UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUCAS ERTEL SOARES

APIS REST VERSUS GRPC

TOLEDO

2025

LUCAS ERTEL SOARES

APIS REST VERSUS GRPC

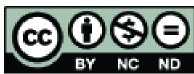
REST APIs Versus gRPC

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Vilson Luiz Dalle Mole

TOLEDO

2025



4.0 Internacional

Esta licença permite download e compartilhamento do trabalho desde que sejam atribuídos créditos ao(s) autor(es), sem a possibilidade de alterá-lo ou utilizá-lo para fins comerciais. Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

LUCAS ERTEL SOARES

APIS REST VERSUS GRPC

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 26/junho/2025

Prof. Dr. Vilson Luiz Dalle Mole
Orientador
COTSI-TD/ UTFPR-TD

Prof. Dr. Fábio Engel de Camargo
Avaliador 1
COTSI-TD/ UTFPR-TD

Profa. Dr. Rosane Fátima Passarini
Avaliadora 2
COTSI-TD/ UTFPR-TD

Prof. Dr. Jefferson Gustavo Martins
Suplente
COTSI-TD/ UTFPR-TD

TOLEDO

2025

Dedico essa obra a deus sem ele nada disso seria possível. "Tudo posso naquele que me fortalece"

AGRADECIMENTOS

Agradeço a Deus, por me sustentar nos momentos de indecisão, por renovar minha força quando pensei em desistir, e por iluminar cada passo dessa jornada.

Sou muito grato ao meu orientador, pela orientação, motivação e calma ao longo do desenvolvimento deste trabalho. Sua ajuda foi importante para que eu conseguisse organizar minhas ideias e avançar com confiança. Agradeço à Universidade, por me proporcionar uma formação sólida e transformadora, e aos professores que contribuíram com conhecimento e inspiração durante minha trajetória acadêmica.

De maneira especial, dedico esse momento à minha mãe e à minha avó, que sempre acreditaram no meu potencial, mesmo diante das dificuldades. Seus gestos de carinho, apoio emocional e incentivo diário foram fundamentais para que eu chegasse até aqui. Gostaria também de deixar registrado que conviver com o TDAH (Transtorno do Déficit de Atenção com Hiperatividade) ao longo da graduação não foi fácil. A desorganização mental, a impulsividade e a dificuldade de concentração estiveram presentes em muitos momentos, mas também me ensinaram a persistir, buscar estratégias, e me conhecer melhor.

Concluir este trabalho representa, para mim, mais do que uma exigência acadêmica: é a prova de que mesmo com obstáculos internos, é possível alcançar grandes conquistas com esforço, apoio e fé. Reconheço a todos que, de alguma forma, estenderam a mão ou acreditaram em mim, o meu mais sincero obrigado.

“Se tornou aparentemente óbvio que nossa tecnologia excedeu nossa humanidade.” Albert Einstein, cientista alemão.

RESUMO

Este trabalho realiza um experimento comparativo entre as arquiteturas REST (*Representational State Transfer*) e gRPC (*Google Remote Procedure Call*), ambas amplamente utilizadas em diferentes tipos de estudos de caso na internet. O modelo REST é conhecido por sua simplicidade e flexibilidade, utilizando métodos HTTP e representações de recursos por URLs, enquanto o gRPC, desenvolvido pelo Google, oferece uma abordagem de alto desempenho baseada em Protocol Buffers e chamadas de procedimento remoto. Para avaliar as diferenças entre as duas arquiteturas, foi desenvolvido protótipos que implementa ambas as abordagens, permitindo a análise de aspectos como desempenho, escalabilidade, e eficiência na troca de dados. Os resultados indicam que o gRPC apresenta moderada vantagem, enquanto o REST se destaca pela facilidade de implementação e ampla adoção. Conclui-se que a escolha entre REST e gRPC deve ser baseada nas necessidades específicas de cada estudo de caso

Palavras-chave: rest; grpc; apis; estudos de caso; desempenho.

ABSTRACT

This study presents a comparative analysis between REST (Representational State Transfer) and gRPC (gRPC Remote Procedure Call) architectures, both widely used in different case studies on the internet. REST is known for its simplicity and flexibility, utilizing HTTP methods and resource representations via URLs, while gRPC, developed by Google, offers a high-performance approach based on Protocol Buffers and remote procedure calls. To evaluate the differences between these architectures, a prototype implementing both approaches was developed, allowing for the analysis of aspects such as performance, scalability, and data exchange efficiency. The results indicate that gRPC offers advantages in terms of latency and resource consumption, while REST excels in ease of implementation and broad adoption. It is concluded that the choice between REST and gRPC should be based on the specific needs of each case study.

Keywords: rest; grpc; apis; case studies; performance.

LISTA DE FIGURAS

Figura 1 – Esquema de uso de uma API. 16

Figura 2 – Representação da comunicação Rest. 17

Figura 3 – Funcionamento do Json em Rest. 20

Figura 4 – Representação da comunicação GRPC. 21

Figura 5 – Funcionamento da *message*(mensagem) em Protobuf. 23

Figura 6 – Arquitetura dos protótipos desenvolvidos 26

Figura 7 – Tempo Médio – REST vs gRPC (Arquivos Pequenos e Grandes) 32

LISTA DE TABELAS

Tabela 1 – Tempo de Envio: REST vs gRPC por Tipo de Arquivo 32

LISTA DE ABREVIATURAS E SIGLAS

Siglas

API	Interface de Programação de Aplicações (Application Programming Interface)
CSV	Valores Separados por Vírgula (Comma-Separated Values)
gRPC	Chamada de Procedimento Remoto da Google (Google Remote Procedure Call)
HTTP	Protocolo de Transferência de Hipertexto (Hypertext Transfer Protocol)
IDE	Ambiente de Desenvolvimento Integrado (Integrated Development Environment)
IDL	Linguagem de Definição de Interface (Interface Definition Language)
JSON	Notação de Objetos JavaScript (JavaScript Object Notation)
PDF	Formato de Documento Portátil (Portable Document Format)
REST	Transferência de Estado Representacional (Representational State Transfer)
RPC	Chamada de Procedimento Remoto (Remote Procedure Call)
STUB	Código Intermediário Gerado para Chamadas Remotas (Stub – Remote Procedure Call Interface)
TCP/IP	Protocolo de Controle de Transmissão/Protocolo de Internet (Transmission Control Protocol/Internet Protocol)
XML	Linguagem de Marcação Extensível (eXtensible Markup Language)

