Сервис-ориентированные архитектуры

Стили API веб-сервисов

RPC веб-сервисы

Глеб Игоревич Радченко

06.02.2021

Сервис-ориентированная архитектура (SOA)

- Сервис ориентированная архитектура – это парадигма организации и использования распределенных функциональных возможностей (сервисов), которые могут предоставляться различными владельцами
  - Определение организации OASIS (Organization for the Advancement of Structured Information Standards)
- По существу, SOA означает, что компоненты приложения представляют собой взаимодействующие сервисы, которые могут быть использованы независимо друг от друга, или же вообще могут быть отделены для реализации других приложений.

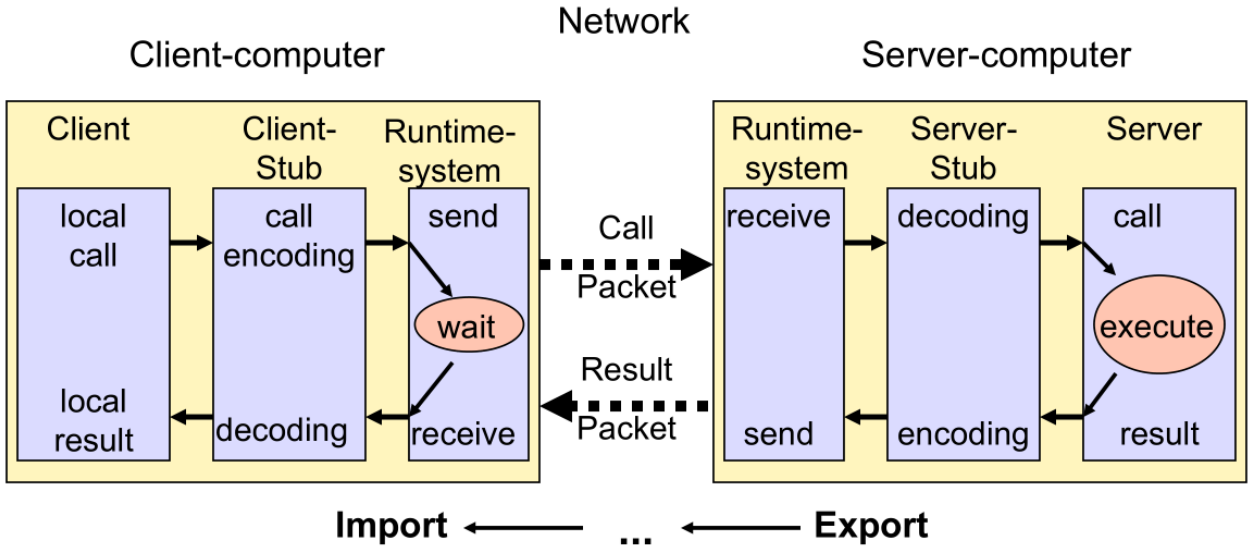
Стили API Веб-сервисов

Стиль Веб-сервиса	Решаемая проблема
1. RPC API	Как клиенты могут выполнить удаленные процедуры посредством HTTP?
2. API сообщений	Как клиенты могут отправлять сообщения (команды, нотификации или другую структурированную информацию) удаленной системе по HTTP, не привязывая себя к удаленным процедурам?
3. API ресурсов	Как клиент может манипулировать данными, предоставляемыми удаленной системой по HTTP, без связывания себя с удаленными процедурами, а также без необходимости создания специального проблемно-ориентированного API?
4. Графовый API	Как сформировать запрос к графу объектов, доступных на удаленном сервере, запросив конкретные поля, необходимые клиенту для обработки?

