# Теоретический отчет: Архитектура сетевого взаимодействия и глубинные механизмы TCP в среде.NET

## Введение

Пятый день третьей недели обучения знаменует собой критический переход в подготовке разработчика C#: отказ от детерминированной, синхронной логики, свойственной локальным вычислениям, в пользу недетерминированной, асинхронной природы распределенных сетевых систем.1 В отличие от работы с файловой системой или памятью, где операции часто атомарны и предсказуемы, сетевое взаимодействие через протокол TCP (Transmission Control Protocol) требует фундаментального сдвига в ментальной модели программиста.

Цель данного документа — предоставить исчерпывающую теоретическую базу, необходимую для выполнения лабораторных работ по созданию TCP-клиентов и серверов. Мы демистифицируем абстракцию «потока», разберем физику движения байтов от сетевой карты до буфера приложения, проанализируем алгоритмы управления трафиком, встроенные в ядро ОС, и детально изучим паттерны работы с классом NetworkStream в экосистеме.NET. Особое внимание будет уделено тем аспектам, которые обычно скрыты от глаз разработчика, но незнание которых приводит к классическим ошибкам: потере данных, «зависанию» соединений и некорректной обработке фрагментированных сообщений.

## Глава 1. Физиология транспортного уровня: Поток против Пакета

### 1.1. Иллюзия пакетной передачи

Распространенным заблуждением среди разработчиков, переходящих от UDP или HTTP-библиотек высокого уровня к «сырым» сокетам, является представление о TCP как о протоколе обмена сообщениями. Это представление ошибочно. TCP — это протокол, ориентированный на **поток** (stream-oriented).2

В контексте операционной системы и сетевого стека это означает следующее: для приложения TCP-соединение выглядит как труба, в которую с одной стороны байты «заливаются» (Write), а с другой — «вытекают» (Read). В этой трубе не существует понятия «границ сообщения». Если приложение-отправитель выполняет две операции записи:

1. Send("Hello")
2. Send("World")

Приложение-получатель может прочитать эти данные в совершенно произвольных комбинациях, зависящих от текущего состояния сети, загрузки процессора и настроек буферов.3 Возможные варианты чтения (Receive):

* Один блок: "HelloWorld" (склейка пакетов).
* Два блока: "Hello", "World" (идеальный, но негарантированный сценарий).
* Фрагментированные блоки: "He", "lloWor", "ld".

### 1.2. Механизм виртуального канала

TCP создает абстракцию надежного канала поверх ненадежного протокола IP. Протокол IP оперирует дискретными пакетами и не гарантирует ни доставки, ни порядка, ни отсутствия дубликатов. TCP берет на себя ответственность за упорядочивание и гарантированную доставку, используя механизм **Sequence Numbers** (SEQ) и **Acknowledgment Numbers** (ACK).

Каждый байт в потоке имеет свой порядковый номер. Когда данные отправляются, они помещаются в сегмент TCP. Если сегмент теряется в сети, стек TCP на стороне отправителя, не получив подтверждения (ACK) в течение определенного тайм-аута (RTO — Retransmission Timeout), выполняет повторную отправку. Этот процесс происходит полностью внутри ядра ОС (Kernel Space) и абсолютно прозрачен для приложения.2 Приложение просто замечает, что операция ReadAsync завершается с задержкой, ожидая, пока «дырка» в потоке заполнится переданными заново данными.

### 1.3. Максимальный размер сегмента (MSS) и MTU

Понимание того, почему поток дробится на части, требует анализа физических ограничений среды передачи данных. Ключевым параметром здесь является **MTU (Maximum Transmission Unit)** — максимальный размер пакета, который может быть передан через канальный уровень (Layer 2) без фрагментации. Для стандарта Ethernet v2 MTU составляет 1500 байт.6

Из этих 1500 байт часть уходит на служебные заголовки:

* Заголовок IP (IPv4): обычно 20 байт.
* Заголовок TCP: обычно 20 байт.

