затрудненная отладка.37 Может иметь значительные потери производительности (до 20%) по сравнению с "сырыми" сокетами BSD из-за накладных расходов, таких как полиморфный исполнитель.43 Более высокое использование памяти на сокет по сравнению с альтернативами, такими как libuv.43

### **Go для сетевого программирования**

Go (Golang) — это статически типизированный, компилируемый, высокоуровневый язык общего назначения, разработанный в Google в 2007 году.40

#### **Особенности, конкурентность и стандартная библиотека**

Go был создан для повышения продуктивности программирования в эпоху многоядерных, сетевых машин и больших кодовых баз.40 Его целью было устранить недостатки C++ и Java, сосредоточившись на простоте, эффективности и сильной поддержке параллельного программирования.48

Go предлагает более простой подход к программированию благодаря чистому синтаксису и сниженной сложности, что может привести к более быстрому времени разработки и снижению затрат на обслуживание.40 Однако эта простота достигается за счет меньшего низкоуровневого контроля и потенциальных пауз, связанных со сборщиком мусора, которые могут быть критичны для приложений с ультранизкой задержкой. В отличие от C++, Go абстрагирует многие низкоуровневые детали, такие как управление памятью и потоками, что упрощает разработку, но ограничивает возможности глубокой оптимизации. В то время как C++ требует от разработчика ручного управления памятью и синхронизацией, Go берет на себя эти обязанности, используя сборку мусора и легковесные горутины с каналами. Это ускоряет разработку и уменьшает количество ошибок, но может привести к непредсказуемым задержкам из-за работы сборщика мусора, что неприемлемо для некоторых сценариев с критически низкой задержкой.40 Таким образом, Go представляет собой компромисс между производительностью и простотой разработки, предлагая высокую продуктивность и встроенную поддержку параллелизма, но уступая C++ в абсолютном низкоуровневом контроле и достижении экстремально низкой задержки. Выбор между ними зависит от приоритетов проекта: скорость разработки и простота масштабирования против максимальной производительности и полного контроля.

Ключевые особенности Go:

* **Простота и читаемость:** Минималистский подход, легкость понимания и использования, что приводит к надежному и поддерживаемому коду.48 Краткий синтаксис, отсутствие излишних скобок.44
* **Параллелизм:** Встроенная поддержка параллелизма через легковесные процессы (горутины) и каналы.40 Горутины управляются средой выполнения Go, что упрощает параллельное программирование по сравнению с управлением потоками в C++.40 Модель параллелизма основана на CSP ("не обменивайтесь данными, используя общую память; обменивайтесь данными, обмениваясь сообщениями").49
* **Управление памятью:** Эффективная сборка мусора снижает утечки памяти и упрощает разработку.44
* **Быстрая компиляция:** Компилируется значительно быстрее, чем C++.44
* **Стандартная библиотека:** Богатая стандартная библиотека включает пакеты для работы с сетью, криптографией, JSON.48
* **Безопасность:** Ограничения кода предотвращают переполнения буфера (например, отсутствие арифметики указателей).44

Недостатки Go:

* **Исторически отсутствовала поддержка дженериков** (добавлена в версии 1.18).49
* Механизм panic/recover не является полноценной системой обработки исключений.49
* Строгое форматирование, навязываемое gofmt.49
* Сборщик мусора может вызывать паузы, влияющие на ультранизкую задержку.40
* Меньший низкоуровневый контроль по сравнению с C++.44

#### **Пакеты net, io, bufio, encoding/binary, time, runtime**

Стандартная библиотека Go предоставляет мощный набор пакетов для сетевого программирования и системного мониторинга:

* **Пакет net:** Предоставляет переносимый интерфейс для сетевого ввода-вывода, включая TCP/IP, UDP, разрешение доменных имен и сокеты Unix-домена.50 Предлагает базовые интерфейсы, такие как Dial, Listen и Accept, а также связанные интерфейсы Conn и Listener.50 Операции ввода-вывода в Go по своей природе блокирующие, но параллелизм достигается с помощью горутин и каналов.51 Внутренний netpoller (инкапсулирующий epoll) управляет неблокирующим сетевым вводом-выводом, паркуя горутины, когда они заблокированы, и пробуждая их, когда ввод-вывод готов.51
* **Пакет io:** Определяет фундаментальные интерфейсы, такие как io.Reader и io.Writer, для чтения/записи потоков байтов.52 Многие типы Go (файлы, сетевые соединения, буферы) реализуют эти интерфейсы.52
* **Пакет bufio:** Предоставляет буферизованный ввод-вывод, реализуя io.Reader и io.Writer.52 Он читает/записывает данные блоками для повышения эффективности, уменьшая количество вызовов к базовым источникам данных.52
* **Пакет encoding/binary:** Реализует простое преобразование между числами и последовательностями байтов, а также кодирование и декодирование переменных целых чисел.54 Он критически важен для двоичного кодирования/декодирования, особенно для кадрирования с префиксом длины.11 Поддерживает порядок байтов BigEndian и LittleEndian.54 Ключевые функции включают PutUint32, Uint32, Read, Write.54
* **Пакет time:** Используется для операций, связанных со временем, включая измерение кругового времени задержки (RTT).19 RTT является ключевым показателем задержки сети.19
* **Пакет runtime:** Предоставляет доступ к определенным реализацией метрикам, экспортируемым средой выполнения Go.57 Это позволяет разработчикам получать подробные сведения о характеристиках производительности своих программ Go, выявлять узкие места и оптимизировать использование ресурсов.57 Например, можно отслеживать использование памяти кучи и стеков горутин для выявления утечек памяти или горутин.57

