

## ESTUDO PRÁTICO DO GRPC

E. A. R. Marques, E. N. Lago, H. Dezani (orientador) \*

e-mail:

[earmarques@gmail.com](mailto:earmarques@gmail.com); [estevan.lago@fatec.sp.gov.br](mailto:estevan.lago@fatec.sp.gov.br); [dezani@fatecristopreto.edu.br](mailto:dezani@fatecristopreto.edu.br).

**Resumo:** Com empresas de TI cada vez mais contratando sistemas de diferentes fornecedores e o crescimento robusto da internet das coisas, torna-se imperativo a integração de sistemas e dados, bem como uma comunicação eficiente de diferentes tipos de dispositivos consumindo serviços na nuvem. Buscamos neste trabalho compreender e comparar os estilos arquiteturais REST e gRPC para concepção de APIs, bem como a implementação de uma arquitetura básica de microsserviços em gRPC avaliando sua interoperabilidade.

**Palavras-chave:** API, REST, gRPC.

**Abstract:** *With IT companies increasingly hiring systems from different vendors and the robust growth of the internet of things, it becomes imperative to integrate systems and data as well as efficient communication of different types of devices consuming cloud services. In this work, we aim to understand and compare the REST and the gRPC architectural styles for API's designing, as well as the implementation of a basic microservices architecture in gRPC evaluating its interoperability.*

**Keywords:** API, REST, gRPC.

## 1. INTRODUÇÃO

Em aplicações monolíticas, todos os recursos e funcionalidades oferecidos estão em uma base de código unitária e indivisível. As dificuldades com este tipo de aplicação surgem quando o *software* precisa evoluir. Modificar e atualizar o código base a fim de incorporar novos serviços e funcionalidades à estrutura existente pode ser complexo e inviável, pois pequenas alterações podem gerar impactos negativos em outras áreas da aplicação.

Como contraponto a aplicação monolítica temos a arquitetura de micro serviços, em que a base de código unitária é fragmentada em componentes de serviços que realizam tarefas específicas, como se fossem uma aplicação autônoma. Os microsserviços encapsulam responsabilidades, são reutilizáveis, comunicam entre si usando protocolos como o HTTP e interagem de forma colaborativa. Com os microsserviços podemos desenvolver sistemas mais eficientes, escaláveis e flexíveis em menor tempo. Estes microsserviços operando em conjunto formam a arquitetura da aplicação.

Os microsserviços são disponibilizados por meio de APIs, usualmente através da rede. A arquitetura amplamente adotada no mercado atualmente é a API REST. Não obstante, há a API gRPC desenvolvida pela Google, uma estrutura mais moderna de conectar serviços que julgamos bastante promissora.

Este artigo aborda os fundamentos das duas arquiteturas, compara suas características destacando as vantagens e desvantagens de ambas, e avalia alguns cenários nos quais seria mais conveniente usar uma ou outra.

## 2. JUSTIFICATIVA

A explosão da internet mudou a realidade da comunicação, trazendo novas demandas que tiveram como resposta as aplicações distribuídas. Empresas tem contratado sistemas de diferentes fornecedores, com efeito, surge a necessidade de se fazer a integração desses sistemas de informação. Mais recentemente, acompanhamos o crescimento da internet das coisas (IoT), onde cada vez mais temos diferentes tipos de dispositivos acessando serviços na Nuvem.

Todos os dias estamos recuperando ou inserindo novos dados em alguma API na Internet. Faz-se necessário um modelo arquitetural simples e eficiente, que ajude a tornar os sistemas escalonáveis e flexíveis e as API tornam isso possível. Uma API modelada de forma correta é simples e intuitiva de ser utilizada.

O desenvolvimento de sistemas distribuídos baseados em API é um padrão amplamente adotado pelo mercado, que por ser bem conhecido fica mais simples de fazer as integrações. Prover e consumir API é independente de linguagem de programação. Uma API desenvolvida em Java pode ser consumida por uma aplicação desenvolvida em qualquer outra linguagem, como Python ou JavaScript. APIs trazem o requisito da interoperabilidade, pois podem interagir entre si além de serem consumidas por outros sistemas.

REST é o estilo arquitetural de desenvolvimento de API mais difundido do mercado e gRPC é uma arquitetura mais moderna, com características interessantes e promissoras, apadrinhada pela gigante Google, é apontada como a arquitetura de API do futuro.

Acreditamos que estudar e implementar estas arquiteturas contribuirá para a formação e auxiliará no crescimento profissional e individual para a busca de oportunidades de alta qualificação no mercado de trabalho.

### 3. OBJETIVO(S)

- Compreender o estilo arquitetural REST e a arquitetura gRPC para construção de APIs;
- Construção de uma arquitetura de microsserviços em gRPC para melhor entendimento da tecnologia;
- Comparar as vantagens e desvantagens das arquiteturas REST e gRPC.

### 4. FUNDAMENTAÇÃO TEÓRICA

Uma API (*Application Programming Interface*) é um software que possui um conjunto de serviços que intermediam o acesso às funcionalidades de um sistema operacional, de alguma aplicação ou mesmo de algum outro serviço. As APIs podem ser bibliotecas de

códigos como a *API Collections* da linguagem Java, onde temos classes que respondem a métodos públicos. Porém, as APIs que estamos interessados são aquelas disponibilizadas via rede ou via web, como a API do *Spotify*, onde podemos pesquisar por artistas ou pelos álbuns do catálogo de música. *Web API* ou *Web Services* é uma API que fornece sua interface de comunicação através da rede (CIRIACO, 2009).

Uma API (*Application Programming Interface*) é um software que possui um conjunto de serviços que intermediam o acesso às funcionalidades de um sistema operacional, de alguma aplicação ou mesmo de algum outro serviço. As APIs podem ser bibliotecas de códigos como a *API Collections* da linguagem Java, onde temos classes que respondem a métodos públicos. Porém, as APIs que estamos interessados são aquelas disponibilizadas via rede ou via web, como a API do *Spotify*, onde podemos pesquisar por artistas ou pelos álbuns do catálogo de música. *Web API* ou *Web Services* é uma API que fornece sua interface de comunicação através da rede (CIRIACO, 2009).

#### 4.1. RPC Legado

A Chamada de Procedimento Remota (*Remote Procedure Call*) RPC foi uma técnica de comunicação entre processos para construir aplicações distribuídas. Uma aplicação cliente invoca remotamente um método como se estivesse fazendo uma chamada de método local. Destacamos pela sua popularidade as implementações baseadas no paradigma da orientação a objeto do CORBA (*Common Object Request Broker Architectures*) (IBM - Brasil, 2012) e a Java RMI (*Remote Method Invocation*) (UFRJ. GTA/UFRJ, 2022), usadas na construção e conexão de serviços e aplicações. Contudo, tanto CORBA quanto Java RMI possuem especificações amplas e complexa (RICARTE, 2002), muito em função de ambos usarem protocolos construídos sobre TCP (JOHANN, 2011), o que dificulta o desenvolvimento da interoperabilidade.

Como reação a essa sofisticação limitante, empresas do porte da IBM e Microsoft desenvolveram a técnica SOAP (*Simple Object Access Protocol*), baseada no padrão da arquitetura orientada a serviço (IBM, 2021). SOAP define interface de serviço e operações sobre serviços (web services) descritos por WSDL (*Web Services Description Language*)

fazendo trocas de mensagens no formato XML (*eXtensible Markup Language*). <sup>2022-1</sup> Teve um crescimento rápido, mas novamente a complexidade dos formatos das mensagens, bem como a complexidade da especificação ao redor do SOAP inviabilizam a agilidade na construção de aplicações distribuídas. No contexto de desenvolvimento das aplicações distribuídas modernas, SOAP *web services* é considerada hoje mais uma tecnologia legada. A maioria das aplicações distribuídas dos dias atuais são desenvolvidas usando o estilo arquitetura REST.

Como reação a essa sofisticação limitante, empresas do porte da IBM e Microsoft desenvolveram a técnica SOAP (*Simple Object Access Protocol*), baseada no padrão da arquitetura orientada a serviço (IBM, 2021). SOAP define interface de serviço e operações sobre serviços (*web services*) descritos por WSDL (*Web Services Description Language*) fazendo trocas de mensagens no formato XML (*eXtensible Markup Language*). Teve um crescimento rápido, mas novamente a complexidade dos formatos das mensagens, bem como a complexidade da especificação ao redor do SOAP inviabilizam a agilidade na construção de aplicações distribuídas. No contexto de desenvolvimento das aplicações distribuídas modernas, SOAP *web services* é considerada hoje mais uma tecnologia legada. A maioria das aplicações distribuídas dos dias atuais são desenvolvidas usando o estilo arquitetura REST.

#### 4.2. REST

REST (*Representational State Transfer*) é um modelo arquitetural usado para desenvolver Web API (OLIVEIRA, 2015). Portanto, não é uma tecnologia, é uma especificação que define a forma de comunicação entre componentes de software na web. Pode-se desenvolver API REST em qualquer linguagem de programação que ofereça bibliotecas de conectividade à rede. REST foi desenhado para utilizar protocolos de comunicação já existentes, como o HTTP. Esta arquitetura foi proposta por Fielding (2000), na qual ele formaliza um conjunto de melhores práticas e regras (*constraints*) para desenvolvimento de Web API.

Há dois papéis bem definidos na arquitetura de uma API: cliente-servidor ou consumidor-provedor. Há o papel de cliente ou consumidor da API, que pode ser uma

aplicação *front-end*, *mobile*, *web* ou até mesmo outra API. A aplicação consumidora envia requisições para o servidor ou provedor da API que por sua vez responde às requisições.

Uma das grandes vantagens decorrente do trabalho do Fielding (2000) foi a separação entre cliente e servidor. Aplicações cliente e servidor devem poder evoluir separadamente, sem qualquer tipo de dependência entre elas, podendo ser substituídas sem interferir em nada na outra, desde que a interface entre elas permaneça inalterada. Dessa forma, temos uma maior flexibilidade e portabilidade, pois o sistema cliente pode evoluir independente do sistema provedor e vice-versa. Esses sistemas podem ser escritos e gerenciados por equipes ou até mesmo por empresas diferentes.

Outra regra ou *constraint* defendida por Fielding (2000) é que as APIs devem ter interface uniforme. Interface é uma espécie de contrato entre as partes, cliente e servidor, em que se estabelece um conjunto de operações bem definidas do sistema. Uma interface uniforme simplifica e desacopla a arquitetura, o que permite que ambos os lados evoluam de forma independente. As operações devem ser identificadas por URIs e deve-se usar um padrão de protocolo de comunicação para se interagir com a API. Usualmente é utilizado o protocolo HTTP e em decorrência, usamos os verbos ou métodos do protocolo para realizar diferentes operações, tais como GET, POST, PUT, DELETE e outros. No contrato também deve estar estabelecido que as respostas têm de ser padronizadas e devem incluir, inclusive, informações de como o cliente deve tratar a mensagem resposta.

Em função do desacoplamento proporcionado pela interface uniforme, REST prevê a possibilidade de sistemas em camadas, onde teríamos outros servidores entre o cliente e o servidor. Esses servidores intermediários podem oferecer uma camada de segurança, balanceamento de carga, *caching*, etc. Essas camadas não devem afetar nem a requisição do cliente nem a resposta do servidor, e mais, o cliente não deve sequer saber quantas camadas intermediária existem.

