



Расскажи подробно и приведи примеры для каждого.

Также расскажи для какого взаимодействия предназначены. Между какими объектами ?
C# Инструменты для взаимодействия

- IAsyncEnumerable
- System.Threading.Channels
- System.IO.Pipeline
- Pipeline (наверное это namedPipeline, но могу ошибаться)
- TcpIP
- gRPC
- Rest
- SignalR

Если что-то забыл дополнни

Вот развернутое объяснение с примерами и назначением каждого из указанных инструментов C# для взаимодействия между объектами, процессами или узлами:

IAsyncEnumerable

- Представляет асинхронный поток элементов, который можно перебирать с помощью `await foreach`.
- Предназначен для асинхронной последовательной работы с коллекциями, где получение элементов может занимать время (например, постраничное получение из API, чтение из файла/потока).
- Взаимодействие — между вызователем и асинхронным источником данных в одном процессе или при асинхронном переборе.
- Пример:

```
public async IAsyncEnumerable<int> GetNumbersAsync() {  
    for (int i = 0; i < 5; i++) {  
        await Task.Delay(100);  
        yield return i;  
    }  
}  
// Потребление:  
await foreach(var number in GetNumbersAsync()) {
```

```
        Console.WriteLine(number);
    }
```

- Используется при асинхронной итерации с отложенной загрузкой данных или с данными из потоков/сервисов.[\[1\]](#) [\[2\]](#) [\[3\]](#)

System.Threading.Channels

- Низкоуровневый асинхронный канал для обмена данными между потоками или компонентами как буферизированный поток данных.
- Подходит для сценариев продьюсер-консьюмер, где производитель пишет данные, а потребитель читает их независимо и асинхронно.
- Взаимодействие — между компонентами/потоками внутри одного процесса или межпроцессного взаимодействия при совместном доступе.
- Пример:

```
var channel = Channel.CreateUnbounded<string>();
var writer = channel.Writer;
var reader = channel.Reader;

_ = Task.Run(async () => {
    await writer.WriteAsync("Hello");
    writer.Complete();
});

await foreach(var item in reader.ReadAllAsync()) {
    Console.WriteLine(item);
}
```

- Отлично для асинхронной передачи данных и конкурентного взаимодействия.[\[4\]](#)

System.IO.Pipelines

- Высокопроизводительный поток для работы с последовательным вводом-выводом, в основном для сетевых приложений и потоков.
- Предназначен для эффективной обработки ввода-вывода, например, при работе с сетевыми сокетами.
- Взаимодействие — между приложением и сетевым потоком или между потоками данных.
- Пример использования с сетевым потоком:

```
var socket = new Socket(...);
await socket.ConnectAsync(...);
var stream = new NetworkStream(socket);
var reader = PipeReader.Create(stream);
var writer = PipeWriter.Create(stream);
// Чтение и запись через reader и writer...
```

- Часто используется в высокопроизводительных сетевых сервисах.^[5]

Named Pipes (возможно, под "Pipeline" понимался NamedPipe)

- Механизм межпроцессного взаимодействия (IPC) на одном или разных компьютерах с помощью именованных каналов.
- Предназначен для обмена байтовыми потоками между процессами.
- Взаимодействие между процессами на одном или нескольких узлах.
- Пример сервера и клиента именованного канала:

```
// Сервер
using var server = new NamedPipeServerStream("my_pipe");
server.WaitForConnection();
// Читаем и пишем в сервер

// Клиент
using var client = new NamedPipeClientStream(".", "my_pipe", PipeDirection.InOut);
client.Connect();
// Читаем и пишем в клиент
```

- Используется для IPC с потоковой передачей данных.^[6]

TCP/IP (через System.Net.Sockets)

- Протокол передачи данных по сети для базовых сокетных соединений.
- Применяется для сетевого взаимодействия между двумя компьютерами или приложениями через TCP.
- Взаимодействие — между клиентом и сервером по сети.
- Пример сервера TCP:

```
var listener = new TcpListener(IPAddress.Loopback, 8080);
listener.Start();
var client = await listener.AcceptTcpClientAsync();
using var stream = client.GetStream();
// Читаем и пишем в stream
```

- Применяется там, где нужны низкоуровневые сетевые коммуникации.^[7]

gRPC

- Фреймворк для высокопроизводительного взаимодействия между микросервисами, использующий HTTP/2 и Protocol Buffers.
- Предназначен для вызова удаленных процедур с поддержкой потоков.
- Взаимодействие — клиент-сервер (часто микросервис-микросервис) с контрактами API.
- Пример (сокращенно):

```
service Greeter {  
    rpc SayHello (HelloRequest) returns (HelloReply);  
}
```

Сгенерированный C# сервис и клиент используют асинхронные методы для вызова.

