Практика 12: технологии распределённых приложений

Юрий Литвинов
yurii.litvinov@gmail.com

04.04.2022

1. protobuf

Рассмотрим типичный и активно использующийся в индустрии пример RPC-системы — Google gRPC. Но поскольку gRPC использует в качестве протокола сериализации протокол от Google, который называется protobuf, сначала кратко рассмотрим его.

Google Protocol Buffers (более известный как protobuf) — это бинарный формат сериализации произвольных данных. Разработан для внутренних нужд сервисов Google, с целью сэкономить трафик и время передачи. Ближайшие аналоги (XML и JSON) хранят данные в текстовом виде, что хорошо в плане отладки, поскольку они человекочитаемы, но не очень хорошо в плане эффективности передачи или места на диске. protobuf же сериализует данные в хитрый бинарный формат (см. https://developers.google.com/protocol-buffers/docs/encoding, особого внимания заслуживает способ хранения целых чисел — Base 128 Varints), что позволяет, по их заявлениям, хранить данные в среднем в 10 раз компактнее формата XML. За это, естественно, надо платить — protobuf-сообщение не содержит в себе практически никакой информации о своей структуре, передаются только номера полей и значения, при этом чтобы однозначно разобрать сообщение, надо знать типы полей, описанные отдельно.

Это самое отдельное описание — декларативное описание структуры сообщения в файле .proto, по которому генерируется код для чтения/записи сообщения. Вот пример .proto-файла:

```
syntax = "proto3";

message Person {
    string name = 1;
    int32 id = 2;
    string email = 3;
}
```

Версий protobuf бывает две — proto2 и proto3, с небольшими синтаксическими отличиями, в реальности используются обе (подробности про отличия — см. в документации).

Инструменты поддерживают обе версии протокола, но первой строкой надо указывать версию протокола. Дальше идёт одно или несколько описаний форматов сообщений (в нашем примере — Person). Сообщения состоят из полей, у каждого поля есть имя, тип и номер. Номера нужны для поддержки изменений в формате записи — сервер, получив сообщение с незнакомыми полями, может их спокойно проигнорировать. Номера полей есть в любой подобной технологии (например, Apache Thrift описывает формат передачи похожим образом), поскольку протоколы связи имеют свойство эволюционировать, и поддержание обратной совместимости в распределённых приложениях весьма важно.

Ещё protobuf умеет типы-перечисления, массивы/списки (с помощью ключевого слова repeated), вложенные сообщения, импорт сообщений из других файлов, даже что-то вроде вариантных записей (с помощью ключевого слова oneof). В общем, довольно развитый мини-язык программирования для описания данных.

На .proto-файлы во время сборки приложения напускается генератор protoc, который генерирует код под целевой язык программирования. Например, для Java он генерирует класс, представляющий сообщение, и строитель (паттерн «Строитель»), который позволяет удобно сообщение сформировать. Поддерживать эти файлы не надо, и даже смотреть на них не надо, они перегенерятся после каждой сборки. В клиентском коде пользоваться ими можно, например, вот так:

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```

Builder-у передаются значения полей, дальше методом build() он делает собственно сообщение, у которого есть метод writeTo, принимающий поток, куда надо записать байтовый массив с сериализованным сообщением. Обычно это сетевой поток, но никто не мешает писать protobuf-сообщения на диск (это вполне валидный и весьма компактный формат сохранения), или вообще куда угодно.

protoc обычно подключается как дополнительный компилятор средствами системы сборки. Например, для gradle и maven есть плагины, укачивающие при сборке protoc как maven-пакет, запускающие его на все .proto-файлы, и подключающие результат генерации как обычные Java- или Kotlin-файлы в проект перед запуском обычного компилятора. То есть настройка процесса сборки — это просто скопировать несколько строк в свой файл с конфигурацией сборки, ничего качать и устанавливать не нужно. Более-менее аналогично оно работает и для других языков (в C++, правда, так и нет пока стандартного пакетного менеджера, там protoc подключить посложнее). Поддерживаются языки Java, Python, Kotlin, Objective-C, C++, Go, Ruby, C#, Dart.

1.1. gRPC

protobuf — это лишь формат сериализации, так что для написания распределённых систем используется обычно вместе с gRPC, который работает поверх protobuf и даже реали-

зован как расширение к компилятору protoc. gRPC уже позволяет описать сервис в терминах поддерживаемых методов, форматы параметров и возвращаемого результата каждого метода в виде protobuf message. Так же, как в protobuf, описание сервиса языконезависимое, на него потом запускается генератор, порождающий клиентскую или сервисную заглушку на целевом языке. Так же, как и protobuf, gRPC поддерживает кодогенерацию в Java, Python, Kotlin, Objective-C, C++, Go, Ruby, C#, Dart. gRPC нынче весьма популярен, во-первых, благодаря тому, что Google его сам активно использует в своей инфраструктуре, во-вторых потому, что это простой и легковесный способ писать распределённые приложения, хотя это именно RPC-система, так что её возможности весьма ограничены. Например, gRPC не умеет наследование message, не умеет даже методы, не возвращающие значение (они симулируются методами, возвращающими библиотечное сообщение Message.Empty).