Uma *constraint* importante da API REST é a característica de *Stateless* (sem estado), isto é, a aplicação não deve possuir estado. A requisição feita ao servidor deve conter tudo que for necessário para que seja devidamente processada. O servidor não pode manter uma sessão ou algum tipo de histórico de uso entre uma requisição e outra. O servidor não pode conhecer o cliente, no sentido de que o servidor não pode armazenar informações da requisição anterior. Não estamos dizendo que não se pode fazer autenticação ou armazenar

informações do cliente, queremos dizer é que o servidor não armazena informações contextuais, tais como: quem é o cliente, se está logado ou não, qual foi a última operação que ele fez. 2022-1

A API pode fazer *caching* das respostas das requisições. A título de exemplo, podemos pensar em uma lista de cidades ou estados. Quantas vezes isso é alterado dentro de um dia, um mês ou até um ano? Portanto, bastaria fazer essa requisição apenas uma vez e a API poderia dizer ao cliente que a resposta pode ser armazenada, e então o cliente guarda os dados em um *cache* interno. Quando o cliente fizer a mesma requisição novamente, o *cache* entra em ação e nem se chega a consumir a rede. Fazendo uso da característica REST de sistema em camadas, este serviço de *caching* pode ser feito por um servidor intermediário entre cliente e servidor atuando como um *proxy*. Isso melhora a escalabilidade da aplicação, bem como a performance, uma vez que diminui o número de *hits* no servidor ou a quantidade de acessos. *Cache* no REST não é uma obrigatoriedade, trata-se apenas de uma possibilidade a ser utilizada quando for necessário.

Outra característica opcional do REST é o código sob demanda. Por não se aplicar na maioria dos casos essa característica é muito pouco utilizada. A ideia é o servidor retorna algum código para ser executado no cliente, por exemplo, o servidor retornaria, junto com os dados tabulados, o código JavaScript responsável por montar um gráfico.

Cabe aqui fazermos uma observação, por vezes encontramos os termos REST e RESTful. REST é estilo arquitetural que possui as *constraints*, ou seja, é a especificação, já RESTful é a API desenvolvida em conformidade com as *constraints*. Os desenvolvedores puristas acreditam que uma REST API deve seguir fielmente os princípios do REST, sem exceção, conforme foi estabelecido por Roy Fielding (2000). Já os desenvolvedores pragmáticos defendem uma abordagem mais prática; eles implementam as *constraints*, mas estão abertos a fazerem concessões nos casos em que, seguir a rigor os canônicos do REST geraria complicações em demasia, que não agregaria tanto valor assim. Os desenvolvedores pragmáticos preferem renunciar à API ser 100% RESTful para tornar o desenvolvimento ou o uso da API mais simples.

REST não restringe uso de protocolos de comunicação, mas para se colocar em prática é preciso usar um, e o mais comum é o protocolo HTTP (*Hypertext Transfer Protocol*).



Quando uma aplicação cliente faz uma requisição ao servidor de API, ela a faz enviando uma mensagem HTTP e o servidor, por sua vez, retorna uma mensagem de resposta ao cliente.

Cada mensagem de requisição enviada pelo cliente possui um método. O método do protocolo possui a semântica da operação a ser efetuada sobre um determinado recurso. É através do método que dizemos ao servidor qual tipo de ação queremos executar em certo recurso identificado pela URI. O protocolo HTTP possui vários métodos, por exemplo, o método GET é usado para recuperar dados, o POST para inserir novos dados, o DELETE para apagar, e o PUT costuma ser usado para modificar, mas por vezes encontramos implementações fazendo inserções.

As mensagens HTTP possuem o campo URI, que identifica o recurso que queremos dentro do servidor. Outro campo que compõe a mensagem é a versão (HTTP/versão) do protocolo. A maioria das APIs REST publicadas usam a versão 1.1, mas poderia ser alguma outra mais recente. A gRPC, que falaremos mais adiante, é concebida sobre HTTP/2.0. Atualmente a última versão do protocolo é a HTTP/3.0

Os cabeçalhos trazem informações no formato de chave e valor. O servidor ou o cliente as utilizam para interpretar a mensagem de requisição ou de resposta. Há vários pares chave-valor pré-definidos no protocolo HTTP, mas é possível criar nossos próprios cabeçalhos customizados. Por último, temos o corpo da mensagem (*payload*), que pode ser opcional, a depender do método utilizado. É no corpo que enviamos os dados do cliente para o servidor e é no corpo da mensagem que estará contido os dados da resposta.

A forma de um cliente interagir com uma API é através de seus recursos (*resources*). Recurso é qualquer coisa disposta na web: um documento, uma página, um vídeo ou uma imagem. É algo que tem importância suficiente para ser referenciada como uma coisa no *software*: um catálogo de produtos ou um único produto, um pagamento ou uma nota fiscal.

REST usa URI (*Uniform Resource Identifier*) para identificar recursos. URI é um conjunto de caracteres que tem por objetivo dar uma identificação para os recursos de forma não ambígua. URL (*Uniform Resource Locator*) é um tipo de identificação de recurso, ou seja, é uma URI, mas além de identificar, ela também fornece a localização onde o recurso está disponível. Em suma, um recurso para ser alcançado precisa de uma URI, ou mais especificamente, precisamos de uma URL para requisitá-lo usando protocolo HTTP.



Os recursos possuem representações, que nada mais são do que códigos que representam o estado atual dos recursos. Como algumas representações usadas, podemos citar o JSON e XML para dados, JPEG para imagens, etc. O mesmo recurso pode ser visualizado ou representado de diferentes formas. Quando o cliente da API faz uma requisição, ele pode especificar qual representação ele deseja, isto é, qual formato ele consegue processar ou interpretar. Esta informação do tipo do formato ou representação é adicionada no cabeçalho HTTP da requisição com o nome de chave *Accept*. O valor dessa chave do cabeçalho é chamado de *media-type*. Tem diversos *media-types* padronizados, como *application/json*, *application/xml*. Essa indicação de qual formato o recurso é retornado é chamada de *content negotiation*. Isto é, o consumidor está negociando com o servidor a representação que quer, e o servidor pode aceitar ou não.

Um conceito importante para API de modo geral é o da Idempotência. Uma operação idempotente significa que ela pode ser aplicada mais de uma vez sem que o resultado da primeira aplicação se altere. O método GET é idempotente, pois não importa quantas vezes se faça o GET sobre um recurso, o resultado da primeira operação não é afetado pelas demais. O método PUT, implementado para atualizar um valor, por exemplo, atualizar o preço de um produto para cinco reais. Aplicar a mesma atualização no preço para cinco reais não afeta o resultado da primeira atualização. É o mesmo caso de quando estamos editando um documento e clicamos no botão salvar seguidas vezes. A ação de salvar o documento da segunda até a enésima vez não altera em nada o resultado do primeiro salvamento, ou seja, o estado do recurso documento não é afetado pelas operações subsequentes, elas não geram efeito colateral na primeira.

Já o método POST que insere novos dados não é idempotente, uma vez que invocações sucessivas alteram o estado do recurso. Método seguro é aquele que não modifica recurso e método idempotente é aquele no qual invocações sucessivas não alteram o estado da aplicação. Logo, o método POST não é seguro nem idempotente. O GET é um método seguro e idempotente. Já o PUT é idempotente, mas não é seguro, pois altera o recurso.

### 4.3. gRPC

gRPC é uma estrutura de desenvolvimento (*framework*) moderna de chamada de procedimento remota (*Remote Procedure Call*) desenvolvida pela Google. Prometida como de alto desempenho, pode ser executada em qualquer ambiente (multiplataforma), multilinguagem de programação, e capaz de conectar serviços de forma eficiente com e entre *Data Centers*, com suporte ao balanceamento de carga, rastreamento, verificação de integridade e autenticação. Indicada para computação distribuída para conectar dispositivos, aplicativos móveis e navegadores a serviços de *back-end* API (GRPC AUTHORS, 2021).

Muito dos ganhos que o gRPC traz se deve ao fato de ter sido construído sobre HTTP/2 (M. BELSHE, 2015), o que trouxe substancial melhora na latência. Com o HTTP/1.x (R. FIELDING, 1999), se o cliente desejar fazer requisições em paralelo, a fim de melhorar o desempenho, era necessário criar múltiplas conexões TCP, pois no HTTP/1.x apenas uma resposta pode ser enviada por conexão. Já a versão HTTP/2.0 oferece multiplexação da conexão TCP e comunicação bidirecional, com várias requisições e respostas sendo feitas na mesma conexão TCP, o que elimina o bloqueio de cabeça de linha (*head-of-line blocking*). HTTP/2.0 ainda realiza enquadramento binário e compressão dos dados do cabeçalho (GRIGORIK, 2013).

O gRPC torna a criação de serviços e aplicações distribuídas facilitada, pois a aplicação cliente faz chamadas a métodos diretamente na aplicação servidora, como se estivesse lidando com objetos locais. Como em muitos sistemas RPC, gRPC baseia-se na ideia de definição de serviço e especificação de métodos que podem ser chamados remotamente. No lado servidor a interface é implementada e executamos o servidor gRPC para tratar as chamadas do cliente. No lado cliente, o cliente tem o *stub*, a representação do servidor no lado cliente e que provê os mesmos métodos do servidor (GRPC AUTHORS, 2021).

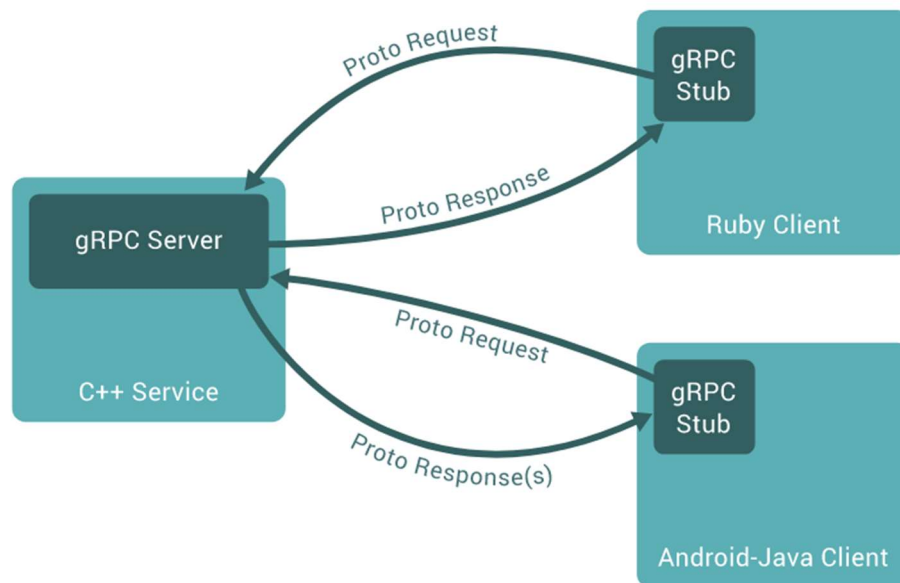


Figura 1 – Visão geral da comunicação gRPC

Fonte: Site desenvolvedores Google, guia, Introdução ao gRPC

gRPC foi projetado para ser naturalmente poliglota. Clientes e servidores gRPC podem executar e conversar uns com os outros em vários ambientes, servidores na *cloud* com clientes *desktop* ou *mobile*, podendo ser escritos em qualquer linguagem com suporte ao gRPC.