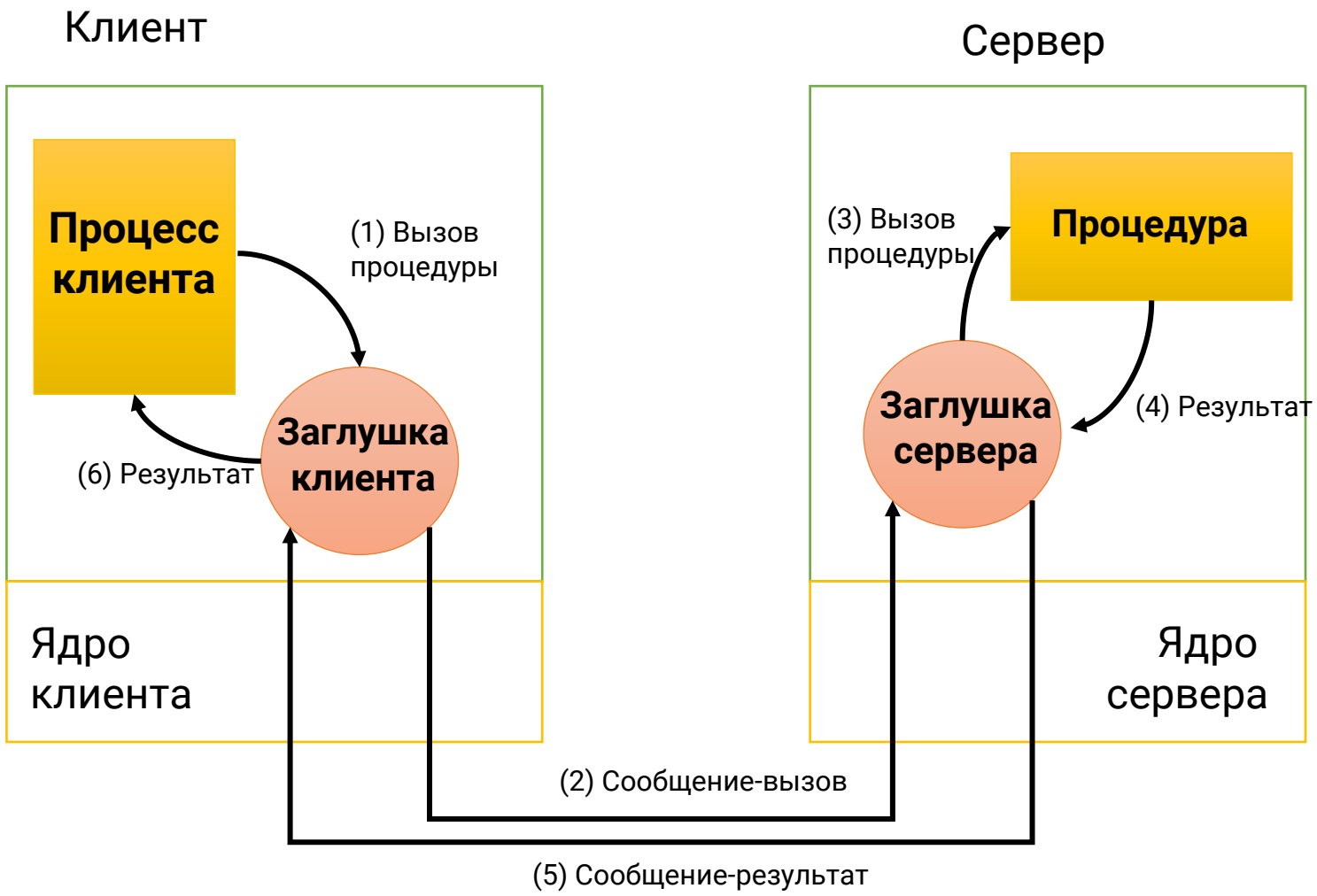
RPC API

RPC

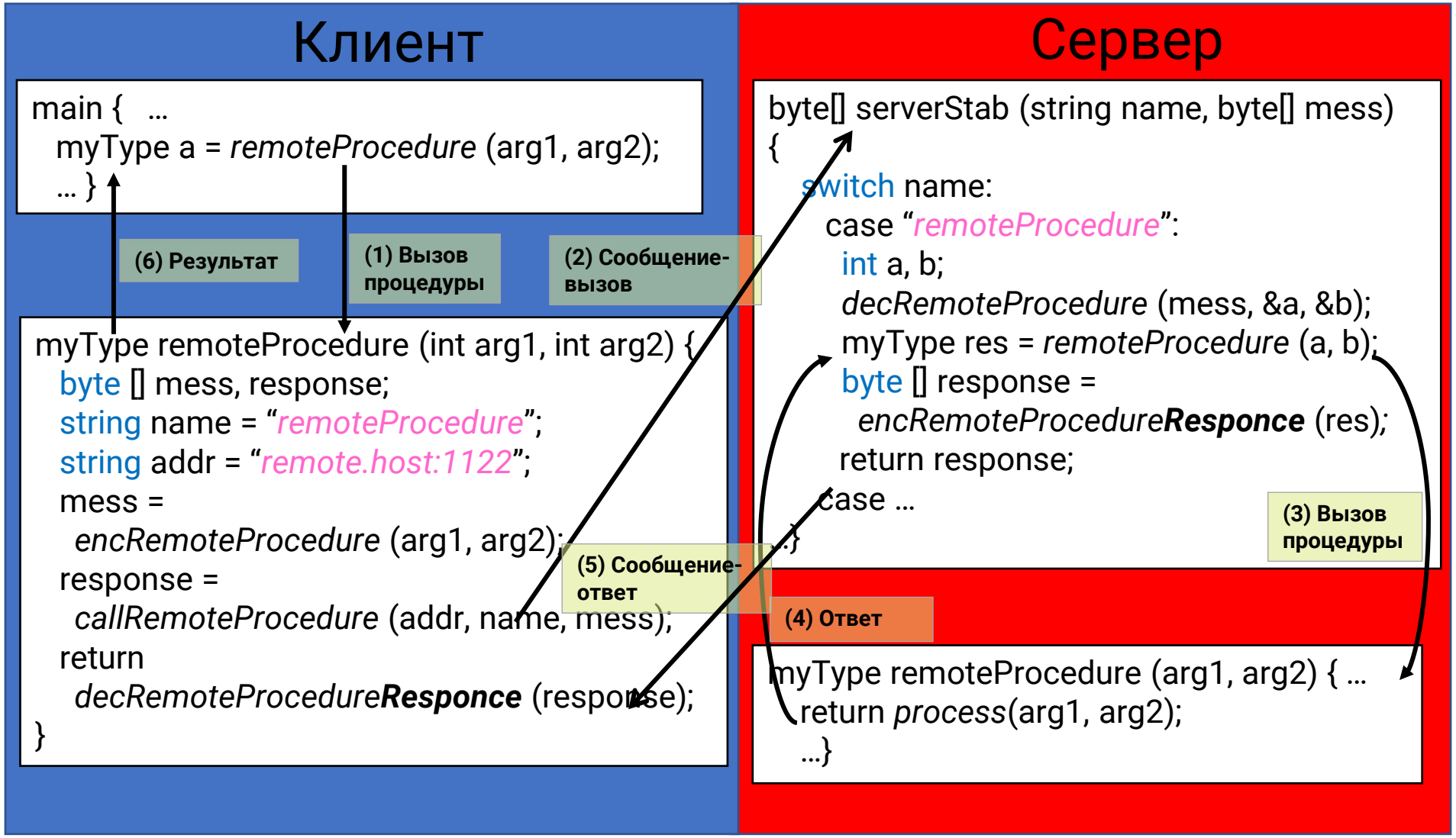
- Разработчики знакомы с концепцией вызова методов
- Хорошо спроектированные процедуры могут выполняться изолированно
- Нет причин, почему бы не выполнять процедуры на удаленной машине



RPC


Клиент
Сервер
Процесс клиента
(1) Вызов процедуры
(3) Вызов процедуры
Процедура
Заглушка клиента
(6) Результат
Заглушка сервера
(4) Результат
Ядро клиента
Ядро сервера
(2) Сообщение-вызов
(5) Сообщение-результат

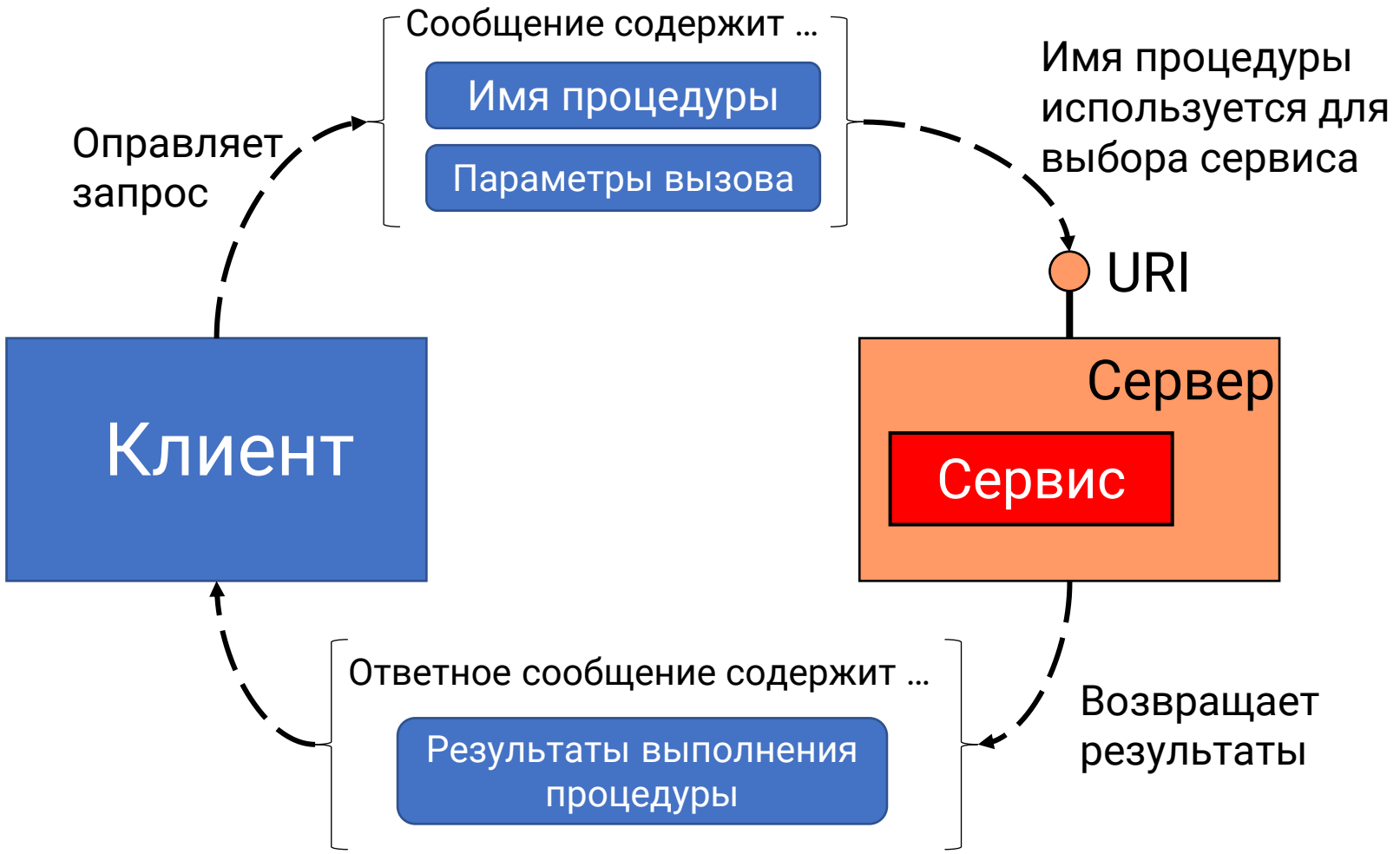
RPC псевдо-код



Стили API Веб-сервисов: RPC API

- Необходимо определить сообщения, которые бы описывали:
  - удаленные процедуры,
  - а также набор элементов, которые формируют набор параметров для удаленной процедуры.
- Клиент должен отправить сообщение с этой информацией по указанному URI для выполнения процедуры.