- Используется для строгого, высокопроизводительного API взаимодействия.^[8]

REST

- Архитектурный стиль взаимодействия по HTTP для доступа к ресурсам сервера через стандартные методы (GET, POST, PUT, DELETE).
- Предназначен для клиент-серверного взаимодействия по HTTP.
- Взаимодействие — между клиентом (например, браузер или API клиент) и REST сервером.
- Пример на ASP.NET Core:

```
[ApiController]  
[Route("api/[controller]")]  
public class ProductsController : ControllerBase {  
    [HttpGet]  
    public IEnumerable<Product> GetProducts() { ... }  
}
```

- Широко распространен для веб-сервисов и API.

SignalR

- Библиотека для реализации двунаправленной связи в реальном времени между клиентом и сервером (WebSocket/Long Polling).
- Предназначена для обмена сообщениями и обновления UI в реальном времени.
- Взаимодействие — между браузером или клиентом и сервером.
- Пример на сервере:

```
public class ChatHub : Hub {  
    public async Task SendMessage(string user, string message) {  
        await Clients.All.SendAsync("ReceiveMessage", user, message);  
    }  
}
```

- Используется для приложений с живой обратной связью — чаты, уведомления и др.

Итог по взаимодействию

Инструмент	Назначение взаимодействия	Межъектами/процессами	Примерное применение
IAsyncEnumerable	Асинхронная итерация коллекций	Вызов и источник данных в одном процессе	Асинхронные потоки данных, API, файлы
System.Threading.Channels	Асинхронный продьюсер-консьюмер с буфером	Потоки/компоненты внутри приложения	Очереди сообщений, параллелизм
System.IO.Pipelines	Высокопроизводительный поток ввода-вывода	Приложение и сетевой/файловый поток	Сетевой ввод/вывод, обработка протоколов
Named Pipes	Межпроцессное взаимодействие через каналы	Отдельные процессы на одном или разных хостах	IPC, каналы обмена байтами
TCP/IP	Низкоуровневое сетевое взаимодействие	Клиент и сервер по сети	Сетевые приложения, сервисы
gRPC	Высокопроизводительный вызов удалённых функций	Микросервисы, клиент-сервер	Микросервисы, API с контрактами
REST	HTTP API для доступа к ресурсам	Клиент и сервер через HTTP	Веб-сервисы, публичные API
SignalR	Реальное время, двунаправленные коммуникации	Веб-клиент и сервер	Чат, обновление UI, уведомления в реальном времени

Если нужны примеры по каждому из пунктов или дополнительное уточнение, можно подробнее углубиться.

**

Я слышал, что очень уместно использовать System.IO.Pipeline вместе с TCP/IP протоколом.

Приведи пример совместного использования System.IO.Pipeline вместе с TCP/IP
В чем здесь смысл ?

Использование System.IO.Pipelines вместе с TCP/IP помогает эффективно и высокопроизводительно работать с потоками данных, передаваемыми по TCP-соединению.

Смысл использования System.IO.Pipelines с TCP/IP

- TCP-сокет работает с потоками байтов. System.IO.Pipelines предоставляет удобный и оптимизированный API для асинхронного чтения и записи этих байтов.
- Pipelines позволяют обрабатывать буфер данных по частям, не создавая лишних копий, что повышает производительность.
- Это особенно важно для сетевых приложений с высокой нагрузкой, где важно быстро и эффективно считывать и отправлять данные.