As chamadas gRPC podem ser de quatro tipos. *Unary RPC* é a forma do HTTP/1.1 trabalhar, onde o cliente envia uma única requisição e obtém uma única resposta. Na *Server Streaming RPC* o cliente faz uma única requisição, porém, o servidor envia um fluxo (*stream*) de mensagens em resposta a solicitação do cliente. No modo *Client Streaming RPC* é o cliente que envia várias mensagens e o servidor retorna apenas uma mensagem de resposta. Na *Bidirecional Streaming RPC* o cliente e o servidor podem processar operações de leitura e escrita de mensagens em qualquer ordem, uma vez que os *streams* são independentes (CORE CONCEPTS... 2021).

As chamadas gRPC podem ser de quatro tipos. *Unary RPC* é a forma do HTTP/1.1 trabalhar, onde o cliente envia uma única requisição e obtém uma única resposta. Na *Server Streaming RPC* o cliente faz uma única requisição, porém, o servidor envia um fluxo (*stream*)

de mensagens em resposta a solicitação do cliente. No modo *Client Streaming RPC* é o cliente que envia várias mensagens e o servidor retorna apenas uma mensagem de resposta. Na *Bidirecional Streaming RPC* o cliente e o servidor podem processar operações de leitura e escrita de mensagens em qualquer ordem, uma vez que os *streams* são independentes (CORE CONCEPTS... 2021).

gRPC permite que se especifique por quanto tempo se pode esperar por uma RPC. Ambos, cliente e servidor, podem determinar de forma independente se a chamada foi concluída, com sucesso ou não, e ambos os lados podem cancelar a chamada RPC em curso a qualquer momento (CORE CONCEPTS... 2021).

As definições de contrato de mensagens e serviço no gRPC são feitas em arquivos com extensões *.proto*. Trata-se de um tipo de Linguagem de Definição de Interface (IDL) (GOOGLE DEVELOPERS, 2021). As mensagens são serializadas usando o *Protobuf*, um formato de mensagem binário eficiente, que resulta em cargas de mensagens pequenas. A partir da definição do arquivo *.proto*, o *framework* gRPC gera código em uma linguagem de programação específica, por meio dos compiladores disponibilizados através de *plugins*.

Cliente e servidor podem estar implementados em linguagens diferentes e rodando em ambientes ou plataformas diferentes. Por exemplo, o compilador do lado cliente irá serializar o objeto Java em uma mensagem *proto request*, que por sua vez, será coletada da rede pelo servidor, e o código gerado pelo seu *plugin* compilador fará a reconstrução do objeto para a linguagem do compilador, por exemplo, C++, Python, JavaScript, Dart ou outras. Processada a requisição, o servidor converte o objeto C++ de sua mensagem de resposta em um *proto response* e retorna ao cliente. O que torna possível a tradução de objetos em diversas linguagens é que ambos implementam a mesma IDL, a mesma definição de interface de serviço, a saber, o arquivo *.proto*, compreendido por todos os *plugins*-compiladores.

#### 4.4. Golang

A concepção da linguagem Golang ou Go, aconteceu em setembro de 2007 através de Robert Griesemer, Ken Thompson e Rob Pike para solucionar desafios de engenharia enfrentados diariamente na Google.

A linguagem foi lançada em novembro de 2009 e atualmente é considerado um sucesso, sendo muito utilizada dentro e fora da Google, com efeitos notáveis em abordagens para concorrência de rede e engenharia de software através de outras linguagens e suas ferramentas.

Go é fortemente usada no coração das soluções de dados da Google para indexação de páginas web ao redor do mundo, fornecendo suporte ao Google Search e mantendo os dados de pesquisa atualizados e abrangentes.

Go também é aplicado em outras grandes tecnologias da Google, como em serviços de otimização de conteúdo do Google Chrome, serviços de hospedagem do Firebase e no seu setor de produção, auxiliando equipes de engenharia a atingir um alto padrão de confiabilidade.

#### 4.5. Dart

Dart é uma linguagem desenvolvida pela Google, lançada na GOTO Conference 2011, voltada ao ágil desenvolvimento de aplicativos, buscando facilitar a experiência do cliente e o aumento de produtividade. Com seu chamado *hot reload* torna-se muito mais fáceis mudanças no código, correções de *bugs*, etc. Além de fornecer suporte a diversas plataformas, como web, dispositivos móveis e *desktop*. O Dart também é a base do Flutter, um *framework* para desenvolvimento de aplicativos multiplataforma.

É uma linguagem com segurança de tipagem, com verificação de tipo estática. A tipagem é obrigatória, porém a anotação de tipos é opcional, pois tem inferência de tipos. Oferece também tipos dinâmicos, que auxiliam na checagem durante execução, muito útil para realização de experimentos.

Outra característica do Dart é o *null safety*, o que significa que valores nulos não serão aceitos, a não ser que o desenvolvedor especifique para que sejam.

#### 4.6. Protocol Buffers

“Os Protocol Buffers fornecem um mecanismo extensível de plataforma neutra e de linguagem neutra para serializar dados estruturados de maneira compatível com versões anteriores e posteriores” (GOOGLE, 2022). Protocol Buffers é mais rápido e mais leve do que JSON.

Sua capacidade é de alguns megabytes, fornecendo a serialização de pacotes de informações para linguagem de baixo nível, sendo amigável ao tráfego e para armazenamento de dados a longo prazo. Possibilitam também a junção de novos dados ao pacote sem a invalidação de dados já existentes. Podem ser usados tanto na comunicação entre servidores como também no armazenamento em disco.

Vantagens que podem ser citadas dos Protocol Buffers são a compactação dos dados, a rápida leitura e análise, poder ser entendido por diversas linguagens de programação e ainda fazer a geração de códigos por meio dos *plugins*.

Uma das vantagens mais notável dos Protocol Buffers é o fato de possibilitar com certa facilidade a comunicação, entre si, de diversas linguagens de programação. Atualmente, de acordo com a Google, há suporte para oito linguagens de programação diferentes, além das linguagens Dart e Go que são de domínio da própria empresa.

Na Figura 2, retirada do site para desenvolvedores da Google, é possível ter um entendimento do fluxo gerado através dos Protocol Buffers:

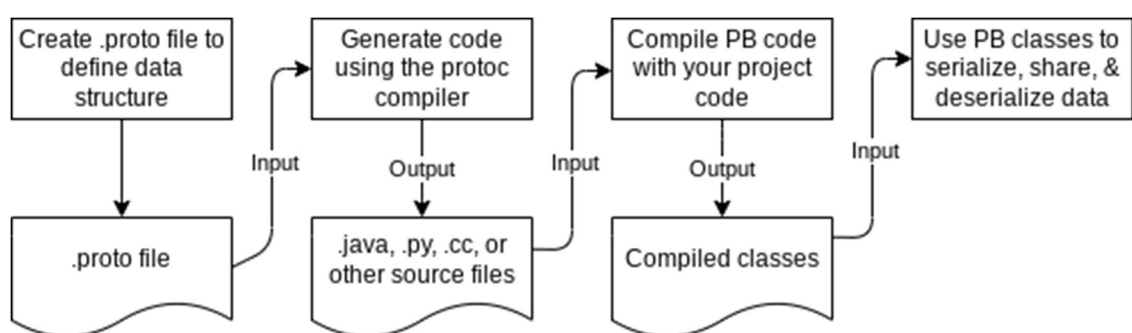


Figura 2 - Protocol buffers workflow

Fonte: Site desenvolvedores Google, guia, visão geral sobre Protocol Buffers

O primeiro passo é definir a interface de serviço, que contém os métodos que permitiram aos clientes fazerem chamadas remotas, os parâmetros dos métodos e as definições das mensagens para serem usadas na invocação. Toda essa definição de serviço e estrutura de dados das mensagens ficam registrados em um arquivo *.proto*.

A seguir, utilizamos o *plugin* específico da linguagem de programação desejada para compilar o arquivo *.proto* e gerar códigos (*protobuf code*), usualmente contendo classes e outras estruturas que refletem a definição do serviço na linguagem específica.

Na terceira etapa, fazemos uso das classes geradas para implementar os métodos declarados e disponibilizar os serviços caso estejamos no lado servidor, ou se estivermos no lado cliente, usamos os códigos para conectarmos à API, a fim de podermos consumir os serviços disponibilizados.

Por último, usamos o protocolo buffers para serializar nossos objetos, enviar pela rede e reconstruí-los do outro lado, sem nos preocuparmos com os detalhes da transmissão e serialização.

## 5. TRABALHOS SIMILARES

No espectro do nosso trabalho, encontramos o desenvolvimento de um aplicativo integrado ao sistema IGUAL - *Innovation for Equality in Latin American Universities* – para que seus usuários possam utilizá-lo através do Facebook, tendo sua integração feita através de Web Services REST. Os objetos de aprendizagem podem ser acessados através da respectiva rede social, tendo também suas funções de curtir, comentar e compartilhar disponíveis para cada objeto. Pretende-se com a implementação aumentar a visibilidade e o fornecimento de cursos através da IGUAL. (VASCONCELLOS; CAMARGO; CECHINEL, 2013).

Utilizando o ecossistema de desenvolvimento Spring Framework para Java, encontramos o trabalho do Gustavo W. Kuhn (2018), em que ele pontua:

Com o grande crescimento no fornecimento de tecnologias de software para dispositivos móveis, vê-se a necessidade da migração de tecnologias atuando de forma monolítica para atuação na nuvem e distribuídas em microsserviços. Através de grandes provedores de tecnologia em microsserviços estas aplicações se tornam escaláveis e de fácil manutenção. Neste projeto, em um aplicativo de chat, utilizando tecnologias como Spring Framework para Java, esta nova abordagem foi tratada de



2022-1  
forma individual, dividida em etapas, como a definição da arquitetura, separação de tarefas para cada serviço desenvolvimento da aplicação para Android, e sua comunicação com os serviços externos foi construída através do protocolo HTTP usando REST. (KUHN, 2018)

Na linha do nosso trabalho em que aborda tanto API REST quanto API gRPC, temos o recente trabalho de conclusão de curso do Fonseca (2021) onde o autor foca no desempenho de APIs.

O aumento da demanda por tecnologias móveis é um fato presente em nossa realidade, aumentando-se também a necessidade por novos métodos que providenciem aparelhos mais rápidos e com disponibilidade de serviços em qualquer lugar onde o usuário estiver, com a terceirização no processamento de dados. A partir desta ideia, busca-se tecnologias que forneçam isto, economia de tempo, energia e espaço, através da análise da realização de cálculos em diferentes ambientes, avaliando-se o impacto sobre o dispositivo, tecnologias como REST e gRPC em servidores nos Estados Unidos da América. Utilizou-se algoritmos do tipo Heap Sort, Bubble Sort e Selection Sort. Nos resultados nota-se que a complexidade do algoritmo cause um impacto sobre o desempenho da aplicação e no consumo da bateria, tanto em soluções remotas como locais. (FONSECA, 2021)

## 6. METODOLOGIA

Para codificação das linguagens de programação Golang, JavaScript e Dart nós utilizamos o editor Neovim na versão 0.9.0. executando sobre terminal shell bash. Para a linguagem Java usamos o editor Eclipse, versão 4.23.0 com gerenciador de dependências Maven.