SUMÁRIO

1 INTRODUÇÃO . . . 12

1.1 Considerações iniciais . . . 12

1.2 Problema da Análise . . . 13

1.3 Objetivos . . . 13

1.3.1 Objetivo geral . . . 13

1.3.2 Objetivos específicos . . . 13

1.4 Justificativa . . . 13

1.5 A Estruturação do Trabalho . . . 14

2 EMBASAMENTO TEÓRICO . . . 15

2.1 APIs - Tradicionais . . . 15

2.2 Web APIs . . . 16

2.3 Transferência de Estado Representacional (*REST*-Representational State Transfer) . . . 16

2.3.1 Processo de Comunicação . . . 18

2.3.2 Notação de Objetos JavaScript(JSON) . . . 20

2.4 Chamada de Procedimento Remoto da Google (*Google Remote Procedure Call*) - gRPC . . . 21

2.4.1 Chamada de Procedimento Remoto (*Remote Procedure Call - RPC*) . . . 22

2.4.2 Buffer de Protocolo (*Protocol buffer*) . . . 22

3 MATERIAIS E MÉTODOS . . . 24

3.1 Descrição Geral . . . 24

3.1.1 Clientes de Testes . . . 24

3.1.2 Servidores . . . 24

3.1.3 Processo de Upload . . . 24

3.1.4 Ambiente de Desenvolvimento Comum . . . 25

3.1.5 Máquina Cliente de teste . . . 25

3.1.6 Servidor de teste . . . 25

3.1.7 Arquitetura Geral dos Protótipos . . . 26

3.2 Protótipo gRPC . . . 26

3.2.1 Definição do Protocol Buffers (Protobuf) . . . 26

3.2.2 Tecnologias Utilizadas no *cliente-gRPC* e *servidor-gRPC* 27
3.2.3 Estrutura do Projeto – *cliente-gRPC* 27
3.2.4 Detalhes da Implementação – *cliente-gRPC* 28
3.2.5 Estrutura do Projeto – *servidor-gRPC* 28
3.2.6 Detalhes da Implementação – *servidor-gRPC* 28
3.3 Protótipo REST 29
3.3.1 Tecnologias Utilizadas – *cliente-REST* 29
3.3.2 Estrutura do Projeto – *cliente-REST* 29
3.3.3 Detalhes da Implementação – *cliente-REST* 29
3.3.4 Tecnologias Utilizadas – *servidor-REST* 30
3.3.5 Estrutura do Projeto – *servidor-REST* 30
3.3.6 Detalhes da Implementação – *servidor-REST* 30
3.4 Captura dos dados 30
3.5 Métricas Avaliadas 31
3.6 Execução dos Testes 31
4 RESULTADOS 32
4.1 Apresentação das informações 32
4.2 Parecer Comparativo 33
5 CONCLUSÃO 34
REFERÊNCIAS 35

1 INTRODUÇÃO

1.1 Considerações iniciais

Em um meio altamente conectado e dependente de tecnologias e sistemas, as aplicações e softwares precisam atender às necessidades crescentes de escalabilidade, desempenho e comunicação. Para alcançar esses objetivos, é fundamental o uso de *APIs* (Interface de Programação de Aplicações – *Application Programming Interfaces*) modernas que possam garantir segurança, escalabilidade, comunicação entre softwares de contextos diferentes e desempenho. Assim, surge a importância de abordar temas como os modelos arquiteturais *REST* (Transferência de Estado Representacional – Representational State Transfer) e *gRPC* (Chamada de Procedimento Remoto da Google – *gRPC Google Remote Procedure Call*), que oferecem soluções para separar conceitos e modelar às APIs.

Ambos são amplamente adotados em diversos domínios de aplicação. REST é conhecido por sua simplicidade e flexibilidade, sendo a escolha predominante em muitas aplicações Web devido à sua ampla compatibilidade e fácil implementação.

Por outro lado, o gRPC, desenvolvido pelo Google, se destaca por oferecer uma comunicação mais eficiente, o seu grande diferencial está na combinação do formato binário e enxuto do Protocol Buffers com a comunicação *full duplex*(envia e recebe ao mesmo tempo) que o HTTP/2 permite. Essa junção permite a troca de mensagens simultâneas entre cliente e servidor, com menos atraso e maior fluidez.

A escolha em comparar REST e gRPC se motiva pela diferença entre os dois paradigmas, adotados no desenvolvimento de Web APIs : um consolidado e amplamente compatível (REST) e outro emergente com foco em eficiência e desempenho (gRPC). REST permanece como o padrão na maioria dos serviços web, especialmente pela simplicidade de implementação e grande suporte de ferramentas. Já o gRPC vem ganhando espaço em contextos que demandam maior desempenho e integração entre sistemas distribuídos com baixa latência.

Apesar de suas características distintas, a decisão sobre qual arquitetura adotar nem sempre é fácil e depende de vários fatores, como o contexto de uso, as necessidades específicas do sistema e os requisitos de desempenho. Neste cenário, torna-se aceitável realizar um experimento comparativo que possa orientar essa escolha de maneira fundamentada. Portanto, a proposta é demonstrar e comparar as arquiteturas REST e gRPC, através da implementação de protótipos que permitam demonstrar suas formas de comunicação e implementação. A partir desta análise, espera-se fornecer argumentos que ajudem desenvolvedores e engenheiros de software a tomar decisões melhores sobre qual tecnologia utilizar em seus projetos.

1.2 Problema da Análise

No cenário atual em que vivemos, muitos sistemas e aplicações dependem diretamente de tecnologias de comunicação e protocolos de troca de dados. Diante disso, os desenvolvedores frequentemente enfrentam dúvidas quanto sobre qual arquitetura de API é mais adequada para seus projetos. Na maioria das vezes, opta-se pela arquitetura REST, amplamente reconhecida por sua simplicidade, vasta documentação e aderência aos princípios da web.

Em compensação, o gRPC surge como uma alternativa moderna, utilizando buffer de protocolos(*Protocol Buffers*) e comunicação em formato binário, oferecendo relativa facilidade de implementação, "desde que seja superada sua curva inicial de aprendizado". No entanto, essa arquitetura nem sempre é percebida como a melhor escolha. Esse experimento contribui para ajudar na dificuldade dos desenvolvedores em selecionar, de forma concreta, qual dos modelos de APIs citados utilizar em suas aplicações.

1.3 Objetivos

1.3.1 Objetivo geral

Desenvolver um experimento comparativo entre os modelos arquiteturais REST e gRPC utilizados em APIs distribuídas através da internet, demonstrando suas funcionalidades e uso.

1.3.2 Objetivos específicos

- Descrever o Modelo REST
- Descrever o Modelo gRPC
- Demonstrar os protótipos de teste em ambos os modelos
- Demonstrar a implementação e resultados do experimento comparativo entre os modelos.

1.4 Justificativa

Nesta pesquisa será realizada um experimento comparativo entre os modelos REST e gRRPC, utilizados em APIs modernas, a fim de amparar dúvidas de desenvolvedores e entusiastas da área sobre seu funcionamento, conceitos, aplicabilidade e suas principais diferenças. Explicando alguns conceitos de comunicação, codificação, uso de protocolos e estrutura de dados, aprendidos no curso de graduação.

1.5 A Estruturação do Trabalho

A estrutura desse estudo consiste nos seguintes tópicos:

Capítulo 2 - Referencial Teórico : Contempla a explicação do conceito de APIs, a diferença de Web APIs e APIs tradicionais, sobre o que é REST e GPRC e as características e conceitos das suas arquiteturas.

Capítulo 3 Materiais e Métodos: Aborda as tecnologias utilizadas, os padrões de organização, estruturação do projeto e a metodologia utilizada para realizar os testes

Capítulo 4 RESULTADOS : A apresentação da aplicação e os resultados obtidos, as coletas de tempo entre os dois paradigmas de comunicação e o parecer comparativo sobre eles.

2 EMBASAMENTO TEÓRICO

Uma API consiste em um contrato de funções que permite a comunicação entre duas aplicações, dispositivos, serviços ou sistemas. Ela atua como uma interface intermediária, transmitindo dados estruturados que podem ser compreendidos por ambas as partes envolvidas. Ela possibilita a comunicação distinta, por um conjunto de funções e protocolos. Para compreender melhor seu conceito, é importante entender os processos que a API utiliza em sua forma de comunicação (COOKSEY, 2023).

