Go x Java e gRPC x REST: Um estudo empírico

Luiz Otávio Soares*
luiz.soares[at]estudante.ufla.br
Universidade Federal de Lavras
Lavras, MG, Brasil

Ricardo Terra*
terra[at]ufla.br
Universidade Federal de Lavras
Lavras, MG, Brasil

Resumo

Este trabalho investiga a influência das linguagens de programação Java e Go e das arquiteturas de comunicação REST e gRPC no desempenho de APIs. A questão de pesquisa #1 investigou qual par *(linguagem,arquitetura-comunicação)* provê melhor desempenho. Por um lado, pares (`Go`, `gRPC`) e (`Java`, `gRPC`) se destacaram em requisições de tamanhos usuais (*StdSize*) possivelmente devido à eficiência na compactação de dados e na redução de latência oferecida pelo gRPC. Por outro lado, pares (`Go`, `REST`) e (`Java`, `REST`) se destacaram em requisições de grande volume (*LargeSize*) provavelmente devido à flexibilidade do REST em lidar com grandes volumes de dados sem estrutura rígida. A questão de pesquisa #2, de forma complementar, investigou a influência de cada fator no desempenho. Um projeto fatorial $2^k r$ concluiu que enquanto a linguagem de programação exerce uma influência de 2%, a arquitetura de comunicação influencia 94%.

CCS Concepts: • **Software and its engineering → Interoperability**; • **General and reference** → *Empirical studies*; *Measurement*.

Keywords: Linguagens de programação, estilo arquitetural de comunicação, estudo empírico

1 Introdução

As APIs (*Application Programming Interfaces*) desempenham um papel crucial no desenvolvimento de sistemas, facilitando a integração e comunicação entre diferentes sistemas de software e plataformas, permitindo assim uma maior modularidade, escalabilidade e reutilização de código [17]. Na prática usual adotada por fábricas de software, utiliza-se o protocolo HTTPS (*Hypertext Transfer Protocol Secure*) para facilitar a comunicação e transferência de dados na Internet e o estilo arquitetural de comunicação REST (*Representational State Transfer*) para organizar e acessar serviços web de maneira padronizada [10; 11]. Isso se consolidou como o padrão usual para a criação de sistemas distribuídos robustos [4; 17; 18].

A ascensão de Go e gRPC, promovida significativamente pelo Google, marca uma evolução nas tecnologias de desenvolvimento de software [9; 12]. Go destaca-se pela sua simplicidade e eficiência no manejo de operações concorrentes, enquanto gRPC revitaliza o conceito de chamadas de procedimento remoto (RPC) ao integrar funcionalidades do HTTP/2, facilitando comunicações rápidas e eficientes por meio de *Protocol Buffers* para serialização de dados. Essas características são particularmente benéficas para microsserviços e aplicações que demandam alto desempenho e baixa latência.

Este artigo, portanto, conduz um estudo empírico que emprega procedimentos estatísticos rigorosos para quantificar as melhorias oferecidas por uma linguagem de programação e um estilo arquitetural de comunicação considerados promissores. Partindo da presunção de que Java é uma linguagem consolidada no mercado, assim como REST é quase sempre adotado como o estilo arquitetural de comunicação, este estudo (i) investiga a combinação mais eficiente entre as linguagens de programação Go e Java e os estilos arquiteturais de comunicação gRPC e REST e (ii) quantifica a influência da linguagem de programação e do estilo arquitetural de comunicação na eficiência.

A partir de quatro implementações – (`Go`, `gRPC`), (`Go`, `REST`), (`Java`, `gRPC`) e (`Java`, `REST`) – de uma mesma aplicação, a questão de pesquisa #1 investigou qual par *(linguagem,arquitetura-comunicação)* provê melhor desempenho e comprovou diferença estatística entre todos os pares. Além disso, as análises indicaram que os pares (`Go`, `gRPC`) e (`Java`, `gRPC`) se destacam em requisições de tamanhos usuais (*StdSize*) possivelmente devido à eficiência na compactação de dados e na redução de latência oferecida pelo gRPC. Por outro lado, pares (`Go`, `REST`) e (`Java`, `REST`) se destacaram em requisições de grande volume (*LargeSize*) provavelmente devido à flexibilidade do REST em lidar com grandes volumes de dados sem estrutura rígida. A questão de pesquisa #2, de forma complementar, investigou a influência de cada fator no desempenho por meio de um projeto fatorial $2^k r$. Essa análise concluiu que a linguagem de programação exerce uma influência de 2%, enquanto que a arquitetura de comunicação influencia 94%.

Este artigo está organizado como a seguir. A Seção 2 provê uma visão geral das linguagens de programação, dos estilos arquiteturais de comunicação e dos métodos estatísticos utilizados. A Seção 3 descreve o estudo empírico incluindo questões de pesquisa, metodologia, coleta de dados, análise de desempenho e influência dos fatores. A Seção 4 apresenta trabalhos relacionados e a Seção 5 conclui e enumera trabalhos futuros.

2 *Background*

Esta seção introduz conceitos fundamentais ao artigo.

2.1 Linguagens de Programação

Go, também conhecida como *Golang*, é uma linguagem de programação lançada publicamente pelo Google em 2009. Projetada por Robert Griesemer, Rob Pike e Ken Thompson, Go visa simplificar o desenvolvimento de software com suporte robusto para programação concorrente e gerenciamento eficiente de memória. A linguagem oferece uma sintaxe limpa e recursos como *goroutines* e *channels*, que facilitam a construção de sistemas distribuídos escaláveis. A capacidade de Go de compilar rapidamente e sua eficiência em tempo de execução a tornam ideal para a criação de serviços de alto desempenho [9] o que justifica a sua ampla adoção da indústria [6].