Utilizamos os *plugins* para compilação de arquivos *protobuf* sintaxe na versão 3 de cada linguagem e o servidor de aplicação NodeJS na versão 14.19.3 para JavaScript. A versão do Java foi a 1.8, a do Golang foi 1.18.1 e Dart SDK 2.15.1.

A máquina utilizada foi um notebook Dell, com processador Intel i7 da 7ª geração de 4 núcleos, com 16G de memória RAM e SSD de 128G. O sistema operacional foi a distribuição Linux Pop!\_OS 21.10.

## 7. DESENVOLVIMENTO

Para o estudo prático nós construímos um sistema distribuído de microsserviços fazendo a integração de sistemas implementados em quatro linguagens de programação diferentes, utilizando para a comunicação e interoperabilidade dos sistemas o gRPC.

Na Figura 3 vemos a arquitetura implementada. A aplicação Java precisa criar uma lista de 10 alunos amostrados aleatoriamente de uma lista interna de 51. Para isso, ele consome uma API que oferece o serviço de sortear números inteiros dentro dos limites de um intervalo. A aplicação Java fornece um intervalo de 0 a 50 e fica fazendo chamadas remotas ao servidor NodeJS até preencher a quantidade desejada.

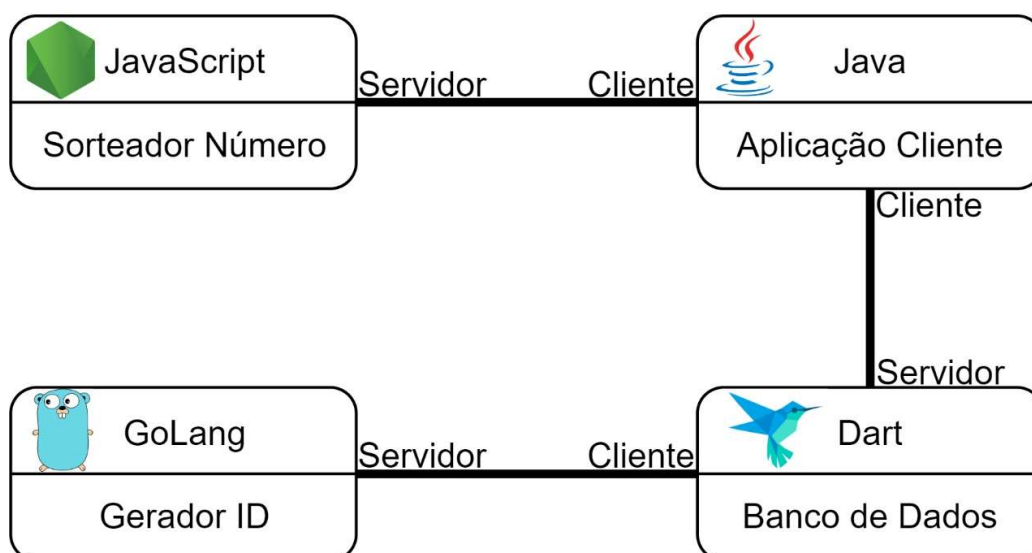


Figura 3 - Exemplo da arquitetura implementada

Depois precisa persistir os alunos selecionados em um banco de dados. Uma API implementada em Dart fará a emulação de um banco de dados oferecendo um serviço de API contendo todos os métodos das operações elementares de um banco: criar, atualizar, buscar por id, listar todos e apagar. O serviço Dart salva os dados apenas em memória.

Para emular a inserção em um banco, usualmente cria-se um número sequencial para o id. Para testarmos a comunicação entre as APIs, a API Dart é cliente de outra API escrita em GoLang. A API GoLang faz o papel de gerador de chave primária, fornecendo números

sequências a cada chamada remota, para que o Dart possa usá-lo como identificador único da entidade Aluno, que a aplicação Java lhe solicitou para salvar no banco.

## 7.1. JavaScript – Sorteador de número

A API em JavaScript irá oferecer um serviço de sorteio de número inteiro dentro de um intervalo fechado.

### 7.1.1. Definição de contrato – sorteio.proto

Primeiramente precisamos da definição do contrato comum entre o servidor JS e os seus clientes, `sorteio.proto`, na Figura 4.

```
1 // sorteio.proto
2
3 syntax = "proto3";
4
5 service SorteioService {
6   rpc SortearNumero (IntervaloRequest) returns (SorteadoResponse) {}
7 }
8
9 message IntervaloRequest {
10  int32 min = 1;
11  int32 max = 2;
12 }
13
14 message SorteadoResponse {
15  int32 numero = 1;
16 }
17
```

Figura 4 - *sorteio.proto*

Na linha 3 estamos informando ao plugin-compilador *protobuffer* que estamos usando a linguagem de definição na versão 3. A palavra *service* denota um serviço em gRPC, logo, na linha 5 estamos declarando o serviço *SorteioService*. Tal como métodos e classe, em que as chamadas aos métodos são realizadas através de instâncias de classe, para invocarmos um método gRPC, ele precisa estar referenciado em um serviço. Podemos declarar vários serviços em um arquivo `.proto`.

Os métodos são declarados dentro do escopo do serviço com a palavra reservada `rpc`. Ela informa ao compilador que o método é um *endpoint* que ficará disponível às chamadas de clientes remotos. Em nosso `sorteio.proto`, nós temos apenas o método `SortearNumero` dentro do serviço `SorteioService`. `SortearNumero` recebe como argumento um tipo de dado chamado `IntervaloRequest` e retorna o tipo de dado `SorteadoResponse`. A sua implementação deverá retornar um número inteiro dentro de um intervalo de domínio.

Os tipos de dados são idealizados como tipos de mensagens, denotados pela palavra reservada `message`. Cada tipo `message` possui campos ou atributos e cada atributo possui um identificador único numérico. Essa numeração sequencial dos atributos é utilizada pelo *framework* gRPC para fazer a serialização e desserialização na ordem correta.

No tipo de mensagem `IntervaloRequest` da linha 9, temos dois atributos de tipo inteiro em sua estrutura de dado. Nos atributos `min` e `max` estarão os limites inferior e superior, respectivamente, com ambos os limites inclusos, dentro dos quais um número deverá ser sorteado. O tipo de mensagem `SorteadoResponse` é o tipo de dado que a chamada remota ao método retornará, armazenado em seu campo `numero` o número sorteado.

### 7.1.2. Servidor gRPC – NodeJS

A principal ferramenta de manipulação de *protofiles* é o `protoc`, o gerador de códigos e *parser* do *protobuf*, que compila os arquivos de definição de contrato `.proto` em arquivos `.pb.js`. Entretanto, com JavaScript vamos gerar o conteúdo dinamicamente com `@grpc/proto-loader`. Ao invés do arquivo ser pré-compilado, com esta ferramenta o arquivo `.proto` é carregado em memória e "parseado" em tempo de execução.

```
server.js JS
1 // server.js
2 const grpc = require("@grpc/grpc-js");
3 const protoLoader = require("@grpc/proto-loader");
4 const PROTO_PATH = "./sorteio.proto";
5
6 const protoObject = protoLoader.loadSync(PROTO_PATH);
7 const sorteioDefinition = grpc.loadPackageDefinition(protoObject);
8 const SorteioService = sorteioDefinition.SorteioService;
9
10 const server = new grpc.Server();
11 server.addService(SorteioService.service, {sortearNumero});
12
```

Figura 5 - server.js

Inicialmente criamos uma representação do arquivo .proto na linha 6 e na linha 7 fornecemos esse objeto para o gRPC para que ele crie uma definição válida, com a qual poderemos criar um serviço de API na linha 8. Na linha 10 instanciamos o servidor e na linha 11 adicionamos o serviço, fornecendo dois parâmetros, o serviço e a lista de funções que implementam os métodos declarados na definição de contrato do .proto, que no nosso caso temos apenas um, sortearNumero. Na Figura 6 temos a implementação do método.

```
13
14 function sortearNumero({ request:{min, max} }, callback) {
15     let sorteado = bingo(min, max);
16     sorteadoResponse = {numero: sorteado};
17     console.log('🍏 API JavaScript - Número Sorteadado:' + sorteado);
18     return callback(null, sorteadoResponse);
19 }
20
21 function bingo(min, max) {
22     return Math.floor( Math.random() * (max - min + 1) ) + min;
23 };
24
```

Figura 6 - Implementação do método sortearNumero

Por fim, colocamos o servidor no ar, ouvindo na porta 50053 com autenticação vazia para o HTTP/2 (Figura 7).

```

24
25 const endereco = 'localhost';
26 const porta = '50053';
27 const pontoAcesso = endereco + ':' + porta;
28
29 server.bindAsync(
30     pontoAcesso,
31     grpc.ServerCredentials.createInsecure(),
32     (error, port) => {
33         console.log("\n🍏 Servidor rodando no ponto acesso " + pontoAcesso);
34         server.start();
35     }
36 );
37

```

*Figura 7 - Colocando o servidor no ar*

## 7.2. Golang – Fornecedor de id

A API em Golang será o provedor de id numérico. A ideia é que o Go fará as vezes do gerador de chave primária numérica sequencial de um sistema gerenciador de bancos de dados. O banco de dados será emulado na linguagem Dart, tratado no capítulo seguinte.

### 7.2.1. Definição de contrato – gerador\_id.proto

Na Figura 8 temos a definição de contrato gerador\_id.proto.

```

1 // gerador_id.proto
2 syntax = "proto3";
3
4 import "google/protobuf/empty.proto";
5
6 option go_package = "github.com/earmarques/tcc_grpc/go_grpc";
7
8 package geradorid;
9
10 service GeradorID {
11     rpc GerarId(google.protobuf.Empty) returns (IdReply) {}
12 }
13
14 message IdReply {
15     int32 goId = 1;
16 }
17

```

*Figura 8 - Definição de contrato gerador\_id.proto*

Na definição de contrato em `sorteio.proto` do JS, procuramos apresentar os elementos estritamente necessários de um arquivo `.proto` – a especificação da sintaxe, as declarações de métodos e serviço e os tipos de dados de mensagem. Já neste capítulo, em `gerador_id.proto` da Figura 8, apresentamos outros elementos da linguagem de definição do *protobuf*. Uma declaração de pacote na linha 8, uma declaração opcional específica da linguagem Golang na linha 6, que diz respeito a organização de pastas e arquivos do projeto no GitHub, e uma declaração de importação de tipo de mensagem externa na linha 4.

A API do Go possui apenas o serviço `GeradorID` declarado na linha 10, que por sua vez possui apenas um método, `GerarID`. O método `GerarID` está declarado na linha 11 e recebe uma mensagem `Empty` e retorna um objeto `message` do tipo `IDReply` que contém em sua estrutura o atributo inteiro `goId` contendo o dado requerido.

A implementação do método `GerarID` não terá argumento, não receberá nenhum parâmetro, entretanto, na sua definição *protobuf* faz-se necessário especificar um tipo de mensagem vazia, equivalente ao `void` de outras linguagens, algo como `message Empty {}`.