Оставшееся место, доступное для полезной нагрузки (данных приложения), называется **MSS (Maximum Segment Size)**.

$$MSS = MTU - (IP\_{Header} + TCP\_{Header})$$

$$MSS = 1500 - (20 + 20) = 1460 \text{ байт}$$

Это означает, что даже если вы попытаетесь отправить буфер размером 1 МБ одним вызовом Stream.WriteAsync, сетевой стек разобьет его на сотни сегментов по 1460 байт (или меньше). Это и есть процесс **сегментации**.6

## Глава 2. Фрагментация, Сегментация и Склейка (Coalescing)

Для построения надежных сетевых приложений критически важно различать процессы, происходящие на разных уровнях модели OSI, так как они имеют разные последствия для производительности и надежности.

### 2.1. TCP Сегментация (Layer 4) против IP Фрагментации (Layer 3)

В документации и технической литературе часто смешиваются термины «фрагментация» и «сегментация», однако между ними лежит пропасть в механике работы.8

| **Характеристика** | **TCP Сегментация (Layer 4)** | **IP Фрагментация (Layer 3)** |
| --- | --- | --- |
| **Уровень** | Транспортный (Transport) | Сетевой (Network) |
| **Инициатор** | Стек TCP на хосте отправителя | Маршрутизатор (Router) в пути или хост |
| **Механизм** | Данные делятся на куски $\le$ MSS. Каждому куску добавляется заголовок TCP. | Пакет делится на части, чтобы пройти через канал с меньшим MTU. Заголовок TCP остается только в первой части. |
| **Потеря пакета** | При потере одного сегмента TCP перезапрашивает только его (Selective ACK). | При потере **одного** фрагмента теряется **весь** исходный пакет, так как его невозможно собрать заново. |
| **Влияние на CPU** | Нормальная работа протокола. | Высокая нагрузка на маршрутизатор (деление) и получателя (сборка). |
| **Статус** | Желательно (штатный режим). | Нежелательно (признак плохой конфигурации сети). |

**Глубинный инсайт:** IP-фрагментация считается вредной практикой в современном интернете. Если маршрутизатор на пути следования пакета имеет MTU меньше, чем у отправителя (например, из-за инкапсуляции в VPN-туннель), и на пакете установлен флаг DF (Don't Fragment), пакет будет отброшен, а отправителю вернется ICMP-сообщение «Fragmentation Needed». Это механизм **Path MTU Discovery (PMTUD)**. Приложения должны быть готовы к тому, что размер MSS может динамически меняться в процессе жизни соединения.10

### 2.2. Склейка пакетов (Receive Side Coalescing)

Если сегментация объясняет, почему данные приходят частями, то склейка (Coalescing) объясняет обратный эффект — почему множество мелких отправок может прийти одним большим куском.

Современные сетевые карты (NIC) и драйверы ОС реализуют технологии **LRO (Large Receive Offload)** и **RSC (Receive Side Coalescing)**. Идея заключается в снижении нагрузки на центральный процессор. Обработка каждого входящего пакета вызывает прерывание (interrupt) процессора. Если на гигабитном канале приходят тысячи мелких пакетов, процессор захлебнется в прерываниях.

RSC позволяет сетевой карте объединять несколько входящих сегментов TCP (принадлежащих одному потоку) в один крупный блок памяти до того, как передать его операционной системе.12

Таким образом, даже если сервер отправил 5 пакетов по 100 байт, ваше приложение.NET в методе ReadAsync может получить один массив размером 500 байт. Это аппаратное подтверждение того факта, что границы пакетов не сохраняются.

## Глава 3. Динамика буферизации и Алгоритмы управления трафиком

Поведение NetworkStream в C# напрямую зависит от сложных алгоритмов, управляющих буферами сокетов в ядре операционной системы. Понимание этих алгоритмов необходимо для диагностики проблем с задержками (latency).

### 3.1. Алгоритм Нагла (Nagle's Algorithm)

В ранние годы развития сетей (1980-е) инженеры столкнулись с проблемой «крошечных пакетов» (tinygrams). Программы типа Telnet отправляли каждый нажатый символ отдельным пакетом.