**Таблица 9: Сравнение C++ и Go для сетевого программирования**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **C++** | **Go (Golang)** |
| **Производительность** | Превосходная, низкоуровневый контроль, подходит для ультранизкой задержки.38 | Очень быстр, но сборщик мусора может вызывать паузы, влияющие на ультранизкую задержку.40 |
| **Параллелизм** | Требует ручного управления потоками, более сложен (но улучшен в C++11+).40 | Встроенная поддержка через горутины и каналы, упрощает параллельное программирование.40 |
| **Управление памятью** | Ручное управление памятью (без сборки мусора), потенциальные утечки.39 | Автоматическая сборка мусора, снижает риск утечек и упрощает разработку.48 |
| **Сложность разработки** | Высокая кривая обучения, более ручное кодирование.39 | Более простой синтаксис, быстрая разработка, сниженная сложность.40 |
| **Время компиляции** | Часто медленнее.44 | Значительно быстрее.44 |
| **Безопасность** | Потенциальные переполнения буфера, требует осторожности.44 | Встроенные ограничения кода предотвращают переполнения буфера (например, нет арифметики указателей).44 |
| **Применение в сети** | Высокочастотная торговля, игровые движки, системное программирование.40 | Сетевые серверы, прокси, балансировщики нагрузки, распределенные системы.40 |

## **Заключение**

Теоретические основы сетевого взаимодействия и программирования раскрывают сложную, но логически структурированную природу современных компьютерных сетей. От абстрактных моделей OSI и TCP/IP до низкоуровневых механизмов TCP и практических аспектов программирования сокетов, каждый компонент демонстрирует продуманные инженерные решения, направленные на обеспечение надежной и эффективной связи.

Обзор сетевых моделей показывает фундаментальный компромисс между теоретической полнотой и практической применимостью. Модель OSI, с ее детальным семиуровневым подходом, служит отличной основой для стандартизации и анализа, в то время как более простая и ориентированная на реальные протоколы модель TCP/IP является движущей силой современного Интернета. Это подчеркивает, что в проектировании систем часто приходится выбирать между идеальной абстракцией и прагматичной реализацией.

Протокол TCP, как краеугольный камень надежной связи, демонстрирует, как надежный сервис может быть построен поверх изначально ненадежной базовой сети. Его ориентированный на соединение характер, трехстороннее и четырехстороннее рукопожатия, а также механизмы подтверждения и повторной передачи (включая адаптивную быструю повторную передачу) являются примерами сложного проектирования, направленного на обеспечение целостности и порядка данных. Важно отметить, что потоково-ориентированный характер TCP требует, чтобы приложения реализовывали свои собственные механизмы кадрирования, что является критическим аспектом проектирования на прикладном уровне.

В области программирования сокетов выбор между потоковыми и дейтаграммными сокетами отражает компромисс между надежностью и скоростью, что напрямую влияет на требования к приложению. Асимметрия операций клиент-серверных сокетов является архитектурной необходимостью для масштабируемых сетевых служб. Эволюция моделей ввода-вывода от блокирующих к неблокирующим и, далее, к мультиплексированию ввода-вывода (такому как select(), poll(), epoll) и событийно-ориентированным архитектурам (с паттерном Reactor) показывает постоянное стремление к повышению масштабируемости и эффективности обработки параллельных соединений. Асинхронный ввод-вывод, с его способностью к перекрывающейся обработке, представляет собой наиболее продвинутую парадигму для высокопроизводительных сетевых приложений.

Наконец, сравнение языков программирования, таких как C++ и Go, для сетевых приложений демонстрирует компромисс между низкоуровневым контролем и производительностью, с одной стороны, и простотой разработки и встроенным параллелизмом, с другой. C++ предлагает беспрецедентную мощь для ультранизкой задержки, но требует высокой квалификации для управления его сложностями. Go, с его горутинами и богатой стандартной библиотекой, обеспечивает высокую продуктивность и эффективный параллелизм, что делает его отличным выбором для многих сетевых задач, хотя он может уступать C++ в экстремальных сценариях с низкой задержкой.

В целом, сетевые технологии постоянно развиваются, адаптируясь к новым требованиям и вызовам. Глубокое понимание этих теоретических основ позволяет разработчикам и инженерам принимать обоснованные решения, проектировать надежные, масштабируемые и высокопроизводительные сетевые приложения, а также эффективно устранять возникающие проблемы.