Вот пример описания несложного gRPC-сервиса[1]:

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.routeguide";
option java_outer_classname = "RouteGuideProto";

service RouteGuide {
    rpc GetFeature(Point) returns (Feature) {}
    rpc ListFeatures(Rectangle) returns (stream Feature) {}
    rpc RecordRoute(stream Point) returns (RouteSummary) {}
    rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}

message Point {
    int32 latitude = 1;
    int32 longitude = 2;
}

message Rectangle {
    Point lo = 1;
    Point hi = 2;
}

message Feature {
    string name = 1;
    Point location = 2;
}

message FeatureDatabase {
    repeated Feature feature = 1;
}
```

[1] Следуя туториалу https://grpc.io/docs/languages/java/basics/ (дата обращения: 08.11.2021).

```
message RouteNote {
    Point location = 1;
    string message = 2;
}

message RouteSummary {
    int32 point_count = 1;
    int32 feature_count = 2;
    int32 distance = 3;
    int32 elapsed_time = 4;
}
```

Тут определяется сервис для туристического приложения, который умеет сообщать о достопримечательностях по данным координатам или в данном прямоугольнике, записывать маршрут (состоящий из маршрутных точек) и аннотировать маршрут, сохраняя на сервер и принимая с сервера заметки относительно интересных точек на маршруте.

Сначала указывается версия синтаксиса .proto-файла, затем опции для кодогенератора (мы хотим .java-файлы, по одному на каждый класс, enum и т.п., в пакете io.grpc.examples.routeguide, при этом чтобы «фасадом» для всего этого был класс RouteGuideProto).

Дальше описание собственно сервиса. Он интересен тем, что показывает все возможные режимы передачи данных — gRPC поддерживает как протоколы вида «запрос-ответ», так и асинхронные потоки сообщений. GetFeature — классический метод в режиме «запрос-ответ», мы посылаем на сервис точку и получаем достопримечательность, которая в этой точке находится. ListFeatures работает в потоковом режиме, мы отправляем прямоугольник и получаем стрим, из которого можем вычитывать (возможно асинхронно) сообщения с достопримечательностями. Между клиентом и сервером открывается канал, куда сервер может писать новые сообщения, и клиент по мере их появления в канале может их оттуда вычитывать. Когда сервер хочет сказать, что больше сообщений не будет, он должен в явном виде закрыть канал (чуть позже это будет видно в примере реализации сервиса).

RecordRoute работает «в другую сторону» — клиент открывает канал, куда время от времени может писать сообщения, а сервер их асинхронно вычитывает. Когда клиент закончил передачу (например, мы доехали до конечной точки маршрута), он должен явно сообщить об этом серверу, и канал будет закрыт. После этого сервер формирует и отправляет клиенту ответ (одно сообщение на сей раз). RouteChat работает в обе стороны — между клиентом и сервером открывается канал, куда и клиент и сервер могут писать сообщения независимо друг от друга, вычитывая их асинхронно. Тут уже любая сторона может инициировать закрытие соединения.

Дальше идут описания форматов сообщений, используемых сервисом (и, в случае FeatureDatabase, заодно и используемых сервером для сохранения данных на диск, судя по всему).

Когда мы сгенерируем код серверной части, запустив protoc (в Java за нас это сделает соответствующий плагин к Maven или Gradle), получим серверную заглушку RouteGuideGrpc.RouteGuideImplBase, в которой реализована вся сетевая часть и объяв-

лены виртуальные методы, по одному на каждый метод сервиса из .proto-файла, которые нам надо реализовать. Вот так может выглядеть реализация GetFeature:

```
private static class RouteGuideService extends RouteGuideGrpc.RouteGuideImplBase {
    ...
    @Override
    public void getFeature(Point request, StreamObserver<Feature> responseObserver) {
        responseObserver.onNext(checkFeature(request));
        responseObserver.onCompleted();
    }
    ...
}
```

Этот метод работает в режиме «запрос-ответ», поэтому принимает прямо объект типа Point (его десериализует из protobuf-запроса как раз RouteGuideImplBase при получении запроса от клиента по сети). А вот чтобы вернуть ответ, мы должны воспользоваться классом `StreamObserver<Feature>`. У StreamObserver всего три метода — onNext(), onCompleted() и onError(), и по сути это реализация паттерна «Наблюдатель». StreamObserver нотифицируют о наступлении события «появились новые данные», он эти данные сериализует и отправляет по сети на клиент. В данном случае мы должны отправить всего один ответ, поэтому в onNext мы передаём нужные данные (которые для нас готовит не показанный тут доменный метод checkFeature, который просто как-то находит достопримечательности в базе, не важно как). Дальше вызовом onCompleted мы сообщаем об окончании передачи.

Вот так может выглядеть реализация ListFeatures:

