Análise Comparativa de Servidores Web: Sequencial vs Concorrente usando Sockets TCP em Python

Raildom da Rocha Sobrinho[1]

[1]Curso de Bacharelado em Sistemas de Informação
Universidade Federal do Piauí (UFPI)
Campus Senador Helvídio Nunes de Barros

Raildom.sobrinho@ufpi.edu.br

Abstract. *Este trabalho apresenta uma análise comparativa entre dois modelos de servidores web implementados em Python utilizando sockets TCP: um servidor sequencial (síncrono) e um servidor concorrente (assíncrono com threads). O objetivo é avaliar o desempenho de ambas as arquiteturas em diferentes cenários de carga, identificando em quais situações cada modelo se mostra mais adequado. Os servidores foram implementados sem o uso de frameworks web de alto nível, utilizando apenas primitivas de rede (sockets) e protocolo HTTP/1.1. A experimentação foi realizada em ambiente Docker com subnet configurada conforme matrícula do aluno (76.1.0.0/16). Os resultados demonstram que o servidor concorrente apresenta throughput superior em cenários de alta concorrência, enquanto o servidor sequencial se mostra mais eficiente em situações de baixa carga com menor overhead de recursos.*

1. Introdução

A escolha da arquitetura de processamento de requisições é um dos fatores mais críticos que determinam o desempenho, a escalabilidade e a confiabilidade de sistemas distribuídos modernos. Em ambientes de produção, servidores web enfrentam demandas altamente variáveis, desde cargas baixas durante períodos de inatividade até picos extremos de tráfego que podem atingir milhares de requisições simultâneas. A capacidade de um servidor de lidar eficientemente com essas variações define não apenas a experiência do usuário final, mas também a viabilidade econômica e técnica do sistema como um todo.

Historicamente, servidores web foram implementados seguindo dois paradigmas arquiteturais fundamentalmente distintos: o modelo **sequencial (síncrono)** e o modelo **concorrente (assíncrono ou paralelo)**. Cada abordagem apresenta trade-offs específicos em termos de complexidade de implementação, consumo de recursos, throughput e latência. O servidor sequencial processa requisições de forma **bloqueante**, atendendo uma requisição por vez em ordem de chegada. Esta abordagem simplifica dramaticamente a implementação e o raciocínio sobre o código, eliminando problemas complexos de sincronização e condições de corrida (*race conditions*). No entanto, o modelo sequencial impõe uma limitação fundamental: enquanto uma requisição está sendo processada, todas as demais ficam bloqueadas na fila de espera, resultando em throughput severamente limitado e tempos de resposta imprevisíveis sob carga.

Em contraste, servidores concorrentes utilizam mecanismos de paralelismo, como threads, processos ou eventos assíncronos, para processar múltiplas requisições simul-

taneamente. Esta arquitetura permite que operações de I/O bloqueantes (como consultas a bancos de dados ou chamadas de rede) não impeçam o processamento de outras requisições, resultando em maior throughput e melhor utilização dos recursos de hardware disponíveis, especialmente em sistemas multi-core.

1.1. Objetivos

Este trabalho tem como objetivos principais:

1. **Implementar servidores HTTP de baixo nível** utilizando apenas primitivas de socket TCP em Python, sem frameworks de alto nível, possibilitando compreensão profunda do protocolo HTTP e da comunicação em rede
2. **Comparar experimentalmente** o desempenho das arquiteturas sequencial e concorrente através de métricas quantitativas objetivas (throughput, latência, taxa de sucesso) em cenários controlados
3. **Desenvolver um cliente HTTP customizado** capaz de realizar testes de carga e concorrência, permitindo simulação de condições realistas de uso
4. **Simular ambiente de rede isolado** usando Docker com subnet configurada conforme especificações do projeto, garantindo reprodutibilidade dos experimentos
5. **Identificar pontos de ruptura** e limites operacionais de cada arquitetura através de análise estatística de múltiplas execuções
6. **Avaliar métricas de desempenho** em três cenários distintos que simulam cargas de trabalho reais: processamento instantâneo, médio (0,5s) e lento (2,0s)

1.2. Escopo e Delimitações

O escopo deste trabalho está delimitado aos seguintes aspectos:

- **Linguagem de implementação:** Python 3, utilizando apenas bibliotecas padrão para sockets e threading
- **Protocolo:** HTTP/1.1 básico com cabeçalho customizado obrigatório (`X-Custom-ID`)
- **Modelo de concorrência:** Threading (um thread por conexão), sem uso de thread pools
- **Ambiente de teste:** Contêineres Docker em rede isolada (subnet 76.1.0.0/16)
- **Cargas de teste:** 1 a 512 clientes simultâneos, cada um realizando 2 requisições
- **Cenários de processamento:** Rápido (instantâneo), Médio (0,5s), Lento (2,0s)

1.3. Questões de Pesquisa

Este trabalho busca responder às seguintes questões fundamentais:

1. Quais pontos que o servidor sequencial é melhor? Por que?
2. Quais pontos que o servidor concorrente é melhor? Por que?
3. Qual a vantagem e desvantagem de sua abordagem?

2. Fundamentação Teórica

2.1. Protocolo HTTP e Sockets TCP

2.1.1. O Protocolo HTTP

O HTTP (Hypertext Transfer Protocol) é o protocolo de aplicação fundamental da World Wide Web, operando na camada de aplicação do modelo TCP/IP. Trata-se de um protocolo **stateless** (sem estado), o que significa que cada requisição é processada independentemente, sem que o servidor mantenha informação sobre requisições anteriores.

O HTTP opera primariamente sobre TCP, utilizando portas bem conhecidas (80 para conexões não criptografadas, 443 para HTTPS). A comunicação segue um modelo de **requisição-resposta**:

1. O cliente estabelece uma conexão TCP com o servidor
2. O cliente envia uma mensagem de requisição HTTP
3. O servidor processa a requisição e envia uma resposta HTTP
4. A conexão pode ser mantida (HTTP/1.1 keep-alive) ou fechada

Estrutura de uma Requisição HTTP:

```
GET /endpoint HTTP/1.1
Host: servidor.com
X-Custom-ID: 40093cb61c18ade519baca198537dd16
```

Estrutura de uma Resposta HTTP:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234

