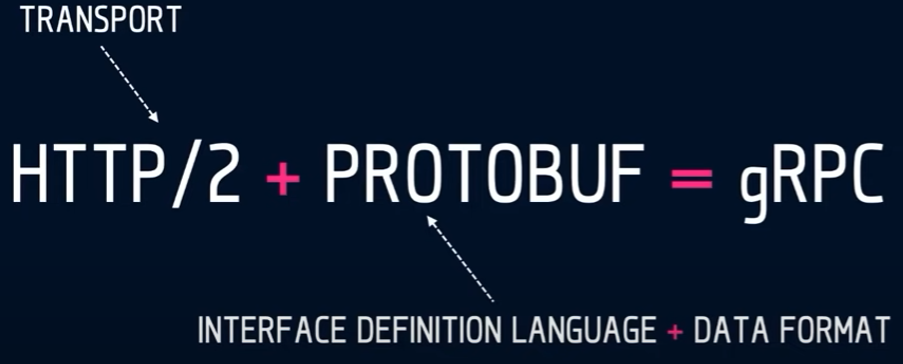
A **Remote Procedure Call** is a specification that allows for remote execution of a function in a different context. RPC extends the notion of local procedure calling but puts it in the context of an HTTP API.

Initial XML-RPC was problematic because ensuring data types of XML payloads is tough. So, later an RPC API started using a more concrete [JSON-RPC](https://www.jsonrpc.org/) specification which is considered a simpler alternative to SOAP. [gRPC](https://grpc.io/) is the latest RPC version developed by Google in 2015. With pluggable support for load balancing, tracing, health checking, and authentication, gRPC is well-suited for connecting microservices.

**Удаленный вызов процедур (RPC)** — это одна из простейших парадигм API, в которой клиент вызывает исполнение блока кода на сервере. В то время как REST рассматривает всё как ресурсы, RPC рассматривает действия. Клиенты обычно передают имя метода и аргументы серверу и получают обратно JSON или XML. RPC позволяет запросить список все возможных методов, используя wsdl, чего не сделать, если использовать тот же REST

 *По сути если заменить Protobuf на JSON и убрать поддержку в веб-сервисах гугла, то получается обычный JSON-RPC*

**PROTOBUF (Protocol Buffers)** – формат сериализации используемый по умолчанию для передачи данных между клиентом и сервером. Используя строгую типизацию полей и бинарный формат для передачи структурированных данных потребляет меньше ресурсов. (Например, JSON понятен человеку, но компьютер должен его распарсить в тот формат, который будет удобен для него. А вот бинарный формат для человека не понятен, зато понятен для компьютера и ему не нужно будет его как то парсить, из за этого экономится время обработки запроса). Время выполнения процесса сериализации/десериализации значительно меньше как и размер сообщений в отличии от JSON/XML.

Для написания protobuf файлов используют язык описания интерфейсов (IDL). Например, чтобы описать структуру данных сообщения, нужно добавить message, имя структуры, а внутри тип, название и номер поля. Номера полей очень важны для обратной совместимости, поэтому не стоит менять их последовательность при добавлении или удалении полей. Старые номера можно резервировать. Пример profile.proto

Файл формата **.proto**



После того как файл с нужной структурой данных создан, необходимо скомпилировать его компилятором для вашего языка программирования, чтобы сгенерировать класс доступа к этим данным. Этот класс будет содержать простейшие методы доступа ко всем полям типа get/set, а также методы для сериализации и десериализации вашей структуры данных в/из массива байтов.

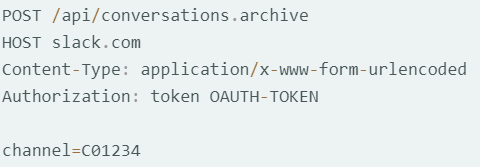
**Типы RPC**:

* **Унарный (Unary RPC)**. Синхронный запрос клиента, который блокируются пока не будет получен ответ от сервера.
* **Серверный стрим (Server streaming RPC)**, при подключении клиента сервер открывает стрим и начинает отправлять сообщения.
* **Клиентский стрим (Client streaming RPC)**. То же самое, что и серверный, только клиент начинает стримить сообщения на сервер.
* **Двунаправленный стрим (Bidirectional streaming)**. Клиент инициализирует соединение, создаются два стрима. Сервер может отправить изначальные данные при подключении или отвечать на каждый запрос клиента по типу “пинг-понга”.