```
private static class RouteGuideService extends RouteGuideGrpc.RouteGuideImplBase {
    ...
    @Override
    public void listFeatures(Rectangle request, StreamObserver<Feature> responseObserver) {
        ...
        for (Feature feature : features) {
            int lat = feature.getLocation().getLatitude();
            int lon = feature.getLocation().getLongitude();
            if (lon >= left && lon <= right && lat >= bottom && lat <= top) {
                responseObserver.onNext(feature);
            }
        }
        responseObserver.onCompleted();
    }
}
```

Тут передаваемых сообщений уже много, поэтому onNext у responseObserver вызывается в цикле. Но по окончании цикла мы должны вызвать onCompleted, чтобы сообщить клиенту, что передача закончена. Обратите внимание, что никто не заставляет передавать все сообщения сразу, между вызовами onNext может пройти значительное время, клиент будет получать сообщения по мере их готовности.

Вот так может выглядеть реализация RouteChat:

```
@Override
public StreamObserver<RouteNote> routeChat(
        final StreamObserver<RouteNote> responseObserver) {
    return new StreamObserver<RouteNote>() {
        @Override
        public void onNext(RouteNote note) {
            List<RouteNote> notes = getOrCreateNotes(note.getLocation());
            for (RouteNote prevNote : notes.toArray(new RouteNote[0])) {
                responseObserver.onNext(prevNote);
            }
            notes.add(note);
        }

        @Override
        public void onError(Throwable t) {
            logger.log(Level.WARNING, "routeChat cancelled");
        }

        @Override
        public void onCompleted() {
            responseObserver.onCompleted();
        }
    };
}
```

Здесь, поскольку канал общения у нас двусторонний, приём и отправка сообщений должны быть симметричны, и поэтому мы и принимаем, и возвращаем StreamObserver. responseObserver, который мы принимаем, как и раньше используется, чтобы писать туда наши данные, которые мы хотим отправить клиенту. Возвращаем мы StreamObserver, который сам gRPC будет дёргать, когда клиент пришлёт нам очередное сообщение. Его метод onNext работает так же, как getFeature и listFeatures — принимает одно сообщение, что-то с ним делает, и если в результате появляется что-то, что надо передать на клиент, пишет это в responseObserver. В отличие от методов, рассмотренных выше, onCompleted тут не вызывается, поскольку нам могут прислать ещё сообщение и мы должны на него отреагировать. Вот когда клиент сообщит о конце передачи, вызовется наш метод onCompleted, и мы корректно закончим передачу. onError вызовется при возникновении какой-либо ошибки (например, клиент отключился, не сказав «onCompleted»).

Обратите внимание, что в этом примере сервер реагирует своим методом onNext на запрос клиента, так что не может «спонтанно» послать сообщение. В реальной жизни, однако, ничто не мешает отдать responseObserver в отдельный поток, и генерить события когда и как ему заблагорассудится. Единственное что надо понять, создаёт ли gRPC-сервер уже отдельный поток для слушания клиента или запускается в основном потоке приложения, в клиентах для разных языков это делается по-разному.

Метод RecordRoute мы рассматривать не будем, потому что он устроен аналогично RouteChat: должен вернуть StreamObserver, который бы в onNext ничего не посылал, а в onCompleted делал бы responseObserver.onNext перед responseObserver.onCompleted.

На клиенте генерируются синхронная и асинхронная заглушки, и можно выбрать, какую использовать (конечно асинхронную, синхронные сетевые запросы противоречат идеологии распределённых приложений). Там даже наследоваться ни от чего не надо, а просто дёргать методы этих заглушек так, будто весь RouteService находится в нашем процессе:

```
public class RouteGuideClient {
    private final RouteGuideBlockingStub blockingStub;
    private final RouteGuideStub asyncStub;

    public RouteGuideClient(Channel channel) {
        blockingStub = RouteGuideGrpc.newBlockingStub(channel);
        asyncStub = RouteGuideGrpc.newStub(channel);
    }

    public void getFeature(int lat, int lon) {
        Point request = Point.newBuilder().setLatitude(lat).setLongitude(lon).build();

        Feature feature;
        try {
            feature = blockingStub.getFeature(request);
        } catch (StatusRuntimeException e) {
            ...
        }
        if (RouteGuideUtil.exists(feature)) {
            info("Found feature called \"{0}\" at {1}, {2}",
                feature.getName(),
                RouteGuideUtil.getLatitude(feature.getLocation()),
                RouteGuideUtil.getLongitude(feature.getLocation()));
        } else {
            info("Found no feature at {0}, {1}",
                RouteGuideUtil.getLatitude(feature.getLocation()),
                RouteGuideUtil.getLongitude(feature.getLocation()));
        }
    }

    public static void main(String[] args) throws InterruptedException {
        String target = "localhost:8980";
        List<Feature> features =
            RouteGuideUtil.parseFeatures(RouteGuideUtil.getDefaultFeaturesFile());
        ManagedChannel channel =
            ManagedChannelBuilder.forTarget(target).usePlaintext().build();
        try {
            var client = new RouteGuideClient(channel);
            client.getFeature(409146138, -746188906);
        } finally {
```