2.1 APIs - Tradicionais

Em resumo, as APIs têm um papel essencial na conexão entre diferentes sistemas e programas, pois facilitam a troca de informações de forma prática e eficiente. Elas tornam possível que diferentes aplicações funcionem de maneira integrada, melhorando sua utilidade no dia a dia. Um exemplo claro disso é a API de interface de usuário do sistema operacional Windows. Ela permite que os desenvolvedores criem e controlem janelas, como aquelas que exibem mensagens, horários ou gráficos solicitados pelo usuário. Além disso, oferece suporte a recursos multimídia, como áudio e vídeo, que tornam os aplicativos mais completos e interativos (Microsoft Corporation, 2024).

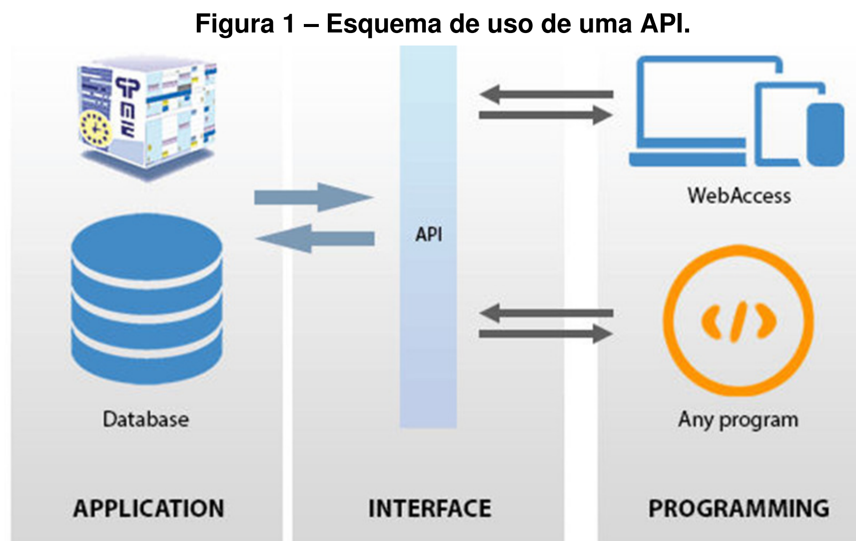
Outro exemplo bastante conhecido é o OpenGL, uma API gráfica voltada para a renderização de elementos tridimensionais. Ela foi desenvolvida pela Silicon Graphics, Inc. (SGI) para ser compatível com diferentes fabricantes de hardware. O OpenGL funciona como uma ponte entre o sistema operacional e o hardware gráfico, otimizando o processo de exibição de imagens em 3D por meio de algoritmos específicos e bem ajustados. Isso permite que os aplicativos apresentem gráficos mais elaborados e com melhor desempenho, especialmente em jogos.

Existe também uma interface de programação bastante utilizada na comunicação entre computadores, conhecida como socket("ponto de comunicação entre dois dispositivos"). Ela funciona dentro da arquitetura do protocolo TCP/IP, mais especificamente entre a camada de aplicação e a de transporte. Essa API serve como um elo entre essas duas camadas, permitindo que os dados enviados por um processo cheguem corretamente até outro processo, mesmo que estejam em máquinas diferentes conectadas pela rede. Em termos simples, os sockets são responsáveis por definir como essa troca de informações vai acontecer, garantindo que a comunicação siga o caminho correto (MARINI; FIGUEIREDO; ROSSETTO, 2013).

2.2 Web APIs

A principal diferença entre Web APIs e APIs tradicionais é que as Web APIs operam no contexto da web e utilizam protocolos específicos para comunicação, como HTTP/HTTPS. Enquanto APIs tradicionais podem usar uma variedade de protocolos e não estão necessariamente limitadas ao ambiente web, as Web APIs são projetadas para serem acessíveis através da internet, facilitando a integração entre diferentes sistemas e aplicações Web (FIELDING; TAYLOR, 2002), conforme representado na Figura 1.

Figura 1 – Esquema de uso de uma API.



Fonte: Araújo (2021).

Além disso, as Web APIs geralmente utilizam formatos de dados como JSON ou XML para a troca de informações, o que as torna altamente compatíveis com navegadores e outras tecnologias web. Elas permitem que desenvolvedores criem aplicações que podem interagir facilmente com serviços web, proporcionando uma maneira padronizada e eficiente de comunicação entre cliente e servidor (MDN Web Docs, 2024).

Por outro lado, APIs tradicionais podem ser usadas em uma ampla gama de contextos, incluindo sistemas locais, dispositivos embarcados e comunicação entre servidores. Elas podem empregar protocolos como FTP, UDP, dependendo das necessidades específicas da aplicação.

Em resumo, enquanto as Web APIs são otimizadas para a comunicação na web, utilizando protocolos e formatos de dados comuns na internet, as APIs tradicionais oferecem maior flexibilidade em termos de protocolos e ambientes de uso.

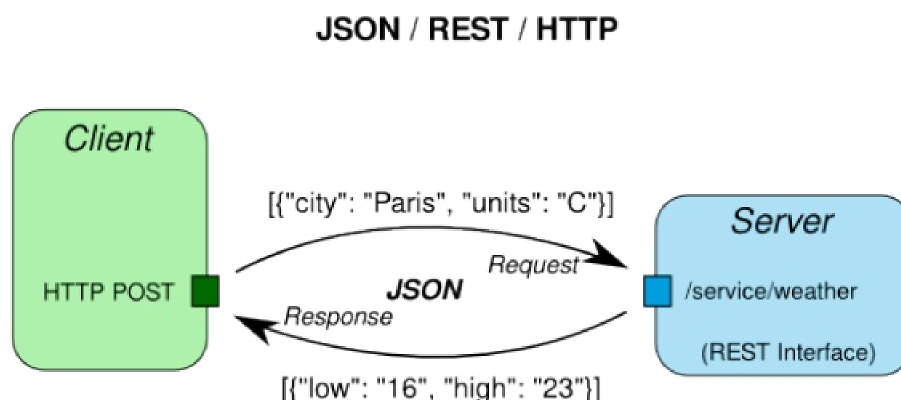
2.3 Transferência de Estado Representacional (*REST*-Representational State Transfer)

Concebida por Roy Thomas Fielding, em sua tese de doutorado no ano 2000 (RIBEIRO; FRANCISCO, 2016). *REST* é um estilo arquitetural, que adiciona restrições, ou seja, modelam seu designer e construção. Segundo Roy, o conceito de web é extenso e escalável e como

um engenheiro ou designer que modela um esboço de um projeto, uns planejam a estrutura e executam e outros começam e vão modelando e avançando, ou seja, há definição de restrições antes no primeiro caso e no segundo essas restrições vão sendo alocadas conforme o necessário (FIELDING, 2000).

Como mencionado anteriormente, *REST* é um estilo arquitetural de software que define um conjunto de regras para a troca de dados entre componentes de software. Sua arquitetura de comunicação é o padrão cliente-servidor (Figura 2), comumente usados na WEB, utilizando o protocolo de comunicação HTTP (*Hypertext Transfer Protocol*), que opera na camada de aplicação do modelo TCP/IP (SERVICES, 2023a). Os principais formatos de comunicação usados pela arquitetura REST são o JSON (*JavaScript Object Notation*-Notação de Objetos JavaScript) uma forma de notação simples de dados que segue o conceito de chave e valor e o XML (*Extensible Markup Language*-linguagem extensível de marcação) que tem o conceito de marcação e sua estrutura é semelhante ao HTML (*HyperText Markup Language* — "Linguagem de Marcação de Hipertexto"). Atualmente, a maioria dos desenvolvedores opta pelo formato JSON por se tratar de um modelo de dados leve e de fácil entendimento (SERVICES, 2023b).