Пример совместного использования System.IO.Pipelines с TCP/IP (C#):

```
using System;
using System.Buffers;
using System.IO.Pipelines;
using System.Net;
using System.Net.Sockets;
using System.Threading.Tasks;

class TcpPipelineExample
{
    private readonly Socket socket;
    private readonly NetworkStream stream;
    private readonly PipeReader reader;
    private readonly PipeWriter writer;

    public TcpPipelineExample(string host, int port)
    {
        var ipHostInfo = Dns.GetHostEntry(host);
        var ipAddress = ipHostInfo.AddressList[^2_0];
        var ipEndPoint = new IPEndPoint(ipAddress, port);

        socket = new Socket(ipEndPoint.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
        socket.Connect(ipEndPoint);
        stream = new NetworkStream(socket);
        reader = PipeReader.Create(stream);
        writer = PipeWriter.Create(stream);
    }

    public async Task SendAsync(byte[] data)
    {
        await writer.WriteAsync(data);
        await writer.FlushAsync();
    }

    public async Task ReceiveAsync()
    {
        while (true)
        {
            ReadResult result = await reader.ReadAsync();
            ReadOnlySequence<byte> buffer = result.Buffer;

            // Обработка прочитанных данных
            foreach (var segment in buffer)
```

```

    {
        Console.WriteLine($"Received {segment.Length} bytes");
        // Можно декодировать данные здесь
    }

    reader.AdvanceTo(buffer.End);

    if (result.IsCompleted)
        break;
}
}

public void Close()
{
    socket.Close();
}
}

```

Объяснение:

- TcpSocket создаёт TCP-соединение.
- NetworkStream обворачивает это соединение в поток.
- PipeReader и PipeWriter работают с этим потоком и обеспечивают эффективное асинхронное чтение/запись.
- При чтении данныечитываются из PipeReader.Buffer в виде ReadOnlySequence<byte> и обрабатываются по частям без лишних копий.

Такой подход снижает накладные расходы на буферизацию и копирование данных, что принципиально важно при сетевых приложениях с высокими требованиями к производительности и частоте передачи данных.

Это пример типичного использования System.IO.Pipelines вместе с TCP/IP для высокопроизводительного обмена байтами по сети. [\[11\]](#)

**

Для межпроцессного взаимодействия какой протокол самый эффективный ?

- по скорости
- по использованию ресурсов
- по эффективности
- внутри Приложения
- между Приложениями на одном Хосте
- между Приложениями на разных Хостах.

Для разных сценариев межпроцессного взаимодействия на C# существуют разные протоколы и механизмы, которые отличаются скоростью, использованием ресурсов и общей эффективностью в зависимости от контекста: внутри приложения, между приложениями на одном хосте или на разных хостах.

1. Внутри одного приложения (межпоточное взаимодействие)

- **Самые эффективные:**
 - **System.Threading.Channels** — асинхронные каналы с буферизацией, очень легковесны, минимальная блокировка, высокая скорость и низкое потребление ресурсов.
 - **ConcurrentQueue, BlockingCollection** — стандартные коллекции для потокобезопасного обмена данными.
- По скорости и ресурсам — использование асинхронных каналов `System.Threading.Channels` почти не имеет накладных расходов и лучше традиционных блокировок.

2. Между приложениями на одном хосте

- **Самый быстрый и эффективный протокол:**
 - **Shared Memory (совместно используемая память)** — максимально быстрая, так как процессы напрямую работают с общим участком памяти без копирования.
 - **Named Pipes (именованные каналы)** — очень быстрый и удобный для передачи потоков данных, не требует сетевого уровня, имеет малые издержки.
- **Скорость:**
 - Shared Memory > Named Pipes > TCP/IP локальные соединения
- **Использование ресурсов:**
 - Shared Memory требует тщательной синхронизации (семафоры), но минимальна накладная часть на передачу данных.
 - Named Pipes — более удобные API, чуть больше издержек.
- Пример: Named Pipes эффективно применяются для обмена сообщениями между процессами Windows на одном хосте. [12] [13]

3. Между приложениями на разных хостах (сетевые взаимодействия)

- **Наиболее эффективные протоколы:**
 - **TCP/IP сокеты** — базовое средство для сетевого взаимодействия, гибкое и универсальное, но требует собственной реализации протокола на уровне приложений.
 - **gRPC** — используется для высокопроизводительного вызова удаленных процедур поверх HTTP/2 с сжатием, меньше накладных расходов из-за бинарного формата.