#### Источники

1. OSI Model vs TCP/IP Model - Check Point Software, дата последнего обращения: мая 25, 2025, <https://www.checkpoint.com/cyber-hub/network-security/what-is-the-osi-model-understanding-the-7-layers/osi-model-vs-tcp-ip-model/>
2. mochieng.files.wordpress.com, дата последнего обращения: мая 25, 2025, <https://mochieng.files.wordpress.com/2012/04/wp_nw_tcp-ip1.pdf>
3. Transmission Control Protocol - Wikipedia, дата последнего обращения: мая 25, 2025, <https://en.wikipedia.org/wiki/Transmission_Control_Protocol>
4. RFC 9293: Transmission Control Protocol (TCP), дата последнего обращения: мая 25, 2025, <https://www.rfc-editor.org/rfc/rfc9293.html>
5. TCP 3-Way Handshake Process | GeeksforGeeks, дата последнего обращения: мая 25, 2025, <https://www.geeksforgeeks.org/tcp-3-way-handshake-process/>
6. TCP Fast Retransmit: Best Practices and Optimization Tips | Orhan ..., дата последнего обращения: мая 25, 2025, <https://orhanergun.net/tcp-fast-retransmit>
7. What is TCP Retransmission? Understanding Network Reliability ..., дата последнего обращения: мая 25, 2025, <https://www.byteplus.com/en/topic/100380>
8. A look at the TCP segment format | cadlag dot org, дата последнего обращения: мая 25, 2025, <https://www.cadlag.org/posts/tcp-segment-format/>
9. Introduction to Serial Framing Formats | Dojo Five, дата последнего обращения: мая 25, 2025, <https://dojofive.com/blog/introduction-to-serial-framing-formats/>
10. TCP Socket Framing Specification - SAP Help Portal, дата последнего обращения: мая 25, 2025, <https://help.sap.com/docs/ABAP_PLATFORM_NEW/05d041d3df1a4595a3c45f57c15e2325/2f9fc283ca4c43b1bb7a54fe504d7fff.html>
11. Read whole message with bufio.NewReader(conn) - Stack Overflow, дата последнего обращения: мая 25, 2025, <https://stackoverflow.com/questions/44159811/read-whole-message-with-bufio-newreaderconn>
12. Type–length–value - Wikipedia, дата последнего обращения: мая 25, 2025, <https://en.wikipedia.org/wiki/Type%E2%80%93length%E2%80%93value>
13. TCP flow control and the sliding window - IBM, дата последнего обращения: мая 25, 2025, <https://www.ibm.com/docs/en/storage-protect/8.1.24?topic=tuning-tcp-flow-control>
14. Flow Control vs. Congestion Control in TCP | Baeldung on ..., дата последнего обращения: мая 25, 2025, <https://www.baeldung.com/cs/tcp-flow-control-vs-congestion-control>
15. What is the TCP three-way handshake process? - Backup Education, дата последнего обращения: мая 25, 2025, <https://backup.education/showthread.php?tid=1757&pid=1754>
16. The three-way handshake via TCP/IP - Windows Server | Microsoft ..., дата последнего обращения: мая 25, 2025, <https://learn.microsoft.com/en-us/troubleshoot/windows-server/networking/three-way-handshake-via-tcpip>
17. In-depth understanding of 3-Way Handshake Connection and 4 ..., дата последнего обращения: мая 25, 2025, <https://www.dingyuqi.com/en/article/3y3tmv0x/>
18. What is a 4-way handshake in TCP? - Quora, дата последнего обращения: мая 25, 2025, <https://www.quora.com/What-is-a-4-way-handshake-in-TCP>
19. What Is Round-Trip Time (RTT)? - JumpCloud, дата последнего обращения: мая 25, 2025, <https://jumpcloud.com/it-index/what-is-round-trip-time-rtt>
20. Program to calculate the Round Trip Time (RTT) | GeeksforGeeks, дата последнего обращения: мая 25, 2025, <https://www.geeksforgeeks.org/program-calculate-round-trip-time-rtt/>
21. TCP Congestion Control Algorithms: Reno, New Reno, BIC, CUBIC ..., дата последнего обращения: мая 25, 2025, <https://www.geeksforgeeks.org/tcp-congestion-control-algorithms-reno-new-reno-bic-cubic/>
22. Congestion Control Algorithms in TCP Acceleration Between ..., дата последнего обращения: мая 25, 2025, <https://support.gfi.com/article/107765-congestion-control-algorithms-in-tcp-acceleration-between-appliances>
23. Network byte order and host byte order - IBM, дата последнего обращения: мая 25, 2025, <https://www.ibm.com/docs/ja/zvm/7.2?topic=domains-network-byte-order-host-byte-order>
24. www2.cs.arizona.edu, дата последнего обращения: мая 25, 2025, <https://www2.cs.arizona.edu/classes/cs525/papers/byteorder.pdf>
25. Socket Programming in Computer Network - Scaler Topics, дата последнего обращения: мая 25, 2025, <https://www.scaler.com/topics/computer-network/socket-programming/>
26. Socket Programming in C | GeeksforGeeks, дата последнего обращения: мая 25, 2025, <https://www.geeksforgeeks.org/socket-programming-cc/>
27. Socket Programming - Dartmouth, дата последнего обращения: мая 25, 2025, <https://www.cs.dartmouth.edu/~campbell/cs50/socketprogramming.html>
28. socket — Low-level networking interface — Python 3.13.3 documentation, дата последнего обращения: мая 25, 2025, <https://docs.python.org/3/library/socket.html>
29. Understanding Blocking and Non-blocking Sockets in C ..., дата последнего обращения: мая 25, 2025, <https://dev.to/vivekyadav200988/understanding-blocking-and-non-blocking-sockets-in-c-programming-a-comprehensive-guide-2ien>
30. Blocking and Nonblocking IO in Operating System | GeeksforGeeks, дата последнего обращения: мая 25, 2025, <https://www.geeksforgeeks.org/blocking-and-nonblocking-io-in-operating-system/>
31. Asynchronous I/O - IBM, дата последнего обращения: мая 25, 2025, <https://www.ibm.com/docs/en/i/7.4.0?topic=concepts-asynchronous-io>
32. Asynchronous I/O - Network Programming, дата последнего обращения: мая 25, 2025, <http://networkprogrammingnotes.blogspot.com/p/asynchronous-io-asynchronous-io-apis.html>
33. Linux – IO Multiplexing – Select vs Poll vs Epoll – Developers Area, дата последнего обращения: мая 25, 2025, <https://devarea.com/linux-io-multiplexing-select-vs-poll-vs-epoll/>
34. I/O multiplexing—select() - IBM, дата последнего обращения: мая 25, 2025, <https://www.ibm.com/docs/en/i/7.4.0?topic=concepts-io-multiplexingselect>
35. Node.js — Don't Block the Event Loop (or the Worker Pool), дата последнего обращения: мая 25, 2025, <https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop>
36. Reactor pattern - Wikipedia, дата последнего обращения: мая 25, 2025, <https://en.wikipedia.org/wiki/Reactor_pattern>
37. The Proactor Design Pattern: Concurrency Without Threads - 1.79.0 - Boost, дата последнего обращения: мая 25, 2025, <https://original.boost.org/doc/libs/1_79_0/doc/html/boost_asio/overview/core/async.html>
38. What is C++ Programming Language? A Compherensive Guide, дата последнего обращения: мая 25, 2025, <https://www.wscubetech.com/resources/cpp/what-is-cpp>
39. A Comprehensive Guide to C++: Advantages and Disadvantages ..., дата последнего обращения: мая 25, 2025, <https://pangea.ai/resources/a-comprehensive-guide-to-c-advantages-and-disadvantages>
40. Choosing the Right Programming Language for Low Latency ..., дата последнего обращения: мая 25, 2025, <https://dev.to/adityabhuyan/choosing-the-right-programming-language-for-low-latency-applications-go-vs-c-20m4>
41. Managing connection lifecycle - Mastering Network Programming with Asio C++: A Comprehensive Guide | StudyRaid, дата последнего обращения: мая 25, 2025, <https://app.studyraid.com/en/read/12426/401286/managing-connection-lifecycle>
42. Synchronization patterns with strands - Mastering Network Programming with Asio C++: A Comprehensive Guide | StudyRaid, дата последнего обращения: мая 25, 2025, <https://app.studyraid.com/en/read/12426/401311/synchronization-patterns-with-strands>
43. ASIO makes for a bloated Networking standard : r/cpp - Reddit, дата последнего обращения: мая 25, 2025, <https://www.reddit.com/r/cpp/comments/h0954a/asio_makes_for_a_bloated_networking_standard/>
44. Golang vs. C++ - A Complete Comparison - Blackdown, дата последнего обращения: мая 25, 2025, <https://www.blackdown.org/golang-vs-c/>
45. Boost.Asio - Scientific Computing Wiki, дата последнего обращения: мая 25, 2025, <https://scicomp.ethz.ch/public/manual/Boost/1.55.0/asio.pdf>
46. Understanding IO Context and its lifecycle - Mastering Network Programming with Asio C++: A Comprehensive Guide | StudyRaid, дата последнего обращения: мая 25, 2025, <https://app.studyraid.com/en/read/12426/401278/understanding-io-context-and-its-lifecycle>
47. Understanding completion handlers - Mastering Network Programming with Asio C++: A Comprehensive Guide | StudyRaid, дата последнего обращения: мая 25, 2025, <https://app.studyraid.com/en/read/12426/401292/understanding-completion-handlers>
48. What is the history and background of the Go language? - Tencent ..., дата последнего обращения: мая 25, 2025, <https://www.tencentcloud.com/techpedia/101475>
49. Go (programming language) - Wikipedia, дата последнего обращения: мая 25, 2025, <https://en.wikipedia.org/wiki/Go_(programming_language)>
50. net - The Go Programming Language - Golang Documentation, дата последнего обращения: мая 25, 2025, <https://documentation.help/Golang/net.htm>
51. Go Internals HTTP request multiplexing in Go - Bits and Bytes, дата последнего обращения: мая 25, 2025, <https://akshay-kumar.hashnode.dev/go-internals-http-request-multiplexing-in-go-1>
52. Golang Bufio (A Complete Guide) - Kelche, дата последнего обращения: мая 25, 2025, <https://www.kelche.co/blog/go/golang-bufio/>
53. Go I/O Readers, Writers, and Data in Motion - VictoriaMetrics, дата последнего обращения: мая 25, 2025, <https://victoriametrics.com/blog/go-io-reader-writer/>
54. binary - The Go Programming Language, дата последнего обращения: мая 25, 2025, <https://www.cs.ubc.ca/~bestchai/teaching/cs416_2015w2/go1.4.3-docs/pkg/encoding/binary/index.html>
55. How to implement encoding methods | LabEx, дата последнего обращения: мая 25, 2025, <https://labex.io/tutorials/go-how-to-implement-encoding-methods-434135>
56. Go Walkthrough: encoding/binary - Go Beyond, дата последнего обращения: мая 25, 2025, <https://www.gobeyond.dev/encoding-binary/>
57. Go memory metrics demystified | Datadog, дата последнего обращения: мая 25, 2025, <https://www.datadoghq.com/blog/go-memory-metrics/>
58. metrics package - runtime/metrics - Go Packages, дата последнего обращения: мая 25, 2025, <https://pkg.go.dev/runtime/metrics>