Java é uma linguagem de programação de propósito geral que é amplamente utilizada para desenvolvimento de software [25]. Criada por James Gosling e lançada pela *Sun Microsystems* em 1995, Java é conhecida por seu princípio de *Write Once, Run Anywhere* (WORA), que permite que o código Java seja executado em qualquer dispositivo que suporte a *Java Virtual Machine* (JVM) [23]. De acordo com o índice TIOBE, que mede a popularidade das linguagens de programação com base em diversos fatores, Java frequentemente aparece entre as linguagens mais utilizadas, destacando sua relevância contínua na indústria [25].

2.2 Estilos arquiteturais de comunicação

O HTTP é o protocolo subjacente usado pela *World Wide Web* para a troca de informações entre navegadores e servidores. O HTTP é um protocolo de aplicação que segue um modelo de requisição-resposta. Um cliente envia uma requisição HTTP para um servidor, que então responde com uma mensagem de *status* e os dados solicitados. Além disso, a simplicidade do HTTP permitiu sua adoção generalizada, e a introdução de HTTP/2 trouxe melhorias significativas em termos de desempenho, incluindo multiplexação de requisições e compressão de cabeçalhos [2; 10].

A arquitetura REST é um estilo de arquitetura de comunicação para sistemas distribuídos, como a *World Wide Web* [11]. Proposta por Roy Fielding em sua tese de doutorado em 2000, REST define um conjunto de restrições que resultam em um sistema escalável, de fácil manutenção e eficiente. As APIs construídas sob a arquitetura REST utilizam métodos HTTP (como GET, POST, PUT e DELETE) para manipular recursos, que são identificados por URIs (*Uniform Resource Identifiers*). Um dos aspectos chave do REST é o uso de JSON (*JavaScript Object Notation*) como formato padrão para a troca de dados entre clientes e servidores [7]. A simplicidade e a padronização de REST têm contribuído para sua ampla adoção em diversas aplicações web [4; 17; 18].

RPC (*Remote Procedure Call*) é um estilo arquitetural de comunicação que permite a um programa executar procedimentos em outro espaço de endereço, geralmente em outro computador na rede, como se fossem chamadas locais [3]. De forma sucinta, RPC abstrai os detalhes da rede, proporcionando uma interface mais simples para os desenvolvedores. O gRPC é uma evolução moderna do RPC desenvolvida pelo Google. Utilizando HTTP/2 como protocolo de transporte, gRPC oferece vantagens como suporte a comunicação bidirecional, multiplexação de requisições e maior eficiência na transmissão de dados [12].

2.3 Procedimentos Estatísticos

Este trabalho emprega uma combinação de testes estatísticos não paramétricos e métodos de análise de variância para explorar as influências significativas e interações entre as variáveis estudadas.

Teste de Shapiro-Wilk. É um método estatístico usado para verificar se uma amostra provém de uma distribuição normal. Esse teste compara a ordem dos dados da amostra com uma distribuição normal esperada, utilizando uma estatística de teste W. Se W está próximo de 1, sugere-se que a amostra pode ser considerada normalmente distribuída. Essencial em pesquisas que dependem de análises paramétricas, o teste de Shapiro-Wilk é particularmente eficaz para amostras pequenas, embora sua sensibilidade possa ser uma limitação para amostras grandes, onde pequenos desvios de normalidade podem levar à rejeição da hipótese nula [20].

Teste de Wilcoxon. É um teste não paramétrico utilizado para comparar duas amostras emparelhadas, a fim de determinar se suas distribuições populacionais diferem significativamente. Para amostras não pareadas utiliza-se a versão *Rank-Sum* do teste [8]. É particularmente útil quando a distribuição dos dados não pode ser assumida como normal, uma condição frequentemente encontrada em dados de desempenho de software. Esse teste foi aplicado para analisar as diferenças de desempenho entre as implementações de REST e gRPC [21].

Teste de Kruskal-Wallis. É um teste empregado quando se deseja comparar três ou mais grupos independentes sem assumir uma distribuição normal dos dados. Esse teste é essencial para avaliar a variação no desempenho entre diferentes configurações linguísticas e arquitetônicas, oferecendo uma análise robusta que complementa as comparações bivariadas do teste de Wilcoxon. Os resultados obtidos fornecem *insights* sobre quais configurações oferecem melhor desempenho em termos de eficiência e velocidade [13].

Projeto fatorial $2^k r$. É um projeto conduzido para a análise de dois ou mais fatores de forma simultânea e seus efeitos interativos sobre a variável resposta. No contexto deste estudo, as variáveis analisadas incluem as linguagens de programação Go e Java, e as arquiteturas de comunicação gRPC e REST, com repetições (r) para garantir a precisão dos resultados. Ele não apenas identifica as configurações mais eficientes, como também revela como essas escolhas interagem entre si para influenciar o desempenho geral [15].

3 Estudo Empírico

Esta seção descreve o estudo empírico realizado para investigar o desempenho de aplicações utilizando as linguagens de programação Go e Java em conjunto com os estilos arquiteturais de comunicação gRPC e REST.

3.1 Questões de Pesquisa

(QP#1) **Qual é a combinação mais eficiente entre as linguagens de programação Go e Java e os estilos arquiteturais de comunicação gRPC e REST para otimizar o desempenho de aplicações?**
Essa questão busca determinar qual par de linguagem e arquitetura de comunicação – (Go, gRPC), (Go, REST), (Java, gRPC) e (Java, REST) – oferece a melhor eficiência em termos de desempenho, em diferentes cenários de aplicação.

(QP#2) **Quanto cada tecnologia, seja linguagem (Go e Java) ou estilo arquitetural de comunicação (gRPC e REST) influencia no tempo de execução das aplicações?**
O objetivo dessa questão é mensurar o quanto as escolhas de linguagem de programação e estilo arquitetural de comunicação impactam no desempenho das aplicações.

Essas questões são fundamentais para compreender as nuances de desempenho, o que pode ajudar desenvolvedores a fazer escolhas corretas para seus projetos.