Стили API Веб-сервисов: RPC API



Обмен сообщениями

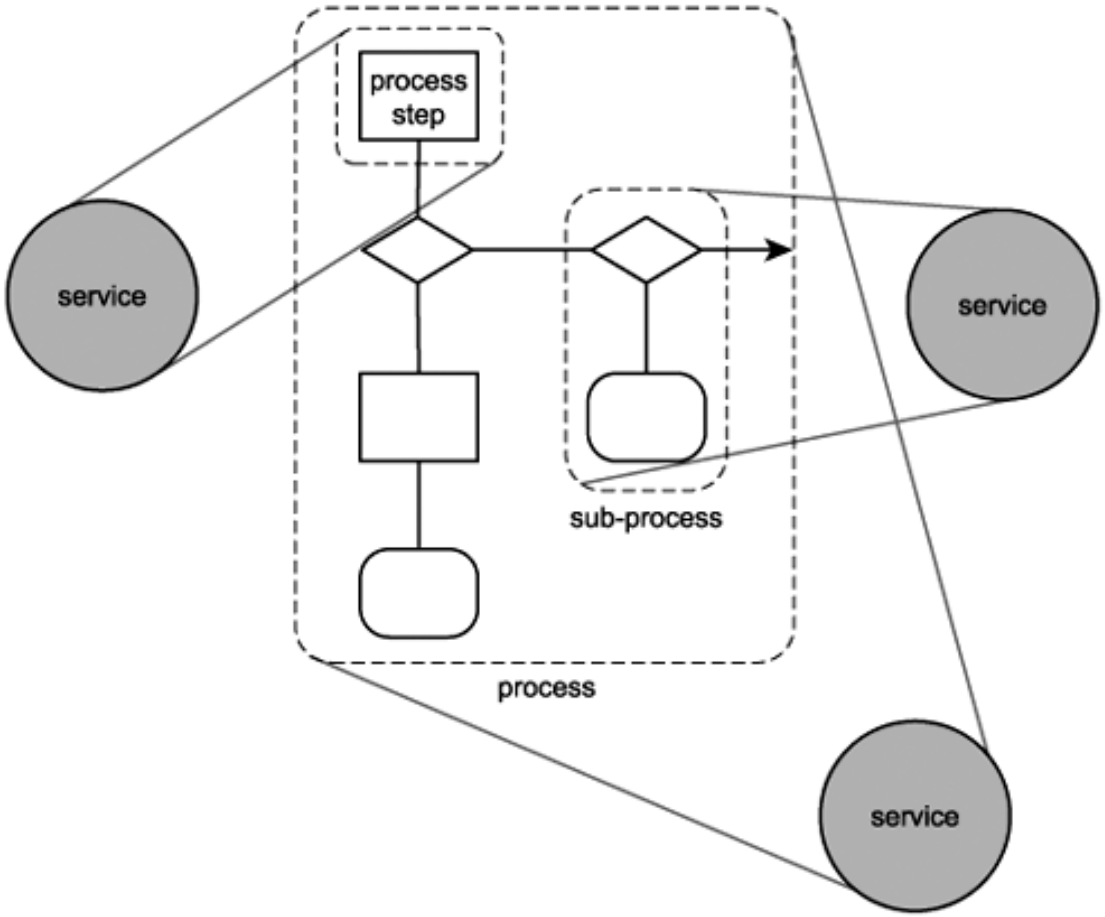
- Использование низкоуровневого транспортного протокола для обмена сообщениями (BSD Socket API) приводило к необходимости сложной трансформации данных, чтобы типы данных, определенные у клиента можно было использовать на сервере.
  - На разных платформах могут использоваться различные методы кодировки данных (ASCII , EBCDIC , UTF-8, UTF-16, Endianness) для представления и хранения данных.
- Технологии CORBA и DCOM решили предложили решения для данной проблемы, посредством собственных абстракций над низкоуровневыми типами данных.
- Но появились проблемы при связи между различными инфраструктурами (CORBA и .NET) + обмен сообщениями через Интернет

RPC платформы

- Применение протокола HTTP решает многие из этих проблем, т.к. обеспечивает возможность взаимодействия клиентов и серверов, базирующихся на различных платформах через Интернет.
- Для организации RPC посредством HTTP существует множество реализаций на разных языках:
  - JAX-WS (Java)
  - WCF (Microsoft .NET)
  - SOAP XML Web-Services (исторически – первый)
  - JSON RPC (простая спецификация)
  - gRPC (высокая эффективность, работа поверх HTTP/2)
  - …
- Они позволяют создавать простые веб-процедуры и клиентов к ним без необходимости понимания форматов данных (XML, JSON), методов кодирования (UTF-8 и др.) или описания структуры сервисов (WSDL).

Основные принципы RPC сервисов

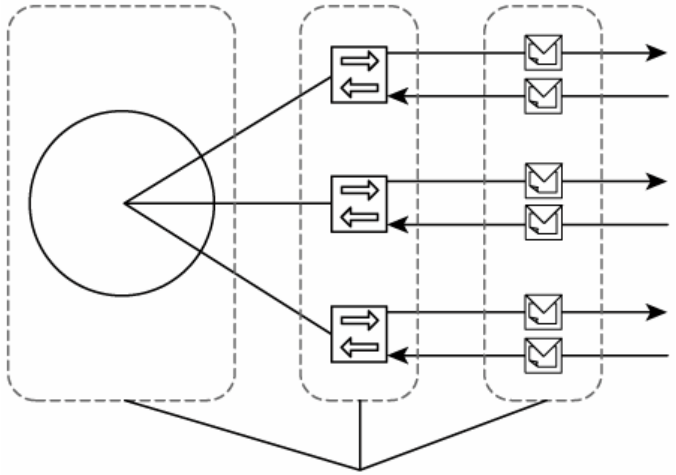
1. RPC-Сервисы инкапсулируют действие



2. Сервисы должны обеспечивать формальный контракт использования

Контракт сервиса предоставляет следующую информацию:

- Адрес конечной точки (service endpoint)
- Все операции, предоставляемые сервисом
- Все сообщения, поддерживаемые каждой операцией
- Правила и характеристики сервиса и его операций



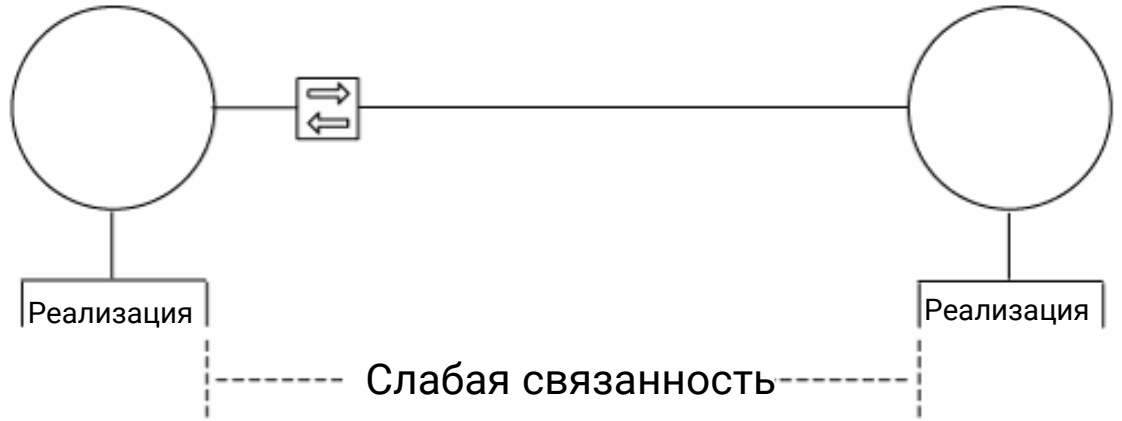
Определяется контрактом сервиса

3. Сервисы должны быть слабосвязаны (loosely coupled)

- В сервис-ориентированных системах выделяют целый ряд видов слабой связности сервисов :
  - организация связи через посредника
  - асинхронный стиль общения
  - модели данных основанные на простых типах
  - распределенное управление логикой процесса
  - динамическая привязка (потребителей и поставщиков сервисов)
  - независимость от платформы
  - независимость управления временем жизни каждого сервиса
  - и др.

