Análise Comparativa de Desempenho entre Servidores Web Sequenciais e Concorrentes em Ambiente Docker

Camila Moura[1]

[1]Universidade Federal do Piauí (UFPI)
Campus Senador Helvídio Nunes de Barros - CSHNB.//

camilamouramoura390@gmail.com

Resumo. *Este artigo apresenta a implementação e a avaliação de desempenho de dois modelos de servidores web: um sequencial (síncrono) e um concorrente (assíncrono) . O objetivo é quantificar as vantagens e desvantagens de cada abordagem , conforme proposto pela disciplina de Redes de Computadores II . Ambos os servidores foram desenvolvidos em Python utilizando a biblioteca de Sockets , implementando o protocolo TCP e estruturando mensagens HTTP . A simulação de rede foi realizada em um ambiente controlado usando Docker, com hosts em containers Ubuntu . Foram definidos três cenários de teste variando a carga e a concorrência, e os resultados foram analisados com base nas métricas de latência média e vazão (throughput) . Os resultados demonstram que, embora o servidor sequencial apresente menor latência em cenários de concorrência nula, o servidor concorrente é fundamental para a escalabilidade, mantendo uma vazão ordens de magnitude superior sob carga média e alta .*

1. Introdução

A arquitetura cliente-servidor é a base da World Wide Web. Nela, servidores aguardam requisições de múltiplos clientes e respondem com os recursos solicitados. A forma como um servidor gerencia essas conexões recebidas define sua eficiência e escalabilidade.

Existem duas abordagens fundamentais para o tratamento de conexões:

Modelo Sequencial (Síncrono): O servidor atende um cliente por vez. Ele aceita uma conexão, processa a requisição, envia a resposta e fecha a conexão antes de poder aceitar a próxima.

Modelo Concorrente (Assíncrono): O servidor é capaz de gerenciar múltiplas conexões simultaneamente. Ao aceitar uma nova conexão, ele a delega a um processo ou thread separado, ficando imediatamente livre para aceitar outras conexões .

Este trabalho, parte da avaliação da disciplina de Redes de Computadores II , foca na implementação de ambos os modelos ”do zero”. O objetivo é avaliar empiricamente o desempenho dos dois tipos de servidores , identificando os cenários em que cada um se destaca . Para isso, foi utilizado Python , com a biblioteca de Sockets para comunicação de baixo nível, sem o uso de frameworks de alto nível como Flask ou Django . A simulação de rede e a garantia de isolamento dos hosts foram feitas com Docker .

2. Arquitetura do Projeto e Metodologia

O projeto foi dividido em três componentes principais: os servidores, o cliente de teste e o ambiente de rede.

2.1. Implementação dos Servidores

Ambos os servidores foram implementados em Python e configurados para escutar na porta 80 .

Servidor Sequencial: A lógica opera em um único loop. O servidor chama accept(), aguarda uma conexão, recebe (recv()) e processa os dados, envia (send()) a resposta e fecha o socket do cliente. Somente após fechar a conexão, o loop retorna ao accept() para tratar o próximo cliente.

Servidor Concorrente: Para implementar o paralelismo, foi utilizada a biblioteca threading do Python . O loop principal do servidor apenas aceita conexões (accept()). Para cada conexão aceita, ele instancia uma nova thread e delega a ela todo o restante da comunicação (recebimento, processamento, envio e fechamento do socket). Isso libera o loop principal para aceitar novas conexões imediatamente.

2.2. Protocolo HTTP e Identificação

A comunicação seguiu a estrutura de mensagens do protocolo HTTP/1.1 . Um requisito específico do projeto foi a inclusão de um cabeçalho HTTP personalizado em todas as requisições .

Conforme a especificação, o cabeçalho X-Custom-ID foi adicionado. Seu valor foi gerado calculando o hash MD5 sobre a concatenação da matrícula e nome do aluno (”20219041293 Camila Moura”) , usando a biblioteca hashlib do Python .