3.2 Metodologia

Conforme ilustrado na Figura 1, o procedimento metodológico adotado pode ser descrito em cinco etapas:

1. Desenvolvimento de quatro implementações de uma mesma aplicação que referem-se às combinações das linguagens de programação e estilos arquiteturais de comunicação empregados neste estudo.
   Resultado esperado: implementações (Go, gRPC), (Go, REST), (Java, gRPC) *e* (Java, REST).
2. Condução de um experimento preliminar para estimar o tamanho de amostra ideal.
   Resultado esperado: obtenção do n ideal.
3. Execução n vezes de cada uma das implementações com requisições de tamanho usual (*StdSize*) e de grande volume (*LargeSize*). Cada execução é repetida r vezes.
   Resultado esperado: oito arquivos – quatro implementações x dois tamanhos de requisição – com n tempos de execução. Cada arquivo contendo r conjuntos de observações.
4. Aplicação do teste de Shapiro-Wilk para verificar a normalidade das distribuições. Caso os dados não sejam normais, (i) aplicação do teste de Wilcoxon *Rank-Sum* para verificar se os tempos de execução de cada uma das implementações são estatisticamente diferentes dentro do mesmo tamanho de requisição e (ii) aplicação do teste de Kruskal-Wallis para avaliar a variação no desempenho entre diferentes implementações, o que complementa as comparações bivariadas do teste de Wilcoxon.
   Resultado esperado: ranking *das implementações com validade estatística.*
5. Condução do projeto fatorial $2^k r$ para determinar a influência dos fatores e suas interações sobre o desempenho das implementações.
   Resultado esperado: percentual de influência de cada fator.

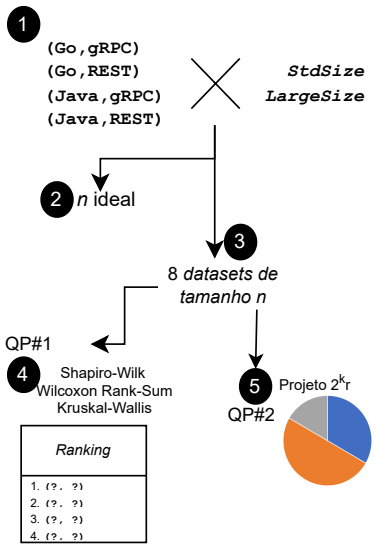

Figura 1. Metodologia

3.3 Ambiente

O ambiente experimental foi configurado de forma a promover o menor erro experimental possível:

Hardware. Intel Core i7 de 9ª geração e 16 GB de RAM.

Sistema operacional. Ubuntu Server com instalação mínima, garantindo que nenhuma outra aplicação interferisse nos testes, proporcionando um ambiente controlado e consistente.

Software. apenas Java 17 e Go 1.16.

Servidor de aplicação. Para Java, utilizou-se o Tomcat 10.1.8 para a implementação REST e gRPC Netty 1.56.1 para gRPC. Go, por outro lado, já possui funcionalidades embutidas para servir HTTP no pacote `net/http`, utilizado em ambas as implementações.

Coleta. Durante a coleta, a aplicação alvo, seja qual for a implementação, foi configurada para rodar com a máxima prioridade (*nice level* de -20), enquanto a aplicação cliente foi configurada com um *nice level* de -19.

3.4 Aplicação Alvo

No âmbito desse experimento, tem-se quatro implementações distintas de uma mesma aplicação, cada uma representando uma combinação única de linguagem de programação e estilo arquitetural de comunicação: (Go, gRPC), (Go, REST), (Java, gRPC) e (Java, REST). O propósito central da aplicação é a implementação de um único *endpoint*, o qual tem a função crítica de receber um conjunto de números e, a partir desse conjunto, selecionar e exibir três elementos em posições aleatórias. Essa metodologia deliberadamente simplificada foi adotada com o objetivo de isolar e quantificar com precisão o impacto exercido tanto pela linguagem de programação quanto pelo estilo arquitetural de comunicação sobre o desempenho geral do sistema. Essa abordagem reduz significativamente a possibilidade de interferência por variáveis externas ao experimento.

Além disso, a operação selecionada para ser realizada pela aplicação foi cuidadosamente escolhida para assegurar que o *payload* completo seja efetivamente transferido para o serviço em questão, prevenindo assim qualquer forma de otimização prematura na leitura e processamento dos dados. Essa premissa é de vital importância para o estudo, uma vez que uma parcela substancial das discrepâncias observadas entre os diferentes estilos arquiteturais advém especificamente da maneira como os dados são encapsulados e transportados por meio da rede. Em consequência, é imperativo avaliar com minúcia o impacto que a transformação desses dados, de sua forma bruta para um formato que seja manejável pela linguagem de programação, tem sobre o desempenho geral.

Uma aplicação é responsável por orquestrar os testes e coletar dados de desempenho. São feitas n requisições para cada um dos seguintes tipos de *payloads*:

O *payload* "*StdSize*" compreende um arranjo de 200 números inteiros com valores aleatórios entre 0 e 1.000, totalizando 0,763 KB. Esse tamanho é representativo do tamanho médio de requisições HTTP comumente observadas em aplicações web, servindo como um *benchmark* relevante para análises de desempenho [24]. O *payload* "*LargeSize*" é concebido para ser exatamente 1.024 vezes maior que o "*StdSize*", refletindo uma carga significativamente mais pesada. Isso eleva o tamanho do *payload* "*LargeSize*" para 781,312 KB.

Essa escala ampliada é crucial para testar o desempenho das implementações sob condições extremas de carga, permitindo uma investigação sobre como diferentes tamanhos de *payload* afetam o desempenho nas aplicações distribuídas.