* Полезная нагрузка: 1 байт.
* Заголовки (IP + TCP): 40 байт.
* Эффективность: ~2.4%.  
  Это приводило к перегрузке сетей. Джон Нагл предложил решение, зафиксированное в RFC 896.14

Суть алгоритма:

Если приложение пытается отправить данные, размер которых меньше MSS, TCP-стек должен задержать отправку и буферизировать эти данные до тех пор, пока не будет выполнено одно из условий:

1. Накопилось достаточно данных для полного сегмента (равного MSS).
2. Пришло подтверждение (ACK) от получателя на все ранее отправленные данные.

Влияние на разработчика:

В приложении C# последовательные вызовы stream.WriteAsync с небольшими порциями данных не приведут к мгновенной отправке пакетов в сеть. Стек будет «склеивать» их в исходящем буфере. Для передачи файлов это повышает пропускную способность. Однако для real-time приложений (игры, биржевые терминалы) это создает неприемлемую задержку.16

### 3.2. Проблема задержанного подтверждения (Delayed ACK)

Ситуация усугубляется взаимодействием с алгоритмом на принимающей стороне — **Delayed ACK**. Чтобы не забивать сеть пустыми пакетами подтверждений, получатель задерживает отправку ACK (обычно на 200 мс), ожидая, что приложение-получатель отправит ответные данные, к которым можно будет «прицепить» (piggyback) этот ACK.17

**Deadly Embrace (Смертельное объятие):**

1. **Отправитель (Nagle):** «Я отправил кусочек данных. У меня есть еще кусочек, но он маленький. Я жду ACK на первый кусочек, чтобы отправить второй».
2. **Получатель (Delayed ACK):** «Я получил кусочек. Я не буду слать ACK сразу, подожду 200 мс, вдруг приложение что-то ответит».
3. **Итог:** Система замирает ровно на 200 мс. Это классическая проблема производительности сетевых приложений.17

### 3.3. Решение: TCP\_NODELAY

Для устранения этой задержки в классе TcpClient (и в сокетах вообще) существует опция NoDelay.

C#

TcpClient client = new TcpClient();  
client.NoDelay = true; // Отключает алгоритм Нагла

Установка NoDelay = true предписывает стеку отправлять данные немедленно, даже если сегмент не заполнен целиком. Это критически важно для задачи 5-го дня (создание Telnet-клиента), где важна отзывчивость интерфейса.19

## Глава 4. Взаимодействие Kernel Space и User Space

Когда вы вызываете методы Read или Write в.NET, вы фактически инициируете сложный процесс перемещения данных между областями памяти с разным уровнем привилегий.

### 4.1. Буферы сокетов (Socket Buffers)

Для каждого активного сокета ядро ОС выделяет два кольцевых буфера:

1. **Receive Buffer (Rx):** Сюда попадают данные, пришедшие из сетевой карты.
2. **Send Buffer (Tx):** Сюда копируются данные из приложения перед отправкой.

Механика WriteAsync:

Когда вы пишете await stream.WriteAsync(buffer), задача завершается (Task completes) не тогда, когда данные дошли до получателя, и даже не тогда, когда они ушли в сеть. Она завершается, когда данные были успешно скопированы из вашего массива byte в Send Buffer ядра.5 Если Send Buffer переполнен (сеть медленная, а мы пишем быстро), WriteAsync будет ждать освобождения места.

Механика ReadAsync:

Операция чтения — это запрос на копирование данных из Receive Buffer ядра в буфер приложения.

* Если Receive Buffer пуст: ReadAsync «засыпает» (возвращает незавершенный Task), регистрируя ожидание в системе (через I/O Completion Ports в Windows или epoll в Linux).
* Как только пакет приходит в Receive Buffer, ядро будит поток из ThreadPool, который копирует данные и завершает Task.22

### 4.2. Скользящее окно (Sliding Window)

Размер Receive Buffer напрямую влияет на скорость передачи данных. TCP использует механизм Sliding Window для управления потоком (Flow Control). Получатель сообщает отправителю: «Мой буфер имеет X свободных байт». Это значение называется Receive Window (RWND).