```
            channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
        }
    }
    ...
}
```

За установление соединения отвечает библиотечный класс ManagedChannelBuilder (часть клиентской библиотеки gRPC для Java), которому надо сказать, куда подключаться (тут это target), что с шифрованием (тут usePlaintext()) и т.п. (тут ничего больше). Дальше мы создаём объект RouteGuideClient, передав ему channel (а то main статический, нам надо сначала «создать себя», чтобы пользоваться нестатическими методами), и вызываем метод getFeature. Конструктор создаст сгенерённые blockingStub и asyncStub, первый из них мы используем в getFeature, передав ему в качестве параметра protobuf-сообщение, полученное с помощью protobuf-builder-а, который мы обсуждали в предыдущем подразделе. Поскольку заглушка синхронная, она тут же отправляет по сети запрос и не возвращает управление, пока не придёт ответ.

Метод RouteChat устроен даже на клиенте более интересно, но рассматривать тут мы его не будем — у нас будет домашняя работа, предполагающая использование двунаправленного асинхронного канала, так что надо оставить на дом что-то, с чем можно поразбираться самостоятельно. Пример кода и некоторые объяснения есть в туториале, так что думаю, что при желании несложно разобраться.

2. Веб-сервисы

2.1. Windows Communication Foundation

Довольно типичный пример технологии, реализующей SOAP — это Windows Communication Foundation, библиотека (точнее, фреймворк) для .NET, которая должна была быть основой всей сетевой коммуникации в .NET-программах, но нынче постепенно вытесняется более простыми штуками типа нового ASP.NET Web APIs. Появилась в .NET 3 (примерно 2006 год), исправно служила человечеству до выхода .NET Core, после чего её некоторое время мучительно переписывали на новую платформу и таки переписали (даже выложили в Open Source: https://github.com/dotnet/wcf). Но, в общем-то, интересна она прежде всего тем, что она используется в корпоративных системах на базе .NET Framework до сих пор очень активно и на её базе сделаны более молодёжные штуки типа того же ASP.NET.

WCF преназначается для быстрого создания веб-сервисов и клиентов для них. Архитектурно она устроена как конфигурируемый фреймворк, состоящий из трёх крупных компонентов: Address, Binding и Contract (так называемые ABCs of WCF). Каждый из компонентов имеет набор абстрактных классов и стандартные их реализации, но можно реализовать и свои, и в конфигурации подсунуть их библиотеке. Так что SOAP не обязателен, можно сделать хоть транспорт на голубиной почте.

© http://www.c-sharpcorner.com

Address отвечает за понятие эндпойнта сервиса и вообще адресацию. Поскольку на одной машине (и даже на одном порту) может быть запущено сразу много сервисов, различаются они URL-ами (обычно частями после слеша).

Binding отвечает за транспорт — транспортный протокол, протокол сериализации, безопасность и подобные вещи.

Contract отвечает за исполнение запроса — либо вызов кода реализации сервиса, либо вызов заглушки, которая должна передать запрос по сети. Именно в этом компоненте реализованы атрибуты, размечающие обычный код на C# так, что по нему потом можно сгенерить WSDL и вызывать по сети[2]:

```
[ServiceContract(Namespace = "http://Microsoft.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);

    [OperationContract]
    double Subtract(double n1, double n2);

    [OperationContract]
    double Multiply(double n1, double n2);

    [OperationContract]
    double Divide(double n1, double n2);
}
```

ServiceContract говорит, что этот интерфейс должен быть доступен как веб-сервис, OperationContract — что данный метод должен быть методом сервиса. Если бы у методов были аргументы не элементарных типов, потребовались бы ещё атрибуты DataContract — чтобы пометить контракты данных (типа message в protobuf, опять-таки).

Дальше этот контракт можно реализовать, уже без всяких атрибутов:

```
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
        => n1 + n2;

    public double Subtract(double n1, double n2)
```

[2] это пример из официальной документации, так что подробности можно посмотреть там.