3.5 Coleta de Dados

O procedimento experimental é delineado em $r = 5$ repetições, cada uma contendo $n = 77$ requisições. Utilizou-se uma fórmula que permitiu definir o número de requisições ideal para a amostra com base em um conjunto de observações preliminares. Considerando um nível de confiança de 95% e 9 graus de liberdade (10 observações preliminares), a variável z tem o valor de 1,833 e representa o valor crítico da distribuição t de *Student* para esse nível de confiança e graus de liberdade especificados. O parâmetro r é definido como o erro tolerável, fixado em 5%. A fórmula geral para o cálculo do tamanho da amostra (sz) é dada por [15]:

$$sz = \left(\frac{100 \cdot z \cdot s}{r \cdot \bar{x}} \right)^2 \tag{1}$$

onde:

- s é o desvio padrão das medições prévias,
- $\bar{x}$ é a média das medições prévias,
- z é o valor t de *Student* para o nível de confiança e
- r é o erro percentual tolerável.

Durante cada repetição, a aplicação calcula o tempo despendido de cada requisição e as armazena em memória. Esse processo é repetido até que todas as repetições previstas sejam concluídas. Na fase final, esses dados são exportados para oito arquivos com n tempos de execução, um arquivo para cada uma das quatro implementações e para cada um dos dois tamanhos de requisição, e importados no R para análises estatísticas [22].

3.6 Análise de Desempenho (*QP#1*)

Esta seção provê respostas à questão de pesquisa #1 por meio da aplicação de procedimentos estatísticos em uma das cinco repetições dos dados coletados conforme descrito na Seção 3.5.

A Figura 2 ilustra um gráfico *boxplot* comparando os tempos de execução das quatro implementações (Go, gRPC), (Go, REST), (Java, gRPC) e (Java, REST) no cenário de requisições *StdSize*. Cada caixa representa a distribuição dos tempos de resposta de diferentes configurações de aplicativos. A parte inferior da caixa indica o primeiro quartil (Q1), ou seja, 25% dos dados são inferiores a esse valor, enquanto a parte superior mostra o terceiro quartil (Q3), com 75% dos dados abaixo desse ponto. O espaço entre Q1 e Q3, conhecido como intervalo interquartílico (IQR), oferece uma medida da dispersão dos dados. A linha dentro da caixa marca a mediana, que é o valor central dos dados. Pontos individuais, ou *outliers*, representam dados que se distanciam significativamente dos demais, geralmente definidos como valores que estão além de 1,5 vezes o IQR a partir dos quartis.

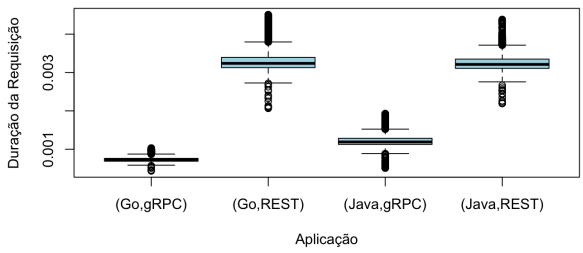

Figura 2. Desempenho em requisições *StdSize*

No gráfico, é possível observar que combinações que utilizam gRPC têm desempenho melhor que as que utilizam REST, além de que a espessura das caixas indicam que existe uma variabilidade menor quando se utiliza a arquitetura de comunicação gRPC.

A Figura 3, de forma complementar, ilustra um gráfico *boxplot* similar, porém no cenário de requisições *LargeSize*. Nota-se no gráfico que aplicações que usam REST possuem desempenho superior as que usam gRPC.

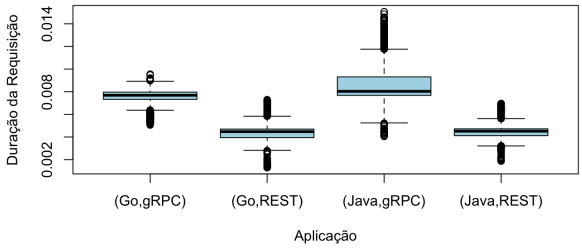

Figura 3. Desempenho em requisições *LargeSize*

A Tabela 1 reporta os dados de mediana e desvio padrão. Nota-se diferenças significativas entre os cenários de requisições *StdSize* e *LargeSize*. Por um lado, nas requisições *StdSize*, a implementação (Go, gRPC) apresentou o menor tempo médio de resposta, destacando-se também por seu baixo desvio padrão, o que indica uma regularidade notável no desempenho. Por outro lado, nas requisições *LargeSize*, a implementação (Go, REST) apresentou o menor tempo médio, complementado por um desvio padrão relativamente baixo, sugerindo relativa estabilidade mesmo sob condições de cargas de grande volume.

Tabela 1. Mediana do desempenho com desvio padrão

	StdSize		***LargeSize***	
Implementação	*Mediana*	*Desvio*	*Mediana*	*Desvio*
(Go, gRPC)	0,00071	0,00006	0,00768	0,00070
(Go, REST)	0,00323	0,00028	0,00447	0,00067
(Java, gRPC)	0,00119	0,00017	0,00803	0,00134
(Java, REST)	0,00321	0,00023	0,00451	0,00066

Normalidade dos dados. A análise de normalidade dos dados desempenhou um papel crucial antes de proceder com testes estatísticos. Para os tempos de execução de todas as oito combinações, o teste de Shapiro-Wilk indicou que os dados *não* seguiam uma distribuição normal.

Como esse teste possui uma sensibilidade grande para pequenos desvios de normalidade [20] quando utilizado em grandes quantidades de amostras (como neste estudo), métodos visuais de análise como gráficos Q-Q e histogramas foram empregados para confirmar a suposição e avaliar a distribuição dos tempos de resposta das implementações. Essas análises confirmaram a não-normalidade dos dados, um achado significativo que orientou a escolha de técnicas estatísticas adequadas para a análise subsequente [13]. A não-normalidade dos dados é uma característica comum em conjuntos de dados reais, especialmente em medições de desempenho de sistemas computacionais, onde fatores externos e a própria natureza das tecnologias podem influenciar grandemente a variação dos dados [14].