Если приложение на стороне получателя читает данные медленнее, чем они приходят, Receive Buffer заполняется. RWND уменьшается до нуля (Zero Window). В этот момент отправитель обязан прекратить передачу. Это защищает получателя от переполнения памяти, но приводит к остановке передачи данных.24

## Глава 5. Работа с NetworkStream в.NET

Класс System.Net.Sockets.NetworkStream является мостом между высокоуровневым кодом C# и низкоуровневыми сокетами. Правильное использование его методов — залог стабильности приложения.

### 5.1. Поведение ReadAsync и частичное чтение

Самая частая ошибка новичков — ожидание, что ReadAsync заполнит весь переданный ему буфер.

C#

byte buffer = new byte;  
int bytesRead = await stream.ReadAsync(buffer, 0, buffer.Length);

Правило: ReadAsync возвращает управление, как только хотя бы один байт доступен для чтения (или при разрыве соединения).

Если вы запросили 1024 байта, а в буфер ядра пришло всего 10 байт (фрагмент пакета или просто маленькое сообщение), ReadAsync запишет 10 байт, вернет значение 10 и завершится. Он не будет ждать остальных 1014 байт.26

Это поведение диктует необходимость использования циклов для чтения сообщений определенной длины (см. Глава 6).

### 5.2. Интерпретация возвращаемого значения: 0 vs Exception

Возвращаемое значение метода ReadAsync (int) является основным индикатором состояния соединения.

1. **bytesRead > 0**: Нормальная работа. Получено указанное количество байт.
2. **bytesRead == 0**: **Graceful Closure (FIN)**. Удаленная сторона корректно закрыла соединение (отправила пакет FIN). Данных больше не будет никогда. Приложение должно выйти из цикла чтения и закрыть сокет.26
3. **IOException (SocketException)**: **Hard Closure (RST)**. Произошла ошибка сети, разрыв кабеля, краш процесса на той стороне или сброс соединения брандмауэром. Соединение разорвано аварийно.30

**Важно:** Нельзя игнорировать случай 0. Бесконечный цикл, который не проверяет 0, при закрытии соединения уйдет в «spin loop» (пожирание 100% CPU), так как ReadAsync будет возвращать 0 мгновенно и бесконечно.29

### 5.3. Тайм-ауты в асинхронном коде

Свойство NetworkStream.ReadTimeout работает только для синхронных методов Read. Для ReadAsync оно игнорируется.

Чтобы реализовать тайм-аут в асинхронном режиме, необходимо использовать CancellationTokenSource с задержкой:

C#

using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(5));  
try {  
 int n = await stream.ReadAsync(buffer, cts.Token);  
}  
catch (OperationCanceledException) {  
 // Обработка тайм-аута  
}

Это стандартный паттерн TAP (Task-based Asynchronous Pattern).31

## Глава 6. Парсинг потока и Фрейминг (Message Framing)

Поскольку TCP не сохраняет границы сообщений, прикладной уровень обязан восстанавливать их самостоятельно. Этот процесс называется **фреймингом** (Framing). Существует два основных подхода.

### 6.1. Подход 1: Разделители (Delimiter-Based)

Используется в текстовых протоколах (HTTP, SMTP, Telnet). Конец сообщения обозначается специальной последовательностью символов (например, \r\n).

**Проблемы реализации:**

* **Коллизии:** Символ-разделитель не должен встречаться в полезной нагрузке.
* **Производительность:** Требуется сканировать каждый полученный байт.
* **Telnet специфика:** В задании с Telnet-клиентом вы столкнетесь с тем, что сервер может присылать не только текст, но и команды согласования опций (Negotiation), начинающиеся с байта 0xFF (IAC - Interpret As Command). Если их не отфильтровать, текст будет содержать «мусор».32

**Рекомендация для задания Б (Telnet):** Читать данные, накапливать их в буфере и искать \r\n или просто выводить поток в консоль по мере поступления, если задача не требует строгой обработки команд.