```
        => n1 - n2

    public double Multiply(double n1, double n2)
        => n1 * n2;

    public double Divide(double n1, double n2)
        => n1 / n2;
}
```

А дальше сказать при старте приложения, что такой-то контракт реализуется таким-то классом, хостится на таком-то URL-е и что про него надо вывесить в сеть его WSDL:

```
static void Main(string[] args)
{
    var baseAddress = new Uri("http://localhost:8000/ServiceModelSamples/Service");
    var selfHost = new ServiceHost(typeof(CalculatorService), baseAddress);

    try {
        selfHost.AddServiceEndpoint(typeof(ICalculator),
            new WSHttpBinding(), "CalculatorService");

        var smb = new ServiceMetadataBehavior();
        smb.HttpGetEnabled = true;
        selfHost.Description.Behaviors.Add(smb);

        selfHost.Open();
        Console.WriteLine("The service is ready. Press <ENTER> to terminate service.");
        Console.ReadLine();

        selfHost.Close();
    } catch (CommunicationException ce) {
        Console.WriteLine($"An exception occurred: {ce.Message}");
        selfHost.Abort();
    }
}
```

`http://localhost:8000/ServiceModelSamples/Service` — это эндпойнт, на котором будет наш сервис. ServiceHost — это веб-сервер внутри процесса (в отличие от внешнего веб-сервера типа IIS или, если не .NET, Apache), он будет слушать порт 8000 нашей машины и все запросы с URL ServiceModelSamples/Service перенаправлять нашему сервису. Как транспорт используется WSHttpBinding (Web Service over HTTP, то есть SOAP на самом деле — в реальной жизни не стоит его использовать, только HTTPS). ServiceMetadataBehavior отвечает за генерацию и публикацию WSDL-документа, который мы говорим, что должен быть доступен по HTTP тоже. По умолчанию WSDL-документ кладётся рядом с сервисом (точнее, на `http://localhost:8000/ServiceModelSamples/Service/wsdl`). Open() запускает всё это дело, дальше мы блокируемся, давая хосту в отдельном потоке делать своё дело, потом закрываем хост.

Обратите внимание, это практически всё, что надо написать для того, чтобы сделать веб-сервис (а всё остальное — это boilerplate, который сгенерит среда разработки). То есть создание инфраструктурной части SOAP-сервиса, если знать, что делать, и пользоваться подходящей IDE — это минут 10 разработки и даже проще чем gRPC на Java.

Со стороны клиента первое, что надо сделать — это сгенерировать клиентскую заглушку. Для этого сервис должен быть запущен и WSDL-описание доступно для скачивания (на самом деле, это опционально, .wsdl-файл можно хоть по почте прислать). В Visual Studio достаточно нажать «Add service reference», указать URL и выбрать пару опций. Если Visual Studio нет (например, разработка идёт под Linux), есть консольные утилиты из .NET SDK, в частности, svcutil:

```
svcutil.exe /language:cs /out:generatedProxy.cs ^
    /config:app.config http://localhost:8000/ServiceModelSamples/service
```

Тут мы говорим, что хотим сгенерировать клиентский прокси в файл generatedProxy.cs на языке C# (никакой интриги), конфигурацию для него в app.config (это на самом деле XML-файл, про который чуть позже), а сам сервис находится по адресу `http://localhost:8000/ServiceModelSamples/service`. Дальше сгенерированную заглушку подключаем к проекту и пишем примерно такой код, чтобы ей воспользоваться:

```
static void Main(string[] args)
{
    var client = new CalculatorClient();

    double value1 = 100.00D;
    double value2 = 15.99D;
    double result = client.Add(value1, value2);
    Console.WriteLine($"Add({value1},{value2}) = {result}");

    client.Close();
}
```

И, собственно, всё. Кажется, что какая-то магия — в gRPC нужно ChannelBuilder, всякие страшные параметры передавать, а тут просто создали заглушку и вперёд. На самом деле, всё дело в app.config, который должен лежать в рабочей папке программы в момент её запуска, там-то всё и написано:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5,Profile=Client" />
    </startup>
    <system.serviceModel>
        <bindings>
            <wsHttpBinding>
                <binding name="WSHttpBinding_ICalculator" />
            </wsHttpBinding>
```

```
        </bindings>
        <client>
            <endpoint
                address="http://localhost:8000/ServiceModelSamples/Service/CalculatorService"
                binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_ICalculator"
                contract="ServiceReference1.ICalculator" name="WSHttpBinding_ICalculator">
            </endpoint>
        </client>
    </system.serviceModel>
</configuration>
```

Руками такое писать страшновато, но нам его сгенерил svcutil, так что если настройки по умолчанию нас устраивают, то можно на него и не смотреть даже. Из интересного тут тэг endpoint, где указаны address, binding и contract, так что если что-то из этого поменяется, надо будет поправить конфиг, не пересобирая приложение. Обратите внимание на хитрую схему с wsHttpBinding — параметр binding на самом деле ссылается на отдельной описание байндинга. Дело в том, что на самом деле байндинг может иметь много параметров (те же параметры безопасности, например) и несколько разных конфигураций, поэтому он вынесен в отдельный тэг.

Вот теперь точно всё, это полная клиентская часть.