Dada essa característica dos dados, optou-se por métodos não paramétricos para as análises seguintes. A escolha foi fundamentada pela capacidade desses métodos de fornecer resultados confiáveis sem a necessidade de pressupostos de normalidade, essenciais em testes paramétricos. Essa decisão metodológica assegura que as comparações de desempenho entre as diferentes combinações de linguagens e estilos arquiteturais de comunicação sejam válidas e não comprometidas por violações das premissas estatísticas [14]. Portanto, a análise de normalidade não apenas informou a escolha dos testes estatísticos apropriados, mas também reforçou a robustez das inferências estatísticas realizadas nesse estudo [13].

Diferenças estatisticamente relevantes. Conforme abordado na Seção 2.3, o teste de Wilcoxon *Rank-Sum* é um método não paramétrico utilizado para comparar duas amostras independentes. O método é ideal para dados que não seguem uma distribuição normal e é aplicado quando se deseja determinar se duas distribuições possuem a mesma mediana sem assumir a normalidade dos dados. Dada a sua capacidade de adaptar-se às condições do estudo, o teste de Wilcoxon pode ser implementado em suas formas condicionais, com classificações naturais e intermediárias, para analisar eficazmente as diferenças entre as medianas [21].

A Tabela 2 reporta os resultados dos testes de Wilcoxon, tanto para requisições *StdSize* quanto para *LargeSize*. Os resultados indicaram diferenças estatisticamente significativas em todas as comparações entre as combinações. A comprovação de todos os pares serem estatisticamente diferentes permite comparar as medianas e classificar os melhores pares de tecnologias.

Tabela 2. Wilcoxon *Rank Sum*: resultados *p*-valor

Comparação	***StdSize***	***LargeSize***
(Java, REST) vs. (Java, gRPC)	< 2,2e-16	< 2,2e-16
(Java, REST) vs. (Go, REST)	4,532e-09	< 2,2e-16
(Java, REST) vs. (Go, gRPC)	< 2,2e-16	< 2,2e-16
(Java, gRPC) vs. (Go, REST)	< 2,2e-16	< 2,2e-16
(Java, gRPC) vs. (Go, gRPC)	< 2,2e-16	< 2,2e-16
(Go, REST) vs. (Go, gRPC)	< 2,2e-16	< 2,2e-16

O teste de Kruskal-Wallis é especialmente útil para analisar variações em dados categóricos, como os avaliados nesse estudo, onde é analisado o impacto de diferentes linguagens de programação e estilos arquiteturais de comunicação no desempenho das aplicações. Esse teste foi empregado em conjunto com o teste de Wilcoxon *Rank-Sum* para reforçar os resultados obtidos e garantir que as comparações entre as implementações sejam válidas.

Tabela 3 reporta os resultados do teste de Kruskal-Wallis. Primeiramente, conforme esperado, o teste indicou que tanto o estilo arquitetural de comunicação quanto a linguagem de programação têm um impacto significativo no desempenho das aplicações, com diferenças estatisticamente relevantes observadas tanto em requisições *StdSize* quanto *LargeSize*. Em segundo lugar, os resultados indicam que tanto a escolha do estilo arquitetural de comunicação quanto a linguagem de programação afetam o desempenho. Por último, mas não menos importante, quando as variáveis de estilo arquitetural de comunicação e linguagem de programação foram combinadas para análise, a interação entre esses dois fatores mostrou um efeito ainda mais pronunciado no desempenho. Estudos como os realizados por Habibzadeh [13] ilustram a aplicabilidade do teste de Kruskal-Wallis em contextos diversos, reforçando a importância de considerar não apenas os efeitos isolados de cada variável, mas também como elas interagem entre si.

Tabela 3. Kruskal-Wallis: resultados

	StdSize / LargeSize	
Comparação	*Chi-quadrado*	*p-valor*
arq. comunicação	21,806 / 22,500	< 2,2e-16 / < 2,2e-16
linguagem	1,239,3 / 356,88	< 2,2e-16 / < 2,2e-16
fator combinado	24,624 / 22,988	< 2,2e-16 / < 2,2e-16

Ranking. Considerando que há uma diferença estatisticamente significativa entre as medianas, é possível classificar o desempenho dos pares. A Tabela 4 exibe a lista das implementações ordenadas da melhor para a pior, considerando as requisições *StdSize* e *LargeSize*. Essa ordenação identifica quais combinações de linguagem e arquitetura de comunicação oferecem o melhor desempenho, auxiliando na escolha das tecnologias mais eficientes para aplicações distribuídas.

Tabela 4. *Rankings* das implementações

Posição	***StdSize***	***LargeSize***
#1	(Go, gRPC), 0,00071s	(Go, REST), 0,00447s
#2	(Java, gRPC), 0,00119s	(Java, REST), 0,00451s
#3	(Java, REST), 0,00321s	(Go, gRPC), 0,00768s
#4	(Go, REST), 0,00323s	(Java, gRPC), 0,00803s

Nesses resultados, observa-se uma influência significativa do tamanho da requisição, *StdSize* ou *LargeSize*, e da combinação entre linguagem e estilo arquitetural de comunicação sobre o desempenho das implementações. Os resultados indicam que (Go, gRPC) mostrou superioridade em requisições *StdSize*, enquanto (Go, REST) e (Java, REST) se destacaram – quase de forma equivalente – nas requisições *LargeSize*.

Essa distinção é particularmente relevante no projeto de sistemas distribuídos, onde a escolha do estilo arquitetural de comunicação pode afetar significativamente a eficiência operacional. Por um lado, a arquitetura de comunicação gRPC, com sua forma de comunicação mais leve e baseado em RPC, provou ser mais eficiente em requisições usuais, possivelmente devido à eficiência na compactação de dados e na redução da latência e da sobrecarga de comunicação. Por outro lado, a arquitetura de comunicação REST mostrou-se mais adaptada para requisições de grande volume, provavelmente devido à flexibilidade do REST em lidar com grandes volumes de dados sem estrutura rígida.