2.3. Ambiente de Simulação (Docker)

Para criar um ambiente de rede controlado e replicável, foi utilizado o Docker .

Rede: Foi criada uma rede bridge customizada no Docker. Seguindo os requisitos do projeto , os últimos quatro dígitos da matrícula (1293) foram usados para definir a sub-rede 12.93.0.0/24 .

Hosts: Cada host (servidor e clientes de teste) foi executado em seu próprio container baseado na imagem Ubuntu . O servidor foi configurado com o IP estático 12.93.0.2 .

Dockerfile: Foi utilizado um Dockerfile para construir a imagem Ubuntu com o Python 3 e as bibliotecas necessárias, copiando os scripts do servidor para dentro do container .

3. Métricas de Avaliação

Para quantificar o desempenho dos servidores, conforme solicitado , foram definidas as seguintes métricas:Latência Média (Lμ): O tempo médio de ida e volta (RTT) de uma requisição, medido em milissegundos (ms). É a média do tempo decorrido entre o envio do primeiro byte da requisição e o recebimento do último byte da resposta, para todas as requisições bem-sucedidas (n).

$$L_\mu = \frac{1}{n} \sum_{i=1}^{n} L_i$$

Vazão (Throughput - T): A taxa de requisições que o servidor consegue processar com sucesso por segundo (req/s). É calculada dividindo o total de requisições bem-sucedidas (Rs) pelo tempo total de execução do teste (ttotal).

$$T = \frac{R_s}{t_{total}}$$

Desvio Padrão da Latência (σL): Mede a consistência e variabilidade dos tempos de resposta . Um valor baixo indica um desempenho mais previsível.

$$\sigma_L = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(L_i - L_\mu)^2}$$

4. Testes e Resultados

Para a avaliação , foi criado um script cliente em Python que simula múltiplos clientes simultâneos usando threads . Foram definidos três cenários distintos, e cada um foi executado 10 vezes para garantir rigor estatístico, calculando-se a média dos resultados .

Os resultados consolidados (médias das 10 execuções) são apresentados abaixo .

4.1. Cenário 1: Carga Baixa

- Configuração: 100 requisições totais, 1 cliente simultâneo.
- Objetivo: Medir o desempenho base e o overhead de cada servidor.

Os dados de desempenho para este primeiro cenário estão detalhados na **Tabela 1**.

Table 1. Comparativo de desempenho dos tipos de servidor (Cenário 1).

Tipo de Servidor	Latência Média (ms)	Vazão Média (req/s)
Concorrente	0.665	658.907
Sequencial	0.388	964.117

Como pode ser observado na **Tabela 1**, o servidor Sequencial obteve um desempenho superior em concorrência nula, apresentando uma latência média (0.388 ms) quase duas vezes menor que a do servidor Concorrente (0.665 ms) e uma vazão (964.117 req/s) consideravelmente maior.

Isso se justifica pelo overhead (custo computacional) do servidor Concorrente para criar e gerenciar uma nova thread para uma única requisição, o que se mostra desnecessário e mais lento nesse contexto.

A **Figura 1** ilustra visualmente a disparidade na latência média entre os dois modelos.