**Приложение А. CPP реализация TCP.**

tcp\_client.cpp

*// benchmark/client/tcp\_client.cpp*

#include "../include/config.hpp"

#include "../common/include/chunk\_reader.hpp"

#include "../common/include/reversal\_utils.hpp"

#include "../common/include/metrics\_aggregator.hpp"

#include "../common/include/tcp\_messaging.hpp"

#include "../common/include/file\_utils.hpp"

#include <boost/asio.hpp>

#include <iostream>

#include <vector>

#include <memory>

#include <array>

#include <string>

#include <filesystem> *// Для std::filesystem*

namespace fs = std::filesystem;

using boost::asio::ip::tcp;

class TCPClient : public std::enable\_shared\_from\_this<TCPClient> {

public:

TCPClient(boost::asio::io\_context& *io\_context*,

const std::string& *host*,

unsigned short *port*,

MetricsAggregator& *metrics*)

: m\_io\_context(io\_context),

m\_socket(io\_context),

m\_resolver(io\_context),

m\_metrics(metrics),

m\_host(host),

m\_port\_str(std::to\_string(port)),

m\_chunk\_reader(config::TEST\_FILE\_NAME, config::CHUNK\_SIZE)

{

m\_total\_chunks\_to\_send = m\_chunk\_reader.total\_chunks();

if (m\_chunk\_reader.file\_size() > 0 && m\_total\_chunks\_to\_send == 0 && m\_chunk\_reader.chunks\_read() == 0) { *// File < chunk\_size*

m\_total\_chunks\_to\_send = 1;

} else if (m\_chunk\_reader.file\_size() == 0) {

m\_total\_chunks\_to\_send = 0;

}

std::cout << "TCPClient: Total chunks to send: " << m\_total\_chunks\_to\_send << " (from file size: " << m\_chunk\_reader.file\_size() << ")" << std::endl;

}

void start() {

auto self = shared\_from\_this();

m\_resolver.async\_resolve(m\_host, m\_port\_str,

[**this**, *self*](const boost::system::error\_code& *ec*, tcp::resolver::results\_type *endpoints*) {

if (!ec) {

do\_connect(endpoints);

} else {

std::cerr << "TCP Client: Resolve error: " << ec.message() << std::endl;

stop\_client\_operations(true);

}

});

}

private:

void stop\_client\_operations(bool *error\_occurred*) {

if (!m\_operations\_stopped) {

m\_operations\_stopped = true;

if (!m\_timer\_stopped\_flag) {

m\_metrics.stop\_timer();

m\_timer\_stopped\_flag = true;

}

boost::system::error\_code ignored\_ec;

if (m\_socket.is\_open()){

m\_socket.shutdown(tcp::socket::shutdown\_both, ignored\_ec);

m\_socket.close(ignored\_ec);

}

if (!m\_io\_context.stopped()) {

std::cout << "TCP Client: Stopping io\_context explicitly." << std::endl;

m\_io\_context.stop();

}

if (error\_occurred) {

std::cout << "TCP Client: Operations stopped due to an error." << std::endl;

} else {

std::cout << "TCP Client: Operations finished successfully." << std::endl;