3.7 Influência dos Fatores (*QP#2*)

Esta seção provê respostas à questão de pesquisa #2 por meio da condução de um projeto estatístico nas cinco repetições dos dados coletados conforme descrito na Seção 3.5.

Para entender o impacto de cada fator no desempenho das aplicações distribuídas, utilizou-se o projeto fatorial $2^k r$ com 2 fatores e 5 repetições. Esse projeto experimental permite a análise sistemática dos efeitos principais e das interações entre os fatores [15]. O projeto $2^k r$ é uma extensão do projeto fatorial 2^k, incluindo repetições para estimar o erro experimental e aumentar a precisão dos resultados.

Os fatores escolhidos para o experimento foram a linguagem de programação e o estilo arquitetural de comunicação. A linguagem de programação (A) variou entre Go (-1) e Java (+1), enquanto o estilo arquitetural de comunicação (B) variou entre gRPC (-1) e REST (+1). Esses fatores são fundamentais para aplicações distribuídas modernas, onde a escolha da tecnologia pode impactar significativamente o desempenho.

Tabela 5 reporta, respectivamente, as tabelas de sinais utilizadas para o projeto fatorial $2^2 r$ para requisições *StdSize* e *LargeSize* incluindo os totais por fator e os totais divididos por 4, também conhecido como efeito do fator. Foram combinadas as duas tabelas de sinais pois apenas as duas última linhas se diferem. Por exemplo, pode-se dizer que o efeito do fator linguagem de programação para requisições *StdSize* foi de 0,00017 e para requisições *LargeSize* foi de 0,00019.

Tabela 5. Projeto Fatorial $2^2 r$ - *StdSize*

I	**Ling.**	**Arq. com.**	**Interação**	**Impl.**
1	-1	-1	1	(Go, gRPC)
1	1	-1	-1	(Java, gRPC)
1	-1	1	-1	(Go, REST)
1	1	1	1	(Java, REST)
0,00973	0,00070	0,00526	-0,00082	$Total_{Std}$
0,00243	0,00017	0,00131	-0,00020	$Total_{Std}/4$
0,02486	0,00079	-0,00599	-0,00049	$Total_{Large}$
0,00621	0,00019	-0,00149	-0,00012	$Total_{Large}/4$

Com os efeitos de cada fator, são calculadas as seguintes sete somas de quadrados para obter de fato as influências de cada fator: (i) SSY *(Soma dos Quadrados Total dos Dados)* que representa a totalidade da variação observada nos dados, somando os quadrados de cada valor de resposta coletado; (ii) SSE *(Erro da Soma dos Quadrados)* que corresponde à variação em Y que não é explicada pelos fatores no modelo

e é calculada subtraindo a contribuição dos fatores e suas interações do SSY; (iii) SS0 *(Soma dos Quadrados do Total)* que é a soma associada ao efeito total, que engloba todos os efeitos sistemáticos observados; (iv) SST *(Soma dos Quadrados dos Tratamentos)* é a variação que é explicada apenas pelos tratamentos, calculada subtraindo SS0 de SSY; (v) SSL *(Soma dos Quadrados da Linguagem)* que quantifica o quanto da variação total em Y é explicada pelas diferenças entre as linguagens de programação utilizadas; (vi) SSAC *(Soma dos Quadrados da Arquitetura de Comunicação)* que avalia o impacto do estilo arquitetural de comunicação sobre a variação nos dados; e (vii) SSLAC *(Soma dos Quadrados da Interação Linguagem e Arquitetura de Comunicação)* que mede a interação entre os fatores Linguagem e Arquitetura de Comunicação, indicando se o efeito de uma variável sobre o desempenho depende da outra.

Dado que E_T é o Efeito Total, E_L é o Efeito Linguagem de Programação, E_{AC} é o Efeito Arquitetura de Comunicação, e E_I: Efeito Interação, as Equações 2 a 8 transcrevem as fórmulas para cada soma de quadrados incluindo os valores resultantes no formato $v_{StdSize}/v_{LargeSize}$, onde $v_{StdSize}$ é o resultado para o projeto fatorial de requisições *StdSize* e $v_{LargeSize}$ de requisições *LargeSize*.

$$SSY = \sum(Y_{ij}^2) = 0,00015/0,00081 \quad (2)$$

$$SSE = SSY - 2^k r\left(E_T^2 + E_L^2 + E_{AC}^2 + E_I^2\right) = 8,84833/5,74272 \quad (3)$$

$$SS0 = 2^k r E_T^2 = 0,00011/0,00077 \quad (4)$$

$$SST = SSY - SS0 = 3,70512/4,65600 \quad (5)$$

$$SSL = 2^k r E_L^2 = 6,20191/7,90081 \quad (6)$$

$$SSAC = 2^k r E_{AC}^2 = 3,47041/4,48847 \quad (7)$$

$$SSLAC = 2^k r E_I^2 = 8,42123/3,10969 \quad (8)$$

Por fim, as influências de cada fator no resultado final são calculadas como porcentagens das somas de quadrados sobre a soma total dos quadrados. A Tabela 6 reporta, portanto, as influências dos fatores para as requisições *StdSize* e *LargeSize*.

Tabela 6. Influência dos Fatores no Desempenho

Fator	*StdSize*	*LargeSize*
Linguagem de programação	1,67%	1,69%
Arquitetura de comunicação	93,66%	96,40%
Interação	2,27%	0,66%
Erro	2,38%	1,23%

A aplicação do projeto fatorial $2^k r$ mostrou que o estilo arquitetural de comunicação é o principal fator que influencia o desempenho de aplicações distribuídas, sendo responsável por aproximadamente 94% do impacto em requisições de tamanho padrão (*StdSize*) e superando 96% em requisições de grande volume (*LargeSize*).