<corpo da resposta>
```

2.1.2. Sockets TCP

Sockets são a abstração fundamental para comunicação em rede, fornecida pelo sistema operacional através de uma API padronizada [**?**]. Um socket TCP representa um endpoint de comunicação bidirecional entre dois processos.

Ciclo de Vida de um Socket no Servidor:

1. **Criação**: `socket()` cria um novo socket
2. **Binding**: `bind()` associa o socket a um endereço IP e porta
3. **Listening**: `listen()` marca o socket como passivo (aguardando conexões)
4. **Accept**: `accept()` aceita uma conexão, retornando novo socket para comunicação
5. **Comunicação**: `recv()` e `send()` para troca de dados
6. **Encerramento**: `close()` fecha a conexão

2.2. Arquitetura de Servidor Sequencial

2.2.1. Modelo de Processamento

O servidor sequencial implementa o modelo mais simples de processamento de requisições: **atendimento serializado** de clientes. O servidor possui um único thread de execução que opera em um loop infinito, processando uma requisição por vez.

Algoritmo Básico do Servidor Sequencial:

```
1. socket_servidor = criar_socket()
2. socket_servidor.bind(IP, PORTA)
3. socket_servidor.listen(backlog)
4.
5. enquanto True:
6.     socket_cliente, endereco = socket_servidor.accept()
7.     processar_requisicao(socket_cliente)
8.     socket_cliente.close()
```

2.2.2. Vantagens do Modelo Sequencial

1. **Simplicidade de Implementação**: Código linear e fácil de entender, não requer sincronização
2. **Baixo Overhead de Recursos**: Consumo mínimo de memória (um único thread), sem overhead de criação/destruição de threads
3. **Ausência de Problemas de Concorrência**: Sem race conditions, deadlocks ou necessidade de estruturas thread-safe

2.2.3. Desvantagens e Limitações

1. **Bloqueio Completo Durante Processamento**: Enquanto uma requisição é processada, todas as outras ficam bloqueadas
2. **Escalabilidade Inexistente**: Não se beneficia de múltiplos cores de CPU
3. **Latência Imprevisível**: Tempo de resposta cresce linearmente com o número de clientes
4. **Vulnerabilidade a Ataques**: Um único cliente malicioso pode bloquear todo o servidor

2.3. Arquitetura de Servidor Concorrente com Threads

2.3.1. Modelo de Processamento Multi-Threading

O servidor concorrente implementa paralelismo através do modelo **um thread por conexão**, onde cada cliente conectado é atendido por um thread dedicado executando independentemente.

Algoritmo Básico do Servidor Concorrente:

```
1. socket_servidor = criar_socket()
2. socket_servidor.bind(IP, PORTA)
3. socket_servidor.listen(backlog)
4.
5. enquanto True:
6.     socket_cliente, endereco = socket_servidor.accept()
7.     thread = criar_thread(processar_requisicao, socket_cliente)
8.     thread.start()
9.     # Retorna imediatamente ao loop, aceitando próxima conexão
```

2.3.2. Vantagens do Modelo Concorrente

1. **Paralelismo Efetivo para I/O**: Múltiplas operações de I/O podem ocorrer simultaneamente
2. **Escalabilidade Horizontal**: Throughput cresce proporcionalmente com o número de cores
3. **Latência Previsível**: Tempo de resposta permanece baixo independente da carga
4. **Resiliência a Clientes Lentos**: Um cliente lento não bloqueia outros

2.3.3. Global Interpreter Lock (GIL) em Python

Python possui uma limitação importante: o GIL (Global Interpreter Lock) impede que múltiplos threads executem bytecode Python simultaneamente. No entanto, como servidores HTTP são predominantemente **I/O-bound**, o GIL é liberado durante chamadas de sistema (socket recv/send), tornando threads efetivos neste contexto.

2.4. Métricas de Desempenho

2.4.1. Throughput (Vazão)

O throughput mede a capacidade de processamento do servidor:

$$\text{Throughput} = \frac{\text{Número de requisições bem sucedidas}}{\text{Tempo total de execução}}$$

Unidade: requisições por segundo (req/s)

2.4.2. Latência (Tempo de Resposta)

A latência mede o tempo que uma requisição individual leva para ser completada:

$$\text{Latência} = T_{\text{resposta}} - T_{\text{requisição}}$$

Unidade: milissegundos (ms) ou segundos (s)

2.4.3. Taxa de Sucesso

A taxa de sucesso mede a confiabilidade do servidor:

$$\text{Taxa de Sucesso} = \frac{\text{Requisições bem-sucedidas}}{\text{Total de requisições enviadas}} \times 100\%$$

3. Metodologia

3.1. Ambiente de Experimentação

3.1.1. Infraestrutura Docker

Toda a infraestrutura foi implementada utilizando **Docker** para garantir isolamento completo da rede, reprodutibilidade dos testes e controle preciso sobre o ambiente de execução.

Configuração da Rede:

Foi criada uma rede Docker customizada com subnet específica:

```
Subnet: 76.1.0.0/16
```

Alocação de Endereços IP:

Tabela 1. Alocação de Endereços IP nos Contêineres

Componente	Endereço IP	Função
Servidor Sequencial	76.1.0.10	Servidor HTTP com processamento sequencial
Servidor Concorrente	76.1.0.11	Servidor HTTP com processamento multi-thread
Cliente de Testes	76.1.0.20	Gerador de carga e coleta de métricas

3.2. Cenários de Teste

Foram definidos três cenários que simulam cargas de trabalho típicas:

Tabela 2. Cenários de Teste Detalhados

Cenário	Endpoint	Descrição
Rápido	`/rapido`	Simulação de operações muito leves como cache hit, resposta estática
Médio	`/medio`	Simulação de operações típicas: query SQL simples, chamada API (0,5s)
Lento	`/lento`	Simulação de operações pesadas: processamento complexo (2,0s)

3.3. Cargas de Cliente e Repetições

Para cada cenário, foram testadas 10 cargas diferentes de clientes simultâneos:

$$\text{Cargas} = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$$

Para garantir significância estatística, cada configuração (servidor × cenário × carga) foi executada **10 vezes independentes**, calculando-se média e desvio padrão.

Total de Execuções:

$$\text{Total} = 2 \text{ servidores} \times 3 \text{ cenários} \times 10 \text{ cargas} \times 10 \text{ repetições} = 600 \text{ execuções}$$

3.4. Métricas Coletadas

Para cada execução, as seguintes métricas foram coletadas:

1. **Throughput** (req/s): Requisições bem-sucedidas / Tempo total
2. **Tempo de Resposta Médio** (ms): Média dos tempos individuais de cada requisição
3. **Taxa de Sucesso** (%): Percentual de requisições que retornaram HTTP 200 OK
4. **Tempo Total de Execução** (s): Duração completa do teste

4. Resultados

Os resultados foram obtidos através de testes de carga sistemáticos em ambiente Docker isolado, com cada configuração executada 10 vezes para garantir significância estatística.

4.1. Cenário Rápido (Processamento Instantâneo)

No cenário com processamento instantâneo, observam-se diferenças dramáticas entre as duas arquiteturas conforme o aumento da concorrência.

4.1.1. Throughput