Na API Dart tratada mais adiante, há também a necessidade desse tipo de mensagem em dois métodos em seu arquivo `aluno.proto`, no método `GetAllAlunos`, cuja implementação não recebe parâmetros bem como no método `DeleteAluno`, que não retorna nada. Sendo assim, teríamos duas declarações de `Empty` distintas, uma em `aluno.proto` e outra em `gerador_id.proto`. Como efeito, o Dart não saberia a qual classe estaríamos nos referindo. Este conflito poderia ser resolvido fornecendo o nome da classe completamente qualificado, mas optamos por enriquecer nosso exemplo fazendo a importação de uma declaração de `Empty` referenciada em um pacote comum de domínio público (linha 4), o que facilita a distribuição de definições de tipos de mensagens.

Uma vez definida a interface no arquivo `.proto`, precisamos compila-lo. Para isso, invocamos `protoc` com o *plugin* do Go digitando em um terminal *shell* a linha de comando da Figura 9.



```
eder@imotep:~/tcc_grpc/go_grpc/protos
$ protoc --go_out=. --go_opt=paths=source_relative \
--go-grpc_out=. --go-grpc_opt=paths=source_relative \
google/protobuf/empty.proto generator_id.proto
```

Figura 9 - Compilando `.proto` Go através do comando `protoc`



Podemos observar os dois arquivos `.proto` utilizados pela API, o local `gerador_id.pronto` e o que fora referenciado externamente, o `empty.proto`. Como resultado de sua execução, teremos os arquivos `.pb.go` gerados pelo *plugin* gRPC e dentro deles as classes *stubs*.

```

eder@imotep:~/tcc_grpc/go_grpc
$ tree .
.
├── go.mod
├── protos
│   ├── gerador_id_grpc.pb.go
│   ├── gerador_id.pb.go
│   ├── gerador_id.proto
│   ├── google
│   │   └── protobuf
│   │       └── empty.pb.go
└──
3 directories, 5 files
eder@imotep:~/tcc_grpc/go_grpc

```

Figura 10 - Arquivos gerados após compilação protoc Go

### 7.2.2. Servidor gRPC – Golang

A implementação Go mostrada da Figura 11 começa com importações básicas típicas e do pacote gRPC, bem como os pacotes contendo as classes *stubs* geradas pelo compilador, uma que atribuímos o *alias* `emptypb` para o pacote externo do google e `pb` para o local. Na sequência, temos uma constante para a porta e na linha 16 a variável `id` que será incrementada a cada chamada.

```
s/main.go
1 package main
2
3 import (
4     "context"
5     "log"
6     "net"
7
8     "google.golang.org/grpc"
9
10    pb "github.com/earmarques/tcc_grpc/go_grpc/protos"
11    emptypb "google.golang.org/protobuf/types/known/emptypb"
12 )
13 const (
14     port = ":50051"
15 )
16 var id int32 = 0
17 // Stub
18 type server struct {
19     pb.UnimplementedGeradorIDServer
20 }
21 // Implementação do método
22 func (s *server) GerarId(ctx context.Context, in *emptypb.Empty) (*pb.IdReply, error) {
23     id++
24     log.Printf("👉 Id=%d", id)
25     return &pb.IdReply{GoId: id}, nil
26 }
27 }
```

Figura 11 – Implementação do servidor Golang

Nas linhas 18 a 20 temos o *stub* do servidor que atenderá as requisições. Nas linhas 22 a 26, a implementação do método `GerarID`, que recebe o contexto e um objeto `Empty`, e retorna um objeto `IdReply` (linha 25) contendo o valor do `id`, após realizado seu incremento na linha 23.

Por fim, temos o método `main`, no qual abrimos o canal gRPC criando uma conexão de rede com protocolo de transporte TCP, ouvindo as requisições na porta definida na constante `port`. Na linha 35 criamos o servidor e na linha 36 o iniciamos.

```
27
28 func main() {
29     // Canal gRPC
30     lis, err := net.Listen("tcp", port)
31     if err != nil {
32         log.Fatalf("Falha ao escutar a conexão: %v", err)
33     }
34     // Instancia o servidor
35     s := grpc.NewServer()
36     pb.RegisterGeradorIDServer(s, &server{})
37
38     log.Printf("👉 Servidor Go ouvindo na porta %s", port)
39
40     if err := s.Serve(lis); err != nil {
41         log.Fatalf("Falha ao prestar o serviço: %v", err)
42     }
43 }
44 }
```

Figura 12 - Método main do servidor Golang

### 7.3. Dart – Banco de dados

A API em Dart irá emular o banco de dados. No relacionamento com o Java, será o servidor do banco de dados, enquanto o Java ficará do lado cliente. Porém, em relação ao Golang, o Dart estará do lado cliente e o Golang do lado servidor, pois o Golang fará o papel de gerador de chave primária sequencial. O Dart trabalha com três arquivos `.proto`. O `gerador_id.proto` para poder consumir os serviços da API em Go. O arquivo `aluno.proto` para o qual proverá as implementações de servidor para a aplicação cliente Java. E por último, o arquivo `empty.proto`, que vem de um pacote externo para servir de definição de tipo `message` comum em `aluno.proto` e em `gerador_id.proto`. Destes três, dois já foram apresentados, só nos falta `aluno.proto`.

#### 7.3.1. Definição de contrato – `aluno.proto`

Na Figura 13 a seguir temos o arquivo `aluno.proto`, onde podemos ver no seu início a especificação da sintaxe e a importação do tipo de `message Empty` de um pacote externo. Na linha 6 temos a declaração do serviço da API, `CrudAlunoService`, e dentro do escopo do serviço, os métodos que representam as operações básicas de um banco de dados. No nosso caso, neste banco emulado (*mock*), temos apenas uma tabela, a entidade `Aluno`.

```

1 // aluno.proto
2 syntax = "proto3";
3
4 import "google/protobuf/empty.proto";
5
6 service CrudAlunoService {
7     rpc CreateAluno(Aluno) returns (Aluno);
8     rpc GetAllAlunos(google.protobuf.Empty) returns (Alunos);
9     rpc GetAluno(AlunoId) returns (Aluno);
10    rpc DeleteAluno(AlunoId) returns (google.protobuf.Empty) {};
11    rpc EditAluno(Aluno) returns (Aluno) {};
12 }
13
14 message AlunoId {
15     int32 id = 1;
16 }
17
18 message Aluno {
19     int32 id = 1;
20     string nome = 2;
21 }
22
23 message Alunos {
24     repeated Aluno alunos = 1;
25 }

```

Figura 13 - aluno.proto

O método `createAluno` recebe como parâmetro o tipo `Aluno`, sem chave primária (`id`) e retorna o objeto (message) `Aluno` persistido, já com `id`. O método `EditAluno` é usado para atualizar um aluno já persistido e, tal qual `createAluno`, recebe o tipo `Aluno` como argumento e retorna o mesmo tipo, `Aluno`. Vemos na definição do tipo `Aluno` (linha 18) que este possui dois atributos, um inteiro (`id`) e outro do tipo `string` (`nome`).

`GetAllAlunos` recebe uma mensagem do tipo `Empty` e retorna uma coleção (`Alunos`) de tipo `Aluno`. Interessante notar como o *protobuf* declara coleções de forma simples. A coleção `Alunos` está declarada na linha 23 e na linha 24 vemos pela primeira vez a palavra-chave `repeated`, usada para denotar coleção do tipo `Aluno`. O método `GetAluno` recebe um outro tipo de mensagem, `AlunoId`, que possui internamente o atributo `id` e retorna um `Aluno`. `DeleteAluno` recebe o mesmo tipo de parâmetro do método `GetAluno`, porém, retorna um objeto vazio, `Empty`.

Uma vez que a definição esteja pronta, precisamos compilar os arquivos `.proto` para que o *framework* `gRPC` gere as classes necessárias à abstração da comunicação pelo protocolo `HTTP/2`.

```
eder@imotep:~/tcc_grpc/go_grpc
$ protoc -I=protos/ --dart_out=grpc:protos/ \
  protos/aluno.proto \
  protos/gerador_id.proto \
  google/protobuf/empty.proto
```

Figura 14 - Compilando .proto Dart através do comando protoc

Na Figura 14 invocamos o compilador `protoc`; com `-I=protos/` informamos onde estão os arquivos com extensão `.proto`, com `--dart_out=grpc:protos/` estamos dizendo para o `protoc` compilar usando o *plugin* do Dart e para descarregar os arquivos gerados na mesma pasta que estão os `.proto`, e por fim, a lista dos arquivos que devem ser compilados.

Após a execução, a pasta `protos/` que continha apenas os dois arquivos `.proto` locais, passa a ter mais nove arquivos, três deles em uma nova pasta referente ao `empty.proto` externo, como podemos ver na Figura 15.

```
eder@imotep:~/tcc_grpc/dart_grpc
$ tree protos
protos
├── aluno.pb.dart
├── aluno.pbenum.dart
├── aluno.pbgrpc.dart
├── aluno.pbjson.dart
├── aluno.proto
├── gerador_id.pb.dart
├── gerador_id.pbenum.dart
├── gerador_id.pbgrpc.dart
├── gerador_id.pbjson.dart
├── gerador_id.proto
└── google
    └── protobuf
        ├── empty.pb.dart
        ├── empty.pbenum.dart
        └── empty.pbjson.dart

2 directories, 13 files
```

Figura 15 - Arquivos gerados após compilação protoc Dart

### 7.3.2. Servidor gRPC de banco de dados e Cliente gRPC de Golang

Para implementarmos o servidor de banco de dados Dart nós estendemos a classe abstrata `CrudAlunoServiceBase`, encontrada no arquivo `aluno.pbgrpc.dart`, gerado pelo *framework* gRPC na compilação. Vemos com mais clareza os conceitos elementares da comunicação do gRPC, canal e *stub*. O compilador criou uma classe que abstrai o canal de comunicação, `ClientChannel` e a classe `GeradorIDClient` que é o *stub*.

Nos foi gerado também a classe `Alunos`, a partir da qual instanciamos essa coleção no objeto `lista`, que será a tabela do nosso banco emulado.

```
server_client_go.dart
1 import 'package:grpc/grpc.dart';
2 import '../protos/aluno.pbgrpc.dart';
3 import '../protos/gerador_id.pbgrpc.dart';
4 import '../protos/google/protobuf/empty.pb.dart' show Empty;
5
6
7 class CrudAlunoService extends CrudAlunoServiceBase {
8   Alunos lista = Alunos();
9   late ClientChannel channel;
10  late GeradorIDClient stub;
11 }
```

Figura 16 - Estendendo classe abstrata `CrudAlunoService`

O único momento em que se precisa gerar uma chave primária é na operação de inserção no banco. Portanto, apenas o método `createAluno` fará uso da API do Go.

```

12 @override
13 Future<Aluno> createAluno(ServiceCall call, Aluno request) async {
14   print('\n API Dart - createAluno -----');
15   late int id;
16   if (request.id == 0) { // buscar id
17     // Obtendo id do servidor Go
18     late var message;
19     channel = ClientChannel('localhost',
20       port: 50051,
21       options: // Aqui não teremos credenciais
22         const ChannelOptions(credentials: ChannelCredentials.insecure()));
23     stub = GeradorIDClient(channel,
24       options: CallOptions(timeout: Duration(seconds: 30)));
25     try {
26       message = await stub.gerarId(Empty());
27     }
28     catch (e) {
29       print('\n\nErro ao obter ID do servidor Go. Talvez o servidor esteja offline\n');
30       print(e);
31     }
32     await channel.shutdown();
33     id = message.goId;
34   }
35   else { // usar o fornecido no request
36     id = request.id;
37   }
38   var aluno = Aluno();
39   aluno.nome = request.nome;
40   aluno.id = id;
41   lista.alunos.add(aluno);
42   print('Método createAluno - novo aluno inserido:');
43   print(aluno.toString());
44
45   return aluno;
46 }
47

```

Figura 17 - Operação de inserção no banco

No corpo do método `createAluno` nós declaramos uma variável inteira, `id`, e verificamos se o valor do `id` foi fornecido no parâmetro `request`. Caso esteja presente, usaremos o que foi enviado, caso contrário usaremos a API Go para gerar o `id` para nós. Em seguida terminamos de preencher o objeto `aluno` e o adicionamos à `lista` (o equivalente a inserir na tabela).