### 6.2. Подход 2: Префикс длины (Length-Prefix)

Стандарт для бинарных протоколов. Каждое сообщение предваряется заголовком фиксированной длины (обычно 4 байта int32), содержащим размер тела сообщения.34

Схема пакета: [Length (4 bytes)][Payload (N bytes)]

Алгоритм чтения (Паттерн Read Loop):

Для корректного чтения такого сообщения необходимо два цикла.

1. **Цикл чтения заголовка:** Читать, пока не наберется ровно 4 байта.
2. **Парсинг длины:** Преобразовать 4 байта в int. **Внимание:** Учитывать порядок байт (Endianness). Сетевой стандарт — Big-Endian, архитектура x86/x64 (C#) — Little-Endian. Необходимо использовать IPAddress.NetworkToHostOrder.34
3. **Цикл чтения тела:** Читать, пока не наберется ровно N байт.

Реализация «Smart Downloader» (Задание А):

В задании требуется скачивание, где протокол (HTTP) уже определен. HTTP использует гибридный подход: заголовки отделяются двойным переносом строки (\r\n\r\n), а тело может определяться заголовком Content-Length (Length-Prefix) или передаваться чанками (Chunked encoding). Использование HttpClient скрывает эту сложность, но если задача подразумевает работу с сокетами, парсинг придется писать вручную.

### 6.3. Паттерн "Read Exactly"

В.NET 7 появился метод Stream.ReadExactlyAsync, который реализует цикл чтения внутри себя.37 Однако для понимания (и для старых версий.NET) важно уметь писать его вручную:

C#

// Концептуальная реализация чтения строгого количества байт  
async Task ReadBytesStrictlyAsync(NetworkStream stream, byte buffer, int count)  
{  
 int totalRead = 0;  
 while (totalRead < count)  
 {  
 int read = await stream.ReadAsync(buffer, totalRead, count - totalRead);  
 if (read == 0) throw new EndOfStreamException("Соединение закрыто до завершения чтения");  
 totalRead += read;  
 }  
}

## Глава 7. Продвинутые аспекты жизненного цикла соединения

### 7.1. Graceful Shutdown (Полузакрытые соединения)

В отличие от Close(), который резко обрывает связь, метод Socket.Shutdown(SocketShutdown.Send) позволяет реализовать корректное завершение сеанса.39

1. Клиент отправляет все данные.
2. Клиент вызывает Shutdown(Send). Это отправляет пакет FIN серверу.
3. Клиент продолжает читать (ReadAsync), ожидая финального ответа сервера.
4. Сервер получает FIN (Read возвращает 0), досылает остатки данных и закрывает сокет.
5. Клиент получает 0 в ReadAsync и закрывает сокет.

### 7.2. Проблема Half-Open соединений

Если кабель выдернули или пропало питание, пакет FIN/RST не может быть отправлен. Сервер может годами держать такой сокет открытым («висячее соединение»), расходуя ресурсы.

* **Решение:** TCP Keep-Alive (на уровне сокета) или Application-Level Heartbeat (периодические пустые сообщения «пинг» внутри протокола приложения). При отсутствии «понга» в течение N секунд сокет принудительно закрывается.41

### 7.3. Управление памятью

При высокой нагрузке постоянное создание буферов new byte для каждого чтения создает огромную нагрузку на сборщик мусора (GC).

* **ArrayPool:** Используйте ArrayPool<byte>.Shared.Rent() для получения буферов и Return() для возврата.
* **System.IO.Pipelines:** Современный стандарт (используется в Kestrel/ASP.NET Core), позволяющий работать с потоковыми данными без лишних копирований и аллокаций.42

## Заключение

Переход к сетевому программированию требует от разработчика понимания того, что сеть — это враждебная среда. Данные фрагментируются, склеиваются, задерживаются и теряются. Абстракция NetworkStream пытается скрыть сложность, но не отменяет физику процесса.