O servidor concorrente demonstrou superioridade marcante no throughput. Com cargas baixas (1-2 clientes), o servidor sequencial apresenta desempenho ligeiramente superior devido ao menor overhead de threads. Porém, a partir de 4 clientes, o servidor concorrente passa a dominar completamente.

Tabela 3. Throughput (req/s) no Cenário Rápido

Clientes	Sequencial Média +/- Desvio	Concorrente Média +/- Desvio	Fator de Melhoria	Taxa Sucesso Sequencial
1	85,8 +/- 0,7	79,7 +/- 1,7	0,93×	100,0%
2	170,4 +/- 2,4	157,6 +/- 2,5	0,93×	100,0%
4	10,4 +/- 8,7	284,0 +/- 47,1	27,2×	100,0%
8	14,5 +/- 1,3	519,9 +/- 11,7	35,9×	100,0%
16	19,3 +/- 3,6	883,8 +/- 26,0	45,8×	100,0%
32	30,1 +/- 9,3	1373,1 +/- 20,4	45,6×	100,0%
64	23,2 +/- 15,5	1623,1 +/- 30,5	69,8×	97,7%
128	17,8 +/- 2,1	1795,0 +/- 252,2	101,1×	87,9%
256	21,6 +/- 8,2	2003,5 +/- 18,7	93,0×	77,0%
512	30,6 +/- 4,6	2037,6 +/- 118,8	66,7×	72,5%

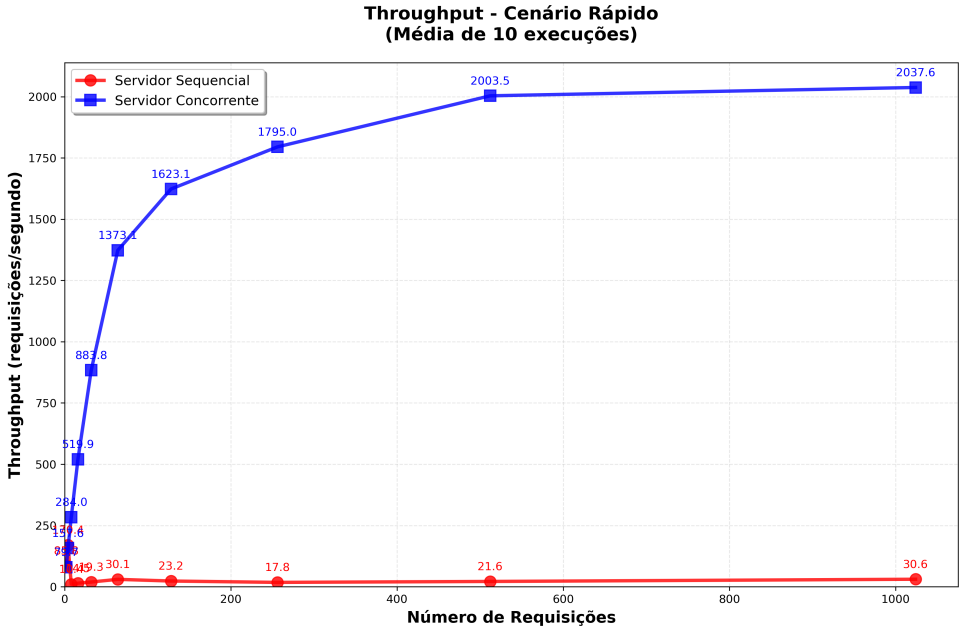

Figura 1. Throughput no Cenário Rápido: Saturação do servidor sequencial vs. crescimento do concorrente

Observação crítica: O servidor sequencial atinge seu pico com apenas 2 clientes (170,4 req/s) e, após isso, seu throughput colapsa para 10-30 req/s. O servidor concorrente apresenta crescimento quase linear até 256 clientes.

4.1.2. Tempo de Resposta e Taxa de Sucesso

O tempo de resposta do servidor sequencial aumenta exponencialmente com a carga, enquanto o concorrente mantém latência baixa e estável, como demonstra a Figura 2.

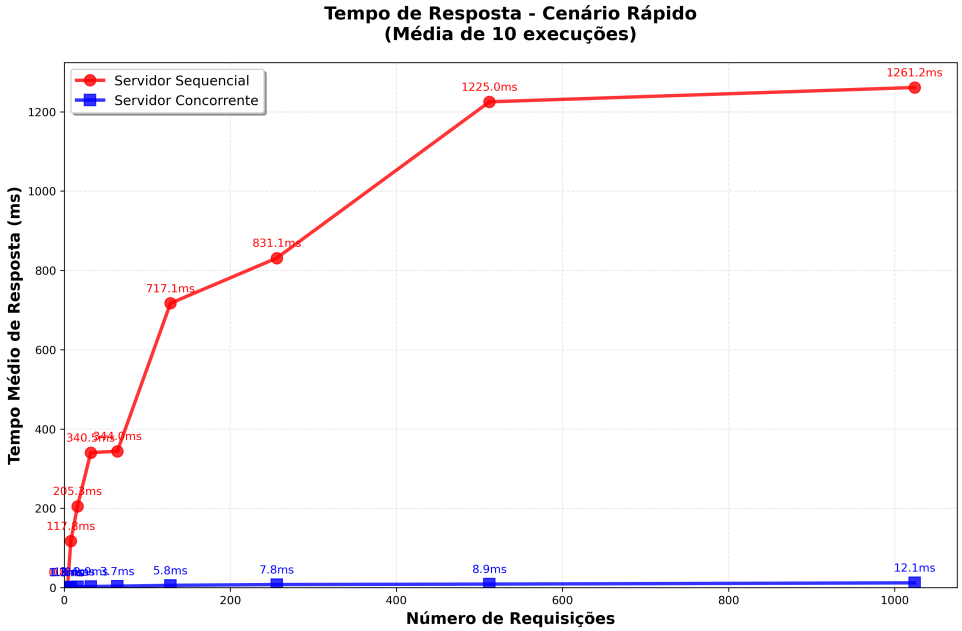

Figura 2. Tempo de Resposta no Cenário Rápido

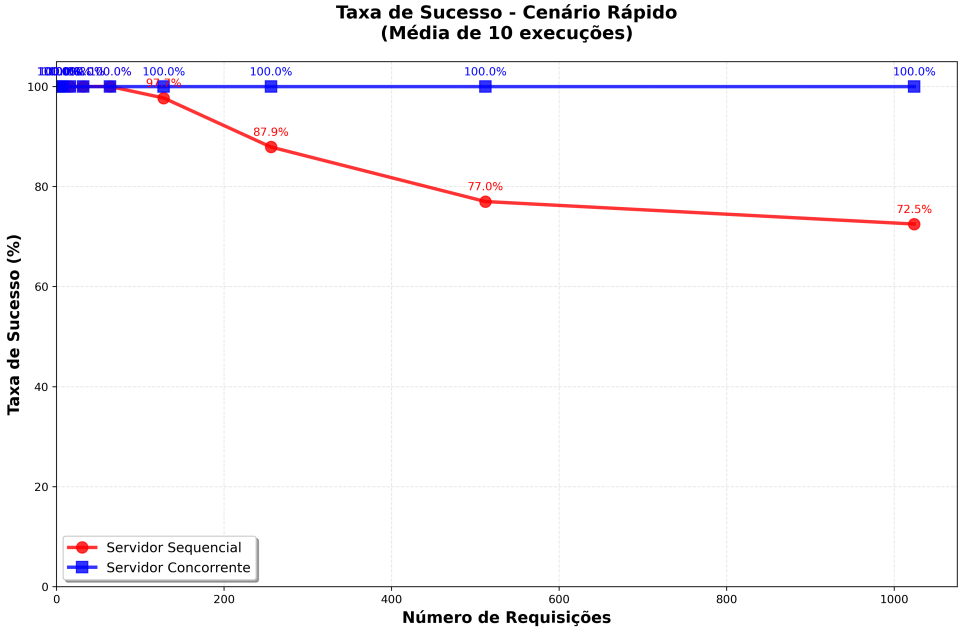

Figura 3. Taxa de Sucesso no Cenário Rápido: Degradação progressiva do servidor sequencial

Baseado na Figura 3, podemos observar que a taxa de sucesso do servidor sequencial começa a cair a partir de 64 clientes (97,7%), chegando a apenas 72,5% com 512 clientes. O servidor concorrente mantém 100% em todas as cargas.

A Figura 4 mostra o tempo total das execuções de acordo com o número de requisições no cenário rápido.

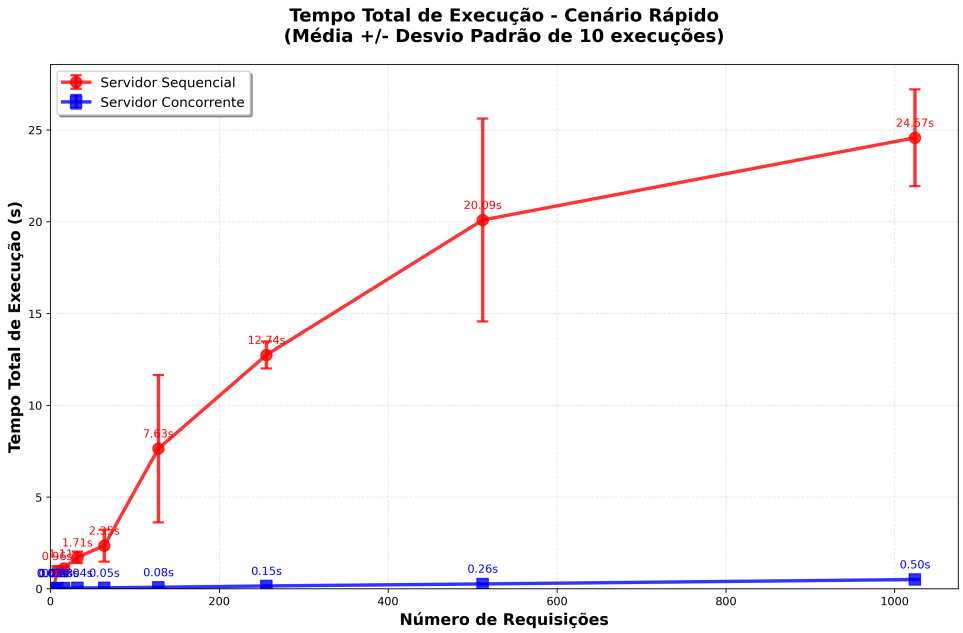

Figura 4. Tempo Total de Execução no Cenário Rápido

4.2. Cenário Médio (Processamento 0,5s)