- REST — более универсальный, но менее эффективный в плане скорости и ресурсов из-за текстового формата (JSON/XML) и постоянных накладных расходов HTTP.
- По скорости и ресурсам:
 - gRPC > TCP/IP чистый > REST
- Пример: gRPC используется в распределенных микросервисах для минимизации задержек и повышения пропускной способности. [14] [15]

Итоговая сравнительная таблица

Сценарий	Протокол/Механизм	Скорость	Ресурсопотребление	Особенности
Внутри приложения	System.Threading.Channels	Очень высокая	Очень низкое	Асинхронный обмен, легковесно
Между приложениями (один хост)	Shared Memory	Максимальная	Низкое	Очень быстрая, требует синхронизации
	Named Pipes	Высокая	Среднее	Удобно, потоковая передача
Между хостами	gRPC	Очень высокая	Низкое	Высокая производительность, бинарный протокол
	TCP/IP	Средняя	Среднее	Универсально
	REST	Низкая	Высокое	Универсально, но медленно и ресурсоемко

Таким образом, для межпроцессного взаимодействия:

- В одном приложении лучше выбирать асинхронные каналы System.Threading.Channels.
- Между приложениями на одном компьютере — shared memory (если приемлема сложность синхронизации) или named pipes.
- Между машинами в сети — gRPC для максимальной эффективности, либо TCP/IP и REST в зависимости от требований к простоте или совместимости.

Это обоснованная практическая рекомендация с учетом скорости, ресурсов и эффективности для разных целей IPC. [16] [17] [18] [12] [14]

**

Я тут про gRPC узнал, что он очень быстрый.

Но мне сказали из некоторых источников, что grpc - не так быстр как кажется потому, что перед передачей следующей порции данных gRpc ЖДЕТ подтверждения что предыдущие данные получены. Поэтому его скорость нужно уменьшить в Два раза, потому что пока он не дождется подтверждения приема отправленного сообщения - он следующее

сообщение не отправит.

Что скажете ?

Ваша информация частично верна, но нужно уточнить детали.

Как работает gRPC с подтверждениями (acknowledgements)

- gRPC опирается на протокол HTTP/2, а тот использует механизм управления потоком (flow control).
- При передаче потоковых данных в gRPC отправитель (клиент или сервер) учитывает размер окна управления потоком. Это окно показывает, сколько данных получатель готов принять.
- Отправитель может отправлять несколько сообщений подряд, пока не исчерпается это «окно». Он НЕ ждет подтверждения отдельного сообщения, а работает с потоком данных, которые получатель «указывает», что может принять.
- Когда данные потребляются на стороне получателя, его окно обновляется (отправляются «подтверждения»), что дает отправителю право послать больше данных.
- По сути, это не «ждать подтверждения каждого сообщения», это управление скоростью передачи — чтобы не перегрузить получателя.

Почему может казаться, что скорость снижается вдвое

- Если поток данных очень маленький или сообщения очень малы, транспорт может ожидать немного, пока окно управления потоком обновится — это называется эффектом «разделения на пакеты» и задержками в TCP/IP и HTTP/2.
- При неправильно настроенных размерах окна (flow control window) или малом размере сообщений это может сильно проявляться.
- При передаче больших потоков данных с нормальной настройкой окна передачи сообщений происходит эффективно, без ожидания между сообщениями.

Итог

- gRPC НЕ ждет подтверждения каждого конкретного сообщения перед отправкой следующего.
- Он использует протокол HTTP/2 с потоковым управлением (flow control) и может отправлять несколько сообщений подряд, пока окно получателя позволяет.
- Если кажется, что пропускная способность занижена вдвое — это, вероятно, из-за маленьких настроек окон, структуры данных или особенностей приложения.
- Можно оптимизировать размеры окна и пакетирование данных для повышения пропускной способности gRPC.

Для деталей и разъяснений можно ссыльаться на документацию и обсуждения gRPC, где описывается flow control и производительность <https://grpc.io/docs/guides/flow-control/> и анализ производительности <https://ably.com/blog/grpc-stream-performance>

Если нужна, могу привести примеры настройки окон и потокового взаимодействия в gRPC.

**

Как gRPC реализует управление потоком и подтверждения на уровне HTTP/2

gRPC реализует управление потоком и подтверждения на уровне HTTP/2 при помощи механизма, называемого flow control (управление потоком). Вот как это работает в деталях:

Управление потоком (Flow Control) в HTTP/2, используемом gRPC

- HTTP/2 делит TCP-соединение на несколько независимых потоков (streams), где каждый поток — это логический канал обмена данными между клиентом и сервером.
- Для каждого потока (и для всего соединения в целом) существует окно управления потоком — это количество данных, которое получатель (клиент или сервер) готов принять без получения дополнительных подтверждений.
- При передаче данных отправитель может передавать данные, пока размер этого окна не исчерпан. Это значит, что несколько фреймов данных (сообщений gRPC) можно отправить подряд без ожидания подтверждения каждого из них.
- Когда получатель обрабатывает данные, он отправляет фрейм WINDOW_UPDATE, увеличивающий размер окна и дающий отправителю сигнал, что можно продолжать передачу.
- Таким образом, подтверждения в виде WINDOW_UPDATE не привязаны к отдельным сообщениям, а контролируют поток всего потока передачи данных, обеспечивая равномерную загрузку и предотвращая переполнение буфера получателя.