Figura 2 – Representação da comunicação Rest.



Fonte: Horvat, Ilic e Nikolic (2018).

As características da arquitetura *REST* consistem em:

- **Interfaces Uniformes (*Uniform Interface*):** Padrões utilizados na transmissão de mensagens onde seguem normas específicas de formato que podem carregar metadados que são servidos conforme a necessidade do cliente, não gerando inconsistência entre a comunicação (SERVICES, 2023b).
- **Sem Estado *(Stateless)*:** Em uma arquitetura REST, cada requisição do cliente para o servidor deve incluir todas as informações necessárias para o servidor compreender e processe a solicitação. Isso significa que o servidor não retém o estado de sessões

anteriores. Cada interação é autônoma, simplificando a escalabilidade e a recuperação de falhas, uma vez que o servidor não precisa recordar interações passadas com o cliente. Essa característica também facilita o balanceamento de carga, pois qualquer servidor pode responder a uma solicitação sem depender de informações armazenadas em outro servidor (SERVICES, 2023b).

- **Capacidade de armazenamento em cache (*Cacheability*):** REST usa o protocolo HTTP que tem a possibilidade de armazenar em cache as informações seja no cliente ou em servidores intermediários possibilitando maior número de respostas para requisições repetidas ou para carregar arquivos maiores como imagem e vídeo onde envia somente a URL ou URI de destino, usando o conceito de hipermídia (SERVICES, 2023b).
- **Código sobre demanda (*Code on demand*) :** No estilo REST, o servidor pode enviar código executável para o cliente, como *scripts JavaScript*. Esse código permite adicionar funcionalidades ao cliente sem alterar seu código fonte. Como exemplo, ao preencher um campo de formulário inválido, o servidor pode enviar um *script* para validar os dados no cliente. Essa lógica complementar é opcional e melhora a flexibilidade da aplicação ("*Status code*") (FIELDING, 2000).

2.3.1 Processo de Comunicação

- ***Endpoints* (Pontos de Extremidade):** Um endpoint se trata de um caminho ou local. Em uma API REST ele permite o acesso pelos desenvolvedores a funcionalidades e recursos que podem ser usados em outros sistemas e aplicações.
- **Solicitações do Cliente:** REST geralmente contém nas solicitações do cliente ("*Client Request*"):
  - **Identificador de Recursos ("Rotas"):** Geralmente representado pela URL, esse identificador indica qual recurso o cliente deseja solicitar para API retornar como resposta (SERVICES, 2023b).
  - **Métodos HTTP:** No contexto da arquitetura REST, os métodos HTTP permitem que o cliente diga ao servidor qual tipo de ação ele deseja realizar sobre um recurso. Em vez de comandos complexos, esses métodos funcionam como instruções simples e padronizadas, facilitando a comunicação entre as partes envolvidas. Entre os métodos mais utilizados, destacam-se cinco que são bastante comuns nesse tipo de interação (SERVICES, 2023b):
    - **GET**: Acessa recursos localizados na URL especificada no servidor. Eles podem armazenar em cache solicitações GET e enviar parâmetros na solicitação REST para instruir o servidor a filtrar dados antes de enviar.

    * **POST**: Utilizado para envio de dados ao servidor, inclui a representação de dados com a solicitação, pode ser enviado uma única vez, senão pode gerar o recurso solicitado várias vezes.
    * **PUT**: Usado para atualizar recursos existentes no servidor. Diferentemente do POST, ele pode repetir a solicitação, gerando o mesmo resultado.
    * **DELETE**: Os clientes usam a solicitação DELETE para remover o recurso. Uma solicitação DELETE pode alterar o estado do servidor. No entanto, se o usuário não tiver autenticação apropriada, a solicitação falhará.
  - **Cabeçalhos HTTP (Headers HTTP):** Os cabeçalhos (*Headers*) HTTP são informações adicionais enviadas com a solicitação ou resposta HTTP para fornecer contexto adicional e controlar a comunicação entre o cliente e o servidor.Os cabeçalhos (*Headers*) HTTP são informações adicionais enviadas com a solicitação ou resposta HTTP para fornecer contexto adicional e controlar a comunicação entre o cliente e o servidor.
    * **Data:** As solicitações da API REST. Podem incluir dados para que os métodos POST, PUT e outros HTTP funcionem com sucesso.
    * **Parâmetros (*Parameters*):** As solicitações da API REST. Incluem parâmetros que dão ao servidor mais detalhes sobre o que precisa ser feito. A seguir estão alguns tipos diferentes de parâmetros:
      · Parâmetros de caminho que especificam detalhes de URL comumente chamado de “ROTAS” (*Route*).
      · Parâmetros de consulta que solicitam mais informações sobre o recurso.
      · Parâmetros de cookies que autenticam clientes rapidamente.

- **Resposta do Servidor (*Server Response*):** Os princípios REST exigem que a resposta do servidor contenha os seguintes componentes principais:
  - **Linha de Status:** A linha de status contém um código de status de três dígitos que comunica o sucesso ou a falha da solicitação. Por exemplo, códigos 2XX indicam sucesso, enquanto códigos 4XX e 5XX indicam erros. Códigos 3XX indicam redirecionamento de URL.

    A seguir estão alguns códigos de status comuns:
    * **200**: Resposta genérica de sucesso
    * **201**: Resposta de sucesso do método POST
    * **400**: Solicitação incorreta que o servidor não pode processar

* **404**: Recurso não encontrado

– **Corpo da Mensagem**: O corpo da resposta contém a representação do recurso. O servidor seleciona um formato de representação apropriado com base nos cabeçalhos da solicitação. Os clientes podem solicitar informações nos formatos XML ou JSON, que definem como os dados são gravados em texto simples.

– **Cabeçalhos**: A resposta também contém cabeçalhos ou metadados sobre a resposta. Eles fornecem mais contexto sobre a resposta e incluem informações como servidor, codificação, data e tipo de conteúdo.

2.3.2 Notação de Objetos JavaScript(JSON)

O REST, utiliza comumente o padrão de comunicação em formato JSON onde tem a forma de notação por chave e valor , como demonstrado na Figura 3 abaixo:

Figura 3 – Funcionamento do Json em Rest.

```
1  {
2      "Person" : {
3
4        "id": 1,
5
6        "name": "Joaquim da Silva",
7
8        "age": 35,
9
10       "height": 1.67,
11
12       "married": true
13
14     }
15 }
```