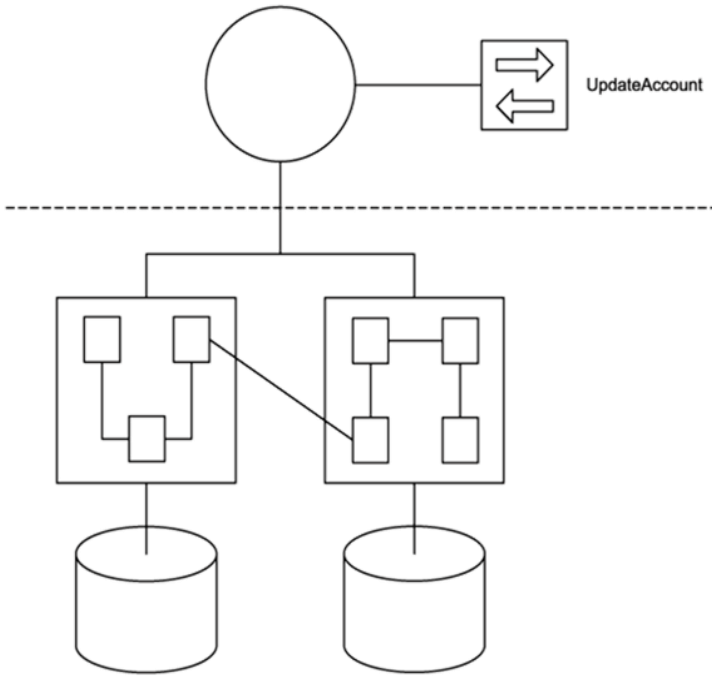
3. Зависимость от интерфейсов а не от реализации – ключ к слабосвязанности

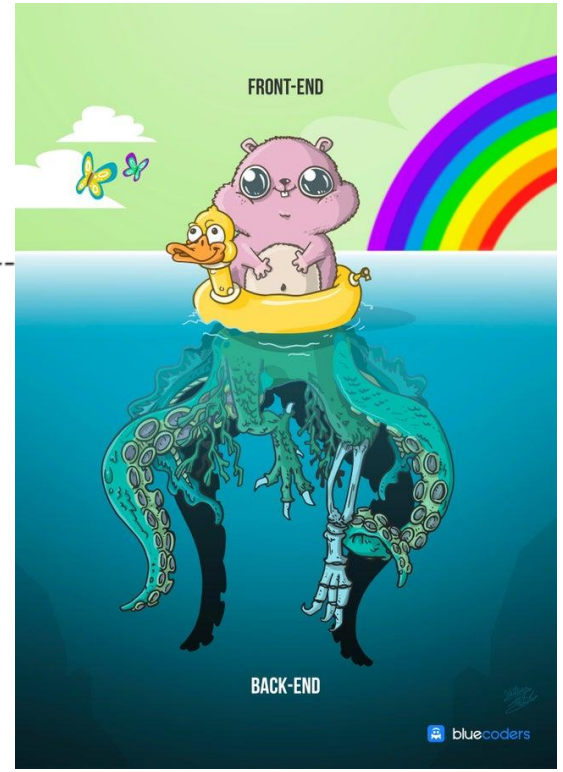
- Ключевой характеристикой реализации сервисов, которая делает остальные методы обеспечения слабой связности возможности является **слабая связность по реализации**
- Слабая связность по реализации сервисов в сервис-ориентированной системе обеспечивается, если сервис может приобретать знания о другом сервисе **исключительно на основе его открытого интерфейса**, оставаясь независимым от внутренней реализации данного сервиса



4. Сервисы должны абстрагировать внутреннюю логику

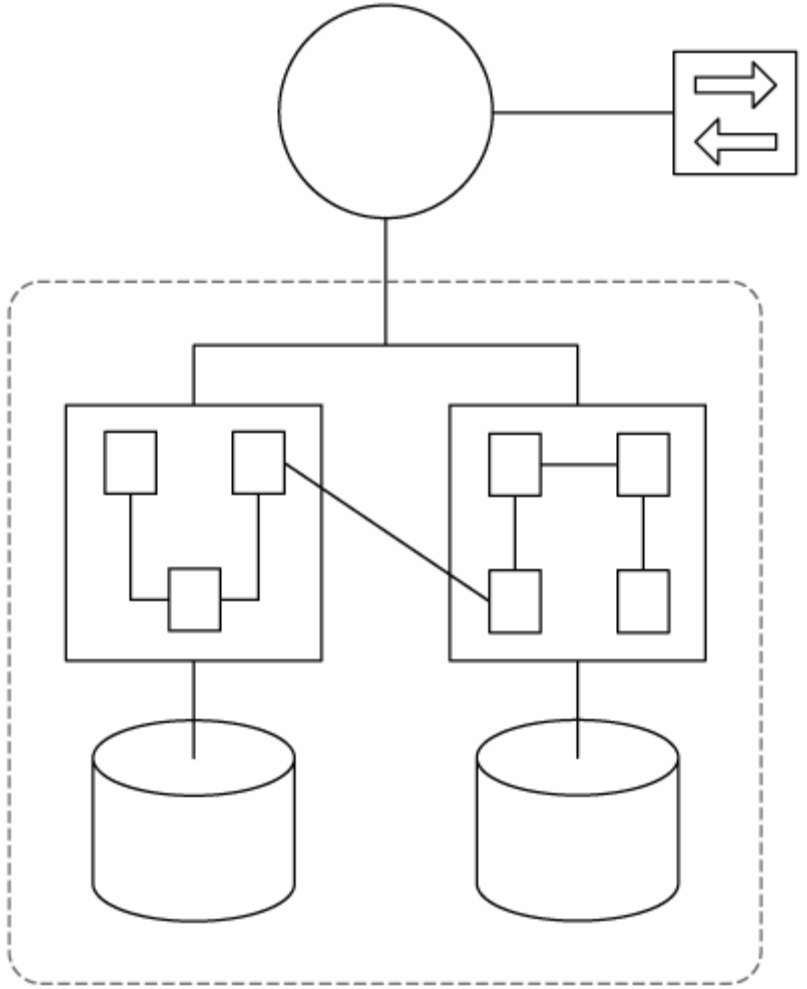
- Каждый сервис должен действовать как «черный ящик»
- Это одно из требований, обеспечивающих слабосвязанность сервисов.



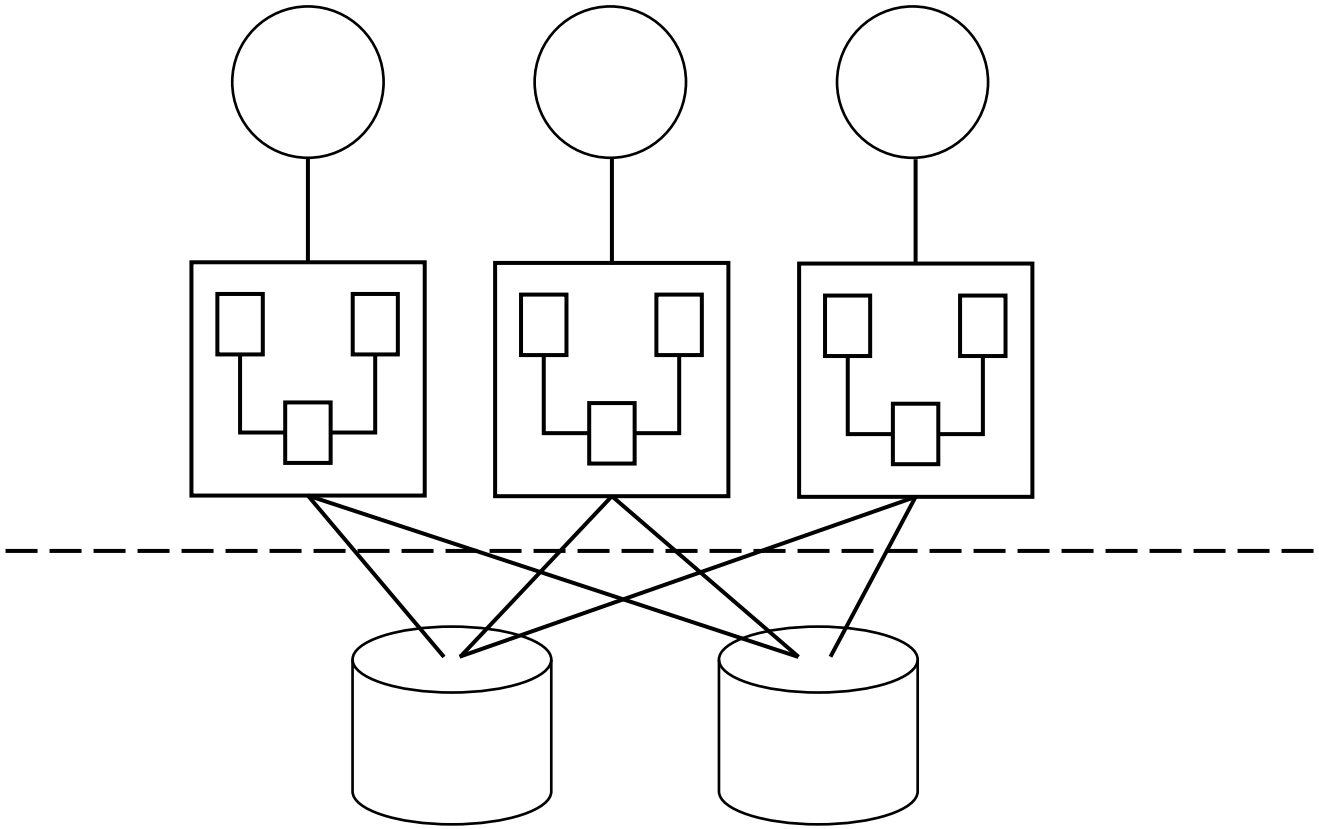


5. Сервисы должны быть автономны

- Область бизнес-логики и ресурсов, используемых сервисом должны быть ограничены явными пределами
- Вопрос автономности – наиболее важный аргумент при распределении бизнес-логики на отдельные сервисы
- Выделяют 2 типа автономности:
  - Автономность на уровне сервиса: границы ответственности сервисов отделены, но они могут использовать общие ресурсы
  - Чистая автономность: бизнес-логика и ресурсы находятся под полным контролем сервиса