3. Очереди сообщений

Мы пока что рассматривали ситуацию, когда клиент и сервер соединяются напрямую. Часто это может приводить к неприятным ошибкам просто из-за кратковременных сетевых отказов, а часто просто не нужно — мы хотим всего лишь отправить сообщение в надежде, что оно будет обработано когда-то в будущем (например, послать нотификацию по почте — сейчас или через полчаса, в случае с электронной почтой обычно разницы нет). В качестве прослойки между клиентом и сервером можно использовать очереди сообщений — системы, обеспечивающие гарантированную доставку сообщений, даже если отправитель и получатель в разное время находятся в онлайне. Обычно очередь сообщений — это отдельный сервер (который простой и не самодельный, поэтому, скорее всего, у него будет высокий аптайм), либо локальное хранилище сообщений у каждого отправителя. Отправитель не шлёт сообщение непосредственно серверу, а кладёт его в очередь и сервер, когда готов, забирает его оттуда (либо очередь сама время от времени пытается его доставить, в зависимости от архитектуры конкретной системы).

Очереди сообщений позволяют нескольким отправителям писать сообщения, и нескольким получателям, соответственно, читать, так что позволяют организовать более интересные способы взаимодействия, чем «точка-точка» (хотя в этом качестве тоже активно используются). НАпример, очередь сообщений может реализовывать модель «издатель-подписчик», когда один источник кидает в неё события, а несколько подписчиков вычитывают (при этом, в зависимости от задачи, одно сообщение может доставляться всем подписчикам, а может и только одному — так, например, можно балансировать нагрузку в распределённых вычислительных системах). Очередь также может выступать в роли шины событий для событийно-ориентированных архитектур — все, кто производят события,

пишут события в очередь (может, в несколько разных очередей, по одной на каждый тип события), все, кто хочет слушать события, просто забирают их из очереди (опять-таки, удаляя событие из очереди или нет).

Очереди при этом обычно имеют развитые возможности по маршрутизации, фильтрации и преобразованию сообщений, реализованные на стороне собственно очереди. Например, разветвители, которые раскидывают сообщения из одной очереди по нескольким, агрегаторы, которые, наоборот, сливают сообщения из нескольких очередей в одну, преобразователи порядка, которые могут переставлять или сортировать сообщения.

Реализаций очередей сообщений довольно много, достойна упоминания, например, MSMQ (Microsoft Message Queuing), очередь сообщений, поставляемая прямо с Windows. Так что если вы не имели идей, что такое очереди сообщений, у вас наверняка есть или когда-то был компьютер, где очередь сообщений была и даже работала. Кстати, MSMQ поддерживается в WCF как Binding.

3.1. RabbitMQ

Одна из самых популярных реализаций очередей сообщений — RabbitMQ[3], называемая на жаргоне «кролик». Это как раз реализация с отдельным сервером (написанным, внезапно, на Erlang, так что чтобы RabbitMQ заработала, ей нужен Erlang runtime). Клиент при отправке сообщения подключается по сети к серверу (никто не мешает задеплоить его на той же машине, что и клиент, впрочем) и сохраняет сообщение там. Сообщение хранится, возможно, будучи записанным на диск, чтобы пережить даже перезапуск сервера RabbitMQ. Когда получатель готов его обработать, он подключается к серверу, указывает из какой очереди забрать сообщение и удалить ли его после обработки, и забирает сообщение. RabbitMQ никакой кодогенерации не использует, сообщения для неё — просто массивы байтов (что, кстати, даёт возможность сериализовать сообщения в формат protobuf, например). Клиентские библиотеки доступны практически для любых языков программирования.

RabbitMQ реализует стандартный протокол AMQP (Advanced Message Queuing Protocol), так что теоретически может работать и со сторонними клиентами, и интегрироваться с другими очередями. Но не уверен, что кто-то пробовал. RabbitMQ имеет развитые возможности по маршрутизации сообщений — на самом деле, сообщение добавляется не в очередь вовсе, а в exchange — именованный и конфигурируемый маршрутизатор, который по умолчанию тут же пересылает сообщение в очередь с таким же именем и больше ничего не делает (так что пользоваться RabbitMQ можно, даже не зная про существование exchange-ей). Однако ему можно сказать раскидывать сообщения по разным очередям, маршрутизовать сообщения по ключам, отправляемым с сообщением клиентом, фильтровать сообщения и творить прочие хорошие вещи.

Но перейдём к примеру кода. Вот отправитель (на C#):

```
using System;
using RabbitMQ.Client;
using System.Text;

class Send
```

[3] Домашняя страница RabbitMQ, https://www.rabbitmq.com/ (дата обращения: 08.11.2021).

```
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using var connection = factory.CreateConnection();
        using var channel = connection.CreateModel();
        channel.QueueDeclare(queue: "hello", durable: false, exclusive: false,
                autoDelete: false, arguments: null);

        string message = "Hello World!";
        var body = Encoding.UTF8.GetBytes(message);

        channel.BasicPublish(exchange: "", routingKey: "hello",
                basicProperties: null, body: body);
    }
}
```