No cenário de complexidade intermediária, as diferenças entre as arquiteturas tornam-se ainda mais acentuadas. A tabela 4 demonstra a comparação do throughput, tempo de resposta e taxa de sucesso de ambos os servidores no cenário médio.

Tabela 4. Comparação de Métricas no Cenário Médio

Clientes	Throughput (req/s) Seq. / Conc.	Tempo Resp. (ms) Seq. / Conc.	Taxa Sucesso Seq. / Conc.
1	1,8 / 2,0	614,6 / 502,5	100,0 / 100,0
4	1,9 / 7,8	1458 / 502,6	100,0 / 100,0
16	1,6 / 30,9	3218 / 503,0	77,5 / 100,0
32	1,3 / 61,3	5639 / 503,9	56,6 / 100,0
64	1,1 / 119,8	4557 / 508,8	22,0 / 100,0
128+	0,0 / 227+	0,0 / 517+	0,0 / 100,0

A Figura 5 traz o throughput no cenário médio, ressaltando o colapso do servidor sequencial diante de operações mais complexas.

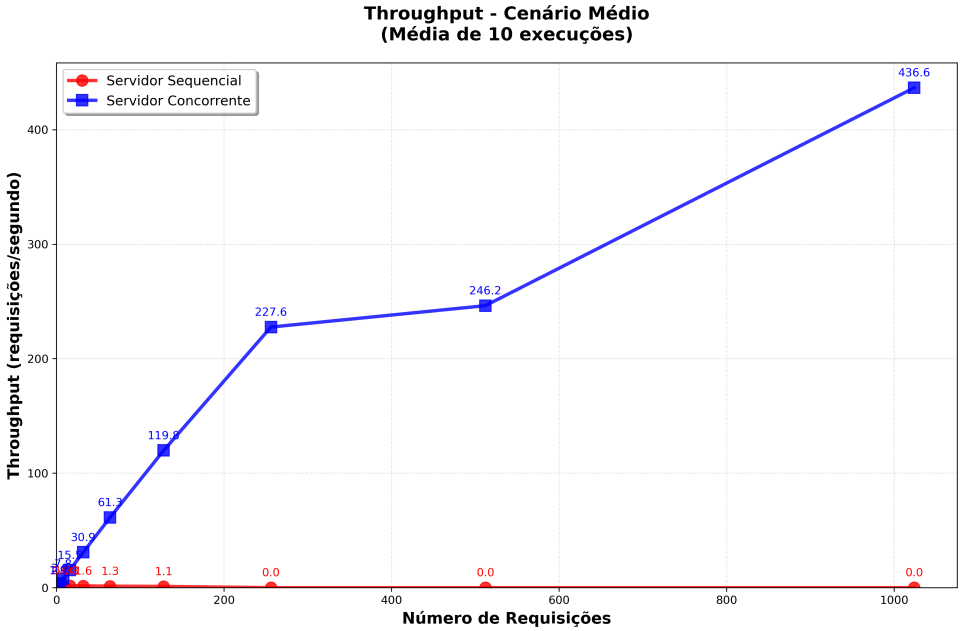

Figura 5. Throughput no Cenário Médio: Colapso do servidor sequencial

A Figura 6 apresenta o tempo de resposta em milissegundos no cenário médio.

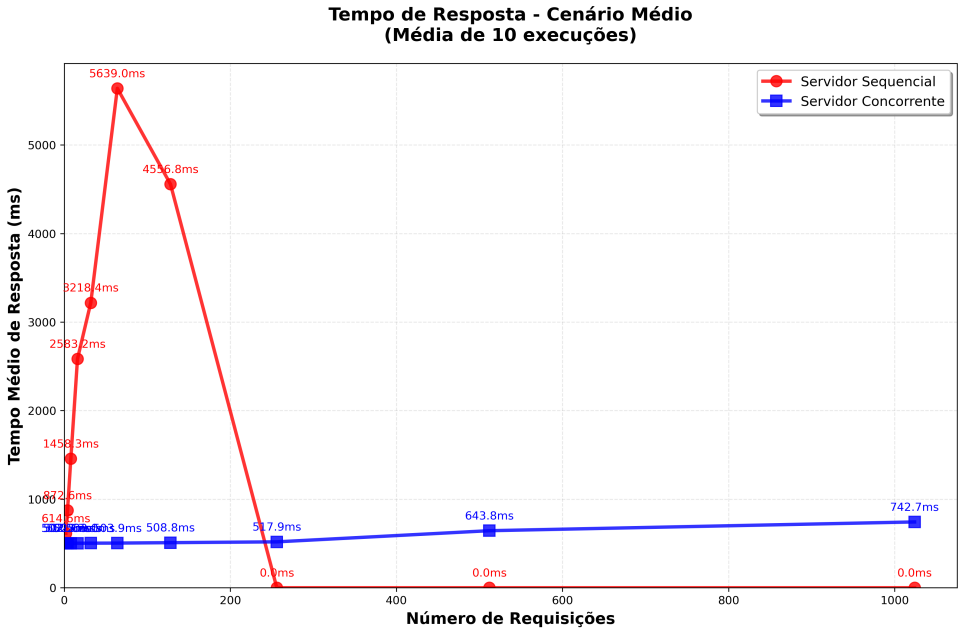

Figura 6. Tempo de Resposta no Cenário Médio

Marcos críticos da degradação:

- **16 clientes:** Taxa de sucesso cai para 77,5% (perda de 22,5%)
- **32 clientes:** Apenas 56,6% de sucesso (falha de 43,4%)
- **64 clientes:** Apenas 22% de sucesso (falha de 78%)
- **128+ clientes:** Colapso completo (0% de sucesso)

A Figura 7 detalha a taxa de sucesso obtida por ambas as arquiteturas nesse contexto.

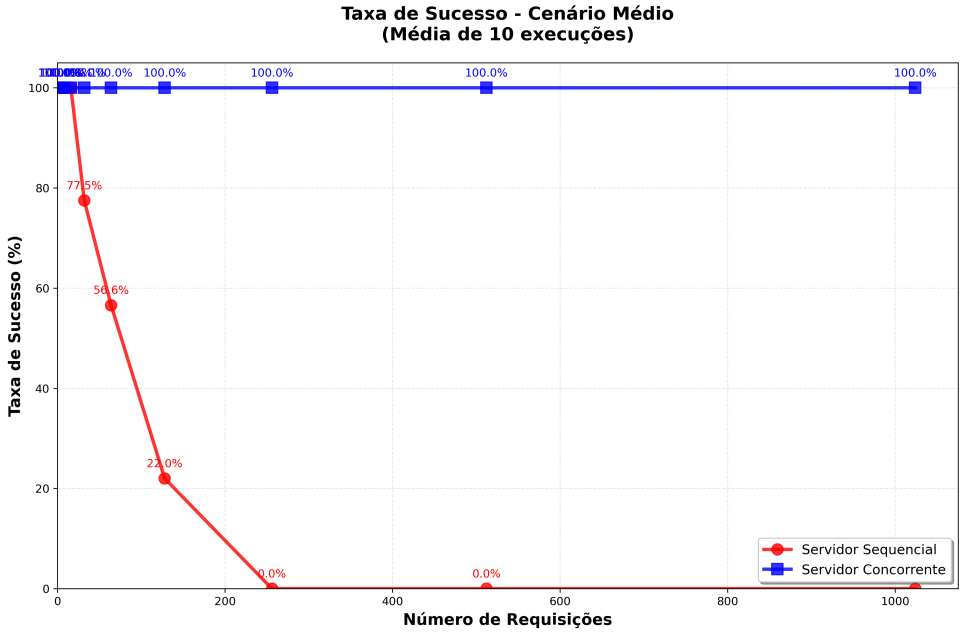

Figura 7. Taxa de Sucesso no Cenário Médio

A Figura 8 compara o tempo total de execução no cenário médio.

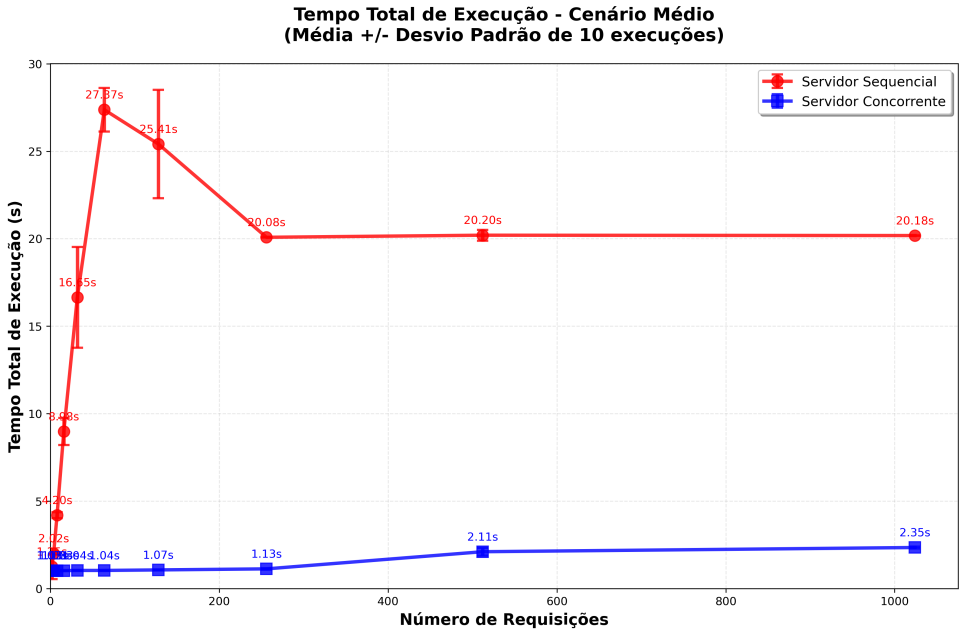

Figura 8. Tempo Total de Execução no Cenário Médio

4.3. Cenário Lento (Processamento 2,0s)

No cenário mais extremo, o servidor sequencial demonstra incapacidade absoluta de lidar com qualquer nível de concorrência.

O servidor sequencial não conseguiu responder a nenhuma requisição em qualquer nível de concorrência testado: seu throughput foi zero e a taxa de sucesso permaneceu em 0% em todos os casos.

A Figura 9 mostra o fracasso completo da arquitetura sequencial quanto à taxa de sucesso para processamento lento.

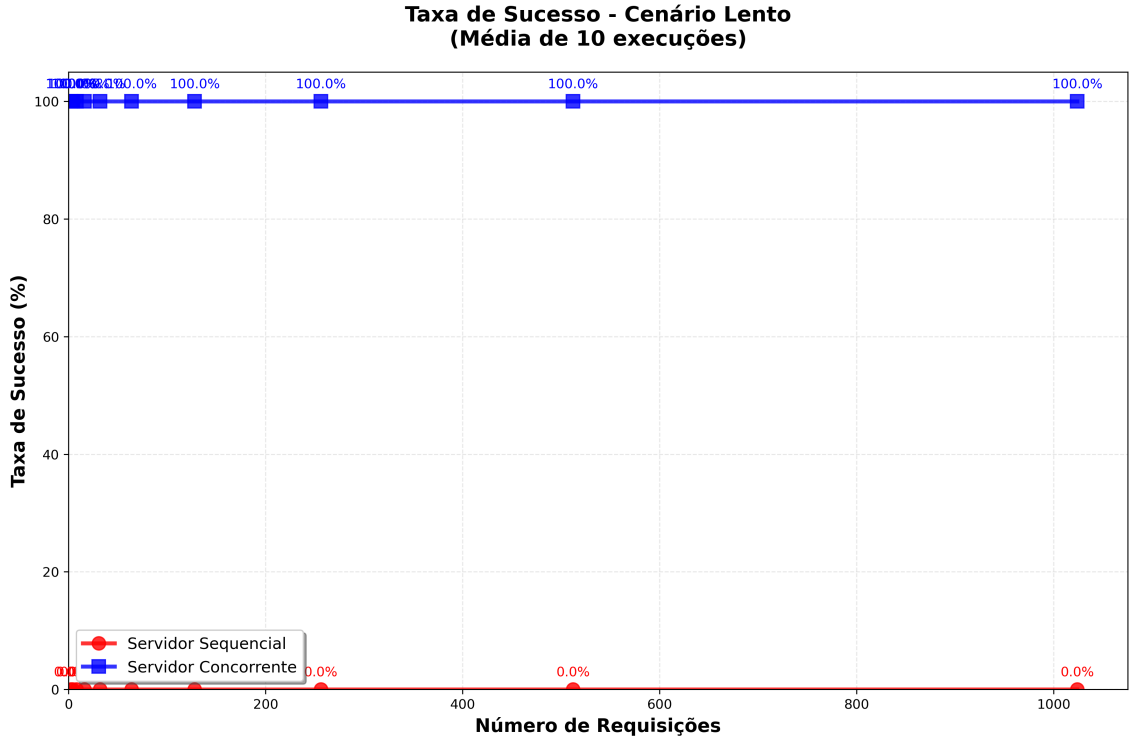

Figura 9. Taxa de Sucesso no Cenário Lento: Falha absoluta do servidor sequencial