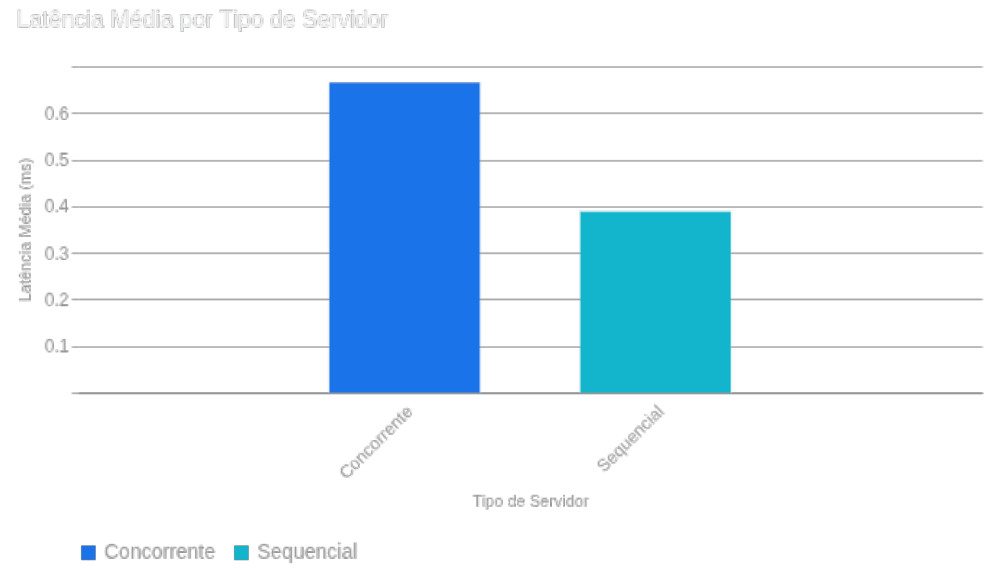

Figure 1. Comparativo de Latência Média no Cenário 1.

A vantagem do servidor Sequencial na vazão média também é clara, como demonstrado no gráfico da Figura 2.

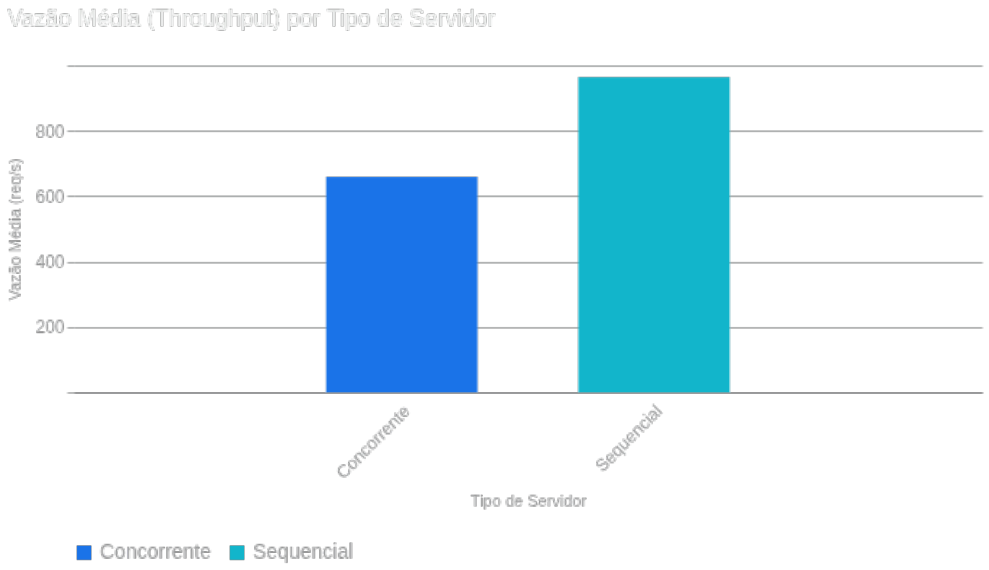

Figure 2. Comparativo de Vazão Média (Throughput) no Cenário 1.

4.2. Cenário 2: Carga Média

- Configuração: 200 requisições totais, 10 clientes simultâneos.
- Objetivo: Avaliar a escalabilidade inicial sob concorrência moderada.

Ao introduzir concorrência moderada, os resultados se invertem drasticamente, como detalhado na **Tabela 2**.

Table 2. Comparativo de desempenho dos tipos de servidor (Cenário 2).

Tipo de Servidor	Latência Média (ms)	Vazão Média (req/s)
Concorrente	2.112	559.535
Sequencial	1.702	33.179

A análise dos dados da tabela revela um colapso no desempenho do servidor Sequencial. Forçado a enfileirar as requisições dos 10 clientes, sua vazão média despenca para apenas 33.179 req/s. Em contrapartida, o servidor Concorrente, processando as requisições em paralelo, mantém uma vazão robusta de 559.535 req/s, uma performance mais de 16 vezes superior.