}

}

}

void do\_connect(const *tcp*::resolver::results\_type& *endpoints*) {

auto self = shared\_from\_this();

boost::asio::async\_connect(m\_socket, endpoints,

[**this**, *self*](const *boost*::system::error\_code& *ec*, const tcp::endpoint& */\*endpoint\*/*) {

if (!ec) {

std::cout << "TCP Client: Connected to "

<< m\_socket.remote\_endpoint().address().to\_string()

<< ":" << m\_socket.remote\_endpoint().port() << std::endl;

m\_metrics.start\_timer();

send\_next\_chunk();

} else {

std::cerr << "TCP Client: Connect error: " << ec.message() << std::endl;

stop\_client\_operations(true);

}

});

}

void send\_next\_chunk() {

if (m\_operations\_stopped) return;

if (m\_chunks\_sent >= m\_total\_chunks\_to\_send || m\_chunk\_reader.eof()) {

if (m\_chunk\_reader.file\_size() == 0 && m\_chunks\_sent == 0) {

std::cout << "TCP Client: Test file is empty. Nothing to send." << std::endl;

} else {

std::cout << "TCP Client: All " << m\_chunks\_sent

<< " chunks processed or EOF reached (ChunkReader EOF: " << m\_chunk\_reader.eof()

<< ", Total to send: " << m\_total\_chunks\_to\_send

<< "). Chunks read by reader: " << m\_chunk\_reader.chunks\_read() << std::endl;

}

stop\_client\_operations(false);

return;

}

m\_current\_chunk\_data = m\_chunk\_reader.read\_next\_chunk();

if (m\_current\_chunk\_data.empty() && !m\_chunk\_reader.eof()) {

std::cerr << "TCP Client: Read empty chunk unexpectedly before EOF (chunks\_sent: " << m\_chunks\_sent << "). Aborting." << std::endl;

m\_metrics.record\_chunk\_verified(false);

stop\_client\_operations(true);

return;

}

if (m\_current\_chunk\_data.empty() && m\_chunk\_reader.eof()) {

std::cout << "TCP Client: Reached true EOF after reading last chunk. Processed " << m\_chunks\_sent << " chunks." << std::endl;

stop\_client\_operations(false);

return;

}

m\_expected\_reversed\_chunk = utils::get\_reversed\_vector\_content(m\_current\_chunk\_data);

m\_metrics.start\_chunk\_rtt\_timer();

auto self = shared\_from\_this();

auto buffers\_to\_send = tcp\_messaging::prepare\_message(m\_current\_chunk\_data, m\_write\_header\_buffer);

boost::asio::async\_write(m\_socket, buffers\_to\_send,

[**this**, *self*](const *boost*::system::error\_code& *ec*, std::size\_t *bytes\_transferred*) {

if (m\_operations\_stopped) return;

if (!ec) {

std::cout << "TCP Client: Sent chunk " << (m\_chunks\_sent + 1) << "/" << m\_total\_chunks\_to\_send

<< " (" << bytes\_transferred - tcp\_messaging::HEADER\_SIZE << " payload bytes)" << std::endl;

do\_read\_header();

} else {

std::cerr << "TCP Client: Write error: " << ec.message() << std::endl;

stop\_client\_operations(true);

}

});

}

void do\_read\_header() {

if (m\_operations\_stopped) return;

auto self = shared\_from\_this();

tcp\_messaging::async\_read\_header(m\_socket, m\_read\_header\_buffer,

[**this**, *self*](const *boost*::system::error\_code& *ec*, std::size\_t ) {

if (m\_operations\_stopped) return;

if (!ec) {

uint32\_t body\_length = tcp\_messaging::parse\_header(m\_read\_header\_buffer);

if (body\_length > config::CHUNK\_SIZE \* 2) {

std::cerr << "TCP Client: Excessive body length in response: " << body\_length << ". Max expected: " << config::CHUNK\_SIZE << ". Closing." << std::endl;

stop\_client\_operations(true);

return;

}

m\_read\_body\_buffer.resize(body\_length); *// Prepare buffer for body*

if (body\_length == 0) {

std::cout << "TCP Client: Received header for 0-length body." << std::endl;

handle\_received\_chunk\_data();

} else {

do\_read\_body();

}

} else {

if (ec == boost::asio::error::eof) {

std::cout << "TCP Client: Server closed connection while reading header." << std::endl;

if (m\_chunks\_sent >= m\_total\_chunks\_to\_send) {

std::cout << "TCP Client: EOF from server, assuming all data processed." << std::endl;

stop\_client\_operations(false);

} else {

std::cerr << "TCP Client: EOF from server before all data processed. Chunks sent: " << m\_chunks\_sent << "/" << m\_total\_chunks\_to\_send << std::endl;

stop\_client\_operations(true);

}

} else {

std::cerr << "TCP Client: Read header error: " << ec.message() << std::endl;

stop\_client\_operations(true);

}

}

});

}

void do\_read\_body() {

if (m\_operations\_stopped) return;

auto self = shared\_from\_this();

boost::asio::async\_read(m\_socket, boost::asio::buffer(m\_read\_body\_buffer.data(), m\_read\_body\_buffer.size()),

boost::asio::transfer\_exactly(m\_read\_body\_buffer.size()),

[**this**, *self*](const *boost*::system::error\_code& *ec*, std::size\_t *length\_read*) {

if (m\_operations\_stopped) return;

if (!ec) {

if (length\_read != m\_read\_body\_buffer.size()) {

std::cerr << "TCP Client: Read body error: Incomplete read. Expected "

<< m\_read\_body\_buffer.size() << ", got " << length\_read << std::endl;

stop\_client\_operations(true);

return;

}

handle\_received\_chunk\_data();

} else {

if (ec == boost::asio::error::eof) {

std::cerr << "TCP Client: Server closed connection while reading body. Expected "

<< m\_read\_body\_buffer.size() << " bytes." << std::endl;

} else {

std::cerr << "TCP Client: Read body error: " << ec.message() << std::endl;

}

stop\_client\_operations(true);

}

});

}

void handle\_received\_chunk\_data() {

if (m\_operations\_stopped) return;

bool verified = (m\_read\_body\_buffer == m\_expected\_reversed\_chunk);

m\_metrics.stop\_and\_record\_chunk\_rtt(m\_current\_chunk\_data.size(), verified); *// Stop RTT timer and record*

if (!verified) {

std::cerr << "TCP Client: ERROR! Chunk " << (m\_chunks\_sent + 1)