5. Антипаттерн : монолитные микросервисы



6. Сервисы не должны использовать информацию о состоянии

- «Чистые» сервисы должны значительно ограничивать объем и время хранения информации (в идеале – только на время вычислений)
- Не зависимость от состояния (Statelessness) позволяет повысить возможности масштабируемости и повторного использования сервисов
- Но это очень жесткое ограничение, которое не всегда удается удовлетворить.
- Чаще всего говорят, что сервис является Stateless, если вся информация, необходимая для обработки запроса, находится в непосредственно в запросе от пользователя

JSON-RPC

JSON-RPC

- JSON-RPC (JavaScript Object Notation Remote Procedure Call — JSON-вызов удаленных процедур) — протокол удаленного вызова процедур, использующий JSON для кодирования сообщений.
- JSON-RPC работает отсылая запросы к серверу, реализующему протокол. Клиентом обычно является программа, которой нужно вызвать метод на удаленной системе. Все передаваемые данные — простые объекты, сериализованные в JSON

JSON-RPC – Запрос и ответ

- Запрос должен содержать три обязательных свойства:
  - **method** — Строка с именем вызываемого метода.
  - **params** — Массив объектов, которые должны быть переданы методу, как параметры.
  - **id** — Значение любого типа, которое используется для установки соответствия между запросом и ответом.

- Ответ должен содержать следующие свойства:
  - **result** — Данные, которые вернул метод. Если произошла ошибка во время выполнения метода, это свойство должно быть установлено в null.
  - **error** — Код ошибки, если произошла ошибка во время выполнения метода, иначе null.
  - **id** — То же значение, что и в запросе, к которому относится данный ответ.

JSON-RPC – пример

--> обозначает данные, отправленные серверу (запрос)

<-- обозначает ответ

```
...
--> {"method": "postMessage", "params": ["Hello all!"], "id": 99}
<-- {"result": 1, "error": null, "id": 99}
<-- {"method": "handleMessage", "params": ["user1", "Ok!"], "id": null}
<-- {"method": "handleMessage", "params": ["user3", "gotta go"], "id": null}
--> {"method": "postMessage", "params": ["I have a question:"], "id": 101}
<-- {"method": "userLeft", "params": ["user3"], "id": null}
<-- {"result": 1, "error": null, "id": 101}
...
```

Реализация JSON-RPC

Название	версия JSON-RPC	Описание	Язык(и), Платформы
JSON-RPC.NET	2.0	Быстрый JSON-RPC сервер. Поддерживает сокеты, именованные сокеты и HTTP с помощью ASP.NET требует Mono или .NET Framework 4.0.	.NET
Jayrock	1.0	Серверная реализация JSON-RPC 1.0 для версий 1.1 и 2.0 Microsoft .NET Framework.	.NET
jsonrpc-c	2.0	Реализация JSON-RPC через TCP сокеты (только сервер).	C
libjson-rpc-cpp	2.0	C++ JSON-RPC фреймворк, поддерживающий клиентскую и серверную стороны через HTTP.	C++
Phobos	2.0	Реализация для Qt/C++. Абстрагирует уровень передачи данных (готовые к использованию классы для TCP и HTTP).	C++
qjsonrpc	2.0	Реализация для Qt/C++. Поддерживает соединения между сообщениями и QObject слотами (как QDBus, qxtrpc) использует новые JSON классы, включенные в Qt5.	C++
JSON Toolkit	2.0	Реализация на Delphi	Delphi
go/net/rpc	?	Реализация JSON-RPC стандартной библиотеки Go	Go
jsonrpc4j	2.0	Java реализация JSON-RPC 2.0 поддерживает как сокеты, так и HTTP соединение.	Java
json-rpc	1.0	Базовая Java/JavaScript реализация, которая хорошо интегрируется в Android/Servlets/Standalone Java/JavaScript/App-Engine приложения.	Java / JavaScript
jpoxy	2.0	Простая Java реализация JSON-RPC созданная для упрощения реализации доступа к POJOs через сырой RPC фреймворк.	Java
JSON Service	2.0	JSON-RPC серверная реализациясс поддержкой Service Mapping Description. Хорошо интегрируется с Dojo Toolkit и Spring Framework.	Java
JSON-RPC 2.0	2.0	Легкая Java library для разбора и сериализации JSON-RPC 2.0 сообщений (open source). Несколько реализация на сайте. (Base, Client, Shell, ...)	Java
java-json-rpc	2.0	Реализация для J2EE серверов.	Java
lib-json-rpc	2.0	Реализация servlet, client, JavaScript	Java
simplejsonrpc	2.0	Простой JSON-RPC 2.0 Servlet, обслуживающий методы класса.	Java
gson-rmi	2.0	Легковесный, независимый от способа передачи RMI фреймворк разработанный для распределенных вычислений.	Java
jsonrpcjs	2.0	JavaScript клиентская библиотека для JSON-RPC 2.0, Не имеет зависимостей.	JavaScript
easyXDM	2.0	Библиотека для cross-domain соединений с поддержкой RPC. Поддерживает все браузеры postMessage, nix, frameElement, window.name, и FIM, очень проста в использовании.	JavaScript
Dojo Toolkit	1.0+	Предоставляет поддержку JSON-RPC	JavaScript
Pmrpc	2.0	JavaScript библиотека для использования в HTML5 браузерах. Реализация JSON-RPC, используя HTML5 postMessage API для передачи сообщений.	JavaScript
qooxdoo	2.0	Имеет JSON-RPC реализацию с опциональными бэк-эндами на Java, PHP, Perl и Python.	JavaScript, Java, PHP, PERL, & Python
JSON-RPC Реализация на JavaScript	2.0	Поддерживает JSON-RPC через HTTP и TCP/IP.	JavaScript
jabsorb	2.0	Легковесный Ajax/Web 2.0 JSON-RPC Java фреймворк, расширяющий JSON-RPC протокол дополнительной ORB функциональностью, такой как поддержка циклических зависимостей.	JavaScript, Java
The Wakanda platform	2.0	Поддерживает JSON-RPC 2.0 клиент внутри Ajax Framework и JSON-RPC 2.0 сервис в серверном JavaScript	JavaScript
Deimos	1.0+2.0	Серверная реализация для Node.js/JavaScript.	JavaScript
Barracuda Web Server	2.0	Barracuda Web Server's интегрированный	Lua
DeferredKit	1.0	Поддерживает JSON-RPC 1.0 клиент.	Objective-C
Demiurgic	2.0	JSON-RPC 2.0 клиент для Objective-C	Objective-C
Oxen iPhone Commons JSON components	1.0	JSON-RPC 1.0 клиент для Objective-C	Objective-C
objc-JSONRpc	2.0	Objective-c JSON RPC клиент. Поддерживает уведомления, простые вызовы и множественные вызовы.	Objective-C
JSON::RPC	2.0	Реализация сервера JSON RPC 2.0	Perl
json-rpc-perl6	2.0	Клиент и сервер.	Perl 6
php-json-rpc	2.0	Простая PHP реализация JSON-RPC 2.0 через HTTP клиента.	PHP
JQuery JSON-RPC Server	2.0	JSON-RPC сервер, специально сделанный для работы с Zend Framework JSON RPC Server.	PHP, JavaScript
jsonrpc2php	2.0	PHP5 JSON-RPC 2.0 базовый класс и пример сервера	PHP
tivoka	1.0 + 2.0	Универсальный клиент/серверная JSON-RPC библиотека для PHP 5+.	PHP
junior	2.0	Client/server библиотека для JSON-RPC 2.0	PHP
json-rpc-php	2.0	Client/server библиотека для JSON-RPC 2.0	PHP
JSONRpc2	2.0	Реализация с «dot magic» для PHP (= поддержка группировки методов и разделения точками)	PHP
GetResponse jsonRPCClient	2.0	Объектно-ориентированная реализация клиента	PHP
zoServices	2.0	PHP, Node.js и JavaScript реализация JSON-RPC 2.0	PHP, JavaScript, Node.js
json-rpc2php	2.0	Серверная и клиентская реализация для PHP. Содержит JavaScript клиент, использующий jQuery	PHP, JavaScript
jsonrpc-php	2.0	JSON-RPC реализация для PHP	PHP
php-json-rpc	2.0	Реализация JSON-RPC 2.0.	PHP
Django JSON-RPC 2.0	2.0	JSON-RPC сервер для Django	Python
Pyjamas		JSON-RPC клиентская реализация.	Python
Zope 3	1.1	JSON RPC реализация для Zope 3	Python
jsonrpclib	2.0	JSON-RPC клиентский модуль для Python.	Python
tornadorpc	2.0	Поддерживает JSON-RPC требует Tornado web server.	Python
tinyrpc	2.0	Поддерживает JSON-RPC через TCP, WSGI, ZeroMQ и др. Разделяет передачу данных от обработки сообщений, может работать без пересылки сообщений.	Python
jsonrpc	2.0	JSON-RPC 2.0 для Python + Twisted.	Python
bjsonrpc	1.0+	Реализация через TCP/IP (асинхронная, двунаправленная)	Python
Barrister RPC	2.0	JSON-RPC реализация клиента и сервера	Python, Ruby, JavaScript (Node.js + web browser), PHP, Java
pyramid_rpc	2.0	Гибкая JSON-RPC реализация интегрированная в Pyramid web application. Работает с Pyramid's системой авторизации.	Python
rjr	2.0	JSON-RPC через TCP/UDP, HTTP, WebSockets, AMQP, и прочие.	Ruby (EventMachine) сервер с Ruby и JavaScript клиентами.
jimson	2.0	Клиент и сервер для Ruby	Ruby
JSON-RPC Objects	1.0+	Реализация только объектов (без клиента и сервера).	Ruby
JSON-RPC RT	2.0	Полная поддержка JSON-RPC 2.0 через TCP.	Windows Runtime (WinRT)
XINS	2.0	С Версии 2.0, поддерживает JSON и JSON-RPC.	XML