A influência da linguagem de programação, embora presente, é menos pronunciada comparativamente ao estilo arquitetural de comunicação. Esses resultados enfatizam a importância de selecionar o estilo arquitetural de comunicação adequado para otimizar o desempenho de sistemas distribuídos, oferecendo um suporte claro para decisões técnicas no desenvolvimento de software distribuído.

A influência da interação foi até superior à influência da linguagem de programação para requisições *StdSize*, porém praticamente nula (0,66%) para requisições *LargeSize*, o que indica que a interação é contextual e dependente do tamanho da requisição, sendo mais relevante em cenários com cargas menores e menos impactante em situações com volumes de dados maiores.

Por fim, a influência do erro é relativamente pequena, indicando que o modelo experimental utilizado é robusto e confiável.

3.8 Ameaças à validade

Pelo menos três ameaças à validade podem ser mencionadas:

Hardware e software. Otimizações das linguagens de programação, de processadores, uso de memória e interferência dos serviços do Ubuntu podem ter afetado as mensurações de tempo. No entanto, pode-se alegar que todas as decisões de projeto do experimento foram tomadas antes de qualquer mensuração e afeta as linguagens de programação e os estilos arquiteturais de comunicação da mesma forma.

Tamanho das requisições. Embora os tamanhos *StdSize* e *LargeSize* tenham sido projetados para representar, respectivamente, cargas médias e substancialmente maiores que a média, a definição de "média" pode não capturar adequadamente a variabilidade dos tamanhos de requisição encontrados em aplicações do mundo real.

Aplicação. No escopo deste estudo, a aplicação alvo foi projetada propositalmente para isolar – ao máximo – o experimento de fatores externos. De forma complementar, experimentos com aplicações que demandem aspectos particulares das linguagens de programação, como concorrência e paralelismo, são vislumbrados como trabalhos futuros.

Essas ameaças destacam a necessidade de cautela ao interpretar os resultados e ao considerar a aplicabilidade das conclusões para outros contextos ou configurações. É essencial reconhecer que, enquanto os resultados fornecem *insights* valiosos sobre o desempenho relativo das tecnologias estudadas sob condições específicas, a generalização desses resultados para outros cenários deve ser feita com uma compreensão clara das limitações e contextos em que os testes foram realizados.

3.9 Replicabilidade

Os aplicativos utilizados e os dados coletados durante o estudo estão publicamente disponíveis em:

https://github.com/PqES/2024_sblp

4 Trabalhos Relacionados

Esta seção descreve trabalhos relacionados ao estudo empírico conduzido neste artigo.

Śliwa e Pańczyk [26] avaliam o desempenho das arquiteturas de comunicação REST, GraphQL e gRPC, destacando-se o REST pela quantidade de transações por segundo e o gRPC pela eficiência na transmissão de dados. Contrariamente, a pesquisa inclui a análise de GraphQL e não emprega análises estatísticas avançadas como o teste de Wilcoxon *Rank-Sum* ou Kruskal-Wallis utilizados neste trabalho. Portanto, enquanto ambos os estudos contribuem para a compreensão do desempenho de diferentes tecnologias de comunicação, o presente trabalho distingue-se pela sua abordagem metodológica e foco analítico específico.

Bora e Bezboruah [5] publicaram dois serviços web distintos, um utilizando a arquitetura SOAP e outro com arquitetura REST, ambos implementados em Java. A análise estatística dos resultados revelou que a arquitetura REST mostrou-se mais eficiente e escalável em comparação com SOAP. Esse resultado é semelhante ao encontrado no presente trabalho, onde a arquitetura REST também mostrou maior eficiência em requisições de grande volume, embora o presente estudo usa uma abordagem estatística mais rigorosa com testes específicos para medir diferenças de desempenho.

Abhinav et al. [1] analisa a concorrência entre Go e Java por meio de experimentos com algoritmos de multiplicação de matrizes e *PageRank*. Os autores identificam que Go apresenta melhor desempenho em menor número de cálculos, enquanto Java supera à medida que o volume de cálculos aumenta. Esse padrão se inverte quando considerado a criação e gerenciamento de *threads*, com Java mostrando vantagem em computações menores e Go melhorando em cenários mais exigentes. Ambas as linguagens possuem características únicas de concorrência, destacando-se em diferentes aplicações. No contexto do presente trabalho, não existem cálculos numerosos sendo realizados nas implementações alvo, isso pode explicar a baixa influência da linguagem de programação nos resultados.

Marx et al. [19] analisam as características de desempenho do HTTP/2 em comparação ao HTTP/1.1. Os experimentos revelam que, apesar das melhorias propostas, o HTTP/2 apresenta limitações significativas em condições de rede adversas, onde a conexão multiplexada pode se tornar um gargalo e os benefícios em termos de desempenho raramente superam os do HTTP/1.1, exceto na redução de sobrecarga. É relevante mencionar que a arquitetura de comunicação gRPC, utiliza o HTTP/2 como base, o que influencia diretamente o seu desempenho em comparação ao REST, que tradicionalmente utiliza o HTTP/1.1.

Kamiński et al. [16] conduzem uma análise comparativa das arquiteturas de comunicação de Internet mais usados, incluindo REST, *WebSocket*, gRPC, GraphQL e SOAP. Utilizando servidores web implementados em Python, a equipe mediu o tempo e a sobrecarga de transferência de dados em operações específicas. Concluiu-se que o gRPC foi o método mais eficiente e confiável, enquanto o GraphQL mostrou-se o mais lento e problemático em termos de uso em servidores web. Esse estudo utilizou cargas pequenas para os comparativos e seus resultados em relação às arquiteturas de comunicação corroboram com os obtidos no presente trabalho com cargas semelhantes.

5 Conclusão

A motivação para este estudo surgiu da necessidade crescente de desenvolver APIs robustas e eficientes que possam atender às exigentes demandas de desempenho e escalabilidade de sistemas de software atuais. Diante disso, foi conduzido um estudo empírico que emprega procedimentos estatísticos rigorosos para quantificar as melhorias oferecidas por uma linguagem de programação e um estilo arquitetural de comunicação considerados promissores dentro de um contexto de sistemas distribuídos, abordando especificamente Go e Java e gRPC e REST.