<< " (original size: " << m\_current\_chunk\_data.size()

<< ", received size: " << m\_read\_body\_buffer.size()

<< ") verification FAILED." << std::endl;

stop\_client\_operations(true);

return;

}

m\_chunks\_sent++;

std::cout << "TCP Client: Chunk " << m\_chunks\_sent << " verified OK. (" << m\_read\_body\_buffer.size() << " bytes)" << std::endl;

if (m\_chunks\_sent >= m\_total\_chunks\_to\_send || m\_chunk\_reader.eof()) {

std::cout << "TCP Client: Successfully processed all " << m\_chunks\_sent << " chunks." << std::endl;

stop\_client\_operations(false);

} else {

send\_next\_chunk();

}

}

boost::asio::io\_context& m\_io\_context;

tcp::socket m\_socket;

tcp::resolver m\_resolver;

MetricsAggregator& m\_metrics;

std::string m\_host;

std::string m\_port\_str;

ChunkReader m\_chunk\_reader;

std::vector<char> m\_current\_chunk\_data;

std::vector<char> m\_expected\_reversed\_chunk;

std::array<char, tcp\_messaging::HEADER\_SIZE> m\_write\_header\_buffer;

std::array<char, tcp\_messaging::HEADER\_SIZE> m\_read\_header\_buffer;

std::vector<char> m\_read\_body\_buffer;

std::size\_t m\_chunks\_sent = 0;

std::size\_t m\_total\_chunks\_to\_send = 0;

bool m\_timer\_stopped\_flag = false;

bool m\_operations\_stopped = false;

};

int main(int *argc*, char\* *argv*[]) {

try {

std::string server\_ip = config::DEFAULT\_SERVER\_IP;

if (*argc* > 1) {

server\_ip = *argv*[1];

std::cout << "TCP Client: Using server IP from argument: " << server\_ip << std::endl;

} else {

std::cout << "TCP Client: Using default server IP: " << server\_ip << std::endl;

}

std::cout << "TCP Client: Target file size: " << config::TOTAL\_FILE\_SIZE / (1024.0\*1024.0) << " MB, Chunk size: " << config::CHUNK\_SIZE / 1024.0 << " KB." << std::endl;

generate\_test\_file\_if\_not\_exists(config::TEST\_FILE\_NAME, config::TOTAL\_FILE\_SIZE);

if (!fs::exists(config::TEST\_FILE\_NAME)) {

std::cerr << "TCP Client: Test file '" << config::TEST\_FILE\_NAME

<< "' could not be created or found. Aborting." << std::endl;

return 1;

}

if (fs::file\_size(config::TEST\_FILE\_NAME) != config::TOTAL\_FILE\_SIZE) {

std::cerr << "TCP Client: Test file '" << config::TEST\_FILE\_NAME

<< "' has incorrect size. Expected " << config::TOTAL\_FILE\_SIZE

<< ", got " << fs::file\_size(config::TEST\_FILE\_NAME) << ". Aborting." << std::endl;

return 1;

}

boost::asio::io\_context io\_context;

MetricsAggregator metrics("CPP\_TCP", config::TOTAL\_FILE\_SIZE, config::CHUNK\_SIZE);

auto client = std::make\_shared<TCPClient>(io\_context, server\_ip, config::TCP\_SERVER\_PORT, metrics);

client->start();

io\_context.run();

std::cout << "TCP Client: io\_context.run() finished." << std::endl;

metrics.print\_summary();

metrics.save\_to\_csv(config::CPP\_OVERALL\_METRICS\_FILE, config::CPP\_CHUNK\_RTT\_METRICS\_FILE);

} catch (const std::exception& e) {

std::cerr << "TCP Client Exception in main: " << e.what() << std::endl;

return 1;

}

return 0;

}

Tcp\_server.cpp

*// benchmark/server/tcp\_server.cpp*

#include "config.hpp"

#include "reversal\_utils.hpp"

#include "tcp\_messaging.hpp"

#include <array>

#include <boost/asio.hpp>

#include <iostream>

#include <memory>

#include <vector>

using boost::asio::ip::tcp;

class TCPSession : public std::enable\_shared\_from\_this<TCPSession> {

public:

TCPSession(tcp::socket *socket*) : m\_socket(std::move(socket)) {

std::cout << "TCP Session: New connection from "

<< m\_socket.remote\_endpoint().address().to\_string() << ":"

<< m\_socket.remote\_endpoint().port() << std::endl;

}

~TCPSession() {

std::cout << "TCP Session: Connection closed with "

<< m\_socket.remote\_endpoint().address().to\_string() *// This*

might throw if socket is already invalid

<< std::endl;

}

void start() { do\_read\_header(); }

private:

void do\_read\_header() {

auto self = shared\_from\_this();

tcp\_messaging::async\_read\_header(

m\_socket, m\_read\_header\_buffer,

[**this**,

self](const boost::system::error\_code &ec,

std::size\_t */\*length - not provided by this handler variant \*/*) {

if (!ec) {

uint32\_t body\_length =

tcp\_messaging::parse\_header(m\_read\_header\_buffer);

std::cout << "TCP Session: Received header for body of length: "

<< body\_length << std::endl;

if (body\_length ==

0) {

m\_read\_body\_buffer.clear();

do\_write();

return;

}

if (body\_length > config::CHUNK\_SIZE \* 2) {

std::cerr << "TCP Session: Excessive body length received: "

<< body\_length

<< ". Max expected around: " << config::CHUNK\_SIZE

<< ". Closing session." << std::endl;

return;

}

m\_read\_body\_buffer.resize(body\_length);

do\_read\_body();

} else {

if (ec == boost::asio::error::eof) {

std::cout << "TCP Session: Client disconnected gracefully (EOF "

"on header read)."

<< std::endl;

} else if (ec == boost::asio::error::operation\_aborted) {

std::cout

<< "TCP Session: Operation aborted (likely server shutdown)."