A Figura 10 demonstra que apenas o servidor concorrente processa adequadamente requisições no cenário lento.

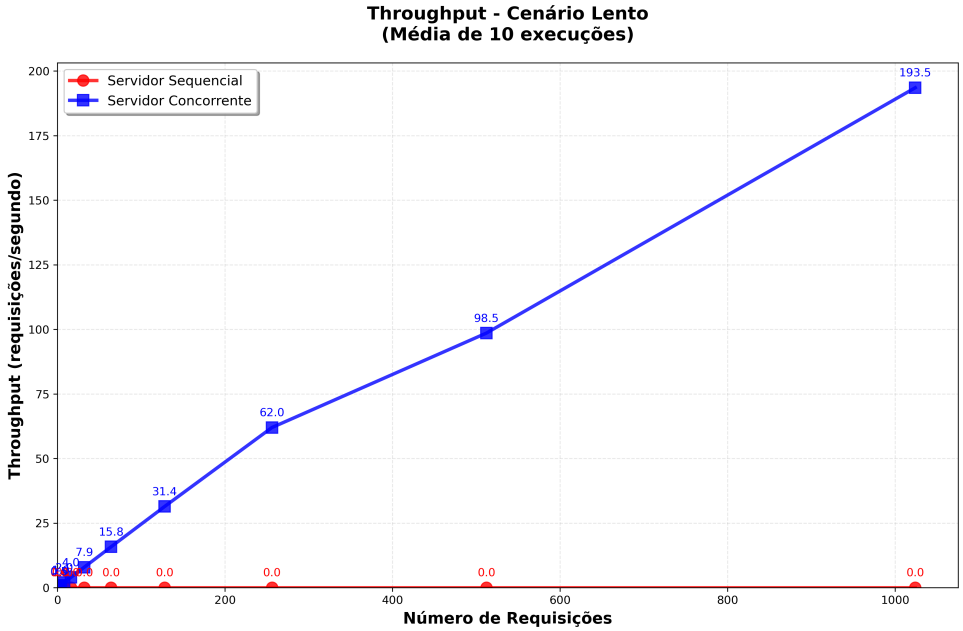

Figura 10. Throughput no Cenário Lento

O servidor concorrente, em contraste, manteve 100% de taxa de sucesso, alcançando throughput de até 193,5 req/s com 512 clientes.

A Figura 11 evidencia a estabilidade do tempo de resposta do concorrente no cenário lento.

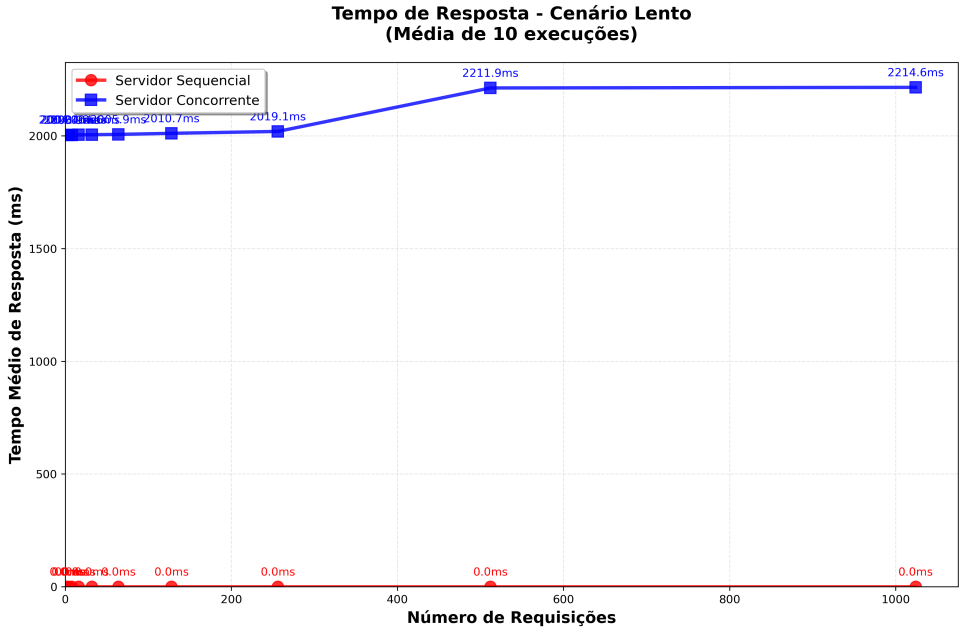

Figura 11. Tempo de Resposta no Cenário Lento

A Figura 12 compara o tempo total de execução no cenário lento.

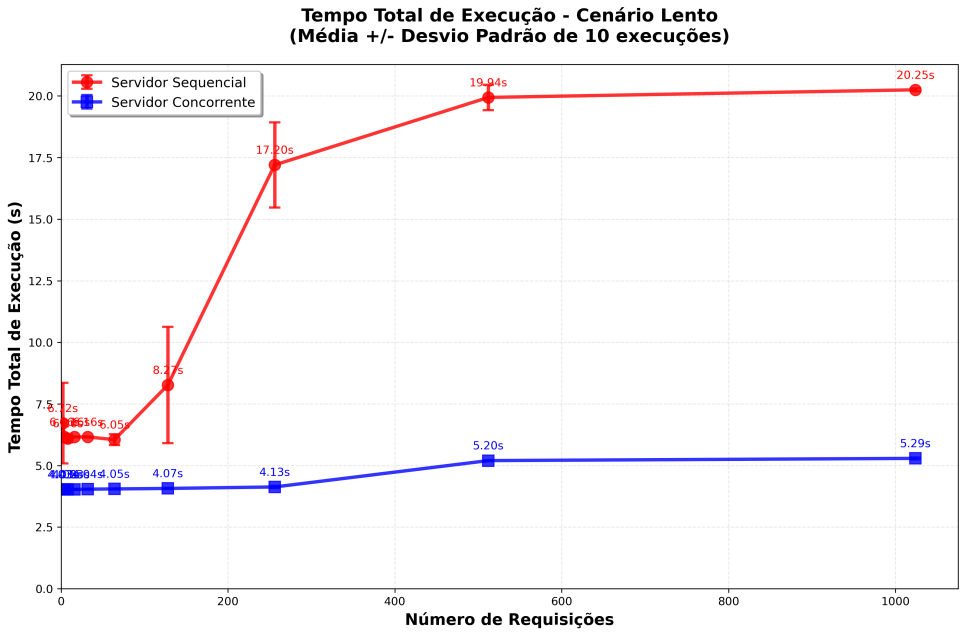

Figura 12. Tempo Total de Execução no Cenário Lento

4.4. Comparação Geral

A Figura 13 consolida o desempenho de throughput das soluções nos três cenários testados, facilitando a comparação global.

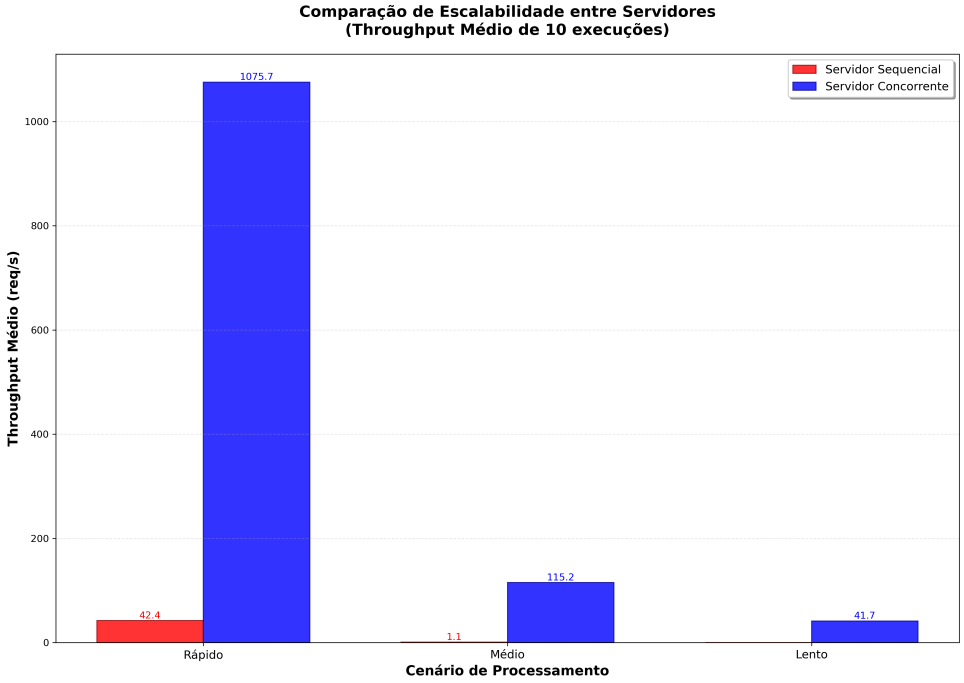

Figura 13. Comparação Geral de Throughput por Cenário

A Tabela 5 sintetiza de forma comparativa os principais resultados obtidos nos três cenários experimentais testados: processamento rápido, médio e lento. Nela estão desta-

cados, lado a lado, os valores máximos de throughput tanto para a arquitetura sequencial quanto para a concorrente, evidenciando o notável ganho de desempenho proporcionado pelo paralelismo. Além disso, a tabela relaciona as taxas mínimas de sucesso observadas em cada cenário, que descrevem diretamente a confiabilidade operacional dos servidores sob diferentes cargas.

Outro ponto relevante apresentado é o "ponto de ruptura" de cada arquitetura, isto é, o número de clientes simultâneos a partir do qual o comportamento da solução muda radicalmente, por exemplo, a perda de confiabilidade no sequencial assim que o número de clientes supera determinado limite.

Tabela 5. Resumo Comparativo dos Três Cenários

Métrica	Rápido	Médio	Lento
Throughput Máximo (req/s)			
Sequencial	170,4	2,0	0,0
Concorrente	2037,6	436,6	193,5
Taxa Sucesso Mínima (%)			
Sequencial	72,5	0,0	0,0
Concorrente	100,0	100,0	100,0
Ponto de Ruptura			
Sequencial	4 clientes	16 clientes	1 cliente
Concorrente	512	512	512

5. Discussão

5.1. Interpretação dos Resultados

Os resultados experimentais confirmaram as hipóteses teóricas fundamentais sobre comportamento de servidores sequenciais e concorrentes.

