gRPC - рецепты счастья

Кузнецов Михаил

План выступления

- Основы gRPC
  - Кто знаком - будет кратко, не засыпайте :)
  - Кто не знаком - на прошлом дотнексте был доклад
- Альтернативы
- Наш опыт в продакшене

Что такое и откуда взялся gRPC?

- Протокол удалённого вызова процедур от Google
- Кроссплатформенная
- Транспорт по HTTP/2
- Поддерживается в .NET, с 3.1 - очень неплохо, в 5.0 - ещё лучше

Какую проблему мы решаем?

- Перешли с монолита на микросервисы - стало много сетевых вызовов
- Упал перформанс - вызвать метод в коде очевидно дешевле сетевого вызова внешнего сервиса
- Kafka/RabbitMQ и прочее - асинхронно и не всегда уместно

А чем это лучше REST?

- Контракты расползаются даже со Swagger, потому что не Contract First
- Много бойлерплейт кода (сериализация, обработка ошибок, Dto)
- Перформанс - не очень
- По сети летает жирный JSON или XML

Что предлагает gRPC?

- Contract First - контракты лежат в репозитории
  - *Nit:* не обязательно, можно Code First, если весь бэк на .NET
- Различные платформы и языки хорошо дружат
- Строгая типизация
- Автогенерация клиента и сервера
  - Контракты не разваливаются, меньше кода поддерживать
  - Встроенная сериализация "из коробки", эффективный бинарный формат
- Лаконичная обработка ошибок
- Удобный стриминг в оба конца (в т.ч. дуплекс!)

Поддержка в .NET

- gRPC C#
  - https://github.com/grpc/grpc/tree/master/src/csharp
  - Обёртка вокруг unmanaged библиотеки
  - Максимальная функциональность / минимальные удобства
    - Все настройки на строках
    - Прелести unmanaged кода
  - Используем местами не в production (эмуляторы/моки/тесты)
- gRPC for .NET
  - https://github.com/grpc/grpc-dotnet
  - Managed
  - Интегрируется в ASP.NET Core
  - Настройки строго типизированы

gRPC for .NET

- Транспорт по HTTP2 - внутри gRPC-клиента сидит System.Net.Http.HttpClient, поэтому многие настройки в привычных полях и классах
  - Авторизация
  - Keep Alive
  - Таймауты
  - И т.д.
- Многие серверные gRPC настройки - настраиваются в Kestrel стандартным способом

Protobuf контракты

- Строгая типизация
- Строго определённая последовательность полей
- Поддержка nullable
- Поддержка коллекций, enum-ов
- Поддержка дефолтных значений
- Вложенные типы
- До 10 раз компактнее, до 100 раз быстрее JSON © Google
  - Кратно компактнее, в 20-30 раз быстрее - наш опыт
  - На уровне MessagePack

```
message Outcome{
    string id = 1;
    int32 selection_kind = 2;
    double price = 3;
    bool disabled = 4;
    map<string, string> properties = 5;
    string raw_id = 6;
}
```

Кроссплатформенность и кодогенерация

- Клиент и сервер в части сериализации/десериализации и транспорта - генерируются
- Поддержка C#, Go, Java, Python, Kotlin, PHP, Ruby
- Никаких проблем с кейсингом, опечатками, форматом (дат, чисел и тд)
- Код обоих сервисов синхронизован с контрактом - гарантируется компилятором