Para consumirmos a API Go, primeiramente estabelecemos o canal de comunicação entre o nosso código Dart e a API, passando como argumento o endereço ip (`localhost`), a porta e as opções do canal. Assim como no JavaScript, nas opções do canal passamos um objeto de credenciais vazias (linhas 19 a 22). Em seguida, na linha 23 instanciamos o `stub`, o objeto que corresponde ao serviço exposto pelo servidor remoto, fornecendo o canal recém-criado, com a opção de aguardar por trinta segundo uma resposta às requisições. Usando um bloco `try-catch` invocamos o método desejado através do `stub`.



Os demais métodos do serviço `CrudAlunoService` não consomem API, apenas manipulam os elementos da lista (Figura 18). Podemos notar que todos os métodos possuem um `ServiceCall`, esse objeto `call` contém metainformações do request.

```
48 @override
49 Future<Alunos> getAllAlunos(ServiceCall call, Empty request) async {
50     //=====
51     print('\n API Dart - getAllAlunos -----');
52     if (lista.alunos.isEmpty) {
53         print('Lista vazia.');
```

```
54     }
55     for (var aluno in lista.alunos) {
56         print('${aluno.id} - ${aluno.nome}');
```

```
57     }
58     print(' ');
59     return lista;
60 }
61 @override
62 Future<Aluno> getAluno(ServiceCall call, AlunoId request) async {
63     print('\n API Dart - getAluno -----');
```

```
64     print('Resgantando o aluno de id:${request.id} ');
65     var aluno = lista.alunos.firstWhere((aluno) => aluno.id == request.id);
66     return aluno;
67 }
68 @override
69 Future<Empty> deleteAluno(ServiceCall call, AlunoId request) async {
70     print('\n API Dart - deleteAluno -----');
```

```
71     print('Apagando o aluno de id:${request.id} ');
72     lista.alunos.removeWhere((aluno) => aluno.id == request.id);
73     return Empty();
74 }
75 @override
76 Future<Aluno> editAluno(ServiceCall call, Aluno request) async {
77     print('\n API Dart - editAluno -----');
```

```
78     print('Editando o aluno de id:${request.id} ');
79     var aluno = lista.alunos.firstWhere((aluno) => aluno.id == request.id);
80     aluno.nome = request.nome;
81     return aluno;
82 }
83 }
```

*Figura 18 - Demais métodos da classe `CrudAlunoService`*

Vimos como o Dart consome uma API gRPC, a seguir, veremos como o Dart disponibiliza seu serviço para chamadas remotas. Na Figura 19 nós temos o método `main`, onde instanciamos o servidor, passando ao construtor a instância do serviço que queremos oferecer, `CrudAlunoService` (linhas 87 e 88). E por fim, iniciamos o servidor na porta 50052 (linha 91).

```

84
85 Future<void> main(List<String> args) async {
86     final server = Server(
87         [CrudAlunoService()],
88         const <Interceptor>[],
89         CodecRegistry(codecs: const [GzipCodec(), IdentityCodec()]),
90     );
91     await server.serve(port: 50052);
92     print('\n🐦 Servidor ouvindo na porta ${server.port}...\n');
93 }
94

```

Figura 19 - Disponibilidade de serviço Dart para chamadas remotas

#### 7.4. Java – Aplicação Cliente

O Java será aplicação cliente de duas API's, da API JavaScript que oferece o serviço de sorteio de número inteiro dentro de um intervalo entre mínimo e máximo, e da API Dart que oferece serviços de banco de dados. Para o Java, mudamos de editor e usamos a IDE Eclipse, muito popular na comunidade. No Eclipse criamos um projeto Maven que irá gerir as dependências e compilar os arquivos .proto. Na Figura 20 vemos a estrutura de pacotes do projeto Java.

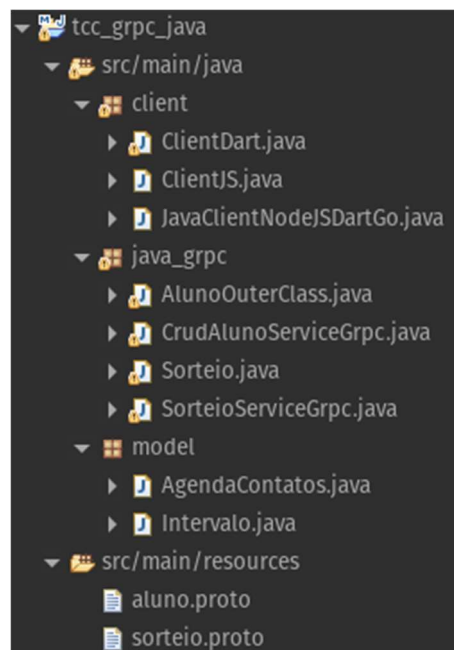
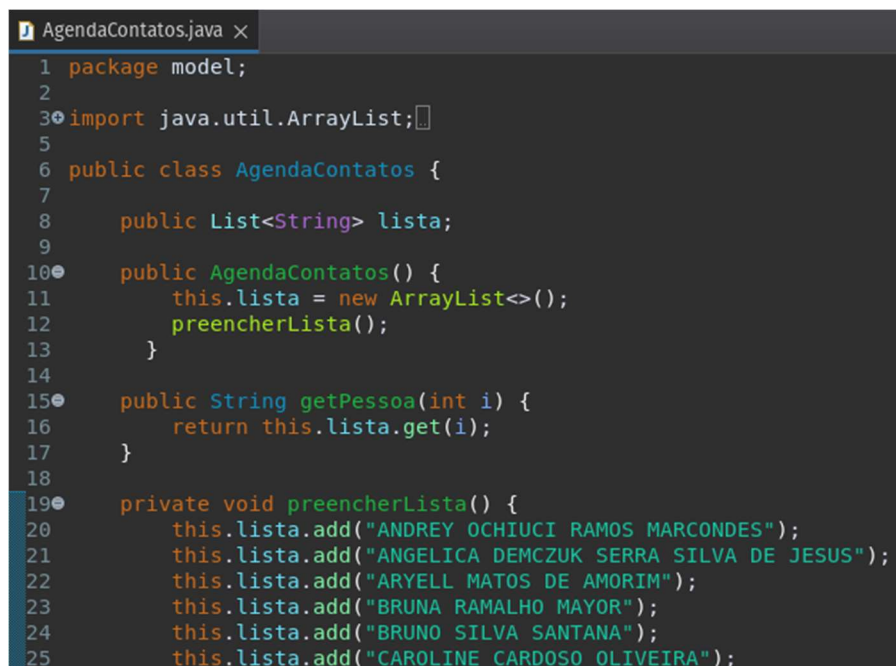


Figura 20 - Estrutura de pacotes projeto Java

A aplicação Java consome duas API e os seus arquivos *protobuf* estão no pacote `resources`. Com `sorteio.proto` a aplicação se comunica com a API NodeJS e `aluno.proto` com a API Dart. Para compilar os arquivos `.proto`, executamos dentro do próprio Eclipse o *Maven-build* e os arquivos gerados vão para o pacote `java_grpc`. Vemos dentro do pacote que foram gerados dois arquivos para cada `.proto`.

No pacote `model` temos duas classes. `AgendaContatos` possui internamente uma lista de nomes de 51 elementos e um método para obtê-los a partir do índice. A classe `Intervalo` possui os atributos inteiros `min` e `max` e será usada na chamada da API JavaScript. A listagem das classes `AgendaContatos` e `Intervalo` seguem as Figuras 21 e 22, respectivamente.



```
1 package model;
2
3 import java.util.ArrayList;
4
5
6 public class AgendaContatos {
7
8     public List<String> lista;
9
10    public AgendaContatos() {
11        this.lista = new ArrayList<>();
12        preencherLista();
13    }
14
15    public String getPessoa(int i) {
16        return this.lista.get(i);
17    }
18
19    private void preencherLista() {
20        this.lista.add("ANDREY OCHIUCI RAMOS MARCONDES");
21        this.lista.add("ANGELICA DEMCZUK SERRA SILVA DE JESUS");
22        this.lista.add("ARYELL MATOS DE AMORIM");
23        this.lista.add("BRUNA RAMALHO MAYOR");
24        this.lista.add("BRUNO SILVA SANTANA");
25        this.lista.add("CAROLINE CARDOSO OLIVEIRA");
```

Figura 21 - Classe *AgendaContatos*

```
Intervalo.java x
1 package model;
2
3 public class Intervalo {
4     public int min = 0;
5     public int max = 0;
6
7     public Intervalo() {}
8
9     public Intervalo(int min, int max) {
10         this.min = min;
11         this.max = max;
12     }
13 }
14
```

Figura 22 - Classe Intervalo

Nós implementamos testes para todos os métodos definidos nos serviços, mas vamos focar na criação de Aluno, pois utiliza as duas APIs. Na linha 129 da Figura 23 obtemos um nome para o aluno invocando o método `sortearNomePessoa`, passando como parâmetro um objeto `Intervalo`.

```
123 // Testes -----
124
125 // Create
126 public void testCreate() {
127     System.out.println("\n☕ API Java - createAluno _____");
128
129     String nome = sortearNomePessoa(new Intervalo(0, 50)); // gRPC - API NodeJS
130
131     // aluno sem id - quem irá fornecer será a API Golang
132     Aluno alunoToCreate = Aluno.newBuilder().setName(nome).build();
133     //teste
134     Aluno alunoCriado = createAluno(alunoToCreate); // gRPC - API Dart
135     System.out.println("Novo aluno criado:");
136     System.out.println(alunoCriado);
137 }
138
```

Figura 23 - Criação de Aluno

Na linha 27 da Figura 24 vemos que o método `sortearNomePessoa` obtém o nome da pessoa do objeto `contatos`, passando para ele o índice da sua lista de contato. Esse índice vem de `getNumeroSorteado` que é o método cliente da API JavaScript.

```

18 public class JavaClientNodeJSDartGo {
19
20     AgendaContatos contatos = new AgendaContatos();
21
22
23     // Sorteio NodeJS -----
24
25     private String sortearNomePessoa(Intervalo intervalo) {
26         int numeroSorteado = getNumeroSorteado(intervalo.min, intervalo.max);
27         String nomePessoa = contatos.getPessoa(numeroSorteado);
28         return nomePessoa;
29     }
30
31     // sortearNumero
32     private int getNumeroSorteado(int min, int max) {
33         // Canal
34         ManagedChannel channel = ManagedChannelBuilder
35             .forAddress("localhost", 50053)
36             .usePlaintext()
37             .build();
38         // Stub: ligação ao servidor de API
39         SorteioServiceGrpc.SorteioServiceBlockingStub sorteioStub = SorteioServiceGrpc
40             .newBlockingStub((Channel)channel);
41         // Montar a requisição
42         Sorteio.IntervaloRequest request = Sorteio.IntervaloRequest.newBuilder()
43             .setMin(5)
44             .setMax(15)
45             .build();
46         // Fazer a requisição
47         Sorteio.SorteadorResponse response = sorteioStub.sortearNumero(request);
48         int sorteado = response.getNumero();
49
50         return sorteado;
51     }
52 }
53

```

Figura 24 - Métodos para criar Aluno: *sortearNomePessoa* e *getNumeroSorteado*

No corpo do método `getNumeroSorteado` criamos o canal de comunicação passando o endereço e a porta. Em seguida conseguimos o *stub* do serviço `SorteioService` passando o canal a um método construtor. O Java adotou o padrão de projeto *Builder* para a instanciação de objetos. Dessa forma, construímos o objeto do `IntervaloRequest` que é passado ao stub para fazer a chamada remota ao servidor NodeJS.