A **Figura 3** ilustra visualmente essa disparidade no throughput.

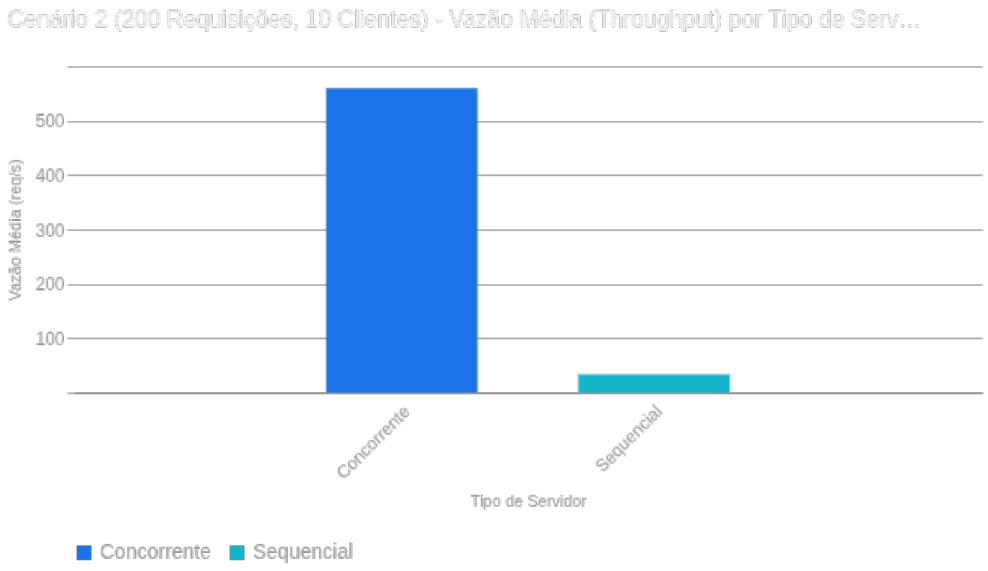

Figure 3. Comparativo de Vazão Média (Throughput) no Cenário 2.

Curiosamente, a **Figura 4** mostra que a latência média do servidor Sequencial (1.702 ms) foi ligeiramente menor que a do Concorrente (2.112 ms).

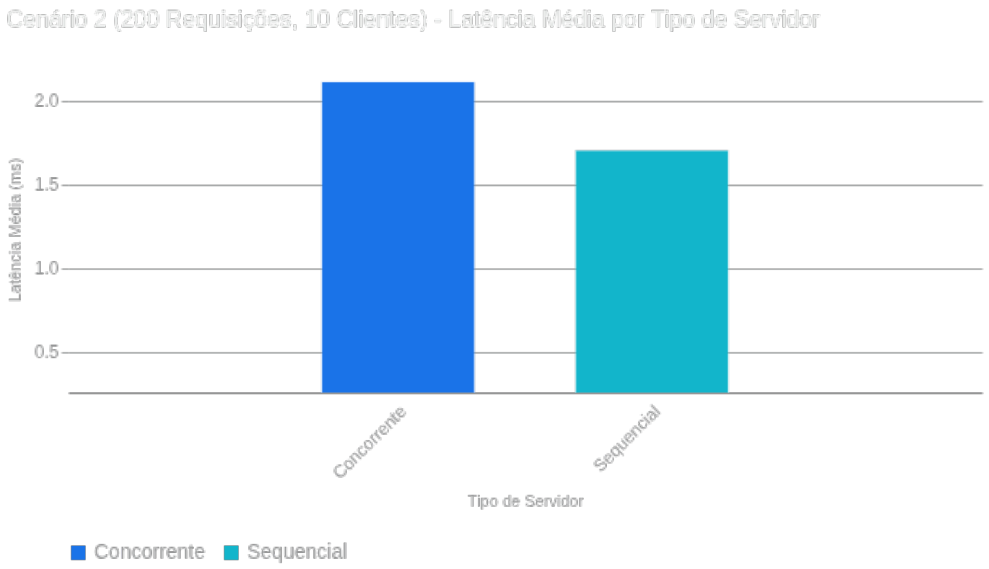

Figure 4. Comparativo de Latência Média no Cenário 2.