Виды вызовов

- Unary call
- Streaming
  - Клиентский
  - Серверный
  - Дуплекс

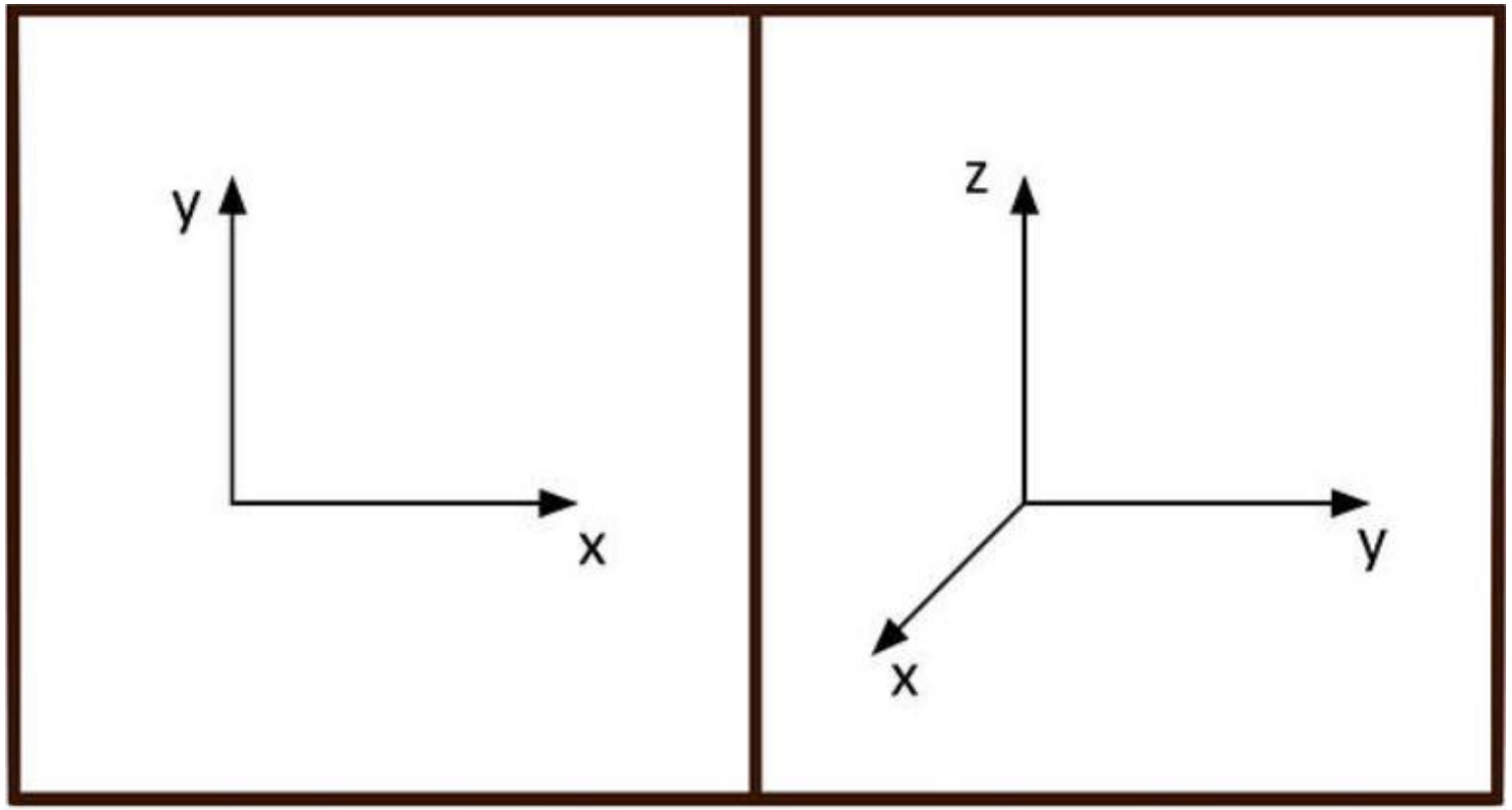
Обсудим альтернативы - Kafka...?

- Нет параметризации запроса (фильтрации).
- Пример - GetUserInfoById(int id) - для тех, кто сейчас онлайн
  - Либо сложный менеджмент топиков
  - Либо дискардим много сообщений
- Отсутствует уведомление о том, что продьюсер перестал обновлять данные.