Почему это эффективно для gRPC

- gRPC может эффективно мультиплексировать множество вызовов на одном TCP-соединении благодаря HTTP/2.
- Использование flow control позволяет избегать потери пакетов и излишнего ожидания, пока данные не будут обработаны.
- В отличие от простого HTTP 1.1, где каждое сообщение блокирует отправку следующего, HTTP/2 в gRPC обеспечивает параллельную передачу нескольких сообщений, улучшая пропускную способность.

Итог

- gRPC НЕ ждет подтверждения каждого отдельного сообщения.
- Управление скоростью передачи контролируется размером окна потока HTTP/2 и обновляется подтверждениями WINDOW_UPDATE.
- Это обеспечивает баланс между скоростью передачи и защищает получателя от перегрузки.
- Такая архитектура позволяет gRPC сохранять высокую производительность и низкую задержку даже при интенсивном обмене потоковыми данными.

Подробности и документация по flow control в HTTP/2 и gRPC:

- <https://grpc.io/docs/guides/flow-control/>
- <https://habr.com/ru/companies/otus/articles/780720/>
- <https://learn.microsoft.com/ru-ru/aspnet/core/grpc/comparison?view=aspnetcore-9.0>
- <https://proselyte.net/http-evolution/>

Если нужен пример или технические детали реализации flow control в gRPC на конкретной платформе, могу помочь с этим дополнительно.

**

Какие инструменты используются для мониторинга flow control в gRPC

Для мониторинга управления потоком (flow control) в gRPC обычно используются следующие инструменты и подходы:

1. Prometheus с экспортёрами для gRPC

- Prometheus — популярный инструмент мониторинга, который может собирать метрики о работе gRPC-сервисов.
- Для сбора метрик gRPC часто добавляют промежуточные слои (interceptors) в сервер и клиент, которые экспортят метрики, включая показатели производительности, задержек и состояния потоков.
- Хотя Prometheus сам не мониторит непосредственно внутренние окна управления потоком HTTP/2, метрики о пропускной способности, загрузке и задержках помогают опосредованно оценить эффекты flow control.
- Для сбора метрик gRPC приложению дополняют HTTP эндпоинт, через который Prometheus считывает данные.

2. Инструменты трассировки (Tracing)

- Инструменты вроде Jaeger, Zipkin, OpenTelemetry позволяют трассировать вызовы gRPC, измерять время на каждом этапе, накладные расходы и узкие места, которые могут быть вызваны flow control.
- Трассировка помогает понять, где задерживается поток сообщений и как распределяются подтверждения.

3. HTTP/2 и TCP уровни

- Для детального анализа flow control зачастую используют инструменты сетевого мониторинга и трассировки TCP-соединений (например, Wireshark), чтобы увидеть WINDOW_UPDATE фреймы HTTP/2, которые реализуют flow control в gRPC.
- Это позволяет анализировать фактический обмен окон и подтверждений на протоколах нижнего уровня.

4. Встроенные средства в gRPC (статистика и логирование)

- В некоторых реализациях gRPC есть встроенные средства логирования и статистики, которые можно включить для получения информации по состоянию соединения и управления потоком.
- В .NET и других платформах часто доступны интерсепторы для расширенного мониторинга вызовов.

Итог

- Для мониторинга flow control в gRPC в реальной практике применяют сочетание: прометеус-метрики с интерцепторами, системы трассировки (Jaeger, OpenTelemetry), сетевые инструменты (Wireshark), а также встроенные логи и статистику gRPC.
- Это позволяет получить и агрегированные показатели, и низкоуровневый сетевой анализ.