Retomando à Figura 23, construímos o objeto `alunoToCreate` na linha 132 e na 134 chamamos o método cliente da API Dart, `createAluno`, listado na Figura 25. A chamada remota ao microserviço em Dart se dá de maneira idêntica à do microserviço em JavaScript, criamos o canal, o usamos para construir o *stub* e por meio deste fazemos a chamada.



```

50 // CRUD Dart -----
51
52 // Create
53 Aluno createAluno(Aluno a) {
54     System.out.println("\n☕ API Java - createAluno _____");
55     Aluno aluno = null;
56     ManagedChannel channel = ManagedChannelBuilder
57         .forAddress("localhost", 50052)
58         .usePlaintext()
59         .build();
60     CrudAlunoBlockingStub alunoStub = CrudAlunoGrpc.newBlockingStub(channel);
61     aluno = alunoStub.createAluno(a);
62     System.out.println("Método createAluno - novo aluno criado:");
63     System.out.println(aluno);
64
65     return aluno;
66 }

```

Figura 25 - Método createAluno cliente da API Dart

## 7.5. Simulação

Na figura 26 vemos o nosso ambiente de execução de testes. Temos do lado direito o Eclipse executando o Java com o *Console* expandido, e à esquerda uma grande janela de terminal que foi multiplexada com o *tmux* em três colunas. A coluna mais à esquerda é dedicada ao Golang, a do meio ao servidor NodeJS e a da direita executa o servidor de banco dados Dart.

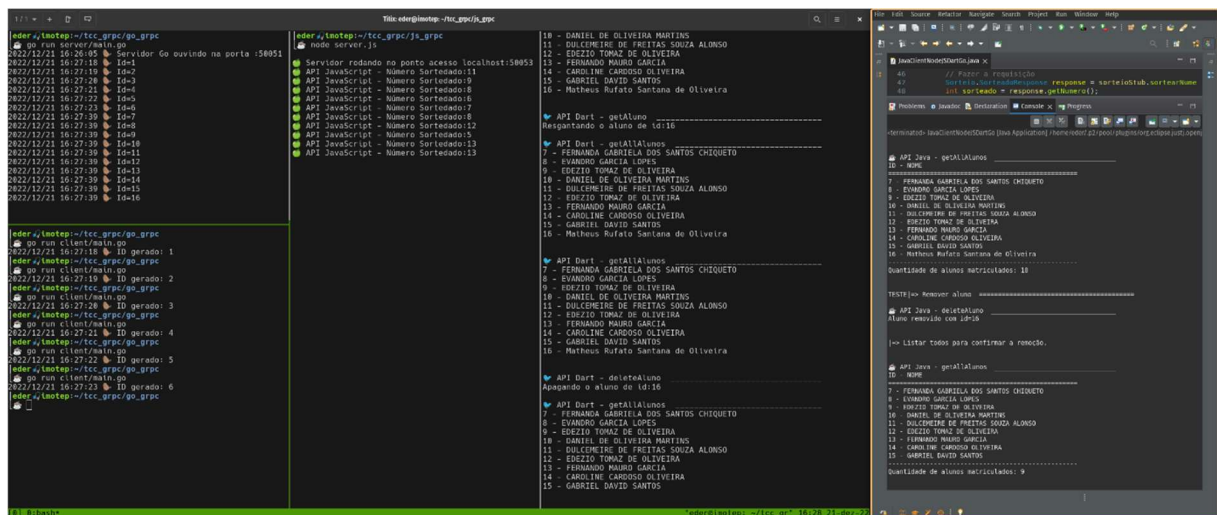


Figura 26 - Ambiente de execução

2022-1

Na figura 27 temos em destaque a coluna do Golang e podemos ver que ela está dividida em dois *shells*, no superior temos o servidor de *id* ouvindo na porta 50051, e no *shell* de baixo executamos por seis vezes um cliente Golang, que com efeito, fez o servidor Golang gerar os seis primeiros *ids*. Em consequência, quando executarmos a API Dart e esta for adicionar um aluno ao banco, e para tanto, precisará solicitar ao servidor Golang o *id*, a API Dart receberá o *id* de número sete, pois os seis primeiros já foram gerados.

```

eder@imotep:~/tcc_grpc/go_grpc
└─ go run server/main.go
2022/12/21 16:26:05  Servidor Go ouvindo na porta :50051
2022/12/21 16:27:18  Id=1
2022/12/21 16:27:19  Id=2
2022/12/21 16:27:20  Id=3
2022/12/21 16:27:21  Id=4
2022/12/21 16:27:22  Id=5
2022/12/21 16:27:23  Id=6
2022/12/21 16:27:39  Id=7
2022/12/21 16:27:39  Id=8
2022/12/21 16:27:39  Id=9
2022/12/21 16:27:39  Id=10
2022/12/21 16:27:39  Id=11
2022/12/21 16:27:39  Id=12
2022/12/21 16:27:39  Id=13
2022/12/21 16:27:39  Id=14
2022/12/21 16:27:39  Id=15
2022/12/21 16:27:39  Id=16

eder@imotep:~/tcc_grpc/go_grpc
└─ go run client/main.go
2022/12/21 16:27:18  ID gerado: 1
eder@imotep:~/tcc_grpc/go_grpc
└─ go run client/main.go
2022/12/21 16:27:19  ID gerado: 2
eder@imotep:~/tcc_grpc/go_grpc
└─ go run client/main.go
2022/12/21 16:27:20  ID gerado: 3
eder@imotep:~/tcc_grpc/go_grpc
└─ go run client/main.go
2022/12/21 16:27:21  ID gerado: 4
eder@imotep:~/tcc_grpc/go_grpc
└─ go run client/main.go
2022/12/21 16:27:22  ID gerado: 5
eder@imotep:~/tcc_grpc/go_grpc
└─ go run client/main.go
2022/12/21 16:27:23  ID gerado: 6
eder@imotep:~/tcc_grpc/go_grpc
└─

```

Figura 27 - Servidor e cliente Go

Na figura 28 temos um trecho da saída padrão do editor Eclipse, mostrando a lista de dez alunos gerada e persistida no banco de dados Dart. Nesse recorte, a API Java fez uma chamada remota ao método `getAllAlunos` da API Dart. Notamos que o *id* obtido pelo Dart, fornecido pela API Golang, foi mesmo o número sete, em decorrência das requisições anteriores feitas com o cliente Golang.



```

TESTE|=> Listar todos os alunos =====
☕ API Java - getAllAlunos _____
ID - NOME
=====
7 - FERNANDA GABRIELA DOS SANTOS CHIQUETO
8 - EVANDRO GARCIA LOPES
9 - EDEZIO TOMAZ DE OLIVEIRA
10 - DANIEL DE OLIVEIRA MARTINS
11 - DULCEMEIRE DE FREITAS SOUZA ALONSO
12 - EDEZIO TOMAZ DE OLIVEIRA
13 - FERNANDO MAURO GARCIA
14 - CAROLINE CARDOSO OLIVEIRA
15 - GABRIEL DAVID SANTOS
16 - GABRIEL DAVID SANTOS
-----
Quantidade de alunos matriculados: 10

```

*Figura 28 - Listagem de alunos: parte da saída da aplicação Java*

Para compor a listagem, a aplicação Java consumiu a API NodeJS que faz o sorteio de números inteiros dentro de um intervalo. A figura 29 traz um recorte da coluna do meio do terminal, onde vemos o sorteador NodeJS respondendo às requisições na porta 50053.

```

☕ node server.js

🍏 Servidor rodando no ponto acesso localhost:50053
🍏 API JavaScript - Número Sorteadado:11
🍏 API JavaScript - Número Sorteadado:9
🍏 API JavaScript - Número Sorteadado:8
🍏 API JavaScript - Número Sorteadado:6
🍏 API JavaScript - Número Sorteadado:7
🍏 API JavaScript - Número Sorteadado:8
🍏 API JavaScript - Número Sorteadado:12
🍏 API JavaScript - Número Sorteadado:5
🍏 API JavaScript - Número Sorteadado:13
🍏 API JavaScript - Número Sorteadado:13

```

*Figura 29 - Servidor NodeJS servindo id na porta 50053*

Na figura 30 temos o início das mensagens de log no terminal do servidor Dart de banco de dados. Vemos que o a API Dart responde na porta 50052 e que seu primeiro aluno inserido possui *id* igual a sete.

```

eder@imotep:~/tcc_grpc/dart_grpc
$ dart bin/server_client.go.dart

Servidor ouvindo na porta 50052...

API Dart - createAluno -----
Método createAluno - novo aluno inserido:
id: 7
nome: FERNANDA GABRIELA DOS SANTOS CHIQUETO

API Dart - createAluno -----
Método createAluno - novo aluno inserido:
id: 8
nome: EVANDRO GARCIA LOPES

API Dart - createAluno -----
Método createAluno - novo aluno inserido:
id: 9
nome: EDEZIO TOMAZ DE OLIVEIRA

```

Figura 30 - Mensagens de log do servidor Dart

## 8. RESULTADOS E DISCUSSÕES

Serviços RESTful, usualmente operam sobre protocolos de transporte baseado em texto, como o HTTP/1.x, e trazem a reboque formatos textuais legíveis a humanos, como o JSON ou XML. Entretanto, a comunicação é um tanto quanto ineficiente, pois converte-se uma estrutura computacional naturalmente binária em texto, para então transmitir texto pela rede, e depois retornar a uma estrutura binária, todavia, máquinas não precisam usar formatos textuais. Já o gRPC usa o protocolo binário *protobuf* para comunicar serviços gRPC a clientes. Cabeçalhos e *payloads* em textos são maiores do que em binários e aumentam a carga sobre a rede. O gRPC implementa o Protocol Buffer sobre o HTTP/2.0, que faz uma comunicação entre processos muito mais rápida e eficiente.

gRPC encoraja uma abordagem de primeiro estabelecermos o contrato. Primeiramente, definimos a interface do serviço e só então trabalhamos nos detalhes da implementação. Diferentemente do OpenAPI/Swagger para a definição de serviços RESTful e do WSDL para os *web services* SOAP, gRPC oferece a IDL, uma linguagem de definição de interface simples, consistente, confiável e escalável para o desenvolvimento de aplicações.