Обсудим альтернативы #2

- OpenAPI
  - Контракты необязательные, но есть
  - Кодогенерация клиента - thirdparty
  - Перформанс хуже, нагрузка на сеть - выше
  - Нет дуплекса
- HttpClient.GetStreamAsync
  - Можно стримить
  - Все остальные проблемы с типизацией, контролем контрактов и сериализацией - остаются

А подводные камни?



Как хотелось бы

Как есть на самом деле

Минусы gRPC

- Нечитаемый человеком payload
- Не интегрируется с браузерами (а с мобилками - может, но опыта нет :-P)
  - Но есть gRPC HTTP API, есть http-gateway
- Не поддерживает броадкаст

HealthCheck gRPC сервиса

- Liveness/Readiness probes, мониторинг
- Отдельный пакет, реализующий стандарт
  - https://github.com/grpc/grpc/blob/master/doc/health-checking.md
  - https://www.nuget.org/packages/Grpc.HealthCheck/
- Статусы и управление статусами из кода

```
_healthGrpcService.SetStatus(string.Empty,
    HealthCheckResponse.Types.ServingStatus.Serving);
```

gRPC и KeepAlive

Defaults Values

Channel Argument	Client	Server
GRPC_ARG_KEEPALIVE_TIME_MS	INT_MAX (disabled)	7200000 (2 hours)
GRPC_ARG_KEEPALIVE_TIMEOUT_MS	20000 (20 seconds)	20000 (20 seconds)
GRPC_ARG_KEEPALIVE_PERMIT_WITHOUT_CALLS	0 (false)	0 (false)
GRPC_ARG_HTTP2_MAX_PINGS_WITHOUT_DATA	2	2

- Тоже стандартизован
- Pings without data / Ping strikes
- Нельзя просто взять и... включить бесконечные пинги

Встраиваем в ASP.NET Core - server

- services.AddGrpc(o => ....); // o is GrpcServiceOptions
  - MessageSize
  - CompressionProviders
  - Interceptors
- services.AddAuthentication()

```
app.UseEndpoints(endpoints ⇒
{
    endpoints.MapGrpcService<UserUIServer>()
        .RequireCors("policy")
        .RequireAuthorization();
});
```

Встраиваем в ASP.NET Core Client – basic

- Можно через new()
- Можно через DI

```
services.AddGrpcClient<UserOnlineService.UserOnlineServiceClient>(
    options ⇒
{
    options.Address = new Uri(Configuration.GetValue<string>("UserOnlineServiceUrl"));
});
```

- С продвинутыми настройками есть хитрости

Встраиваем в ASP.NET Core Client – wrong

```
services.AddGrpcClient<AgentService.AgentServiceClient>((sp, o) ⇒
    {
        o.Address = new Uri(settings.Uri);
        // bad !
        o.ChannelOptionsActions.Add(c ⇒ c.HttpHandler = new HttpClientHandler()
        {
        });
    })
```

Встраиваем в ASP.NET Core Client – advanced

```
services.AddGrpcClient<AgentService.AgentServiceClient>((sp, o) ⇒
    {
        o.Address = new Uri(settings.Uri);
    })
    .ConfigurePrimaryHttpMessageHandler(_ ⇒ new SocketsHttpHandler
    {
        SslOptions = new SslClientAuthenticationOptions
        {
            ClientCertificates = new X509CertificateCollection { TryGetCertificate() }
        },
        PooledConnectionIdleTimeout = Timeout.InfiniteTimeSpan,
        KeepAlivePingDelay = TimeSpan.FromSeconds(10),
        KeepAlivePingTimeout = TimeSpan.FromSeconds(15),
        ConnectTimeout = TimeSpan.FromSeconds(30),
        EnableMultipleHttp2Connections = true
    });
```

Корректный shutdown gRPC-сервера.