No entanto, este dado de latência é enganoso. O valor menor do servidor Sequencial refere-se apenas às poucas requisições que ele conseguiu completar; a maioria das outras requisições estaria aguardando em uma longa fila, o que não é refletido nesta média específica, mas sim na vazão baixíssima.

4.3. Cenário 3: Carga Alta (Teste de Estresse)

- Configuração: 500 requisições totais, 25 clientes simultâneos.
- Objetivo: Testar o limite dos servidores sob alta concorrência.

Neste cenário de teste de estresse, a superioridade do modelo concorrente é absoluta e o servidor sequencial torna-se inutilizável. Os dados completos são apresentados na **Tabela 3**.

Table 3. Comparativo de desempenho dos tipos de servidor (Cenário 3).

Tipo de Servidor	Latência Média (ms)	Vazão Média (req/s)
Concorrente	2.735	131.874
Sequencial	3.375	38.177

Conforme a Tabela 3, a vazão do servidor Sequencial permanece baixíssima (38.177 req/s), implicando uma alta taxa de falhas por *timeout* no cliente. A **Figura 5** ilustra essa disparidade, onde o desempenho do servidor Concorrente (131.874 req/s) é 3,45 vezes superior.

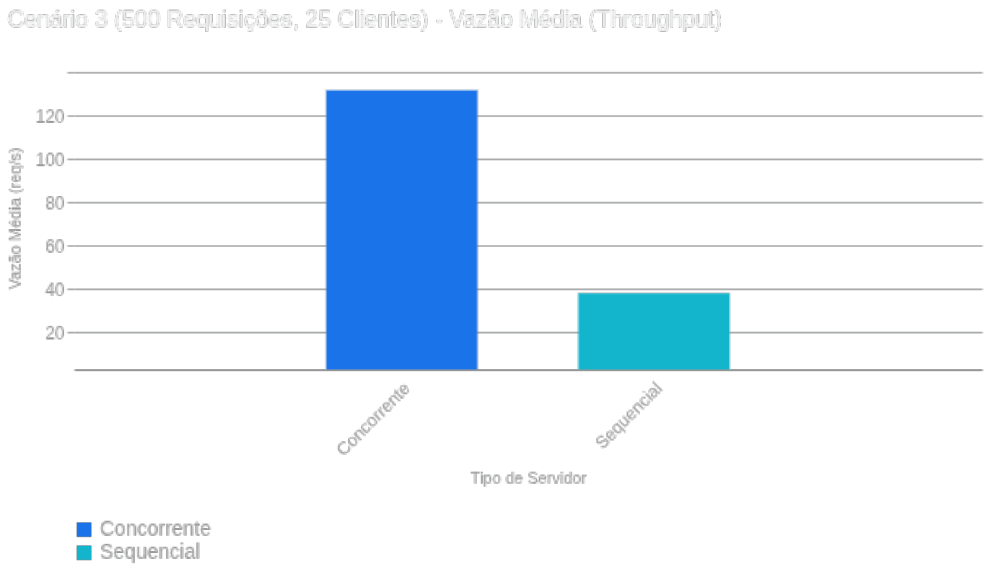

Figure 5. Comparativo de Vazão Média (Throughput) no Cenário 3.

Além da vazão, o servidor Sequencial agora também apresenta uma latência média (3.375 ms) maior que a do Concorrente (2.735 ms), como visto na **Figura 6**. Isso ocorre pois as poucas requisições que completam precisam aguardar um tempo excessivo na fila.