**Правила RPC:**

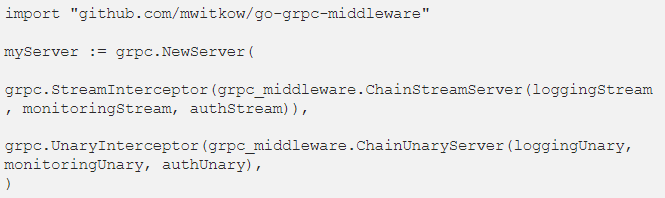
* Эндпоинты содержат имя выполняемой операции.
* Вызовы API выполняются с помощью наиболее подходящего HTTP-глагола: GET для запросов только для чтения и POST для других.

**Пример**:



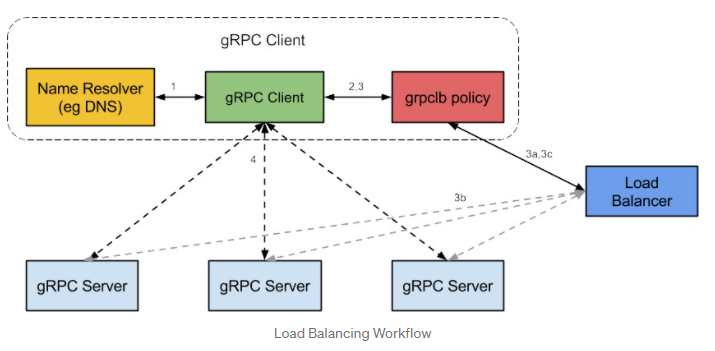
**Перехватчики (Interceptors)**

Выступают этакой заменой мидлварям, служат для перехвата вызовов и встраивания необходимых условий. Отлично подходят для логирования запросов, сбора метрик, аутентификации и т.д. Пример подключения логов, мониторинга и аутентификации для всех типов запросов используя пакет go-grpc-middleware:

****

**Балансировка нагрузки (Load Balancing)**

Выполняется на стороне клиента. Клиент использует простой “round-robin” алгоритм для передачи запросов по списку полученному от LB сервера. При желании на стороне LB сервера можно организовать более сложный алгоритм выдачи списка бэкенд сервисов клиенту использую LB политики.

****

**RPC/SOAP vs REST**



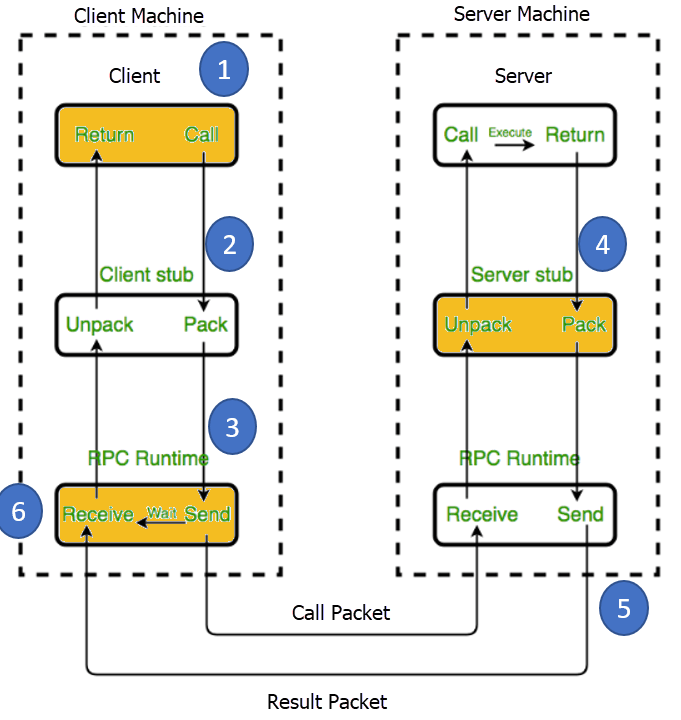
**SOAP и RPC** использует разные команды (SendUserMessage, GetUnits, …) что бы манипулировать сервисом. При этом используется HTTP методы POST(для всех остальных requests)/GET(для read-only requests) для RPC и POST для SOAP

REST сфокусирован больше на самих ресурсах (реальных данных, которые передаются в json формате). Для манипуляции сервисом используются разные HTTP методы – GET, POST, PUT, DELETE