Для успешного выполнения заданий 5-го дня запомните ключевые тезисы:

1. **Поток бесконечен и бесструктурен.** Всегда используйте фрейминг (границы сообщений).
2. **Чтение не гарантировано.** Всегда используйте циклы или ReadExactly, чтобы собрать полное сообщение.
3. **Задержки неизбежны, но управляемы.** Используйте NoDelay для интерактивности и буферизацию для пропускной способности.
4. **Закрытие — это процесс.** Обрабатывайте возвращаемое значение 0 и исключения раздельно.

Эти знания формируют фундамент для построения высоконагруженных распределенных систем, устойчивых к реалиям физических сетей.

#### Источники

1. План обучения C# третьей недели
2. Services and Segment structure in TCP - GeeksforGeeks, дата последнего обращения: декабря 3, 2025, <https://www.geeksforgeeks.org/computer-networks/services-and-segment-structure-in-tcp/>
3. Working with TCP Streams in Wireshark Dissectors - Jonas Lieb, дата последнего обращения: декабря 3, 2025, <https://www.jonaslieb.de/blog/wireshark-tcp/>
4. C# TCP client sending but not receiving messages - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/27850537/c-sharp-tcp-client-sending-but-not-receiving-messages>
5. Deep Dive into TCP Sockets: How Data Travels Under the Hood | by Harshith Gowda, дата последнего обращения: декабря 3, 2025, <https://medium.com/@harshithgowdakt/deep-dive-into-tcp-sockets-how-data-travels-under-the-hood-7c16f6b2bf95>
6. Difference between MTU and MSS - PyNet Labs, дата последнего обращения: декабря 3, 2025, <https://www.pynetlabs.com/difference-between-mtu-and-mss/>
7. What is MSS (maximum segment size)? - Cloudflare, дата последнего обращения: декабря 3, 2025, <https://www.cloudflare.com/learning/network-layer/what-is-mss/>
8. дата последнего обращения: декабря 3, 2025, <https://medium.com/@gulernilay088/fragmentation-vs-segmentation-ba3f4229ed4f#:~:text=In%20summary%2C%20fragmentation%20occurs%20at,efficient%20transmission%20across%20the%20network.>
9. Fragmentation vs Segmentation - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@gulernilay088/fragmentation-vs-segmentation-ba3f4229ed4f>
10. What Is MTU & MSS | Fragmentation Explained - Imperva, дата последнего обращения: декабря 3, 2025, <https://www.imperva.com/learn/application-security/what-is-mtu-mss/>
11. Understanding MTU, MSS, Frame / Packets / TCP and UDP - From a beginner perspective (confused) - Cisco Community, дата последнего обращения: декабря 3, 2025, <https://community.cisco.com/t5/switching/understanding-mtu-mss-frame-packets-tcp-and-udp-from-a-beginner/td-p/2687894>
12. Receive Segment Coalescing (RSC) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh997024(v=ws.11)>
13. Receive Side Coalescing for Accelerating TCP/IP Processing - AMiner, дата последнего обращения: декабря 3, 2025, <https://static.aminer.org/pdf/PDF/000/251/175/receive_side_coalescing_for_accelerating_tcp_ip_processing.pdf>
14. It's always TCP\_NODELAY. Every damn time. - Marc's Blog - brooker.co.za, дата последнего обращения: декабря 3, 2025, <https://brooker.co.za/blog/2024/05/09/nagle.html>
15. Nagle's algorithm - Wikipedia, дата последнего обращения: декабря 3, 2025, <https://en.wikipedia.org/wiki/Nagle%27s_algorithm>
16. Tiny Packets, Big Delays: How Nagle's Algorithm Works | by Aryan Walia | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@ariyanwaliya/tiny-packets-big-delays-how-nagles-algorithm-works-ef83203c4089>
17. TCP\_NODELAY & Nagle's Algorithm | ExtraHop, дата последнего обращения: декабря 3, 2025, <https://www.extrahop.com/blog/tcp-nodelay-nagle-quickack-best-practices>
18. TCP delayed acks vs Nagle's algorithm - Software Engineering Stack Exchange, дата последнего обращения: декабря 3, 2025, <https://softwareengineering.stackexchange.com/questions/408318/tcp-delayed-acks-vs-nagles-algorithm>
19. StreamSocketControl.NoDelay Property (Windows.Networking.Sockets) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/uwp/api/windows.networking.sockets.streamsocketcontrol.nodelay?view=winrt-26100>