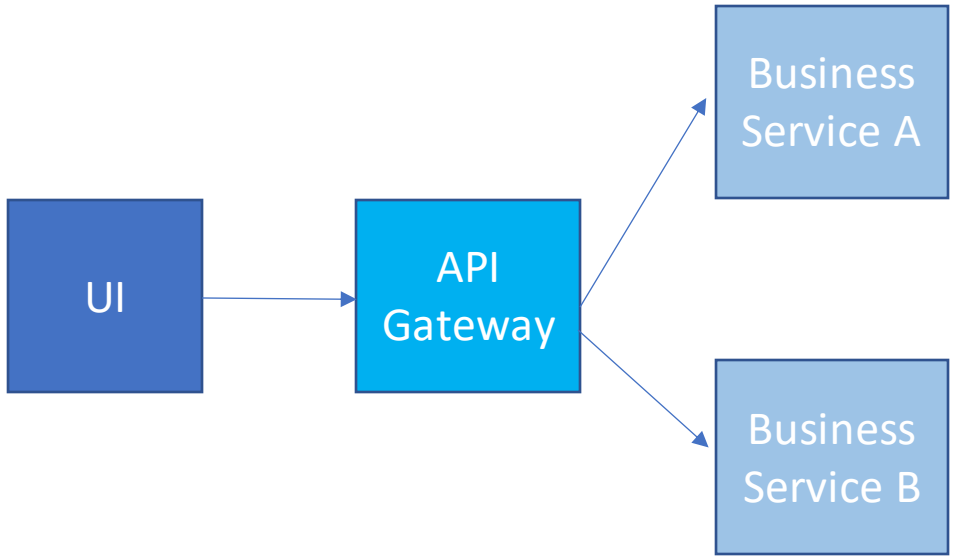
- Долго боролись и не понимали, что мы делаем не так - клиент не узнаёт о том, что сервер умер (а должен!)
- В документации на MSDN про завершение работы ничего нет :)
- Что мы узнали
  - gRPC-сервера на python - всё ок
  - Go - по-разному
  - .NET - всё плохо
- Что мы нашли, сниффая трафик? GOAWAY.
- А чем отличались реализации на Go?
- Квиз. Какой метод правильный - Stop() или GracefulStop()?
- RST_STREAM

Корректный shutdown. Итог.

- Ну хорошо, а как быть с dotnet?
- IHostApplicationLifeTime.ApplicationStopping
- CancellationToken.Register
- k9s может вас смутить :)

Зачем (и как) мы отказались от типизации

- Проблема - есть API Gateway - фронт на websockets, бекенды - gRPC-серверы
- Не хотим дорабатывать его при разработке новой фичи
- Решение в виде универсального протокола. Строгую типизацию частично теряем. И перформанс.
- В конфиге прописывается мэппинг Name на { Url, ServiceName }



Обработка ошибок и специфика отладки

- Неправильный protobuf - беда, ничего не понять. Первое что надо проверить :)
- Ловим исключения RpcException, проверяем статус. 16 штук и все понятные
- Расширенные логи через стандартную конфигурацию - помогают