RESTful carece de interfaces fortemente tipadas bem definidas. <sup>2022-1</sup> Quando desenvolvemos serviços RESTful não há uma exigência estrita de definição de serviço e de tipos de informação que seja compartilhada entre aplicações. A aplicação RESTful espera por um formato de texto vindo pela rede, o que em tecnologias políglotas desiguais, pode acarretar erros em tempo de execução e problemas na interoperabilidade. Alguns tipos inteiros em C++ são dependentes de arquitetura de máquina, por exemplo, e peculiaridades dessa ordem podem ocasionar incompatibilidades. Em contraste, gRPC é fortemente tipada. No contrato do serviço definimos claramente os tipos que serão usados na comunicação. Isso ajuda a tornar mais estáveis as aplicações concebidas para operar na nuvem, uma vez que a tipagem estática evita a maioria de erros em tempo de execução e de interoperabilidade. Além disso, gRPC é naturalmente poliglota, porque foi projetada para trabalhar com múltiplas linguagens de programação.

O estilo arquitetural REST possui um conjunto de “boas práticas”, as *constraints*, que devemos seguir para fazer do serviço verdadeiramente RESTful. No entanto, as *constraints* não estão impostas enquanto parte da implementação de protocolos, como o HTTP. Na prática, a maioria dos serviços chamados RESTful não seguem a rigor os princípios REST, são apenas serviços HTTP expostos na web. Demandaria muito tempo de uma equipe de desenvolvimento manter a consistência e fidelidade aos preceitos de um serviço RESTful.

gRPC oferece suporte nativo a *streaming*, tanto do lado cliente quanto do lado servidor, ou a ambos simultaneamente, mantendo o canal aberto e performando uma comunicação bidirecional. Em oposição, o REST praticado sobre o HTTP/1.x suporta apenas comunicação unidirecional, precisando abrir e fechar conexão a cada requisição.

O gRPC faz parte da *Cloud Native Computing Foundation* – CNCF e está integrado às tecnologias desse ecossistema. Podemos dizer que gRPC está maduro, pois tem passado por pesados testes na Google e sua adoção por outras empresas de alta tecnologia como Netflix, Docker, Kubernetes, Cisco entre outras, ratifica sua maturidade. No entanto, pode não ser a melhor escolha em alguns cenários.

O ecossistema gRPC ainda é relativamente pequeno, se comparado ao dueto protocolo HTTP/1.1 e REST. Outro ponto relevante é que o suporte a gRPC em navegadores ainda está nos estágios iniciais. Em relação a aplicações *mobile*, o gRPC está bem integrado no Firebase

e no Flutter, mas não em todas as tecnologias de desenvolvimento *mobile*, sendo ainda preferível o REST em cenários mais voltados ao *front-end*. 2022-1

gRPC é fortemente utilizado no *back-end*, para comunicação inter-processos das aplicações internas das empresas. Caso a intenção seja expor aplicações ou serviço para serem consumidos por clientes externos através da internet, acreditamos que o REST/HTTP seja o mais indicado, além do fato do REST ser mais familiar à comunidade de desenvolvedores e o gRPC ainda pouco conhecido. E nos casos em que houver uma modificação drástica da definição do serviço, normalmente será necessário recriarmos os códigos gerados pelo *framework* gRPC, tanto do lado do cliente quanto do lado do servidor, e isto pode se tornar uma complicação dentro do processo de desenvolvimento.

## 9. CONCLUSÕES

Podemos constatar a forte interoperabilidade entre APIs implementadas em linguagens diferentes. As dificuldades de implementação das tecnologias de chamadas remotas do passado, como CORBA e RMI adivinham da complexidade dos programadores terem de se preocupar com detalhes da comunicação, como conexão, o controle do fluxo de dados e serialização. Com o gRPC toda essa complexidade fica abstraída a cargo do *framework*, uma vez que os *plugins* compiladores geram todo o código necessário a comunicação. O gRPC fornece a interface para o uso da infraestrutura lógica de baixo nível, gerando a abstração necessária por meio de classes do modelo do domínio. Com isso o programador está livre para focar mais na regra de negócio e menos na comunicação.

Notamos uma diferença na clareza dos conceitos elementares da comunicação do gRPC nas interfaces expostas pelas bibliotecas implementadas em cada linguagem. Vemos que o JavaScript fez um ótimo trabalho de abstração carregando e compilando arquivos *.proto* em memória, usando função de *callback*, tão familiar entre a comunidade JS, para fazer o *payload*, de tal forma que com poucas linhas o programador já pode usar o gRPC em seu código. Entretanto, com exceção de *protobuf*, não fica claro os conceitos de *Channel* e *Stub*, quais classes geradas e disponibilizadas pelos *plugins* fariam esses papéis. Em contraponto, vemos no Java os conceitos de *Channel* e *Stub* explicitamente nos nomes das

classes geradas, como em `ManagedChannel`, `CrudAlunoBlockingStub` e `SorteioServiceBlockingStub`.

Vislumbramos um cenário positivo com a adoção do gRPC em larga escala, como é hoje com o REST. Com o gRPC as equipes expõem seus dados e funcionalidades por meio de interfaces de serviço e devem se comunicar por meio dessas interfaces. Não será permitida nenhuma outra forma de comunicação entre processos: sem *links* diretos, sem leituras diretas do armazenamento de dados de outra equipe, sem modelo de memória compartilhada, sem *backdoors*. A única comunicação permitida é por meio de chamadas de interface de serviço pela rede. Todas as interfaces de serviço, sem exceção, devem ser projetadas desde o início para serem externalizáveis. Ou seja, a equipe deve planejar e projetar para poder expor a interface aos desenvolvedores do mundo exterior.

Salientamos que a tecnologia gRPC ainda precisa expandir sua disponibilidade a mais linguagens e que o suporte nos navegadores ainda é incipiente, o que restringe sua utilização ao *back-end* e aplicações *mobile*. Todavia, é apenas uma questão de tempo, porque a perspectiva de menor latência, menor carga computacional na serialização de objetos e menor tráfego pela rede são argumentos fortes para esse suporte se materializar.

Em âmbito geral, constatamos que a partir dos estudos teóricos realizados e do conhecimento adquirido durante a implementação da arquitetura de microserviços, podemos concluir que os resultados acadêmicos foram satisfatórios, tendo proporcionado um bom entendimento da tecnologia gRPC e de seus benefícios relacionados às outras arquiteturas de implementação de API's como o REST.

---

## REFERÊNCIAS

2022-1

BELSHE, M. Internet Engineering Task Force. **Hypertext Transfer Protocol Version 2 (HTTP/2)**. 2015. Disponível em: <https://datatracker.ietf.org/doc/HTTP/rfc7540>. Acesso em: 06 dez. 2021.

CIRIACO, D. **O que é API?** 2009. Acessado em 10/04/2018. Disponível em: <http://www.tecmundo.com.br/programacao/1807-o-que-e-api-.htm>. Acesso em: 04 dez. 2021.

CORE CONCEPTS, ARCHITECTURE AND LIFECYCLE: **An introduction to key gRPC concepts, with an overview of gRPC architecture and RPC life cycle**. 2021. Disponível em: <https://grpc.io/docs/what-is-grpc/core-concepts/>. Acesso em: 06 dez. 2021.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. Tese (Doutorado) — University of California, Irvine, 2000.

FIELDING, R. Network Working Group. **Hypertext Transfer Protocol -- HTTP/1.1**. 1999. Disponível em: <https://datatracker.ietf.org/doc/HTTP/rfc2616>. Acesso em: 06 dez. 2021.

FONSECA, Gabriel Magno França da. **Análise de Desempenho Durante Descarregamento Computacional de Aplicações Móveis**. 2021. Disponível em: <http://bib.pucminas.br:8080/pergamumweb/vinculos/00009c/00009cfb.pdf>. Acesso em: 06 dez. 2021.

GOOGLE DEVELOPERS. **Protocol Buffers: language guide. Language Guide**. 2021. Disponível em: <https://developers.google.com/protocol-buffers/docs/overview>. Acesso em: 06 dez. 2021.

GOOGLE. go.dev, 2020. **Using Go at Google**. Disponível em: <https://go.dev/solutions/google/>. Acesso em: 22 de junho de 2022.

GOOGLE. dart.dev, 2022. **Dart overview**. Disponível em: <https://dart.dev/overview>. Acesso em: 23 de junho de 2022.

GOOGLE. developers.google.com, 2022. **Protocol Buffers**. Disponível em: <https://developers.google.com/protocol-buffers/docs/overview>. Acesso em: 23 de junho de 2022.

GRIGORIK, Ilya. **Making the Web Faster with HTTP 2.0: HTTP continues to evolve**. 2013. Disponível em: <https://queue.acm.org/detail.cfm?id=2555617>. Acesso em: 06 dez. 2021.

GRPC AUTHORS. **INTRODUCTION to gRPC: An introduction to gRPC and protocol buffers**. GRPC Authors. Disponível em: <https://grpc.io/docs/what-is-grpc/introduction/>. Acesso em: 06 dez. 2021.

IBM. IBM – Brasil, 2022. **Common Object Request Broker Architecture (CORBA)**. Disponível em: <https://www.ibm.com/docs/pt-br/integration-bus/10.0?topic=corba-common-object-request-broker-architecture>. Acesso em: 27 de junho de 2022.

IBM. IBM – Brasil | IBM, 2022. **O que É SOAP?**. Disponível em: <https://www.ibm.com/docs/pt-br/integration-bus/10.0?topic=services-what-is-soap>. Acesso em: 27 de junho de 2022.

JOHANN, Marcelo. **Web Services and Corba**. Porto Alegre: UFRGS, 2011. Disponível em: <https://www.inf.ufrgs.br/~johann/sisop2/gvgo2q20112.pdf>. Acesso em: 27 de junho de 2022.

KUHN, Gustavo Weber. **Aplicativo Android para mensagens instantâneas utilizando microserviços REST**. 2018. 43 f. Trabalho de Conclusão de Curso



(Especialização em Redes de Computadores e Teleinformática) - Universidade Tecnológica <sup>2022-1</sup>  
Federal do Paraná, Curitiba, 2018. Disponível em:  
[http://repositorio.utfpr.edu.br/jspui/bitstream/1/20013/1/CT\\_CEREC\\_I\\_2018\\_04.pdf](http://repositorio.utfpr.edu.br/jspui/bitstream/1/20013/1/CT_CEREC_I_2018_04.pdf). Acesso  
em: 06 dez. 2021.

OLIVEIRA, P. H. C. **Desenvolvimento de um gerador de api rest seguindo os principais padrões da arquitetura.** 2015.

ORACLE. Oracle Help Center, 2022. **Getting Started Using Java RMI.** Disponível em:  
<https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html>.  
Acesso em: 27 de junho de 2022.

RICARTE, Ivan L. M., 2002. **Arquitetura CORBA.** Disponível em:  
<https://www.dca.fee.unicamp.br/cursos/PooJava/objdist/idlcorba.html>. Acesso em: 27 de  
junho de 2022.

VASCONCELLOS, Bruno Campos de; CAMARGO, Sandro da Silva; CECHINEL, Cristian. **Desenvolvimento de Um Aplicativo Integrado ao Facebook e ao Sistema do Projeto IGUAL para Compartilhamento e Recomendação de Objetos de Aprendizagem.** 2013. Disponível em: <http://www.tise.cl/volumen9/TISE2013/610-612.pdf>. Acesso em: 06 dez. 2021.