OpenRPC – открытый стандарт для описания JSON-RPC API

- Стандарт OpenRPC Был представлен в 2019 году командой Ethereum Classic Labs Core для формирования единого подхода к описанию JSON-RPC 2.0 API
- Сайт проекта: https://open-rpc.org/
- Обеспечивает:
  - стандартный формат описания интерфейса JSON-RPC сервиса
  - поддерживает генерацию клиентской (для языков rust, typescript, go и python) и серверной части по описанию API
  - предоставление «песочницы», где можно посмотреть примеры и проверить работу собственного API: https://playground.open-rpc.org/

Структура OpenRPC документа

Название поля	Тип	Описание
openrpc	string	**Обязательно**. Строка должна содержать номер версии используемой спецификации OpenRPC, например: `"1.0.0-rc1"`
info	Info Object	**Обязательно**. Описание метаданных API: название приложения, его краткое описание, лицензия и др.
servers	[Server Object]	Массив объектов, содержащих описание серверов целевой системы. Каждый сервер определяется именем и URL.
methods	[Method Object | Reference Object]	**Обязательно**. Описание методов, предоставляемых API. Каждый метод определяется названием, описанием параметров вызова и описанием объекта-результата. Возможно добавление дополнительных полей с общим текстовым описанием, ссылками на спецификации а также примерами.
components	Components Object	Элемент для хранения различных схем для спецификации, включая используемые схемы данных и примеры их использования
externalDocs	External Documentation Object	Дополнительная внешняя документация

Пример OpenRPC описания: общая часть

```
{
  "openrpc": "1.0.0-rc1",
  "info": {
    "version": "1.0.0",
    "title": "Petstore",
    "license": {
      "name": "MIT"
    }
  },
  "servers": [
    {
      "url": "http://localhost:8080"
    }
  ],
```

Пример OpenRPC описания: описание методов

```
 "methods": [
    {
      "name": "list_pets",
      "summary": "List all pets",
      "tags": [
        {
          "name": "pets"
        }
      ],
      "params": [
        {
          "name": "limit",
          "description": "How many items to return at one time
(max 100)",
          "required": false,
          "schema": {
            "type": "integer",
            "minimum": 1
          }
        }
      ],
      "result": {
        "name": "pets",
        "description": "A paged array of pets",
        "schema": {
          "$ref": "#/components/schemas/Pets"
        }
      },
```

```
"errors": [
      {
        "code": 100,
        "message": "pets busy"
      }
    ],
    "examples": [
      {
        "name": "listPetExample",
        "description": "List pet example",
        "params": [
          {
            "name": "limit",
            "value": 1
          }
        ],
        "result": {
          "name": "listPetResultExample",
          "value": [
            {
              "id": 7,
              "name": "fluffy",
              "tag": "poodle"
            }
          ]
        }
      }
]
```

Пример OpenRPC описания: описание схем

```
"components": {
    "contentDescriptors": {
      "PetId": {
        "name": "petId",
        "required": true,
        "description": "The id of the pet to retrieve",
        "schema": {
          "$ref": "#/components/schemas/PetId"
        }
      }
    },
```

```
"schemas": {
  "PetId": {
    "type": "integer",
    "minimum": 0
  },
  "Pet": {
    "type": "object",
    "required": [
      "id",
      "name"
    ],
    "properties": {
      "id": {
        "$ref": "#/components/schemas/PetId"
      },
      "name": {
        "type": "string"
      },
      "tag": {
        "type": "string"
      }
    }
  },
  "Pets": {
    "type": "array",
    "items": {
      "$ref": "#/components/schemas/Pet"
    }
  }
}
```

gRPC

gRPC

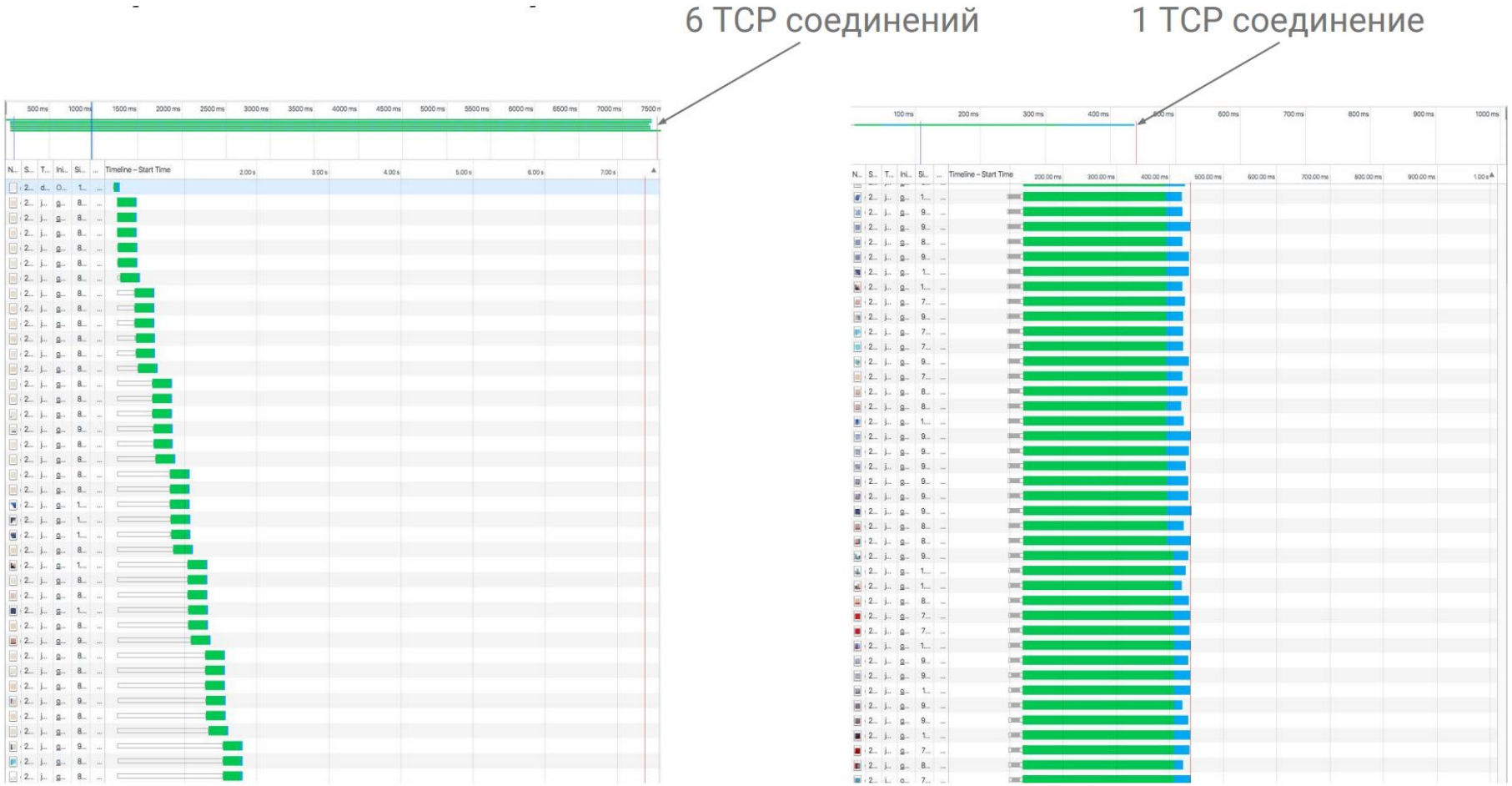
- gRPC - это высокопроизводительный фреймворк разработанный компанией Google для вызов удаленных процедур (RPC), работает поверх HTTP/2.
- Для обмена данными используется технология Google Protocol Buffers