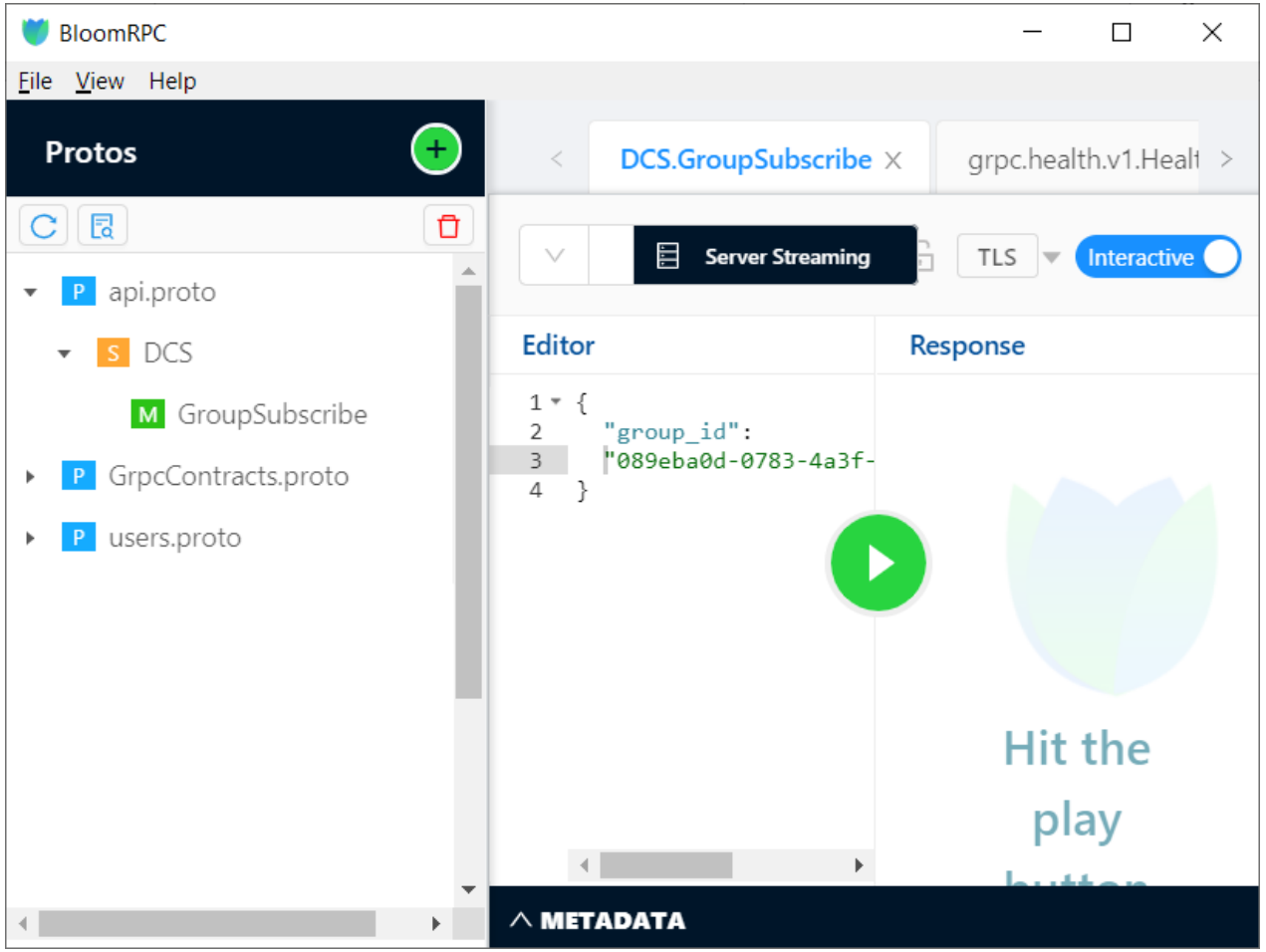
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Grpc": "Debug"
    }
  }
}
```

Разберём некоторые статусы

- Их можно возвращать самим, но многие кидает библиотека
- **Internal** – самый понятный
- **Canceled** - не только срабатывание Cancellation в таске отправки, но и некоторые ошибки при работе с клиентом
- **NotFound** - вообще ничего нет. **Unimplemented** – service есть, а вот такого method - нет

Инструментарий

- BloomRPC
  - Как Postman/Fiddler для gRPC сервисов
  - Учитывая специфику фреймворка - достаточно загрузить .proto файл и указать эндпоинт, чтобы отправить запрос



Немного о производительности

- На одном ядре - сотни сообщений в секунду. Не синтетика - реальная нагрузка. Накладные расходы на gRPC - трудноизмеримы на фоне других действий.
- Есть проблемы с большими сообщениями и LOH
  - UnsafeByteOperations - можно попробовать
- До десятков тысяч gRPC-стримов между двумя нодами - всё ок

Подведём итоги

- .NET Core 3.1+
- Микросервисы
  - Бонусом - на разных стеках
- Болит
- Стоит рассмотреть gRPC

Спасибо

Жду ваши вопросы!