20. c# - When to disable Nagle's algorithm for TcpClient? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/78321459/when-to-disable-nagles-algorithm-for-tcpclient>
21. TCP Sockets send buffer size efficiency - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/28785437/tcp-sockets-send-buffer-size-efficiency>
22. Direct buffers and zero copy - Google Groups, дата последнего обращения: декабря 3, 2025, <https://groups.google.com/g/mechanical-sympathy/c/A3WvNcpFjF0>
23. User space buffer and Kernel buffer - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/33629903/user-space-buffer-and-kernel-buffer>
24. TCP flow control and the sliding window - IBM, дата последнего обращения: декабря 3, 2025, <https://www.ibm.com/docs/en/storage-protect/8.1.25?topic=tuning-tcp-flow-control>
25. TCP Series — TCP Receive Buffer and Receive Window | by Brunda K - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/brundas-tech-notes/tcp-series-tcp-receive-buffer-and-receive-window-f7f3547be251>
26. NetworkStream.ReadAsync Method (System.Net.Sockets) | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets.networkstream.readasync?view=net-10.0>
27. Partial and zero-byte reads in DeflateStream, GZipStream, and CryptoStream - .NET, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/core/compatibility/core-libraries/6.0/partial-byte-reads-in-streams>
28. Reading Zero Bytes from Empty System.Net.Sockets.NetworkStream Halts Execution, дата последнего обращения: декабря 3, 2025, <https://developercommunity.visualstudio.com/content/problem/711370/reading-zero-bytes-from-empty-systemnetsocketsnetw.html>
29. Tcpclient stream.ReadAsync does not return 0 when socket/stream closes? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/33662928/tcpclient-stream-readasync-does-not-return-0-when-socket-stream-closes>
30. NetworkStream exceptions and return codes - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/28484946/networkstream-exceptions-and-return-codes>
31. Asynchronous Network Operations are Pegging CPU with 0-length reads : r/csharp - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/csharp/comments/70v1gz/asynchronous_network_operations_are_pegging_cpu/>
32. nonsense char on telnet client - General Guidance - Arduino Forum, дата последнего обращения: декабря 3, 2025, <https://forum.arduino.cc/t/nonsense-char-on-telnet-client/474400>
33. RFC 854 - Telnet Protocol Specification - IETF Datatracker, дата последнего обращения: декабря 3, 2025, <https://datatracker.ietf.org/doc/html/rfc854>
34. Winsock Programmer's FAQ: How to Use TCP Effectively - Tangentsoft, дата последнего обращения: декабря 3, 2025, <https://tangentsoft.com/wskfaq/articles/effective-tcp.html>
35. Length-prefix Framing experimental — CAF 0.19.5 documentation - Read the Docs, дата последнего обращения: декабря 3, 2025, <https://actor-framework.readthedocs.io/en/0.19.5/net/LengthPrefixFraming.html>
36. How do I prefix the length of message in TCP/IP - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/1339270/how-do-i-prefix-the-length-of-message-in-tcp-ip>
37. Stream.ReadAtLeastAsync Method (System.IO) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.stream.readatleastasync?view=net-10.0>
38. Stream.ReadExactlyAsync Method (System.IO) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.stream.readexactlyasync?view=net-10.0>
39. close vs shutdown socket? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/4160347/close-vs-shutdown-socket>
40. Socket.Shutdown(SocketShutdown) Method (System.Net.Sockets) | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.shutdown?view=net-10.0>
41. Dealing with half-opened TCP connections · Issue #622 - GitHub, дата последнего обращения: декабря 3, 2025, <https://github.com/vibe-d/vibe.d/issues/622>
42. System.IO.Pipelines - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/io/pipelines>