A pesquisa foi impulsionada por duas principais questões de pesquisa: (i) qual a combinação mais eficiente entre as linguagens de programação Go e Java e os estilos arquiteturais de comunicação gRPC e REST e (ii) qual a influência da linguagem de programação e do estilo arquitetural de comunicação na eficiência.

A questão de pesquisa #1 evidenciou que Go apresenta desempenho superior tanto em requisições *StdSize* quanto *LargeSize*, comparado ao Java. Esse resultado sugere que a seleção de Go em combinação com gRPC, pode oferecer vantagens substanciais em termos de eficiência, especialmente em ambientes que requerem comunicações frequentes e de alto desempenho.

A questão de pesquisa #2 conclui que – de fato – a influência da arquitetura de comunicação é de 94% no desempenho. Isso indica que, ao substituir REST por gRPC, existem cenários em que fábricas de software podem ter uma melhoria de desempenho em seus sistemas sem necessitar de uma reescrita de código em uma linguagem de programação diferente.

Trabalhos futuros incluem: (i) avaliar a eficiência com requisições de diversos tamanhos a fim de encontrar pontos de transição onde uma arquitetura de comunicação pode superar outra; (ii) avaliar a eficiência com diferentes frequências de solicitações, a fim de obter os limiares de desempenho de REST e gRPC em situações de estresse; (iii) avaliar em diferentes configurações de infraestrutura como rede, CPU, memória, etc.; e (iv) avaliar o desempenho com algoritmos específicos no servidor, a fim de medir o impacto de funcionalidades especificas das linguagens, como paralelismo e gerenciamento de memória.

Agradecimentos: Esta pesquisa é apoiada pela FAPEMIG (APQ-03513-18).

Referências

[1] P Y Abhinav, Avakash Bhat, Christina Terese Joseph, and K Chandrasekaran. Concurrency analysis of go and java. In *5th International Conference on Computing, Communication and Security (ICCCS)*, pages 1–6, 2020.

[2] M. Belshe, R. Peon, and M. Thomson. Hypertext transfer protocol version 2 (http/2). Internet Requests for Comments, 2015.

[3] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[4] Marek Bolanowski, Kamil Żak, Andrzej Paszkiewicz, Maria Ganzha, Marcin Paprzycki, Piotr Sowiński, Ignacio Lacalle Úbeda, and Carlos Palau. *Eficiency of REST and gRPC Realizing Communication Tasks in Microservice-Based Ecosystems*, pages 97–108. IOS Press, Amsterdam, Netherlands, 2022.

[5] Abhijit Bora and Tulshi Bezboruah. A comparative investigation on implementation of restful versus soap based web services. *International Journal of Database Theory and Application*, 8(3):297–312, 2015.

[6] Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor, and Ken Thompson. The Go programming language and environment. *Communications of the ACM*, 65(5):70–77, 2022.

[7] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

[8] Somnath Datta and Glen A Satten. Rank-sum tests for clustered data. *Journal of the American Statistical Association*, 100(471):908–915, 2005.

[9] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, Boston, MA, 2012.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Internet Requests for Comments, 1999.

[11] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[12] Google. grpc: A high performance, open source universal rpc framework. https://grpc.io, 2015.

[13] F. Habibzadeh. Data distribution: Normal or abnormal? *Journal of Korean Medical Science*, 39(e35):1–8, 2024.

[14] Olena S. Holovnia, Natalia O. Shchur, I. A. Sverchevska, Yelyzaveta M. Bailiuk, and O. Pokotylo. Interactive surveys during online lectures for it students. *Contemporary Educational Technology*, 13(1):ep56, 2023.

[15] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, NY, USA, 1991.

[16] Lukasz Kamiński, Maciej Kozłowski, Daniel Sporysz, Katarzyna Wolska, Patryk Zaniewski, and Radosław Roszczyk. Comparative review of selected internet communication protocols. *Foundations of Computing and Decision Sciences*, 48(1):39–56, 2023.

[17] Kamal Kumar, Anuj Kumar Jain, Raj Gaurang Tiwari, Nitin Jain, Vinay Gautam, and Naresh Kumar Trivedi. Analysis of api architecture: A detailed report. In *12th International Conference on Communication Systems and Network Technologies (CSNT)*, pages 880–884, 2023.

[18] Yunhyeok Lee and Yi Liu. Using refactoring to migrate rest applications to grpc. In *ACM Southeast Conference*, page 219–223, 2022.

[19] Robin Marx, Maarten Wijnants, Peter Quax, Axel Faes, and Wim Lamotte. Web performance characteristics of http/2 and comparison to http/1.1. In *Web Information Systems and Technologies*, pages 87–114, 2018.

[20] Hélio Amante Miot. Avaliação da normalidade dos dados em estudos clínicos e experimentais. *Jornal Vascular Brasileiro*, 16(2):88–91, 2017.

[21] F. Pokropp. A note on conditional wilcoxon tests with natural and mid-ranks. *Biometrical Journal*, 34(8):859–863, 1992.

[22] R Core Team. R: A language and environment for statistical computing, 2019.

[23] S. Singhal and B. Q. Nguyen. The java factor. *ACM*, 41(6):34–37, 1998.

[24] The Chromium Projects. Spdy protocol. https://www.chromium.org/spdy/spdy-whitepaper, 2012. Acessado em 13 de fevereiro de 2024.

[25] TIOBE Software. Tiobe index for may 2024. https://www.tiobe.com/tiobe-index/, 2024.

[26] Mariusz Śliwa and Beata Pańczyk. Performance comparison of programming interfaces on the example of rest api, graphql and grpc. *Journal of Computer Sciences Institute*, 21(21):356–361, 2021.