- https://grpc.io/

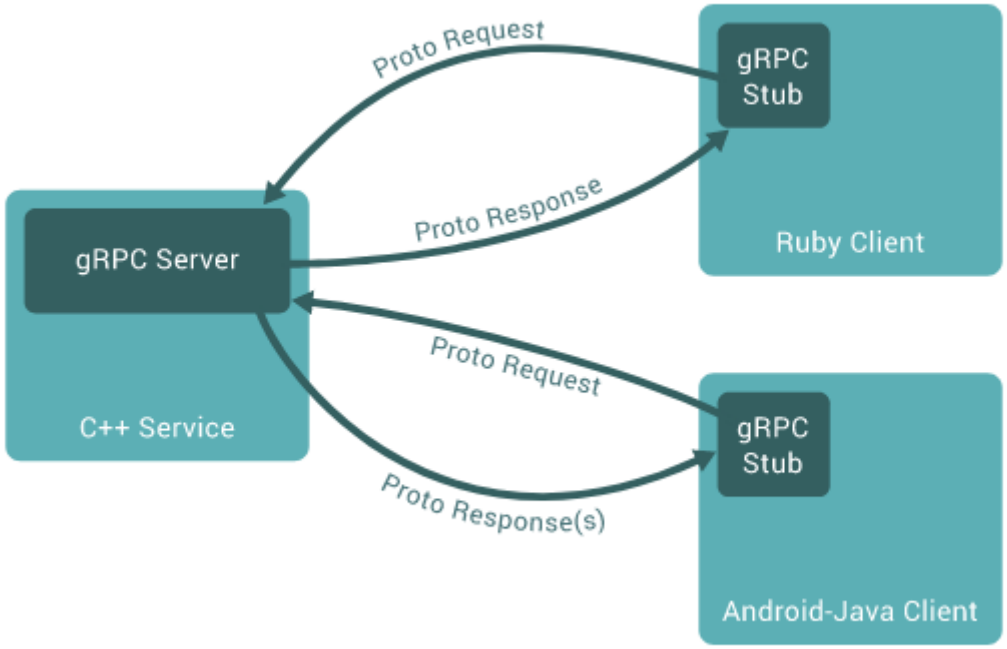
HTTP/2

- HTTP/2 – это вторая версия протокола HTTP, которая была утверждена в 2015 году.
- Является вариацией на тему протокола SPDY, который был предложен Google и реализовывался в инициативном порядке в ряде браузеров, основанных на Chromium с 2012 года.
- URI, методы (GET, PUT и др.), коды ошибок, заголовочная информация – остается в том же виде, но существенно меняется реализация обмена данными:
  - бинарный вместо текстового
  - мультиплексирование — передача нескольких асинхронных HTTP-запросов по одному TCP-соединению
  - сжатие заголовков методом HPACK
  - Server Push — несколько ответов на один запрос
  - приоритизация запросов
  - Хотя это и не требуется стандартном, но подавляющее число клиентов реализуются на базе “HTTP/2 over TLS”

HTTP/2



Концепция gRPC



Proto-файл

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "com.grpc.search";
option java_outer_classname = "SearchProto";
option objc_class_prefix = "GGL";
package search;

service Google {
  // Search returns a Search Engine result for the query.
  rpc Search(Request) returns (Result) {}
}
message Request {
  string query = 1;
}
message Result {
  string title = 1;
  string url = 2;
  string snippet = 3;
}
```

Генерация кода из protobuf

- Go:
- go get github.com/golang/protobuf/protoc-gen-go

```
> protoc ./search.proto --go_out=plugins=grpc:../golang/search
```

- Java
- protobuf-gradle-plugin/protobuf-maven-plugin

```
> gradle generateProto
> mvn protobuf:compile
```

Service Definition

`weather.proto`

gRPC Java runtime

Сервисная реализация

`WeatherServiceImplBase.java`

Запрос и ответ

`WeatherRequest.java`
`WeatherResponse.java`

Клиентские библиотеки

`WeatherServiceStub.java`
`WeatherServiceFutureStub.java`
`WeatherServiceBlockingStub.java`

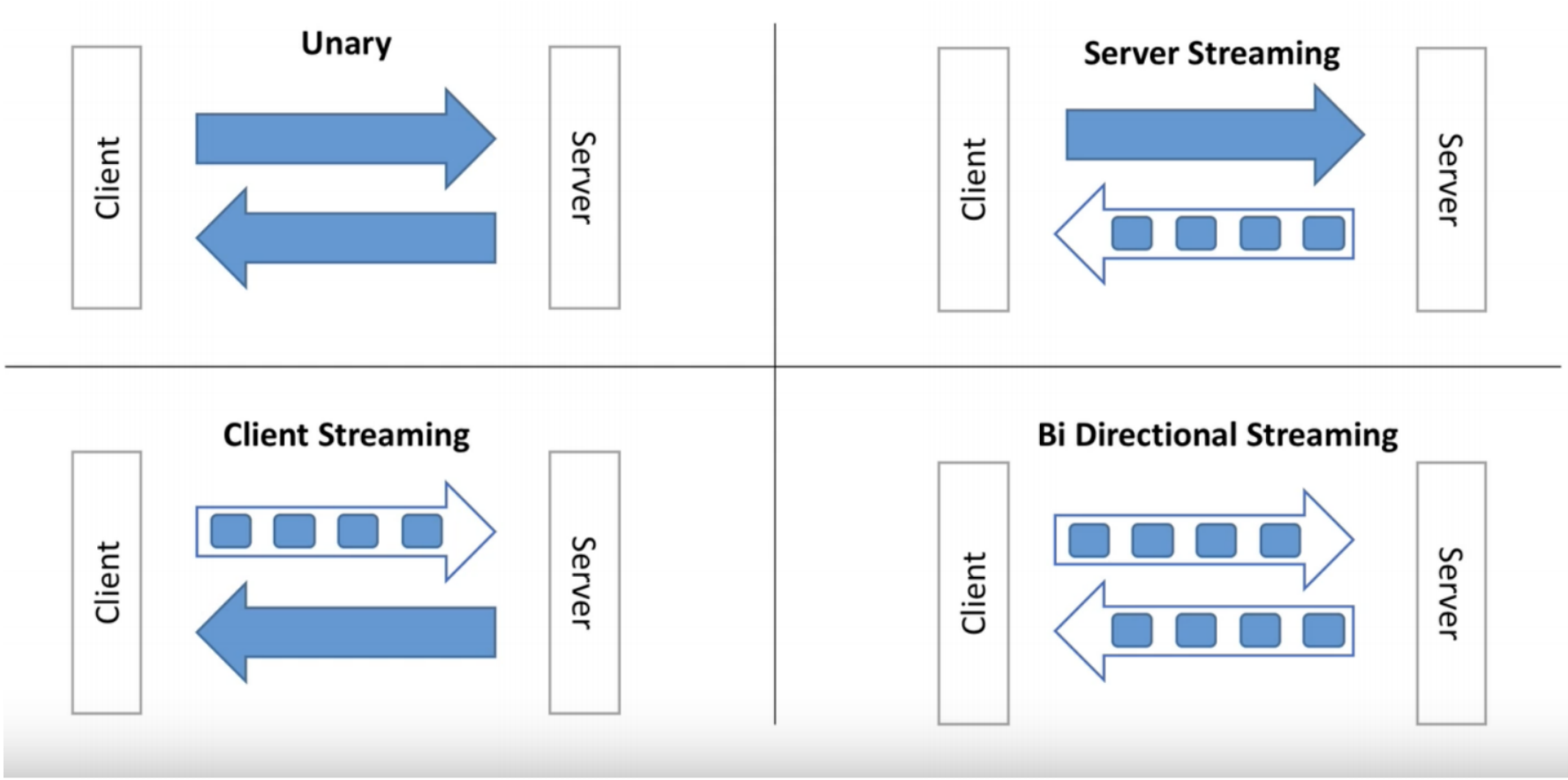
Сгенерированный код на Java