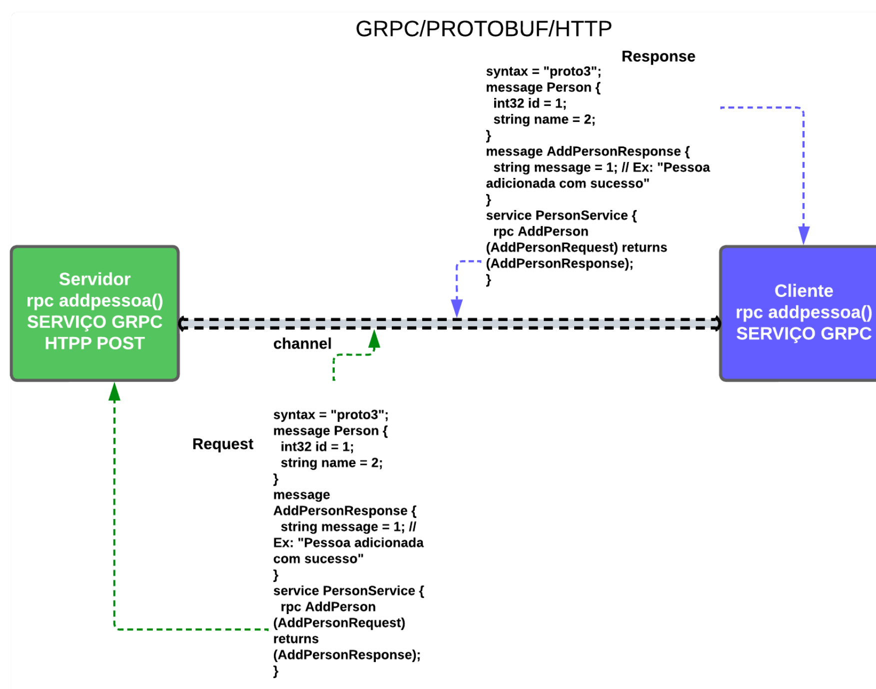
Fonte: Autoria própria (2025).

Neste exemplo, a estrutura de dados no formato JSON(*JavaScript Object Notation*), organiza eles em um par de chave e valor. A chave principal do objeto pessoa(*Person*), está a estrutura de atributos: identificador único da pessoa(“id”), com o valor um em número inteiro. O nome(*name*) da pessoa separada por um tipo texto(*string*) com o valor “Joaquim da Silva”. Idade(*age*), que representa o valor da idade da pessoa e armazena o valor trinta e cinco em número inteiro. Altura(*height*), em ponto flutuante(*float*) da pessoa, que é um e sessenta e sete. E a chave casada(*married*) com os valores verdadeiro(*true*) ou falso(*false*).

2.4 Chamada de Procedimento Remoto da Google (*Google Remote Procedure Call*) - gRPC

O grpc foi criado originalmente pelo Google e foi baseado em chamadas de procedimento remoto onde os métodos e serviços são definidos no arquivo com extensão *.proto*. A Figura4 apresentada ilustra a comunicação entre cliente e servidor utilizando a arquitetura gRPC, com mensagens definidas em Protobuf. Nesse exemplo, o cliente invoca remotamente o método addpessoa(), enviando uma estrutura Person contendo um identificador e um nome. Essa requisição é serializada em formato binário por meio do Protobuf e transmitida ao servidor por um canal. O servidor, recebe os dados, processa a requisição e retorna uma mensagem de resposta (AddPersonResponse) indicando o sucesso da operação (Google, 2025). Todo o fluxo ocorre sem a necessidade de endpoints como no REST. A Figura 4 também apresenta a definição dos serviços ou métodos de chamada de procedimento remoto (*Remote Procedure Call - RPC*) e das mensagens em, que servem como contrato entre as partes envolvidas.

Figura 4 – Representação da comunicação GRPC.



Fonte: Autoria própria (2025).

O gRPC é uma boa ferramenta para construir sistemas distribuídos, independentemente da plataforma ou linguagem utilizada. Ele é ideal para cenários como sistemas de baixa latência, desenvolvimento de clientes móveis. Sua flexibilidade e desempenho o tornam uma escolha sólida para uma variedade de casos de uso. Ele usa um modelo para definir as estruturas de dados em um arquivo “.proto”, usando o compilador de Protocol Buffer (chamado “protoc”) para

gerar classes de acesso aos dados em sua linguagem de programação preferida. Essas classes permitem que você preencha, serialize e recupere mensagens do tipo definido no “.proto”. Por exemplo, se você estiver usando C++, o “protoc” irá gerar uma classe correspondente ao seu arquivo “.proto”. Essa classe facilita a manipulação dos dados definidos no arquivo (YONG, 2023).

2.4.1 Chamada de Procedimento Remoto (*Remote Procedure Call - RPC*)

É uma tecnologia baseada em cliente servidor criada na década de 80 com objetivo de desenvolver aplicações, cliente-servidor sem a necessidade de programar diretamente na camada de transporte (FACOM, 2019). Seu conceito principal tenta abstrair as requisições e respostas, em procedimentos semelhantes a linguagens de programação de alto nível que abstrai suas funções em interfaces e fazem as chamadas quando necessário. “Sendo um dos seus objetivos principais possibilitar aos desenvolvedores fazerem chamadas semelhantes às outras funções e objetos comumente usados nas suas aplicações, porém ele depende que esse procedimento seja processado em outro ambiente podendo gerar atrasos e necessitando da resposta do servidor. Não podendo passar parâmetros como referência, pois esses métodos serão processados em outra máquina (MARIO, 2020).

Uma característica do RPC, é que o mesmo tem que prover uma Linguagem de Definição de Interface (IDL — Interface *Definition Language*). Ou seja, precisa ter uma interface detalhada que explica os procedimentos para o cliente informando seus tipos e dados de entradas e saídas, denominando se *Procedure Signatures* de suma importância para geração das *stubs* , sendo responsáveis pela complexidade (MARIO, 2020).

Os *stubs* são umas das partes mais importantes do RPC. Tendo a tarefa de isolar os detalhes de comunicação de rede do programador. Resumidamente, em vez de seu programa chamar uma função diretamente, ele usa um "cliente stub", na qual pega os valores que você quer enviar, coloca em um “pacote” com a identificação ou nome do método que você quer usar e manda ele para o servidor. Quando ele chega, um "organizador"identifica qual é o pedido e passa a tarefa para o "servidor stub". Ele pega seus valores, roda a função, pega o seu resultado e manda de volta, como se tivesse rodado ela localmente. (MARIO, 2020).

2.4.2 Buffer de Protocolo (*Protocol buffer*)

O gRPC utiliza uma definição de interface (IDL) chamada *Protocol Buffer* (Buffer de Protocolo). Diferente do JSON, que segue a convenção de chave e valor, o *Protocol Buffer* usa mensagens para encapsular atributos (variáveis ou definições de dados a serem armazenados) e seus tipos (atribuição do dado que irá receber, seja número, texto ou lógica booleana - verdadeiro ou falso). Os *Protocol Buffers* são mais eficientes e compactos do que o JSON, o que

os torna ideais para comunicação de alto desempenho. Eles permitem a definição de estruturas de dados, que podem ser serializadas e desserializadas rapidamente. Na Figura 5 abaixo, é demonstrado como definir uma classe (“forma de objeto que contém atributos”) Person (“Pessoa”) com seus atributos e tipos (YONG, 2023).

Figura 5 – Funcionamento da *message*(mensagem) em Protobuf.

```
1   syntax = "proto3";
2
3   message Person {
4
5       int32 id = 1;
6
7       string name = 2;
8
9       int32 age = 3;
10
11      float height = 4;
12
13      bool married = 5;
14
15  }
```

Fonte: Autoria própria (2025).

Neste exemplo, a mensagem *Person* possui cinco atributos: `id` (identificador), *name* (nome), *age* (idade), *height* (altura) e *married* (casado). Cada atributo é identificado por um número único, o qual é utilizado para a serialização e identificação dos dados. Os *Protocol Buffers* (Buffers de Protocolo) são empregados no gRPC para definir, de forma clara e padronizada, a estrutura das mensagens trocadas entre cliente e servidor, garantindo uma comunicação eficiente entre eles.