<< std::endl;

} else {

std::cerr << "TCP Session: Error reading header: " << ec.message()

<< std::endl;

}

}

});

}

void do\_read\_body() {

auto self = shared\_from\_this();

boost::asio::async\_read(

m\_socket,

boost::asio::buffer(m\_read\_body\_buffer.data(),

m\_read\_body\_buffer.size()),

boost::asio::transfer\_exactly(

m\_read\_body\_buffer.size()),

[**this**, *self*](const boost::system::error\_code &*ec*,

std::size\_t *length\_read*) {

if (!ec) {

std::cout << "TCP Session: Read body of size " << length\_read <<

std::endl;

if (length\_read != m\_read\_body\_buffer.size()) {

std::cerr << "TCP Session: Incomplete body read. Expected "

<< m\_read\_body\_buffer.size() << ", got " << length\_read

<< ". Closing." << std::endl;

return;

}

utils::reverse\_vector\_content(m\_read\_body\_buffer);

do\_write();

} else {

if (ec == boost::asio::error::eof) {

std::cout

<< "TCP Session: Client disconnected while reading body."

<< std::endl;

} else if (ec == boost::asio::error::operation\_aborted) {

std::cout << "TCP Session: Operation aborted (likely server "

"shutdown) while reading body."

<< std::endl;

} else {

std::cerr << "TCP Session: Error reading body: " << ec.message()

<< std::endl;

}

}

});

}

void do\_write() {

auto self = shared\_from\_this();

auto buffers\_to\_send = tcp\_messaging::prepare\_message(

m\_read\_body\_buffer, m\_write\_header\_buffer);

boost::asio::async\_write(

m\_socket, buffers\_to\_send,

[**this**, *self*](const boost::system::error\_code &*ec*,

std::size\_t *bytes\_transferred*) {

if (!ec) {

std::cout << "TCP Session: Wrote response of "

<< (bytes\_transferred - tcp\_messaging::HEADER\_SIZE) <<

" payload bytes." << std::endl;

do\_read\_header(); *// Ready for the next message from this client*

} else {

if (ec == boost::asio::error::operation\_aborted) {

std::cout << "TCP Session: Operation aborted (likely server "

"shutdown) while writing."

<< std::endl;

} else {

std::cerr << "TCP Session: Error writing response: "

<< ec.message() << std::endl;

}

}

});

}

tcp::socket m\_socket;

std::array<char, tcp\_messaging::HEADER\_SIZE> m\_read\_header\_buffer;

std::vector<char> m\_read\_body\_buffer;

std::array<char, tcp\_messaging::HEADER\_SIZE> m\_write\_header\_buffer;

};

class TCPServer {

public:

TCPServer(boost::asio::io\_context &*io\_context*, unsigned short *port*)

: m\_io\_context(io\_context),

m\_acceptor(io\_context, tcp::endpoint(tcp::v4(), port)) {

std::cout << "TCP Server listening on port " << port << std::endl;

do\_accept();

}

private:

void do\_accept() {

m\_acceptor.async\_accept([**this**](const *boost*::system::error\_code &*ec*,

tcp::socket *socket*) {

if (!ec) {

std::make\_shared<TCPSession>(std::move(socket))->start();

} else {

if (ec == boost::asio::error::operation\_aborted) {

std::cout

<< "TCP Server: Accept operation aborted (server shutting down?)."

<< std::endl;

return;

}

std::cerr << "TCP Server: Accept error: " << ec.message() << std::endl;

}

if (m\_acceptor.is\_open()) {

do\_accept();

} else {

std::cout

<< "TCP Server: Acceptor is closed. Not accepting new connections."

<< std::endl;

}

});

}

boost::asio::io\_context &m\_io\_context;

tcp::acceptor m\_acceptor;

};

int main() {

try {

boost::asio::io\_context io\_context;

TCPServer server(io\_context, config::TCP\_SERVER\_PORT);

boost::asio::signal\_set signals(io\_context, SIGINT, SIGTERM);

signals.async\_wait([&](const *boost*::system::error\_code & ,

int ) {

std::cout << "TCP Server: Shutdown signal received. Stopping io\_context."

<< std::endl;

if (!io\_context.stopped()) {

io\_context.stop();

}

});

io\_context.run();

std::cout << "TCP Server: io\_context.run() finished. Server has shut down."

<< std::endl;

} catch (const std::exception &e) {

std::cerr << "TCP Server Exception in main: " << e.what() << std::endl;

return 1;

}

return 0;

}

**Приложение B. Go-реализация TCP.**

server/[main.go](http://main.go)

package main

import (

"src/go\_tcp\_benchmark/pkg/config"

"src/go\_tcp\_benchmark/pkg/messaging"

"src/go\_tcp\_benchmark/pkg/reversal"

"io"

"log"

"net"

)

func handleConnection(conn net.Conn) {

defer conn.Close()

remoteAddr := conn.RemoteAddr().String()

log.Printf("Server: Accepted connection from %s\n", remoteAddr)

for {

receivedData, err := messaging.ReadMessage(conn)

if err != nil {

if err == io.EOF {

log.Printf("Server: Client %s disconnected gracefully.\n", remoteAddr)

} else {

log.Printf("Server: Error reading from client %s: %v\n", remoteAddr, err)

}

return

}

log.Printf("Server: Received %d bytes from %s\n", len(receivedData), remoteAddr)

reversal.ReverseBytesInPlace(receivedData)

err = messaging.WriteMessage(conn, receivedData)

if err != nil {

log.Printf("Server: Error writing to client %s: %v\n", remoteAddr, err)

return

}

log.Printf("Server: Sent %d reversed bytes to %s\n", len(receivedData), remoteAddr)

}

}

func main() {

*// "tcp" - протокол, ":" + config.TCPServerPort - слушать на всех интерфейсах на этом порту.*

listener, err := net.Listen("tcp", ":"+config.TCPServerPort)

if err != nil {

log.Fatalf("Server: Failed to listen on port %s: %v\n", config.TCPServerPort, err)

}

defer listener.Close()

log.Printf("Go TCP Server listening on port %s\n", config.TCPServerPort)

for {

conn, err := listener.Accept()

if err != nil {

log.Printf("Server: Failed to accept connection: %v\n", err)

continue

}

*//Горутин для параллельного выполнения*

go handleConnection(conn)