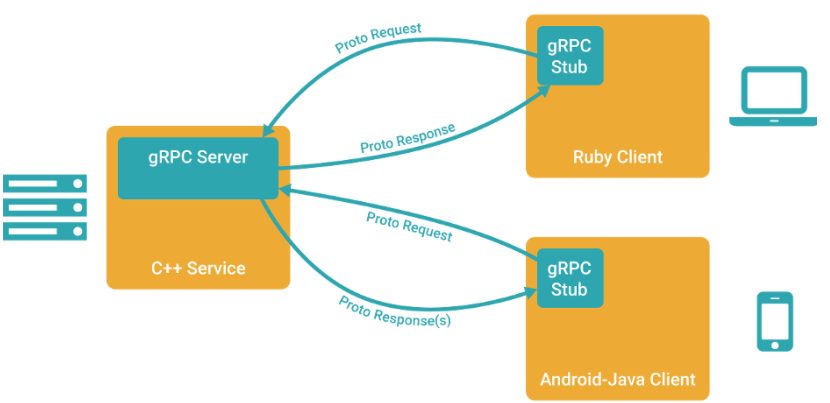
Особо популярный для взаимодействия между микросервисами, так как коммуникация между ними через HTTP сложно, к тому же они могут быть написаны на разых языках, и проще использовать в коде обычный вызов функции и так и передать данные. Не нужно писать документацию, так как из кода и так все понятно

How RPC works

A client invokes a remote procedure, serializes the parameters and additional information into a message, and sends the message to a server. On receiving the message, the server deserializes its content, executes the requested operation, and sends a result back to the client. The server stub and client stub take care of the serialization and deserialization of the parameters.



gRPC does communication via stubs. Both for communication between clients and our application and between microservices within our app



Концепция gRPC

Вообще, gRPC — это абстрактная спецификация. Она описывает абстрактную RPC (remote procedure call), то есть удаленный вызов процедуры, которая обладает определенными свойствами. Сейчас мы их перечислим. Первое свойство — поддержка как одиночных вызовов, так и стриминга. То есть все сервисы, которые реализовывают эту спеку, поддерживают оба варианта. Следующий пункт — наличие метаданных, то есть чтобы вместе с полезной нагрузкой вы могли бы передать какие-то метаданные — условно, заголовки. И — поддержка отмены запроса и таймаутов из коробки.  
  
Также она предполагает, что описание сообщений и самих сервисов осуществляется через некий Interface Definition Language или IDL. Также спецификация описывает wire-протокол поверх HTTP/2, то есть gRPC предполагает работу только поверх HTTP/2.

Реализация gRPC

Есть типичная реализация gRPC, которая используется в большинстве случаев. У нас она также используется, и сейчас мы ее посмотрим. В качестве IDL используется proto-формат. gRPC плагин для proto-компилятора позволяет из proto-описания получать исходники, сгенерированных сервисов. И существуют runtime-библиотеки на разных языках — Java, С++, Python. В общем, практически все популярные языки поддерживаются, runtime-библиотеки для них существуют. И в качестве сообщений, которыми обмениваются сервисы, используется proto-сообщение, стилизованные сообщения по схеме protobuf.

Фичи

**Строгая типизация**

Строгая типизация, то есть сообщение proto message, это строго типизированные сообщения. Те, кто с protobuf когда-то работал, знает, что там можно описывать поля в своем сообщении типами. Типы существуют как примитивные, так и строковые, массивы байт. Они могут быть скалярные, могут быть векторные. И, на самом деле, сообщения могут, в качестве поля, содержать другие сообщения, что достаточно удобно, в общем, можно представить любую модель.

**Обратная совместимость**

Про обратную совместимость. Хочется заметить, что proto IDL это формат, в который заложена обратная совместимость из коробки, то есть он задумывался с заделом на обратную совместимость, и Google выпустил версию proto3, которая по сравнению с proto2 еще больше улучшает обратную совместимость. Там, плюс, есть всякие спецификации, как и что можно менять так, чтобы обратную совместимость сохранять в каких-то нетривиальных кейсах.  
  
Есть возможность значений по умолчанию, можно добавлять новые поля и у потребителя ничего не требуется, собственно, изменять. Все поля в proto3 опциональные и их можно, допустим, удалять, и обращение к удаленному полю не вызывает ошибок на клиенте.