Если нужно, могу подробно расписать пример интеграции Prometheus для gRPC мониторинга или показать, как активировать трассировку с OpenTelemetry для gRPC. [43] [44] [45]



1. <https://learn.microsoft.com/en-us/archive/msdn-magazine/2019/november/csharp-iterating-with-async-enumerables-in-csharp-8>
2. <https://danimacan.com/how-to-use-iasyncenumerable-in-c-for-paged-apis/>
3. <https://www.red-gate.com/simple-talk/development/dotnet-development/working-with-iasyncenumerable-in-c/>
4. <https://dev.to/petersaktor/a-concise-guide-to-asynchronous-data-flow-with-c-channels-45af>
5. <https://the0x00.dev/c%23/hivemq/c-System.IO.Pipelines-Cheatsheat/>
6. <https://dev.to/volkanalkilic/using-named-pipes-in-c-for-interprocess-communication-4kp1>

7. <https://clouddevs.com/c-sharp/tcp-ip-applications-for-networking/>
8. <https://roshancloudarchitect.me/how-to-implement-grpc-in-c-a-step-by-step-guide-6ab3bbf663e2>
9. <https://www.youtube.com/watch?v=qTetsXmZLk0>
10. <https://stackoverflow.com/questions/62731425/how-to-use-iasyncenumerable-in-repository-class>
11. <https://the0x00.dev/c%23/hivemq/c-System.IO.Pipelines-Cheatsheat/>
12. <https://stackoverflow.com/questions/2002835/whats-the-fastest-ipc-method-for-a-net-program>
13. <https://dev.to/volkanalkilic/using-named-pipes-in-c-for-interprocess-communication-4kp1>
14. <https://learn.microsoft.com/en-us/aspnet/core/grpc/interprocess?view=aspnetcore-9.0>
15. <https://roshancloudarchitect.me/how-to-implement-grpc-in-c-a-step-by-step-guide-6ab3bbf663e2>
16. <https://github.com/cloudtoid/interprocess>
17. <https://www.geeksforgeeks.org/operating-systems/ipc-shared-memory/>
18. <https://www.geeksforgeeks.org/operating-systems/inter-process-communication-ipc/>
19. <https://www.apriorit.com/dev-blog/web-python-ipc-methods>
20. <https://stackoverflow.com/questions/21101243/interprocess-communication-on-the-same-box-communication-between-2-application>
21. <https://stackoverflow.com/questions/3742334/using-ipc-for-different-systems>
22. https://www.reddit.com/r/dotnet/comments/j27bya/what_is_available_for_interprocess_communication/
23. <https://learn.microsoft.com/en-us/windows/win32/ipc/interprocess-communications>
24. <https://github.com/grpc/grpc/issues/12539>
25. <https://github.com/grpc/grpc-go/issues/1043>
26. <https://github.com/grpc/grpc-node/issues/2641>
27. <https://ably.com/blog/grpc-stream-performance>
28. <https://stackoverflow.com/questions/65731855/sending-grpc-requests-without-waiting-for-response>
29. <https://grpc.io/docs/guides/flow-control/>
30. <https://yifan-online.com/en/km/article/detail/7765>
31. <https://stackoverflow.com/questions/57062815/how-to-implement-lossless-grpc-streaming-calls>
32. <https://grpc.io/docs/guides/keepalive/>
33. <https://grpc.io/docs/guides/performance/>
34. <https://habr.com/ru/companies/otus/articles/780720/>
35. <https://learn.microsoft.com/ru-ru/aspnet/core/grpc/comparison?view=aspnetcore-9.0>
36. <https://yandex.cloud/ru/docs/glossary/grpc>
37. <https://babok-school.ru/blog/grpc-api-implementation-example/>
38. <https://proselyte.net/http-evolution/>
39. <https://habr.com/ru/companies/otus/articles/730740/>
40. <https://purpleschool.ru/knowledge-base/article/grpc-transport>
41. https://www.reddit.com/r/dotnet/comments/r2ekfj/practicalreallife_usecases_for_grpc/
42. <https://testgrow.ru/articles/article64>
43. <https://tproger.ru/articles/grpc-integration-experience>
44. <https://grpc.io/docs/guides/flow-control/>

45. <https://my-js.org/docs/cheatsheet/system-design-101>
46. <https://platformmv.sbertech.ru/docs/public/BPM/4.10/common/documents/about/index.html>
47. <https://www.protokols.ru/WP/rfc9232/>
48. <https://habr.com/ru/companies/otus/articles/657017/>
49. <http://new.groteck.ru/images/catalog/61601/d9c4c1d91ab02094a7d0f6183f2125ea.pdf>
50. <https://habr.com/ru/hubs/java/posts/page4/>
51. <https://weril.me/nms/>
52. <https://testgrow.ru/articles/article64>