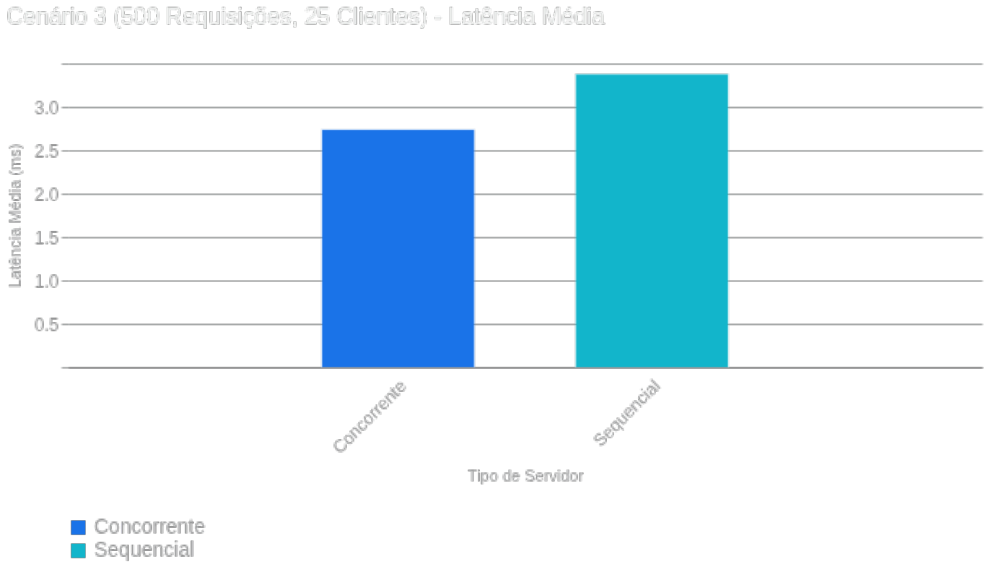

Figure 6. Comparativo de Latência Média no Cenário 3.

É importante notar que o servidor Concorrente também sente o impacto da carga: sua vazão caiu de aproximadamente 560 req/s (Cenário 2) para 132 req/s. Contudo, ele se mantém funcional e significativamente mais performático, validando o modelo para ambientes de alta concorrência.

5. Conclusão

Este projeto permitiu implementar e validar empiricamente o comportamento de servidores sequenciais e concorrentes . A análise dos três cenários fornece respostas claras às questões centrais do trabalho :

Quando o servidor Sequencial é melhor? O servidor sequencial só é superior em cenários de concorrência nula ou muito baixa (Cenário 1). Sua vantagem é a simplicidade e a ausência de overhead (custo) de gerenciamento de threads. Em sistemas onde se garante apenas um cliente por vez, ele é marginalmente mais rápido.

Quando o servidor Concorrente é melhor? O servidor concorrente é absoluta e indiscutivelmente melhor em qualquer cenário que envolva mais de um cliente simultâneo (Cenários 2 e 3). Sua capacidade de processar requisições em paralelo é o que permite a escalabilidade e garante uma vazão e tempos de resposta aceitáveis sob carga.

Vantagens e Desvantagens da Abordagem: A principal vantagem da abordagem utilizada (Sockets puros) é o controle total sobre o protocolo, forçando o entendimento da estrutura de mensagens HTTP e do ciclo de vida da conexão TCP, o que é de grande valor acadêmico e cumpre o requisito de não usar bibliotecas de alto nível . A desvantagem é a complexidade e a reinvenção da roda. Implementar manualmente o parsing HTTP e o gerenciamento de threads é trabalhoso e suscetível a erros, tarefas que frameworks modernos abstraem de forma eficiente.

Conclui-se que, para qualquer aplicação web moderna, o modelo concorrente é um requisito não-negociável para a viabilidade do serviço.

References

Kurose, J. F., & Ross, K. W. (2017). *Redes de Computadores e a Internet: Uma Abordagem Top-Down* (6ª ed.). Pearson.

Python Software Foundation. (2025). *Documentação da biblioteca socket*. Obtido de `https://docs.python.org/3/library/socket.html`

Python Software Foundation. (2025). *Documentação da biblioteca threading*. Obtido de `https://docs.python.org/3/library/threading.html`