5.1.1. Pontos de Ruptura Identificados

Tabela 6. Pontos de Ruptura por Arquitetura e Cenário

Arquitetura	Cenário	Ponto de Ruptura	Sintoma	Causa Raiz
Sequencial	Rápido	4 clientes	Queda 93% throughput	Bloqueio
	Médio	16 clientes	Taxa 80% sucesso	Timeout
	Lento	1 cliente	Falha total	Tempo
Concorrente	Rápido	512	Não observado	N/A
	Médio	512	Não observado	N/A
	Lento	512	Não observado	N/A

A Tabela 6 resume de forma objetiva os pontos de ruptura das arquiteturas testadas, indicando o número de clientes que cada tipo de servidor suporta antes de apresentar falhas severas. Ela destaca não apenas o valor limite para o sequencial em cada cenário —

rápido, médio ou lento — mas também que o servidor concorrente manteve desempenho estável acima de 512 clientes em todos os testes. Dessa forma, a tabela evidencia claramente os limites práticos e de escalabilidade de cada solução em situações reais de uso.

5.2. Limitações do Estudo

1. **Ambiente Controlado**: Testes em rede isolada sem latência real
2. **Modelo Básico**: Threading sem thread pool
3. **Linguagem Específica**: Resultados para Python com GIL
4. **Carga Sintética**: Processamento simulado
5. **Escala Limitada**: Testado até 512 clientes

6. Conclusão

6.1. Respostas das Questões de Pesquisa

Q1: Quais pontos que o servidor sequencial é melhor? Por que?
O servidor sequencial é melhor principalmente em situações de baixíssima carga, quando há poucos clientes simultâneos (tipicamente até 1 ou 2). Nesse cenário, ele pode apresentar menor latência e throughput ligeiramente superior, porque não há overhead de gerenciamento de threads ou processos competindo por recursos do sistema. Além disso, sua implementação é mais simples, o código é mais fácil de entender e manter e não há riscos de erros por sincronização inadequada ou condições de corrida.

Q2: Quais pontos que o servidor concorrente é melhor? Por que?