**Клиент и сервер из коробки**

Еще одна фича gRPC — клиент и сервер генерируются при помощи proto-компилятора и gRPC-плагина на основе proto-описания. Есть возможность в моменте, когда пишется код, выбрать какой клиент будет использоваться. То есть выбрать асинхронный или синхронный клиент, в зависимости от того, какого рода код вы пишите. Например, для реактивного кода асинхронный клиент очень подходит. И эта возможность есть для любого языка. То есть один раз вы пишите proto-описание, после этого вы можете генерировать клиент для любого языка, и не нужно их как-то отдельно разрабатывать. Распространять интерфейс для своего сервиса можете просто в виде proto-описания. Любой потребитель может сгенерировать себе клиент сам.

**Отмена запроса и дедлайны**

Про отмену запроса и дедлайны хотел бы отметить, что запрос можно отменить на сервере и на клиенте. Если мы понимаем, что все, запрос нам дальше выполнять не нужно, то мы его можем отменить. Есть возможность выставить таймаут на запрос. В gRPC в большинстве runtime-библиотек в качестве понятия таймаута используется дедлайн. Но по факту это то же самое. То есть это время, когда запрос должен завершиться.  
  
И что самое интересное, это то, что сервер может узнать, как об отмене запроса, так и об истечении таймаута и перестать выполнять запрос на своей стороне. Это очень круто, такого, мне кажется, нигде особо больше нет.  
  
Про документацию хотел заметить, что поскольку в IDL для gRPC используется proto-формат, это обычный код. Там можно писать комментарии, в том числе очень развернутые. И нужно понимать, что вашим пользователям для того, чтобы сделать интеграцию с вашим сервисом, нужно иметь этот proto-формат у себя, и он попадет к ним вместе с комментариями, они не будут лежать где-то в другом месте. Это очень удобно. И можно расширять это описание, то есть это такая удобная фича, что документация идет рядом с кодом, примерно так же, как она может лежать рядом с методами в виде javadoc или любых других комментариев.

### RPC Pros

**Straightforward and simple interaction.** RPC uses GET to fetch information and POST for everything else. The mechanics of the interaction between a server and a client come down to calling an endpoint and getting a response.

**Easy-to-add functions.** If we get a new requirement for our API, we can easily add another endpoint executing this requirement: 1) Write a new function and throw it behind an endpoint and 2) now a client can hit this endpoint and get the info meeting the set requirement.

**High performance.** Lightweight payloads go easy on the network providing high performance, which is important for shared servers and for parallel computations executing on networks of workstations. RPC is able to optimize the network layer and make it very efficient with sending tons of messages per day between different services.

### RPC Cons

**Tight coupling to the underlying system.**An API’s abstraction level contributes to its reusability. The tighter it is to the underlying system, the less reusable it will be for other systems. RPC’s tight coupling to the underlying system doesn’t allow for an abstraction layer between the functions in the system and the external API. This raises security issues as it’s quite easy to leak implementation details about the underlying system into the API. An RPC’s tight coupling makes scalability requirements and loosely coupled teams hard to achieve. So, the client either worries about any possible side effects of calling a particular endpoint or tries figuring out what endpoint to call because it doesn’t understand how the server is naming its functions.

**Low discoverability.** In RPC there’s no way to introspect the API or send a request and start understanding what function to call based on its requests.

**Function explosion.**It’s so easy to create new functions. So, instead of editing the existing ones, we create new ones ending up with a huge list of overlapping functions that are hard to understand.

### RPC use cases

The RPC pattern started being used around the 80s, but this doesn’t automatically make it obsolete. Big companies like Google, Facebook ([Apache Thrift](https://thrift.apache.org/)), and Twitch ([Twirp](https://twitchtv.github.io/twirp/docs/intro.html)) are using RPC high-performance variates internally to perform extremely high-performance, low-overhead messaging. Their massive microservices systems require internal communication to be clear while arranged in short messages.

**Command API.** An RPC is the proper choice for sending commands to a remote system. For instance, a Slack API is very command-focused: Join a channel, leave a channel, send a message. So, the designers of the Slack API modeled it in an RPC-like style making it small, tight, and easy to use.

**Customer-specific APIs for internal microservices**. Having direct integration between a single provider and consumer, we don’t want to spend a lot of time transmitting a lot of metadata over the wire, like a REST API does. With high message rate and message performance, gRPC and Twirp are strong cases for microservices. Using HTTP 2 under the hood, gRPC is able to optimize the network layer and make it very efficient with sending tons of messages per day between different services. However, if you’re not aiming at high network performance, but rather at a stable API contact between teams publishing highly distinctive microservices, REST will ensure that.

**Для API**, предоставляющего действия, которые сложно инкапсулировать в CRUD операциях.