3 MATERIAIS E MÉTODOS

Neste capítulo, descrevem-se as tecnologias utilizadas e os procedimentos de implementação dos protótipos desenvolvidos para a comparação entre os modelos REST e gRPC. Ambos, compostos por cliente e servidor, foram desenvolvidos no mesmo ambiente de programação, diferenciando-se apenas quanto às bibliotecas e estruturas específicas de cada arquitetura. Os projetos tiveram como finalidade o envio e recebimento de arquivos de diferentes tamanhos (pequenos e grandes), com a estimativa do tempo de transferência de cada operação. Os dados coletados foram armazenados em arquivos no formato `.csv`, os quais serviram de base para esse experimento.

3.1 Descrição Geral

3.1.1 Clientes de Testes

Os clientes foram desenvolvidos visando permitir a seleção de arquivos para upload. Após a seleção, os arquivos são enviados aos respectivos servidores (REST e gRPC), sendo contabilizado o tempo desde o início da transmissão até o recebimento da resposta do servidor.

3.1.2 Servidores

Implementou-se dois servidores distintos para o recebimento de arquivos:

- **Servidor REST**: Responsável por receber os arquivos por meio de uma API REST e confirmar ao cliente o recebimento da requisição.
- **Servidor gRPC**: Utiliza o protocolo gRPC para o recebimento de arquivos via chamadas de procedimento remoto, retornando ao destino a confirmação da entrega.

3.1.3 Processo de Upload

1. Os clientes de testes de ambas tecnologias envia os arquivos selecionados para os seus respectivos servidores(REST e gRPC), registrando o tempo de envio e o tamanho dos arquivos.
2. Cada servidor processa o envio(*upload*) e salva os arquivos conforme especificado.
3. Após o processamento, os servidores retornam ao cliente o tempo de término do *upload* e o tamanho dos arquivos que trafegaram pela requisição.

3.1.4 Ambiente de Desenvolvimento Comum

Os projetos desenvolvidos, tanto para REST quanto para gRPC, utilizam a seguinte base tecnológica:

- **Linguagem:** Java 8+;
- **Gerenciador de dependências:** Maven;
- **IDE:** IntelliJ IDEA;
- **Organização de pacotes:** conforme estrutura padrão do Maven, com `src/main/java` para código-fonte, `src/test` para testes e `src/main/resources` para recursos.

3.1.5 Máquina Cliente de teste

A máquina de teste utilizada para envio dos arquivos:

- **Sistema operacional:** Windows 11;
- **Memória ram:** 16GB;
- **Processador:** 10 núcleo de processador;
- **Linguagem de programação:** Java 8+

3.1.6 Servidor de teste

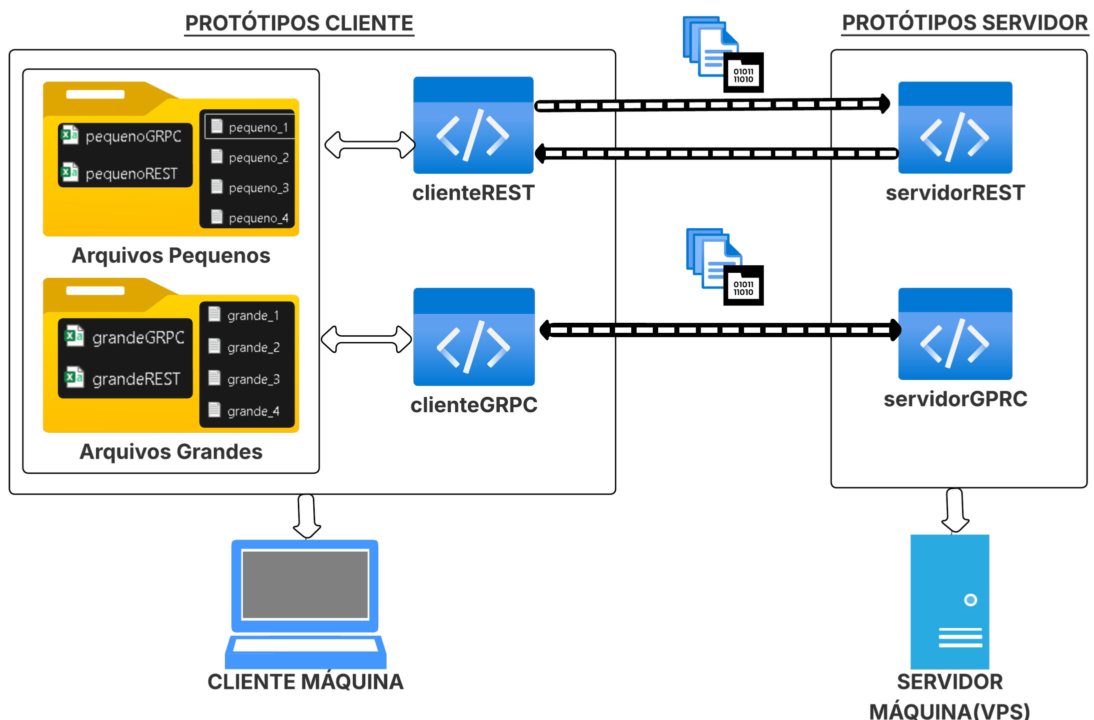
Os testes foram realizados em um servidor virtual privado (VPS), configurado com os seguintes recursos:

- **Sistema operacional:** Ubuntu 22.04;
- **Memória ram:** 2GB;
- **Processador:** 2 núcleo de processador;
- **Linguagem de programação:** Java 8+

3.1.7 Arquitetura Geral dos Protótipos

A Figura 6 apresenta a arquitetura geral desenvolvida para os protótipos REST e gRPC. Ambos seguem uma lógica semelhante de comunicação entre cliente e servidor, diferenciando-se nas tecnologias empregadas para o transporte e serialização dos dados.

Figura 6 – Arquitetura dos protótipos desenvolvidos



Fonte: Elaborado pelo autor.

3.2 Protótipo gRPC

3.2.1 Definição do Protocol Buffers (Protobuf)

O meio de comunicação adotado no protótipo gRPC baseia-se no uso do *Protocol Buffers* (Protobuf), responsável por estruturar e serializar as mensagens trocadas entre cliente e servidor. Os arquivos de definição, com extensão *.proto*, estão localizados na pasta *resources* e definem os seguintes elementos:

- **Mensagens (message):**
  - `File`: representa o conteúdo do arquivo em bytes;

- `FileUploadRequest`: encapsula o arquivo e seus metadados;
- `FileUploadResponse`: contém o nome do arquivo e o status de envio;
- `FileNameWithType`: identifica o nome e a extensão do arquivo.

- **Enumeração (enum):**
  - `UploadStatus`: define os estados possíveis da operação (`SUCCESS` ou `FAILED`).

- **Serviço RPC (service):**
  - `uploadFileUnary`: método remoto responsável pelo envio de arquivos em chamada única.

3.2.2 Tecnologias Utilizadas no *cliente-gRPC* e *servidor-gRPC*

Ambos os protótipos na linguagem de programação *Java* e com as seguintes dependências, especificadas no arquivo `pom.xml`:

- `grpc-netty-shaded`;
- `grpc-protobuf`;
- `grpc-stub`;
- `protobuf-java`;
- `protobuf-java-util`.