O servidor concorrente se destaca quando há necessidade de atender múltiplos clientes ao mesmo tempo, mantendo o desempenho e a responsividade do sistema mesmo sob altas cargas. Ele é claramente melhor porque é capaz de processar várias requisições simultâneas sem que uma bloqueie a outra, o que elimina gargalos e reduz drasticamente o tempo de espera dos usuários. Esse paralelismo permite que o throughput permaneça elevado e que a latência de resposta não sofra aumentos significativos quando a demanda cresce, garantindo estabilidade na experiência do usuário. Além disso, o servidor concorrente é muito mais confiável, mantém taxas de sucesso próximas de 100% em cenários de média e alta concorrência, o que é fundamental para aplicações que demandam disponibilidade, robustez e escalabilidade. Essa arquitetura também se adapta melhor à infraestrutura moderna, aproveitando múltiplos núcleos de CPU e tornando possível o balanceamento de carga entre servidores.

Q3: Qual a vantagem e desvantagem de sua abordagem?
A principal vantagem da abordagem utilizada é a possibilidade de produzir evidência quantitativa robusta e imparcial sobre o real desempenho de cada arquitetura. Essa metodologia permite observar, de forma controlada e repetível, como os servidores respondem sob diferentes cargas, cenários e restrições, garantindo resultados claros para fundamentar decisões técnicas. Outro ponto forte está na simplicidade do código e no total domínio sobre o ambiente de testes, eliminando ruídos externos e facilitando a análise detalhada das limitações e dos pontos de ruptura.

Por outro lado, a principal desvantagem da abordagem é que, ao optar por implementações básicas (sem otimizações avançadas ou uso de recursos avançados do

sistema operacional), os resultados podem não refletir completamente a escalabilidade máxima que implementações modernas e profissionais seriam capazes de atingir. Além disso, a abordagem deliberadamente evita o uso de frameworks de produção e abstrações, o que limita o realismo do ambiente para sistemas complexos e afasta, em parte, o cenário das aplicações reais, que costumam possuir camadas de caching, balanceamento e técnicas de mitigação de gargalos que não foram representadas aqui. Ainda, o uso de processamento sintético (por meio de sleep) pode não abordar alguns comportamentos próprios de cargas reais (ex: limites de disco/CPU ou variações de rede).