Создаём подключение, сообщая ConnectionFactory имя хоста, на котором запущен сервер. Порт указывать не надо, он по умолчанию 5672. Дальше создаём канал — канал представляет виртуальное соединение по протоколу AMPQ внутри физического подключения, которое описывается классом Connection. Теоретически один Connection может поддерживать несколько каналов. Дальше мы внутри канала объявляем очередь с именем «hello», говорим (durable: false), что она должна сохранять сообщения только в память, а не на диск (так менее надёжно, но быстрее), что очередь не надо удалять при закрытии соединения (exclusive: false), что очередь не надо удалять, когда читатель от неё отписывается (autoDelete: false), не передаём дополнительных аргументов. Если очередь с таким именем на сервере уже есть, QueueDeclare ничего не делает, однако если параметры очереди не совпадают, RabbitMQ вернёт ошибку. Дальше мы сериализуем наше сообщение и публикуем его в очередь (методом BasicPublish) с ключом «hello», по которому exchange поймёт, что его надо отправить в очередь «hello».

Теперь получатель:

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Text;

class Receive
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using var connection = factory.CreateConnection();
        using var channel = connection.CreateModel();
        channel.QueueDeclare(queue: "hello", durable: false, exclusive: false,
            autoDelete: false, arguments: null);
```

```
        var consumer = new EventingBasicConsumer(channel);
        consumer.Received += (model, ea) =>
        {
            var body = ea.Body;
            var message = Encoding.UTF8.GetString(body);
            Console.WriteLine(" [x] Received {0}", message);
        };
        channel.BasicConsume(queue: "hello", autoAck: true, consumer: consumer);
    }
}
```

Тут тоже, создаём подключение к локалхосту, объявляем очередь (на случай, если получатель будет запущен раньше отправителя, если нет, то QueueDeclare ничего не делает). А дальше мы создаём асинхронного получателя сообщений и запускаем блокирующий метод BasicConsume, указывая ему очередь, которую надо слушать, автоматически подтверждать доставку и удалять сообщение из очереди, ну и consumer, который будет вызываться при появлении в очереди сообщения. Собственно получатель — это лямбда-функция с двумя аргументами, подписанная на событие Received у consumer. model — это канал, ea — собственно полученное сообщение, у него мы берём тело (байтовый массив), десериализуем и печатаем на экран.

Опять-таки, это прямо готовые программы отправителя и получателя, да и сам RabbitMQ с параметрами по умолчанию запустить ничего не стоит (особенно из Docker-контейнера), так что завести это всё в базовом варианте — дело десяти минут. Впрочем, правильно всё настроить (например, авторизацию по сертификатам — вы же не хотите, чтобы кто угодно мог подписываться на сообщения или писать сообщения в очередь) может быть уже не так тривиально.

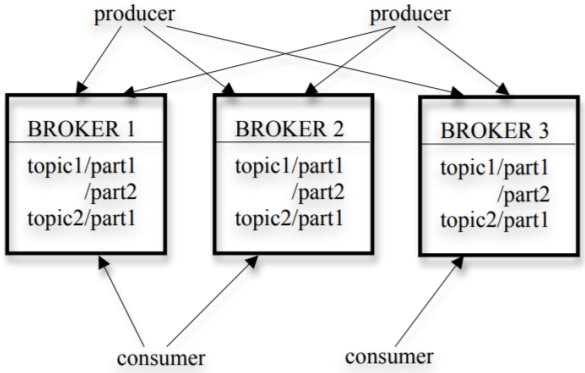
3.2. Apache Kafka

Apache Kafka занимает схожую с RabbitMQ экологическую нишу и вполне сравнима с ней по популярности, так что выбор между Kafka и RabbitMQ — вполне серьёзная задача (хотя часто определяется личными предпочтениями). Kafka использует несколько другой подход к работе с сообщениями, она на самом деле не очередь, а распределённый лог событий. Сообщение в Kafka посылается на сервер в целом так же, как в RabbitMQ, не не вычитывается получателем, а просто записывается и хранится до тех пор, пока политика чистки логов его не удалит (по умолчанию это происходит по таймеру, где-то раз в семь дней, но ничто в принципе не мешает хранить событий хоть вечно). Каждый подписчик по сути просто читает лог, получая сообщения последовательно (как будто бы из стрима), и сам хранит индекс последнего прочитанного сообщения (технически не совсем сам, потому что хочется, чтобы если получатель умрёт, он мог запуститься и продолжить — так что индекс хранится на сервере, но привязан к получателю). Так что несколько получателей могут вычитывать сообщения независимо, и каждый получатель может отмотать назад индекс и вычитать сообщения снова (чего нельзя в RabbitMQ и что бывает удобно, чтобы повторно проиграть последовательность событий — например, заново заполнить базу данными).

Делается так для того, чтобы избежать необходимости чрезмерного общения сервера

и клиента и достичь практической немутабельности хранимых данных. Что позволяет получить большую выгоду от распределённости — Kafka может быть запущена на кластере и хранить разные части лога на разных физических машинах, выдавая события читателям параллельно. Да и не только в распределённости дело, такой подход позволяет весьма эффективно хранить данные, поэтому даже один экземпляр Kafka работает быстрее, чем экземпляр RabbitMQ. Кроме того, такой подход хорошо ложится на событийные стили и подход «Event Sourcing» — когда система вообще не хранит состояние, а хранит просто последовательность событий, по которому состояние можно построить. Поэтому Kafka очень популярна там, где состояние не очень важно, событий много, и надо уметь быстро их агрегировать и что-то по ним считать — например, в разных приложениях «интернета вещей», где оконечные устройства могут сыпать сотнями тысяч событий в минуту. Однако Kafka немного сложнее в настройке и несколько хуже в маршрутизации сообщений, поэтому RabbitMQ популярнее в информационных системах, где событий не очень много, но надёжная доставка адресату (и только ему) критична. Впрочем, не стоит думать, что сценарии типа балансировщика нагрузки на Kafka не реализовать — в Kafka есть понятие «группа подписчиков», они координируются между собой так, чтобы каждое событие доставлялось только одному участнику группы.

Архитектурно Kafka устроена как набор *брокеров*, обслуживающих запросы издателей и подписчиков, и хранящих каждый свою часть лога:



Лог разделён на *топики* — каналы, куда можно писать и откуда можно читать отдельно от других. Топики разделены на *разделы* — логические куски топиков. Отправитель может послать событие в конкретный раздел, по умолчанию события распределяются равномерно. Кроме того, для события можно указать ключ, тогда раздел, в который оно попадёт, определяется по хешу ключа, так что все сообщения с одним ключом попадут в один раздел. Подписчики из одной группы получают события из разных разделов (см. `https://stackoverflow.com/questions/38024514/understanding-kafka-topics-and-partitions`, там хорошее объяснение с картинками). Топики и разделы хранятся на одном или нескольких брокерах, которые балансируют между собой нагрузку и собственно хранят на диске данные (для Kafka принципиально, что на диске — она даже кеширование в оперативной памяти не использует, надеясь на функциональность кеширования файловой системы — и товарищи в статье выше говорят, что это реально хорошее решение).

Как обычно, пример кода на C#, отправитель:

```
using Confluent.Kafka;

var config = new ProducerConfig { BootstrapServers = "localhost:9092" };

using var p = new ProducerBuilder<Null, string>(config).Build();
try
{
    var deliveryResult = await p.ProduceAsync(
        "test-topic", new Message<Null, string> { Value = "test" });
}
catch (ProduceException<Null, string> e)
{
    Console.WriteLine($"Delivery failed: {e.Error.Reason}");
}
```

Тут мы создали отправителя, который должен связаться с брокером на localhost:9092 (9092 — стандартный порт Kafka). Дальше мы паттерном «Строитель» конфигурируем строитель, который будет отправлять сообщения без ключа (Null) и со значением строкового типа. ProduceAsync асинхронно отправит сообщение на брокер (если не отправится, мы поймаем исключение и поругаемся).

Вот получатель:

```
using Confluent.Kafka;

var conf = new ConsumerConfig {
    GroupId = "test-consumer-group",
    BootstrapServers = "localhost:9092",
    AutoOffsetReset = AutoOffsetReset.Earliest
};

using var consumer = new ConsumerBuilder<Ignore, string>(conf).Build();
consumer.Subscribe("my-topic");

var cts = new CancellationTokenSource();
Console.CancelKeyPress += (_, e) => {
    e.Cancel = true;
    cts.Cancel();
};

try {
    while (true) {
        var consumeResult = consumer.Consume(cts.Token);
        Console.WriteLine($"Consumed message '{consumeResult.Message.Value}'.");
    }
}
catch (OperationCanceledException) {
```

```
    consumer.Close();
}
```

Получатель более долго настраивается — ему помимо адреса брокера надо передать идентификатор группы и политику сброса индекса сообщения (которая применяется, при первом запуске, или если на сервере логи подчистились и текущий индекс перестал быть валиден). Тут мы говорим, что хотим начать вычитывать сообщения начиная с первого в топике (если топик сброшен, старые сообщения удалились и мы в любом случае получим то, что ещё не видели). Можно попросить с последнего — тогда мы при включении не будем вынуждены обрабатывать историю, а сразу начнём получать свежие события.

Дальше опять с помощью паттерна «Строитель» создаём получателя, подписываемся на топик «my-topic», создаём токен отмены (это стандартный механизм .NET, позволяющий прерывать асинхронные операции) и запускаем бесконечный цикл потребления сообщений. пока нас не прервут по нажатию на Ctrl-C.

Как видим, в клиентском коде на самом деле даже проще, чем RabbitMQ. Зато если в RabbitMQ достаточно было просто запустить сервер (ну, для базовых сценариев), то в Kafka потребуется их аж два — сама Kafka и Zookeeper (сервис координации группы брокеров — вообще, они собираются от него избавиться, вроде, чтобы можно было просто Kafka запускать).