3.2.3 Estrutura do Projeto – *cliente-gRPC*

A aplicação cliente foi organizada de forma modular, com os seguintes pacotes e responsabilidades:

- `com.utfpr.service`: contém a classe `HttpUploadService`, responsável pela comunicação com o servidor gRPC;
- `com.utfpr.utils`: agrupa classes utilitárias como `CSVWriter` e `MultipartFile`;
- `com.utfpr.MainUpload`: ponto de entrada da aplicação.

3.2.4 Detalhes da Implementação – *cliente-gRPC*

A classe `HttpUploadService` estabelece o canal de comunicação com o servidor utilizando a interface `FileUploadServiceBlockingStub`. A requisição é construída com base nos dados do arquivo e enviada por meio do método remoto `uploadFileUnary`. A resposta é recebida e interpretada de forma síncrona.

A classe `MainUpload` é responsável por orquestrar o envio dos arquivos. Os parâmetros fornecidos ao programa incluem:

- Caminho dos arquivos (`path`);
- Nome base do arquivo (`fileBaseName`);
- Tipo de conteúdo/extensão (`contentType`);
- Quantidade de execuções (`MAX`).

A aplicação calcula o tempo de envio, registra os resultados em arquivos CSV e finaliza a execução após o processamento de todos os arquivos.

3.2.5 Estrutura do Projeto – *servidor-gRPC*

O servidor foi estruturado com base nas seguintes configurações:

- **Grupo identificador (groupId):** `com.utfpr`;
- **Artefato identificador (artifactId):** `grpcServer`;
- **Diretórios:** `src/main/java`, `src/test`, e `src/main/resources` (onde está localizado o arquivo `upload.proto`);
- **Pacotes lógicos:**
  - `com.utfpr.grpc.upload`: contém as classes geradas a partir da compilação do arquivo `.proto`;
  - `com.services.upload`: implementa a lógica do serviço gRPC, incluindo o processamento das requisições.

3.2.6 Detalhes da Implementação – *servidor-gRPC*

A classe principal do servidor é a `FileUploadService`, que implementa o método `uploadUnary()` definido no arquivo `upload.proto`. Esse método atua como ponto de entrada remoto e é responsável por:

- Receber os metadados do arquivo (nome, tipo e tamanho);
- Processar o conteúdo do arquivo em bytes;
- Retornar o status da operação por meio do `responseObserver`, utilizando os valores da enumeração `UploadStatus`.

A estrutura garante que a comunicação ocorra conforme os padrões definidos pelo Protobuf.

3.3 Protótipo REST

3.3.1 Tecnologias Utilizadas – *cliente-REST*

O cliente REST foi implementado utilizando a biblioteca `OkHttp` (`com.squareup.okhttp.*`), utilizada no envio de requisições HTTP em aplicações javas.

3.3.2 Estrutura do Projeto – *cliente-REST*

A organização do projeto cliente segue os seguintes pacotes e funções:

- `com.utfpr.service`: contém a classe `HttpUploadService`, responsável pelas requisições HTTP;
- `com.utfpr.MainUpload`: ponto de entrada do sistema, responsável pela execução do envio de arquivos.

3.3.3 Detalhes da Implementação – *cliente-REST*

A operação de envio via REST é realizada pelas seguintes classes:

- **HttpUploadService**: define os parâmetros de tempo limite (timeout) para conexão, leitura e escrita, além de construir e executar as requisições HTTP do tipo POST com os arquivos;
- **MainUpload**: inicia o processo de envio, calcula o tempo de resposta e armazena os resultados em arquivo.

3.3.4 Tecnologias Utilizadas – *servidor-REST*

O servidor REST foi desenvolvido utilizando o framework *Spring Boot*, com as seguintes dependências principais:

- `spring-boot-starter-web`;
- `spring-boot-devtools`.

3.3.5 Estrutura do Projeto – *servidor-REST*

A estrutura do servidor contempla:

- Rota de recebimento: `/upload`;
- Pacote principal: `br.com.utfpr.controllers`;
- Classe controladora: `FileUploadController`, anotada com:
  - `@RestController`;
  - `@PostMapping`: define o método de recebimento;
  - `@RequestMapping`: configura a rota base.

3.3.6 Detalhes da Implementação – *servidor-REST*

O método de upload do servidor é responsável por:

- Confirmar o recebimento com um status HTTP apropriado;
- **Observação:** os arquivos não são armazenados em disco, pois o objetivo do experimento é exclusivamente medir o tempo de envio e resposta.

3.4 Captura dos dados

Para registrar os resultados, foi elaborada uma classe utilitária denominada `CSVWriter`, responsável pela geração de arquivos no formato `.csv`. Essa classe realiza a gravação estruturada dos dados em colunas específicas, sendo elas: `File` (nome do arquivo), `Size` (tamanho do arquivo em bytes) e `Millis` (duração da operação em milissegundos).

A `CSVWriter` recebe como parâmetros o *caminho de destino* e o *nome do arquivo de saída*. A estrutura dos dados registrados baseia-se na classe `Upload`, localizada no pacote `com.utfpr.model`, a qual encapsula os atributos `filename`, `duration` e `size`.

A interação entre essas duas classes permite que os resultados sejam exportados de maneira padronizada, o que facilitou a análise.

3.5 Métricas Avaliadas

A análise dos resultados considerou as seguintes métricas:

- **Tempo médio de envio em milissegundos**;
- **Tempo mínimo e máximo de envio em milissegundos**;

A ferramenta R foi utilizada para importar os arquivos `.csv`, calcular as estatísticas e gerar os gráficos apresentados na Seção 4.

3.6 Execução dos Testes

Para o experimento comparativo entre as arquiteturas REST e gRPC, foram coletadas tempo de envio entre a máquina local(cliente) e servidor virtual privado("*Virtual Private Server*") em milissegundos de arquivos com variados tamanhos: pequenos de 2 megabytes á 20 megabytes, como também grandes de 20 megabytes á 50 megabytes, por meio da execução dos protótipos desenvolvidos. Com algumas execuções para cada cenário. Os resultados foram organizados conforme referenciado na Seção 3.4.

4 RESULTADOS

Este capítulo apresenta os resultados obtidos com os testes realizados a partir dos protótipos REST e gRPC e parecer comparativo entre eles.

4.1 Apresentação das informações

A Tabela 1 apresenta os resultados obtidos com os testes de envio de arquivos utilizando as arquiteturas REST e gRPC. Observa-se que o gRPC apresentou desempenho um pouco relevante no envio de arquivos pequenos, com tempos médios um pouco inferior e menor variação nos valores mínimo e máximo.