```
public final class Request extends
      com.google.protobuf.GeneratedMessageV3 implements
      // @@protoc_insertion_point(message_implements:search.Request)
       RequestOrBuilder {

   public static final int QUERY_FIELD_NUMBER = 1;
   private volatile java.lang.Object query_;
   /**
    * <code>string query = 1;</code>
    */
   public java.lang.String getQuery() {
       java.lang.Object ref = query_;
       if (ref instanceof java.lang.String) {
           return (java.lang.String) ref;
       } else {
           com.google.protobuf.ByteString bs =
               (com.google.protobuf.ByteString) ref;
           java.lang.String s = bs.toStringUtf8();
          query_ = s;
          return s;
       }
   }
```

1400 строк кода

Типы gRPC API



Service – описание службы для gRPC

- Простой RPC-запрос, где клиент отправляет запрос на сервер, используя заглушку, и ожидает ответа, как при обычном вызове функции.

```
rpc Search(Request) returns (Result) {}
```

- Потоковый RPC на стороне сервера: клиент отправляет запрос на сервер и получает поток для считывания последовательности сообщений обратно. Клиент читает из возвращенного потока, пока больше нет сообщений

```
rpc Search(Request) returns (stream Result) {}
```

- RPC потоковой передачи на стороне клиента: клиент записывает последовательность сообщений и отправляет их на сервер, используя предоставленный поток. Как только клиент закончил писать сообщения, он ждет, пока сервер прочитает их все и вернет свой ответ

```
rpc Search(stream Request) returns (Result) {}
```

- Двунаправленный потоковый RPC, где обе стороны отправляют последовательность сообщений, используя поток чтения-записи. Два потока работают независимо, поэтому клиенты и серверы могут читать и писать в любом порядке

```
rpc Search(stream Request) returns (stream Result) {}
```

Пример обмена данными в потоковом формате

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHello();
Console.WriteLine("Type a name then press enter.");

while (true)
{
        var text = Console.ReadLine();
        // Send and receive messages over the stream
        await call.RequestStream.WriteAsync(new HelloRequest { Name = text });
        await call.ResponseStream.MoveNext();

        Console.WriteLine($"Greeting: {call.ResponseStream.Current.Message}");
}
```

Клиент

```
public override async Task SayHello(IAsyncStreamReader<HelloRequest> requestStream,
IServerStreamWriter<HelloReply> responseStream, ServerCallContext context)
{
        await foreach (var request in requestStream.ReadAllAsync())
        {
                var helloReply = new HelloReply { Message = "Hello " + request.Name };
                await responseStream.WriteAsync(helloReply);
        }
}
```

Сервер

Performance best practices with gRPC. https://docs.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-5.0

Практика gRPC: потоковый режим обмена данными

- Не стоит злоупотреблять потоковым режимом обмена данными. Он хорош, если удовлетворяются следующие условия:
  - требуется высокая пропускная способность и низкие задержки
  - gRPC и HTTP/2 – это узкое горлышко в обмене данными
  - клиент отправляет и получает данные в виде gRPC сообщений
- Необходимо учитывать следующие особенности:
  - Поток может быть прерван сервисом или при ошибке связи. Необходимо перезапускать поток самостоятельно
  - RequestStream.WriteAsync не безопасен в многопоточном режиме. В поток можно записать только одно сообщение за раз
  - Клиент и сервер могут отправлять и получать только один определенный тип сообщений. Можно обойти, используя примитивы Protobuf, такие как Any или oneof, но их использование должно быть оправдано

Performance best practices with gRPC. https://docs.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-5.0

Практика gRPC: переиспользование каналов

- При работе с gRPC открытые каналы следует использовать повторно. Повторное использование канала позволяет мультиплексировать вызовы через существующее HTTP/2 соединение.
- Если для каждого вызова gRPC создавать новый канал, то количество времени, затрачиваемое на его прохождение, может значительно увеличиться. Каждый вызов потребует нескольких сетевых раундов между клиентом и сервером для создания нового HTTP/2 соединения:
  - Открытие сокета
  - Установление TCP-соединения
  - Ведение переговоров TLS
  - Запуск HTTP/2 соединения
  - Сделать вызов gRPC
- Каналы безопасны для совместного использования и повторного использования между вызовами gRPC:
  - gRPC клиенты являются легковесными объектами и не нуждаются в кэшировании или повторном использовании.
  - На базе одного канала можно создать несколько gRPC клиентов, включая различные типы клиентов. Клиенты, созданные на базе одного канала, могут безопасно использоваться несколькими потоками и совершать несколько одновременных вызовов.

Performance best practices with gRPC. https://docs.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-5.0

Практика gRPC: балансировка нагрузки

- Классические балансировщики работают на уровне транспортных протоколов (L4, TCP/UDP). Это позволяет распределять запросы HTTP/1.1 (т.к. каждый такой запрос идет в отдельном TCP-соединении), но не позволяет распределять запросы HTTP/2, которые передаются в одном TCP соединении.
- В этом случае, все gRPC запросы будут передаваться на одну конечную точку.
- Для балансировки можно применять 2 стратегии:
  - балансировка на уровне клиента
  - балансировщик (прокси) на уровне L7 (прикладной уровень)

Performance best practices with gRPC. https://docs.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-5.0

Практика gRPC: балансировка на уровне клиента

- При балансировке нагрузки со стороны клиента клиент знает о конечных точках.
- Для каждого вызова gRPC он выбирает отдельную конечную точку для отправки вызова.
- Балансировка нагрузки со стороны клиента является хорошим выбором, когда важна малая задержка. Между клиентом и сервисом нет прокси-сервера, поэтому вызов отправляется в сервис напрямую.
- Недостатком балансировки нагрузки на стороне клиента является то, что каждый клиент должен следить за доступными конечными точками, которые он должен использовать.

Performance best practices with gRPC. https://docs.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-5.0

Практика gRPC: балансировщик (прокси) на уровне L7

- Прокси L7 (прикладной) работает на более высоком уровне, чем L4 (транспортный).
- L7 прокси понимает HTTP/2, и может распределять вызовы gRPC, мультиплексированные к прокси по одному HTTP/2 соединению, по нескольким конечным точкам.
- Использование прокси проще, чем балансировка нагрузки со стороны клиента, но может добавить дополнительную задержку gRPC вызовам.
- Можно попробовать использовать один из следующих прокси:
  - Envoy – попоулярный прокси с открытым исходным кодом.
  - Linkerd – сервис-меш для платформы Kubernetes.
  - YARP: A Reverse Proxy - прокси с открытым исходным кодом на .NET.
- Если хотите подробнее про разницу L7 и L4 прокси – рекомендую эту статью: Igor Olemskoi. Введение в современную балансировку сетевой нагрузки и проксирование

Performance best practices with gRPC. https://docs.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-5.0

Спасибо за внимание!

Готов ответить на ваши вопросы!