}

}

client/[main.go](http://main.go)

package main

import (

"src/go\_tcp\_benchmark/pkg/config"

"src/go\_tcp\_benchmark/pkg/fileutils"

"src/go\_tcp\_benchmark/pkg/messaging"

"src/go\_tcp\_benchmark/pkg/metrics"

"src/go\_tcp\_benchmark/pkg/reversal"

"bytes" *// Для bytes.Equal*

"flag" *// Для парсинга аргументов командной строки*

"fmt"

"io"

"log"

"net"

"os"

"time"

)

func main() {

serverIP := flag.String("server\_ip", config.DefaultServerIP, "IP address of the TCP server")

flag.Parse()

log.Printf("Go TCP Client: Using server IP: %s:%s\n", \*serverIP, config.TCPServerPort)

*// 1. Генерация тестового файла*

fileutils.GenerateTestFileIfNotExists(config.TestFileName, config.TotalFileSize)

info, err := os.Stat(config.TestFileName)

if err != nil || info.Size() != config.TotalFileSize {

log.Fatalf("Go TCP Client: Test file '%s' issue. Size: %d (expected %d), Error: %v. Aborting.",

config.TestFileName, info.Size(), config.TotalFileSize, err)

}

*// 2. Инициализация агрегатора метрик*

metricsAggregator := metrics.NewMetricsAggregator("Go", config.TotalFileSize, config.ChunkSize)

*// 3. Установка соединения с сервером*

*// net.Dial подключается к адресу по указанной сети ("tcp").*

*// Возвращает net.Conn и ошибку.*

serverAddr := fmt.Sprintf("%s:%s", \*serverIP, config.TCPServerPort)

conn, err := net.DialTimeout("tcp", serverAddr, 10\*time.Second) *// Таймаут на подключение*

if err != nil {

log.Fatalf("Go TCP Client: Failed to connect to server %s: %v\n", serverAddr, err)

}

defer conn.Close() *// Гарантируем закрытие соединения.*

log.Printf("Go TCP Client: Connected to %s\n", serverAddr)

conn.SetReadDeadline(time.Now().Add(60 \* time.Second)) *// Пример: 60 сек на чтение*

conn.SetWriteDeadline(time.Now().Add(60 \* time.Second)) *// Пример: 60 сек на запись*

*// 4. Инициализация ChunkReader*

chunkReader, err := fileutils.NewChunkReader(config.TestFileName, config.ChunkSize)

if err != nil {

log.Fatalf("Go TCP Client: Failed to create ChunkReader for '%s': %v\n", config.TestFileName, err)

}

defer chunkReader.Close()

*// 5. Запуск таймера и отправка/получение чанков*

metricsAggregator.StartTimer()

totalChunksToSend := chunkReader.TotalChunks()

if chunkReader.FileSize() == 0 && totalChunksToSend == 0 {

log.Println("Go TCP Client: Test file is empty. Nothing to send.")

}

for i := 0; ; i++ {

chunkData, err := chunkReader.ReadNextChunk()

if err != nil {

if err == io.EOF { *// Конец файла*

if len(chunkData) == 0 {

log.Println("Go TCP Client: EOF reached, all data read from file.")

break

}

} else {

log.Printf("Go TCP Client: Error reading chunk %d from file: %v. Aborting.\n", i, err)

metricsAggregator.StopTimer()

metricsAggregator.PrintSummary()

metricsAggregator.SaveToCSV()

os.Exit(1)

}

}

if len(chunkData) == 0 && chunkReader.EOF() {

if i == 0 && chunkReader.FileSize() == 0 {

break

}

if i > 0 && chunkReader.EOF() {

break

}

}

expectedReversedChunk := reversal.GetReversedBytes(chunkData)

rttStartTime := time.Now()

err = messaging.WriteMessage(conn, chunkData)

if err != nil {

log.Printf("Go TCP Client: Error writing chunk %d to server: %v. Aborting.\n", i, err)

metricsAggregator.StopTimer()

metricsAggregator.PrintSummary()

metricsAggregator.SaveToCSV()

os.Exit(1)

}

*// Получаем ответ от сервера*

receivedData, err := messaging.ReadMessage(conn)

if err != nil {

log.Printf("Go TCP Client: Error reading response for chunk %d from server: %v. Aborting.\n", i, err)

if err == io.EOF && i >= int(totalChunksToSend)-1 {

log.Println("Go TCP Client: Server closed connection after all expected chunks, as expected.")

}

metricsAggregator.StopTimer()

metricsAggregator.PrintSummary()

metricsAggregator.SaveToCSV()

os.Exit(1)

}

rtt := time.Since(rttStartTime)

*// Записываем метрики для чанка*

metricsAggregator.RecordChunkRTT(i, rtt)

metricsAggregator.RecordChunkProcessed(len(chunkData)) *// Размер исходного чанка*

*// Верификация*

*// bytes.Equal сравнивает два среза байт.*

verified := bytes.Equal(receivedData, expectedReversedChunk)

metricsAggregator.RecordChunkVerified(verified)

if !verified {

log.Printf("Go TCP Client: ERROR! Chunk %d (original size: %d, received size: %d) verification FAILED.\n",

i, len(chunkData), len(receivedData))

*// Остановка на первой ошибке*

metricsAggregator.StopTimer()

metricsAggregator.PrintSummary()

metricsAggregator.SaveToCSV()

os.Exit(1)

}

log.Printf("Go TCP Client: Chunk %d (size %d) sent, received, and verified OK. RTT: %s\n", i, len(chunkData), rtt)

if chunkReader.EOF() && len(chunkData) > 0 { *// Если это был последний чанк с данными*

log.Printf("Go TCP Client: Processed final chunk %d from file.\n", i)

break *// Выходим из цикла после обработки последнего чанка*

}

}

*// 6. Остановка таймера и вывод результатов*

metricsAggregator.StopTimer()

metricsAggregator.PrintSummary()

err = metricsAggregator.SaveToCSV()

if err != nil {

log.Fatalf("Go TCP Client: Failed to save metrics to CSV: %v\n", err)

}

log.Println("Go TCP Client finished successfully.")

}