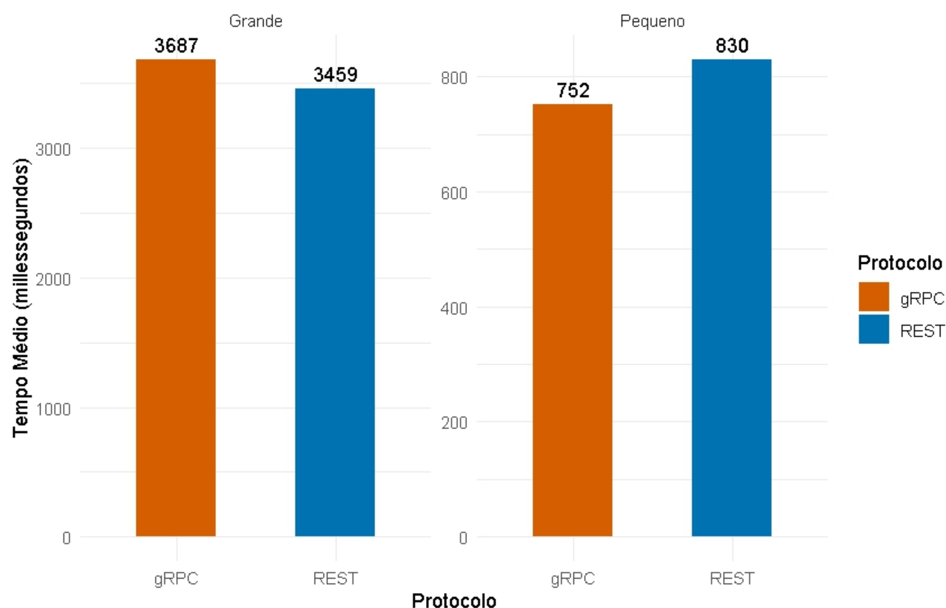
Tabela 1 – Tempo de Envio: REST vs gRPC por Tipo de Arquivo

Tipo de Arquivo	Protocolo	Tempo Médio (ms)	Mínimo (ms)	Máximo (ms)
Pequeno	REST	830,96	263	3271
Pequeno	gRPC	752,27	212	3290
Grande	REST	3459,77	1433	7324
Grande	gRPC	3687,29	1805	8887

Além da tabela, foi elaborado um gráfico para representar visualmente os resultados obtidos nos testes de envio de arquivos por meio dos protocolos REST e gRPC. A Figura 7 apresenta, para cada cenário avaliado, o tempo médio de envio.

Essa visualização facilita a comparação entre os protocolos ao mostrar, de forma, que o gRPC apresenta tempos médios inferiores ao REST, em arquivos pequenos.

Figura 7 – Tempo Médio – REST vs gRPC (Arquivos Pequenos e Grandes)



Fonte: Autoria própria (2025).

4.2 Parecer Comparativo

Com base nos dados obtidos nos experimentos, observa-se uma sutil vantagem do protocolo gRPC em relação ao REST.

O tempo médio de envio utilizando gRPC foi inferior ao registrado com REST em um dos cenários, com destaque para o envio de arquivos pequenos, onde a diferença foi de aproximadamente 48 milissegundos.

Contudo, o protocolo REST ainda se mostra viável em cenários onde:

- A integração com diferentes sistemas e linguagens são necessários;
- O volume de dados é reduzido;
- A simplicidade de implementação é mais relevante que o desempenho.

O gRPC, por outro lado, é mais indicado para aplicações que:

- Demandam alta performance e baixa latência;
- Trabalham com grande volume de dados;
- Estão integradas em ambientes com suporte a microsserviços e comunicação assíncrona.

Portanto, conclui-se que, embora ambos os protocolos sejam funcionais, o gRPC apresenta pouca vantagem. A escolha ideal deve considerar o contexto específico da aplicação, suas restrições técnicas e seus objetivos operacionais.

5 CONCLUSÃO

Este experimento comparativo teve como finalidade comparar duas abordagens comumente utilizadas para a construção de Web APIs: REST (Representational State Transfer) e gRPC (gRPC Remote Procedure Call). Através da criação de protótipos, implementação deles e da realização de testes entre eles, foi possível identificar as características específicas de cada abordagem.

REST tem como pontos fortes a simplicidade e a facilidade de implementação e ampla compatibilidade com diversos ambientes e facilidade de implementação, o que o torna uma escolha prática para projetos com equipes menos especializadas ou demandas de rápida entrega. Seu uso de formatos textuais como JSON e a comunicação sobre HTTP facilitam o desenvolvimento e a manutenção, além de contar com ampla documentação e suporte da comunidade. Por outro lado, o gRPC apresentou leve superioridade em aspectos como desempenho. Contudo, seu uso requer maior conhecimento técnico e enfrenta barreiras como uma curva de aprendizado acentuada e menor flexibilidade em ambientes diferentes.

Durante o desenvolvimento deste experimento, algumas dificuldades se destacaram, como a complexidade inicial da tecnologia gRPC e do uso do Protocol Buffers. A definição de métricas de avaliação que fossem relevantes e aplicáveis a ambas tecnologias. Entretanto, a familiaridade prévia com Java e o uso de APIs REST, a documentação clara do gRPC e a possibilidade de reutilização de partes do código entre os protótipos contribuiu para a elaboração do experimento.

Conclui-se, portanto, que não há uma solução universalmente superior entre REST e gRPC. A escolha entre as duas arquiteturas deve ser guiada pelos requisitos técnicos do sistema, pela infraestrutura disponível, pelo perfil da equipe de desenvolvimento e pelos objetivos do projeto.

Como sugestão para trabalhos futuros, propõe-se a ampliação deste experimento com a inclusão de critérios como integração com sistemas legados e testes em ambientes com diferentes restrições de rede.

REFERÊNCIAS

ARAúJO, R. **API: Teoria e questões**. [*S.l.*], 2021. 1 p. Disponível em: https://blog.grancursosonline.com.br/api-teoria-e-questoes/.

COOKSEY, B. **AN INDROUCTION API**. 2023. 6-7 p. Disponível em: https://cdn.zapier.com/storage/learn_ebooks/e06a35cfcf092ec6dd22670383d9fd12.pdf.

FACOM. **RPC - Remote Procedure Call**. 2019. 25 p. Disponível em: https://www.facom.ufu.br/~faina/MCC_Crs/SOD_2S01/DwLd_SOD/rpc-prt1.pdf.gz.

FIELDING, R. T. **REST: architectural styles and the design of network-based software architectures**. 2000.

FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern web architecture. **ACM Transactions on Internet Technology**, v. 2, n. 2, p. 115–150, maio 2002. Disponível em: http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf.

Google. **gRPC Overview**. 2025. https://grpc.io/docs/. Acesso em: 9 jul. 2025.

HORVAT, Z.; ILIC, V.; NIKOLIC, M. **Web Server and QR Decoder Applications for Xilinx FPGA Boards**. 2018.

MARINI, T. dos S.; FIGUEIREDO, J. A. O. de; ROSSETTO, A. G. de M. **Estudo Comparativo da Comunicação de Dados em Dispositivos Móveis: Métodos Web services e Sockets**. 2013. 46–46 p.

MARIO. **Remote Procedure Calls**. 2020. 13 p. Disponível em: http://www.deinf.ufma.br/~mario/grad/redes2/rpc_cap.pdf.

MDN Web Docs. **Introdução às Web APIs**. 2024. Disponível em: https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Extensions/Client-side_APIs/Introduction.

Microsoft Corporation. **Lista de API do Windows**. 2024. https://learn.microsoft.com/pt-br/windows/win32/apiindex/windows-api-list. Acessado em 20 de julho de 2024.

RIBEIRO, M.; FRANCISCO, R. **Web Services REST Conceitos, análise e implementação**. 2016.

SERVICES, A. W. **gRPC vs REST: Understand the Differences**. 2023. Acesso em: 08 ago. 2024. Disponível em: https://aws.amazon.com/pt/compare/the-difference-between-grpc-and-rest/.

SERVICES, A. W. **O que é uma API RESTful?** 2023. Disponível em: https://aws.amazon.com/what-is/restful-api/.

YONG, T. Z. **Introduction to gRPC**. 2023. 1 p. Disponível em: https://grpc.io/docs/what-is-grpc/introduction/.